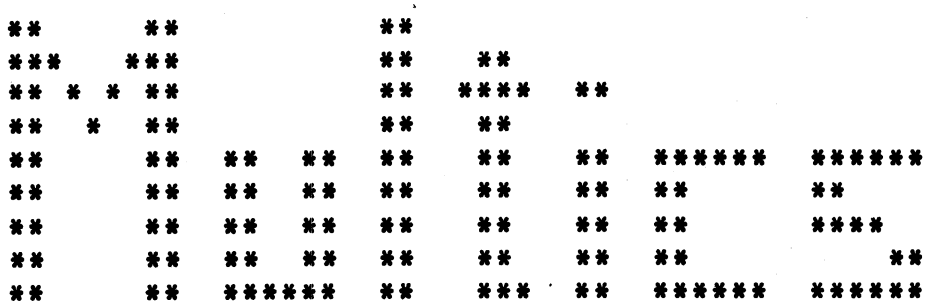
Honeywell

SYSTEM DUMP ANALYSIS
PROGRAM LOGIC MANUAL



RESTRICTED DISTRIBUTION

PREPRINT EDITION

PREPRINT EDITION

Honeywell

SYSTEM DUMP ANALYSIS
PROGRAM LOGIC MANUAL

SERIES 60 (LEVEL 68)

MULTICS

RESTRICTED DISTRIBUTION

SUBJECT:

Guide to Analyzing System Failure and Malfunction

SPECIAL INSTRUCTIONS:

This Program Logic Manual (PLM) describes certain internal modules constituting the Multics System. It is intended as a reference for only those who are thoroughly familiar with the implementation details of the Multics operating system; interfaces described herein should not be used by application programmers or subsystem writers; such programmers and writers are concerned with the external interfaces only. The external interfaces are described in the Multics Programmers' Manual, Commands and Active Functions (Order No. AG92), Subroutines (Order No. AG93), and Subsystem Writers' Guide (Order No. AK92).

As Multics evolves, Honeywell will add, delete, and modify module descriptions in subsequent PLM updates. Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions.

This PLM is one of a set, which when complete, will supersede the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96).

DATE:

June 1975

ORDER NUMBER:

AN53, Rev. 0

PREFACE

Multics Program Logic Manuals (PLMs) are intended for use by Multics system maintenance personnel, development personnel, and others who are thoroughly familiar with Multics internal system operation. They are not intended for application programmers or subsystem writers.

The PLMs contain descriptions of modules that serve as internal interfaces and perform special system functions. These documents do not describe external interfaces, which are used by application and system programmers.

Since internal interfaces are added, deleted, and modified as design improvements are introduced, Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions. To help maintain accurate PLM documentation, Honeywell publishes a special status bulletin containing a list of the PLMs currently available and identifying updates to existing PLMs. This status bulletin is distributed automatically to all holders of the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96) and to others on request. To get on the mailing list for this status bulletin, write to:

Large Systems Sales Support
Multics Project Office
Honeywell Information Systems Inc.
Post Office Box 6000 (MS A-85)
Phoenix, Arizona 85005

This PLM is intended for use by those persons who wish to analyze Multics crashes or various system anomalies. Since problem analysis of this sort requires a working familiarity with many parts of the Multics system, it is presumed that the reader of this PLM has this familiarity. In particular, the following PLMs should have been read before attempting to understand this PLM:

Process and Processor Control	Order No. AN60
Storage System	Order No. AN61
Supervisor Input/Output	Order No. AN65
System Initialization	Order No. AN70
Multiprogramming and Scheduling	Order No. AN73
Bootload Operating System (BOS)	Order No. AN74

The reader is warned at the outset that this manual is not intended to be a complete decision tree over which one may travel with a dump to arrive at the cause for the crash. Instead, it lists those items most generally referenced when analyzing a dump and gives some direction as to where to look next for the cause of the problem.

The System Debuggers' Handbook, Order No. AN87, should be consulted for the detailed formats of status words, control unit data, fault codes, and other information read or shared by hardware. Also included therein are some formats peculiar to the Multics software environment, such as stack frame formats. No attempt is made to duplicate any of that information in this manual. That publication is a crucial tool in any attempt at crash analysis.

The following PLMs are referenced frequently in this manual. For convenience, their titles are shortened as follows:

System Tools	Tools PLM
Storage System	Storage System PLM
Supervisor Input/Output	Supervisor I/O PLM
System Initialization	Initialization PLM
Multiprogramming and Scheduling	Multiprogramming PLM
System Debuggers' Handbook	Debuggers' Handbook PLM

Multics requires the use of a Front-End Network Processor (FNP) to handle two-way information transmission between an IOM data channel and remote terminals. The FNPs referenced in this document may be either DATANET 355 Front-End Network Processors or DATANET 6600 Front-End Network Processors; their use with the Multics system is completely interchangeable.

CONTENTS

Page

ILLUSTRATIONS

TABLES

SECTION I

CRASH PROCEDURES

This section covers the information necessary to understand how Multics crashes (i.e., returns to BOS), how dumps are taken, and how these dumps are processed.

RETURNING TO BOS

There are six ways in which Multics can crash. The first, and most common, way is for some type of fatal error to be discovered by Multics. When this happens, a brief message that describes the surface cause of the crash (e.g., LOCK NOT EQUAL PROCESSID) is typed on the operator's console and then BOS is reentered. To effect this reentry, syserr (which typed the message on the operator's console) calls privileged_mode_ut$bos_and_return. This program picks up the interrupt pattern called scs$sys_trouble_pattern and executes a SMIC instruction. The handler for this "system trouble" interrupt is the interrupt interceptor ii.

The first task performed by ii is to send system trouble interrupts to any other processors that may be running. The other processors take this interrupt and it is processed by ii. However, ii sets a variable, trouble_processor, in its linkage section so that when it handles the trouble interrupts on the other processors, it recognizes this fact and does not get into a loop sending trouble interrupts.

After the initial processor has informed the other processors of the impending crash, it saves all the system controller masks for system controllers for which it is control processor. In addition, the trouble interrupt machine conditions on the Processor Data Segment (PRDS) are copied into trouble_save_data on the PRDS. Both of these save operations are performed so that Multics may be restarted from BOS with a GO command. Since this is generally not done in the case of a crash, no more will be said about it at this time. As each processor enters ii after receiving the trouble interrupt from

the initial processor, it saves the masks of system controller for which it is control processor and copies the trouble interrupt data to trouble_save_data on its PRDS.

Clearly only one processor can return to BOS and this processor is the bootload CPU. All other processors execute a DIS instruction in ii. Regardless of which processor is the initial processor to enter ii, it is the bootload CPU that returns to BOS.

The method used to enter BOS is as follows: the bootload CPU loops for a while to allow all pending I/O operations to quiesce. Since this loop is inhibited, the lockup fault vector is patched to ignore any lockup faults taken during this loop. Once the loop is completed, the two instructions, an SCU and a TRA instruction pair, as picked up from location 4 of the BOS toehold (absolute location 4004) are patched into the DERAIL fault vector. The previous contents of this fault vector are saved prior to doing so. Finally, a DERAIL instruction is executed. Since the derail fault vector was patched, the SCU is done and BOS is entered in absolute mode via the TRA instruction. (See the Bootload Operating System (BOS) PLM, Order No. AN74, for information about the BOS toehold and the steps taken by BOS when entered at location 4004.)

The second way that Multics can crash and enter BOS is for an error to occur that cannot be reported by a syserr message on the operator's console. These errors are the arrival of an interrupt while the processor is using the PRDS as a stack, a page fault while the processor is using the PRDS as a stack, certain faults while the processor is using the PRDS as a stack, or a spurious trouble interrupt. More information is provided later about these errors. When any of the above errors occur, control goes either immediately to the system trouble code of ii that was just described, or it gets to this code by the forcing of a sys_trouble interrupt. In either case, BOS is reentered as described above.

Another way that BOS can be entered is by an execute fault. An execute fault is caused by the depression of the EXECUTE FAULT button on any processor maintenance panel. The handler for this fault is ii and the fault is treated in exactly the same way as a system trouble interrupt. In addition, the fault data stored by the execute fault is stored in the same place on the PRDS as the system trouble interrupt data.

Another way that BOS can be entered is by the manual execution of an XED instruction at location 4000 octal (the BOS toehold). The two instructions in the toehold are executed by placing an inhibited XED 4000 in the processor DATA switches (octal pattern 004000717200). The processor is put in MEM step mode and then the EXECUTE SWITCHES pushbutton is depressed once and the STEP button is depressed several times. Then the processor is taken out of MEM step mode and the STEP button is

depressed once more to cause the two instructions at location 4000, an SCU and TRA, to be executed.

It should be pointed out that of the last two ways mentioned for entering BOS, the execute fault is the normal way used to crash the system. This would be done for example when it is noticed that Multics is in a loop. The execute fault method ensures that all processors are stopped via system trouble interrupts before returning to BOS. The manual XED of location 4000 is used when running only one processor and when it is desired to start Multics again with a BOS GO command after perhaps doing some patching or dumping of Multics from BOS.

It is also possible to return to BOS in a restartable manner during initialization, under control of patterns in the processor DATA switches. This causes the sending of a sys_trouble interrupt to the bootload CPU from pmut$call_bos. For more details see the System Initialization PLM, Order No. AN70.

The final way that BOS can be entered is via an explicit call to hphcs_$call_bos, which may be invoked by the Initializer bos command, or certain failures in administrative ring initialization. Any sufficiently privileged process can call this entry from the user ring.

TAKING A DUMP

When BOS is entered at its toehold via any of the six ways just described, it saves the state of the processor and all live registers. In addition, it copies the first 40000 octal locations of the low-order memory to the BOS partition on disk so that most BOS commands can run without destroying the Multics core image. It should be noted that some BOS commands will destroy the Multics core image (i.e., use memory above location 40000). These BOS commands should not be run if it is desired to preserve the Multics core image. These commands are LD355, SAVE, RESTOR, TEST, and LOADDM. If it is necessary to run one of these commands and yet still preserve the Multics core image, the BOS commands CORE SAVE and CORE RESTOR can be used to save the Multics core image on tape. See the BOS PLM for details.

There are two primary ways to dump Multics. The first way is to use the BOS DUMP command as described in the BOS PLM. The DUMP command dumps various segments of one or more processes to either the line printer or tape. The BOS DMP355 command can be used to dump the DATANET 6600 Front-End Network Processor (FNP) on the printer or tape. This, of course, is a rather time-consuming process and lengthens the time involved in bringing the system up again after a crash. There are, however, times when this must be done. Most typical is the case where Multics crashes before the previous FDUMP command (see below) has been processed. In this case, the most reasonable alternative would be to take a dump on tape. This tape can be printed after

bringing Multics back up by using the print_dump_tape (pdt) command. The print_dump_tape command is described in Section VIII.

The more usual procedure for taking dumps is to use the BOS FDUMP (Fast DUMP) command. FDUMP writes the Multics "core image" out to the DUMP partition on disk. The dump is preceded by a small directory that describes where to find various segments. The term "core image" is used inexactly in this case. What actually happens is that the FDUMP command scans the Active Process Table (APT) (described partially in Section V and more completely in the Multiprogramming PLM, and dumps all processes that are in the running state, have the dbr_loaded bit on in their Active Process Table Entry (APTE), or have the stop_pending bit on in their APTE. If FDUMP is called with the SHORT argument, those processes with the dbr_loaded bit on in their APTE (usually one per processor) are dumped in the normal fashion (i.e., all write permit segments are dumped) and <u>all</u> other processes have their descriptor segment, Process Data Segment (PDS), and Known Segment Table (KST) dumped. If FDUMP is called with the LONG argument, <u>all</u> processes are dumped in the normal fashion.

To dump a process, the FDUMP command scans the descriptor segment of that process and dumps each segment with the write permit bit on in its Segment Descriptor Word (SDW). Since all processes share at least the segments of ring 0, FDUMP only dumps a segment once, even if all processes being dumped have an SDW with the write permit bit on for that segment. Of course, not all pages of each segment are in main memory, so FDUMP interprets each Page Table Word (PTW) for a segment. If the fault bit is off, FDUMP can dump that page of the segment from core. If the fault bit is on, FDUMP interprets the secondary storage or Paging Device address that is stored in the PTW and it reads the page from that location and dumps it.

Figure 1-1 below depicts the layout of the DUMP partition following execution of the BOS FDUMP and FD355 commands. Once the FDUMP and/or FD355 commands are executed, standard crash recovery procedures can be initiated (e.g., Emergency Shutdown (ESD), Salvager, etc.) and the system bootloaded again. To process the fdump (the image produced by the FDUMP command), the command copy_dump (described in Section VIII) must be used. This command uses the gate hphcs_ and therefore is generally executed by Initializer.SysDaemon, to determine whether the DUMP partition

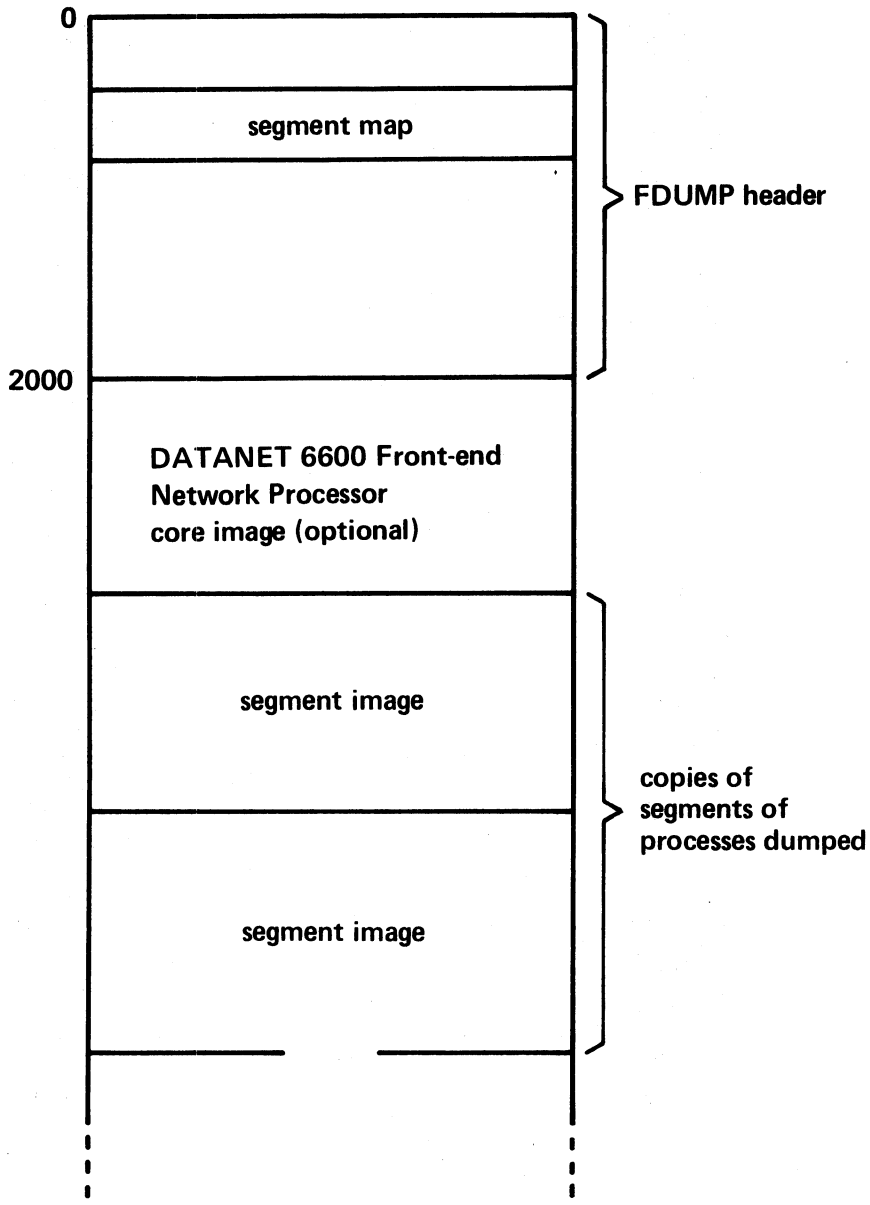


Figure 1-1. Format of DUMP Partition

contains a valid dump (dump.valid = "1"b--see include file bos_dump.incl.pl1). If it does, the Multics fdump is copied into one or more segments in the directory >dumps. These segments have the name date.time.n.erf_no where date is in the form MMDDYY, time is in the form HHMMSS, n is a number, starting at 0, incremented by one each time an FDUMP is taken, and erf_no is the error report form number as extracted from dump.erfno. If there is a valid FNP dump (dump.valid_355 = "1"b), it is copied into a segment in >dumps named date.time.0.erf_no.355. The error report form number is maintained (incremented each time) by the BOS FDUMP command. The number can be set to a new value (e.g., 1) at any time by typing FDUMP n where n is the new error report form (ERF) number (crash number).

DUMPING THE INITIALIZER PROCESS

Although the FDUMP command is the normal way to take a dump, there is one important circumstance when FDUMP should not be used. When the initializer process runs into trouble (e.g., will not respond to any commands) and the system must be crashed, the crash should usually be brought about by an execute fault. Of course, whatever process is running on the bootload processor is the process that actually returns to BOS as described above. If an FDUMP were to be taken, the initializer process would only be dumped if it were running on any of the processors. Otherwise, the only way to cause it to be dumped would be to use the LONG argument to FDUMP. This dumps every process. It also produces a massive dump if a number of processes exist in the system.

A better way to ensure that the initializer process is dumped is to dump it using the BOS DUMP command. To do this, one must switch the Descriptor Base Register (DBR) value used by BOS. This value is set initially to the DBR value of the process that returned to BOS. To find the DBR value of the initializer process, one must use the BOS PATCH command. The DBR for any process is stored in its APTE at the symbol apte.dbr (see include file apte.incl.pl1). (A further description of an APTE is found in Section V.)

The APTE for the initializer process is always the first APTE in the APT. One may find the offset of the first APTE by looking at the value assigned to the segdef apt in the data segment tc_data. Given these listings or offsets, it is possible to look at tc_data with the PATCH command and find the DBR value for the initializer process. Then the DUMP command should be used. Use the DBR command of DUMP to set the BOS DBR value to the DBR of the initializer. Then take a dump of the initializer process on the printer or tape.

PROCESSING AN FDUMP

If a crash occurs, the Multics and possibly the FNP core images are in several segments in the directory >dumps. There are two commands that can be used to print these dumps. One command, online_dump, or od, is used to print the Multics dump. The other command, online_dump_355, or od_355, is used to process the FNP dump. These command descriptions can be found in Section VIII.

Various useful subroutines used by the online dump facility are also discussed in Section VIII.

If it is desired to merely examine portions of the fdump from a terminal, the command ol_dump (see Section VIII) should be used.

SECTION II

CRASH ANALYSIS

This section provides some basic knowledge necessary to anyone analyzing a dump regardless of the cause for the crash.

EXAMINATION OF REGISTERS

The first block of information available in either an fdump or a dump printed by the BOS DUMP command is the state of various processor registers. The first register of interest is the Procedure Segment Register (PSR). The PSR contains the segment number of the procedure that actually returned to BOS. In all but one case, this should be the segment number of ii. The only case in which this is not true is when BOS is entered by a manual transfer (XED of location 4000). In this case, the PSR is at whatever it is when the processor is stopped to perform the manual execution of the XED instruction. Listed along with PSR is the instruction counter (IC), the Ring Alarm Register (RALR), the A and Q registers, the exponent register, the Interval Timer register, and the index registers. In the case of entry to BOS from ii, only the Interval Timer register has any real possible interest.

Since Multics can enter BOS and be subsequently restarted with a GO command, BOS saves all registers. It also saves all interrupts that come in after Multics has entered BOS. These interrupts are set again (via a SMIC instruction) if a GO command is executed. The interrupts are printed in the dump in the form of the word INTER followed by 12 octal digits. The first 32 bits of the 12 octal digits correspond to the setting of interrupt cells 0-31.

Following the interrupt word in the dump are the values in each of the eight pointer registers. When BOS is entered by ii, pointer register 2 (bp) points to the actual machine conditions that were stored when the cause of the crash actually happened. For example, in the case of a crash with a message, pointer register 2 points to the fault data stored by the system trouble interrupt.

After the pointer registers, the contents of the PTW and SDW associative memories are printed. This data is printed in an interpreted format. Figure 1-10 (SDW Associative Memory Register Format), Figure 1-11 (SDW Associative Memory Match Logic Register Format), Figure 1-12 (PTW Associative Memory Register Format), and Figure 1-13 (PTW Associative Memory Match Logic Register Format) in the Debuggers' Handbook PLM contain the layout of the associative memories as stored in memory. Generally, the associative memory contents are of little use except in debugging hardware problems. One thing to check for if associative memory problems are suspected is nonunique usage counts (i.e., two associative memory slots having the same usage number). Another possibility is for two slots to have the same contents (e.g., two slots in the SDW associative memory pointing to the same segment).

Following the associative memory printout is an interpreted description of what memories are attached to the bootload processor and the size of each memory. The information is printed in two columns. The first column contains the beginning address, 0 mod 64, of a memory. The second column contains the size of that memory in 64-word blocks. There are eight entries in each column, one for each processor port. Listed below is a sample printout for a system with 128k on each of the first three processor ports.

```
COREBLOCKS:     FIRST     NUM
                    0     4000
                 4000     4000
                10000     4000
               NO MEM
               NO MEM
               NO MEM
               NO MEM
               NO MEM
```

Following the display of the memory layout is a printout of the memory controller masks for the memory on each processor port. A memory mask is stored as 72 bits. Bits 0-15 contain the settings of bits 0-15 of the interrupt enable register for a memory. Bits 32-35 contain bits 0-3 of the port enable register for a memory. Bits 36-51 contain the settings of bits 16-31 of the interrupt enable register. Bits 68-71 contain bits 4-7 of the port enable register.

The last set of registers that are stored are the four sets of history registers. These history registers are stored for the Operations Unit (OU), Control Unit (CU), Appending Unit (APU), and Decimal Unit (DU) or EIS portion of the processor. See Figure 1-34 (CU History Register Format), Figure 1-35 (OU History Register Format), Figure 1-36 (APU History Register Format), and "DU History Register Format" in Section I in the Debuggers' Handbook PLM, for formats of these history registers.

The last set of information that is printed with the registers is an interpretive layout of the descriptor segment. Each SDW is printed in an expanded format. Along with the SDW is printed the reference name(s) of the segment. For a directory, this is a full pathname. Segments with null reference names only have no names printed with the SDW. SDWs that are all zero (directed fault zero or segment fault) are not printed.

LAYOUT OF MACHINE CONDITIONS

Whenever any type of fault or interrupt occurs, the state of the processor is saved. This involves saving all live registers and the state of the Control Unit. In all cases, the fault data is saved as shown in Figure 2-1 below. The format of the EIS pointer and length data, as stored by the SPL instruction, is found in Figure 1-26 (EIS Pointers and Lengths Format, Word 0), Figure 1-27 (EIS Pointers and Lengths Format, Word 1), Figure 1-28 (EIS Pointers and Lengths Format, Word 2), Figure 1-29 (EIS Pointers and Lengths Format, Word 3), Figure 1-30 (EIS Pointers and Lengths Format, Word 4), Figure 1-31 (EIS Pointers and Lengths Format, Word 5), Figure 1-32 (EIS Pointers and Lengths Format, Word 6), and Figure 1-33 (EIS Pointers and Lengths Format, Word 7) in the Debuggers' Handbook PLM.

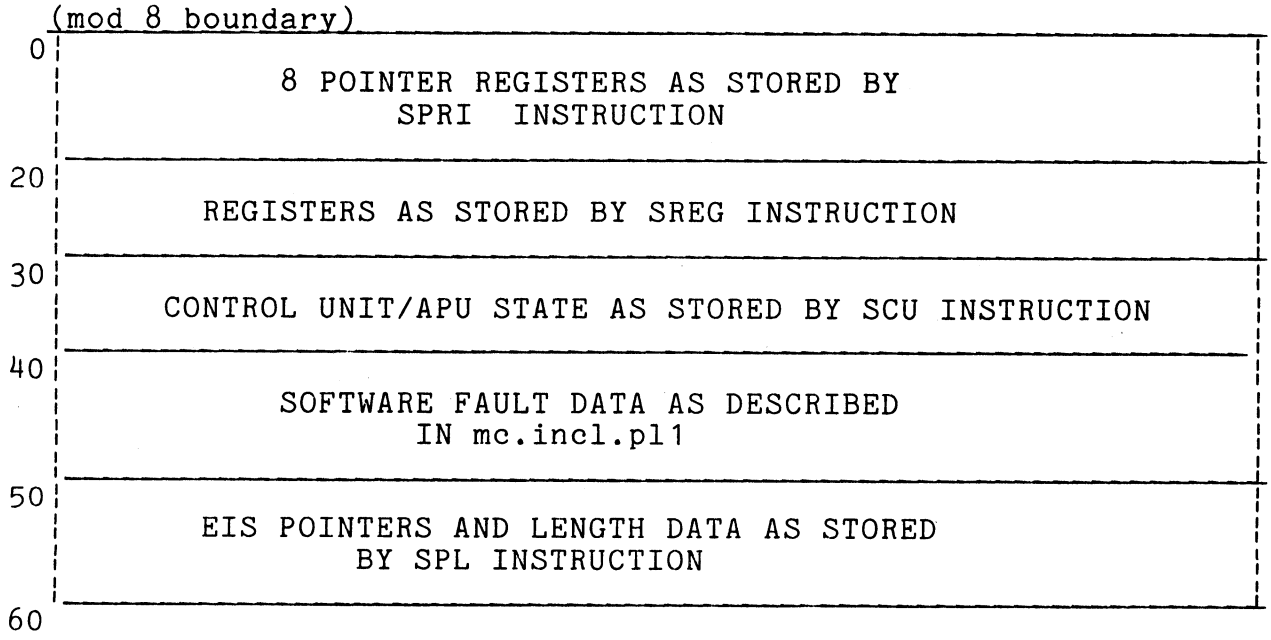


Figure 2-1. Format of Machine Conditions

There are several items in the machine conditions that should be inspected when hardware troubles are suspected. In case of ring violations, the ring fields in the SCU data, PPR.PRR (the ring of execution) and TPR.TRR (the ring of reference) should be checked. Another value to examine is the RALR that is stored by the SREG instruction. The value in the RALR should never be greater than or equal to PPR.PRR. The appending unit status bits (Figure 1-38 (scu Data Format, Word 0), Figure 1-39 (scu Data Format, Word 1), Figure 1-40 (scu Data Format, Word 2), Figure 1-41 (scu Data Format, Word 3), Figure 1-42 (scu Data Format, Word 4), Figure 1-43 (scu Data Format, Word 5), Figure 1-44 (scu Data Format, Word 6), and Figure 1-45 (scu Data Format, Word 7) in the Debuggers' Handbook PLM) are also of interest when attempting to discover what cycle was being executed by the appending unit when an APU produced failure or fault occurred. Another point of interest is that when a fault is taken by the even instruction of an even/odd instruction pair, words six and seven of the SCU data hold the two instructions. In the case of a fault following some levels of indirection, the even instruction (word six of the SCU data) may exist in an altered form since some address modification may have been performed before taking the fault. In the case of a fault taken by the odd instruction of an even/odd instruction pair, word six of the SCU data contains the odd instruction although it may have had some address modification done as just described. Word seven may contain that instruction or a later instruction due to the instruction fetch lookahead feature.

One final note on the addresses and instructions in fault data: certain classes of faults (e.g., parity, store, some cases of command) are detected by the CPU port logic while processing a data request generated by some other unit of the processor. Due to the internal overlap of the CPU, the instructions being processed by the control unit may be several instructions <u>after</u> the instruction that caused the store or other fault. Hence, SCU data for these faults may not be taken as deterministically specifying the faulting reference.

STACK HEADER

Another data base that a person doing crash analysis must be familiar with is the stack. Ring 0 uses both the PDS and PRDS as stacks, and each ring of each process has a stack. (The stack assigned to ring 0 in this assignment is the PDS.) Every stack has a header. The first sixteen words of the stack header are essentially unused and are not discussed further. There are two items in the stack header that are of interest to someone reading a dump. The stack begin pointer points to the first valid stack frame on the stack. On the user ring stacks, the first frame is usually immediately after the stack header. On the PDS and PRDS however, there are intervening data items. The other item of general interest in the stack header is the stack end pointer. It points to the location of the stack where the next frame may

be started. Figure 7-5 (Stack Header Layout) in the Debuggers' Handbook PLM depicts the layout of the stack header. There are three other important pointers in the stack header. The BAR mode sp (stack header location 32 octal) is a place to store the stack pointer register (PR6) before entering BAR mode. This is done since BAR mode programs can validly destroy the word offset portion of the stack pointer register. However, the stack pointer is needed by the signaller, so it is stored here so that it can be reloaded if a fault is taken by the BAR mode program. The translator operator tv pointer is a pointer to a transfer vector of language operator pointers. The PL/I operator pointers (call, push, etc.) have their own locations in the stack header but the transfer vector exists so the pointers to operators for other translators (e.g., BASIC) can be found in a defined way. The ALM segment operator_pointers_ contains the current transfer vector. Finally, the unwinder pointer is provided so that an unwinder program can be found in each ring.

STACK FRAME

The other element of a stack that one must be familiar with is the stack frame. The first forty words are reserved in each stack frame (see Figure 7-6 (Stack Frame Layout) in the Debuggers' Handbook PLM). The first sixteen words can be used for saving the active pointer registers when making a call to another program. The ALM CALL macro saves its pointer registers here and restores them upon return. Since PL/I in general does not depend upon registers across a call, it does not save all the pointer registers. Words sixteen through nineteen contain the pointers that thread the stack frames together on a stack in a forward and backward manner. One can start from the stack begin pointer in the stack header and using the forward pointer in each frame, it is possible to "trace" the stack. Similarly, starting at any given stack frame, it is possible to trace forward or backward using the forward or backward pointers in each stack frame.

In general, the backward pointer of each stack frame is a standard ITS pointer. In the case of some special frames, bits 24-29 are used for flags. Listed below are the flags:

bit 24	on if frame belongs to signal.
bit 25	on if this is a signal caller frame.
bit 26	on if next frame is signaller's.
bit 27	on if this frame was made by the linker. Used for trap before link and trap before first reference.
bit 28	on if this frame belongs to a program that is part of the support environment (e.g., signal_).
bit 29	on if a condition is established in this frame.

At location 24 octal is a pointer to the return point in the program that created the stack frame if that program called out. This pointer is used by the return operator to return to the caller. At location 26 octal is the entry pointer. This is used by the entry sequence code of PL/I. Upon entry to a PL/I program, that program calls the entry operator of PL/I. The entry point to the PL/I program is saved at this double word. Location 32 octal contains a pointer to the operator segment for most translators. However, ALM programs use this double word as a place to store the linkage section pointer. When an ALM program does a call, the call operator reloads pointer register 4 (the linkage section pointer) from this location (it is saved there by the ALM push operator). The reason it is reloaded before calling is in case the ALM program is calling a Version 1 PL/I program that is bound into the same segment as the ALM program. In this case, the standard Version 1 entry sequence that loads the linkage section pointer register is not invoked so that the ALM program must ensure that it is correct. When Version 1 PL/I programs cease to exist, this will no longer be a requirement.

Following the entry pointer is a pointer to the argument list of the procedure that owns the stack frame. The format of an argument list is discussed below. The next two words at locations 34 and 35 octal are reserved. At location 36 octal in the stack frame are two 18-bit relative addresses. These addresses are relative to the base of the stack frame. The first relative address points to a series of 16-word on unit blocks in the stack frame. Each on unit block contains a 32-character condition name, a chain pointer, and some flags. Listed below is the PL/I declaration for an on unit block:

```
dcl 1 on_unit based aligned,
    2 name ptr,
    2 body ptr,
    2 size fixed bin,
    2 next bit (18) unaligned,
    2 flags unaligned,
      3 pl1_snap bit (1) unaligned,
      3 pl1_system bit (1) unaligned,
      3 pad bit (16) unaligned,
    2 file ptr;
```

Details of this may be found in the Limited Command Environment PLM, Order No. AN78. The second relative address is for compatibility with older systems and is discussed no further in

this document. At location 31 in the stack header is a word entitled operator return offset. In fact, this word really consists of two 18-bit halves. The left-most 18 bits contain the translator ID. This is a number that tells what translator compiled the program that owns the stack frame. The various IDs are as follows:

0 Version 2 PL/I
1 ALM
2 Version 1 PL/I
3 signal caller frame
4 signaller frame

The right half of the word holds the address in the program at which a called operator returns. This is useful in debugging, for it describes the return address for certain calls to pl1_operators_. If a crash occurs and the machine conditions show that some fault or interrupt was taken by pl1_operators_, X0 contains the return address for the operator that took the fault or interrupt. If the operator was forced to use X0, then the operator return pointer in the stack frame contains the return address. This cell is zeroed when an operator restores X0 from it. Hence, if this cell is nonzero, it contains the return address. If zero, X0 contains the return address. Given this, one can look at the program that called the operator to determine why the fault occurred.

The last reserved area in the stack frame is at location 40 octal. Here the registers are stored by an SREG instruction. Again, PL/I does not generally save registers since it does not depend upon their contents across a call. The ALM CALL macro however saves the registers here in stack frame. A person tracing a stack should be aware of course that certain programs do not have stack frames. These programs are most typically ALM programs that do not call the push operator. These programs "borrow" the stack frame of their caller and hence should not write into it. Such programs cannot perform standard calls since the call operator writes into the stack frame.

ARGUMENT LIST

Every standard Multics call must construct a standard argument list. Pointer register 0 (ap) is set to point to the argument list. The callee saves this argument pointer in his stack frame as described previously. The argument list format is described below. Figure 2-2 lists the types of argument descriptors.

The argument list must begin in an even word boundary. The pointers in the argument list need not be ITS pointers, however, they must be pointers that can be indirected through. Hence, packed (unaligned) pointers cannot be used.

The i'th argument pointer points at the i'th argument directly. The i'th descriptor pointer points at the descriptor for the i'th argument. The format for a descriptor is as follows:

F — is a flag specifying that this is a new type descriptor. It is a 1 if it is a PL/I Version 2 descriptor and 0 for the old format descriptors.

type — specifies the data type of the variable being described. The PL/I documentation contains a mapping of the actual codes used.

P — indicates, if 1, that the data item is packed.

nd — is the number of dimensions of an array. The array bounds follow the descriptor head in a format described in the PL/I documentation.

size — holds the size (in bits or characters) of string data, the number of structure elements for structure data, or the scale and precision (as two, 12-bit fields) for arithmetic data.

Version 1 Descriptors

Value	Type
1	single precision real integer
2	double precision real integer
3	single precision real floating-point
4	double precision real floating-point
5	single precision complex integer (2 words)
6	double precision complex integer (4 words)
7	single precision complex floating-point (2 words)
8	double precision complex floating-point (4 words)
13	pointer data
14	offset data
15	label data
16	entry data
17-24	arrays of types 1-8
29-31	arrays of types 13-15
514	structure
518	area
519	bit string
520	character string
521	varying bit string
522	varying character string

(514, 518-522 are data types that are not Multics standard)

Figure 2-2. PL/I Descriptors

Value	Type
523	array of structures
524	array of areas
525	array of bit strings
526	array of character strings
527	array of varying bit strings
528	array of varying character strings

(523-528 are data types that are not Mutlics standard)

Version 2 Descriptor

Value	Type
1	real fixed binary short
2	real fixed binary long
3	real float binary short
4	real float binary long
5	complex fixed binary short
6	complex fixed binary long
7	complex float binary short
8	complex float binary long
9	real fixed decimal
10	real float decimal
11	complex fixed decimal
12	complex float decimal
13	pointer
14	offset
15	label
16	entry
17	structure
18	area
19	bit string
20	varying bit string
21	character string
22	varying character string
23	file

Figure 2-2 (cont). PL/I Descriptors

SECTION III

CRASHES WITH NO MESSAGE

This section describes a deterministic algorithm for ascertaining the immediate reason for a system return to BOS with no message. It is the intent of this section and the next to describe an appropriate course of action for determining the immediate cause of a crash, when presented with a dump.

The first quantity to inspect is the PSR in the registers printed by BOS, or online_dump. If the PSR contains the segment number of any segment other than the interrupt interceptor (ii), then the system did not return to BOS of its own volition, and a manual transfer to BOS (XED 4000 or XED 4002) was made by the operator. (Specifically, the PSR/Instruction counter should point to the instruction in ii following the derail instruction that causes the return to BOS).

If the PSR/ICTC points to the correct place in ii, pointer register 2, as printed by BOS or online_dump, points to the machine conditions that caused the bootload processor to return to BOS. If the fault/interrupt code in the second word of the SCU data in these machine conditions is anything other than a system trouble interrupt (octal 44 as it appears there), these machine conditions represent one of the following cases:

1. An execute fault was taken by the bootload processor, i.e., the operator pressed the EXECUTE FAULT pushbutton on this processor. The fault/interrupt code is 37, octal.

2. An interrupt was taken by the interrupt interceptor (ii) while running on the PRDS. The value of sp (pointer register 6) in these machine conditions is an address on the PRDS, and the fault/interrupt code is even (interrupt). A masking problem should be suspected.

3. A fault is detected during initialization, before the mechanism to handle that fault is set up.

If the fault/interrupt code in the SCU data pointed to by pointer register 2 as given by BOS reflects the sys_trouble code, some other module or processor caused this interrupt. All modules that send a sys_trouble interrupt execute NOP instructions (octal 000 000 011 000, but sometimes with direct (03 or 07) modifiers) immediately after sending this interrupt. Hence, if the SCU even/odd instruction words (6 and 7) do not have NOP instructions, one should assume that some processor other than the bootload processor first sent a sys_trouble interrupt. The machine conditions at prds$sys_trouble_data for all running processes should be inspected to find one that was interrupted out of NOPs. (It is possible, however, for a processor to be executing some NOP loop, such as certain locking code at the time a sys_trouble interrupt is received from another processor. If, in a multi-CPU dump, many such sets of sys_trouble data are found, this should be suspected, and the set of conditions that identifies NOP after a SMIC instruction sending sys_trouble found.)

When the processor that started the sys_trouble broadcast has been found, the program that sent the first sys_trouble interrupt must be identified. This can be done by inspecting the PSR in the machine conditions for this first sys_trouble interrupt. If it is the segment number of the fault interceptor (fim), some fault was encountered that required paged programs to handle properly, while running on the PRDS. Pointer register 2 in these machine conditions points to the machine conditions stored at the time of the fault. (If such a fault should happen with the page table lock set while running on a temp-wired PDS, the fim does not detect a problem but attempts to process the fault, usually causing a page fault with the page table lock set, with a resulting crash message, "PAGE: MYLOCK ERROR ON GLOBAL LOCK". In later systems, the fim checks for this case and sends sys_trouble.)

If the PSR identifies bound_page_control, or wired_fim in earlier systems, a page fault was taken in an invalid circumstance. Pointer register 0 in the system trouble machine conditions points to the page fault machine conditions.

If the PSR identifies privileged_mode_ut (in bound_priv_1 in later systems) an explicit call was made to pmut$call_bos. This is always done in the case of fatal crashes with a message, in which case syserr makes this call. One should identify the owner of the stack frame pointed to by sp (pointer register 6) in the sys_trouble data. By <u>owner</u>, we mean the procedure indicated by the return pointer (location 24 octal). (pmut does <u>not</u> push a frame in this case.) If the owner is bound_error_wired (which contains syserr) then a call was probably made to print out a

crash message. The arguments to syserr in a preceding frame should be inspected. In this case, either the message was printed out by the operator's console, or some difficulty may have been encountered in trying to print it out. Otherwise, it may be assumed that privileged_mode_ut was called by some program in the outer rings, and a stack trace should determine the problem.

SECTION IV

CRASHES WITH A MESSAGE

When Multics crashes after printing a message on the operator's console, that message is always printed by syserr. After printing the message, syserr calls privileged_mode_ut$bos_and_return (in bound_priv_1), which sends a system trouble interrupt to the current processor. The receipt of this system trouble interrupt sends similar interrupts to all other processors in the system. When analyzing a dump, look at the system trouble machine conditions on PRDS of each processor. One set of such machine conditions has a PSR equal to the segment number of bound_priv_1. In addition, the even and odd instructions in the SCU data are both NOP instructions since privileged_mode_ut executes NOPs waiting for the system trouble interrupt to go off.

Once the correct machine conditions have been found, pointer register 6 (the stack pointer) contains a pointer to the syserr stack frame. If the segment number in pointer register 6 is for the PRDS, the previous stack frame belongs to the caller of syserr. If, however, the segment number is for the PDS, syserr uses a different convention. syserr makes its stack frame at location 30000 octal on the PDS. It does this so that possibly valuable stack history is not overwritten by its stack frame. This would happen if it laid its frame down right after the frame of its caller. An examination of the stack frame at 30000 shows that it has two frames following it. The first is for wire_stack, a program that wires pages of the PDS so that syserr does not take a page fault running on the PDS. The second is for syserr_real, the program that actually prints the message. Further examination of the stack frame at 30000 shows that the back pointer points to the stack frame of the caller of syserr. This frame is usually quite far back on the stack with the intervening area holding the stack history. To examine this history, it is necessary to know the old forward pointer in the stack frame of the syserr caller since the current forward pointer points to 30000 now. The old forward pointer is saved in location 26 octal of the frame of the caller of syserr. Given this old forward pointer then, it is possible to examine the stack history to see the last set of calls before the syserr call.

SECTION V

MAJOR SYSTEM DATA BASES

This section describes those parts of the system data bases that one might wish to examine after a crash.

SYSTEM SEGMENT TABLE

The System Segment Table (SST) is a variable size (via configuration card) unpaged segment. It holds all the page tables in the system. In addition, it holds control blocks for core management, for paging device management, and for active segment management. Many of the data items in the SST contain the addresses of other items. These addresses are expressed as 18-bit relative pointers to the SST. Figure 5-1 below gives the general layout of the SST.

SST Header

The SST header consists of various control variables and meters. The meters are defined in the include files sst.incl.pl2 and cnt.incl.pl1. These meters are not discussed further in this document. It would be useful to have a copy of this include file in hand before reading further. The first item of interest in the SST header is the page table lock, sst.ptl. This lock is set when taking a page fault and remains set until page control has completed processing the page fault (e.g., initiated a disk read for the page). The page table lock is also used at other times and it locks the header, core map, paging device map, and page tables. It is also a lock on those parts of an ASTE needed by page control. If any processor attempts to lock the page table lock and it is already locked, that processor loops waiting for the lock to be unlocked.

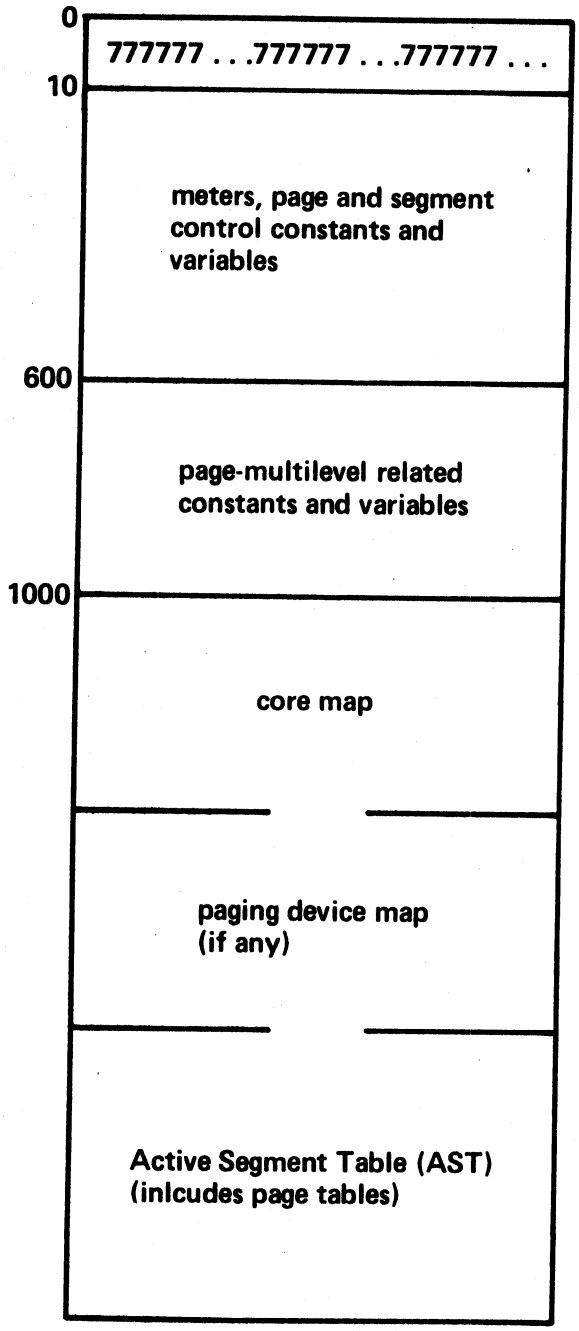


Figure 5-1. Layout of System Segment Table (SST)

Following the page table lock is the AST lock (sst.astl). This lock is generally used only by segment control. Instead of being a loop lock, the AST lock is a wait lock. This means that if a process finds the AST lock locked, it gives up the processor and informs the traffic controller that it wishes to WAIT on this lock. When the process that locked the AST lock is finished, it notifies all processes that are waiting on the lock. The wait/notify mechanism and locking mechanism are described in the Multiprogramming and Scheduling PLM, Order No. AN73. The SST variable, sst.astl_event, contains the event upon which processes contending for the AST lock should wait.

The next item of interest is the pointer to the beginning of the AST, sst.astap. As described below, there is a page table following each ASTE. The maximum size of a page table is 256 PTWs. Clearly it would be wasteful to allocate a maximum size page table for every active segment. Consequently, the ASTEs are broken up into pools where all the ASTEs in a pool have the same number of PTWs. The current pool sizes are 4, 16, 64, and 256. Each element in the array sst.level consists of a pointer to the used list (described below) of ASTEs of a pool, and the number of ASTEs in the pool. There are also special pools for various classes of supervisor and initialization segments (see the Storage System PLM, Order No. AN61 and the Initialization PLM).

The next item of general interest in the SST header is the set of control words for the core map. The variable sst.cmp is a pointer to the start of the core map, and sst.usedp is a relative pointer to the least-recently-used core map entry of the used list (described briefly below and fully in the Storage System PLM. Another variable of interest is sst.fsdctp, a pointer to the ASTE for the FSDCT. There is also a pointer to the ASTE for the root in sst.root_astep. There is room now in the header for a block of meters. Following the meters is a block of information used in management of the paging device. The variable sst.pdmap is a pointer to the paging device map in the SST, sst.pdhtp is a pointer to the paging device hash table, and sst.pdusedp points to the least-recently-used entry of the paging device used list.

Core Map

Directly following the SST header is the core map. The core map consists of a 4-word Core Map Entry (CME) for each 1024-word core block that is configured. Even if a memory is not online, if it is to be added dynamically, each page in the memory must be represented by a CME. Figure 5-2 below depicts a CME.

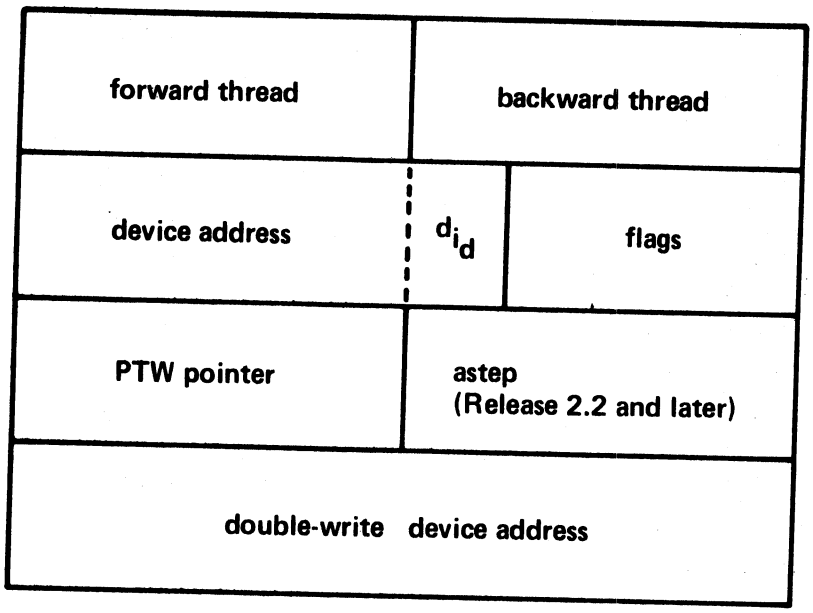


Figure 5-2. Core Map Entry

The first word contains the forward and backward threading pointers. These pointers (actually, they are addresses relative to the base of the SST) are used in the implementation of the Least Recently Used (LRU) algorithm for core management. The header variable, sst.usedp, points to the head of this circular list and in fact points to the CME that represents the block of core least recently used. The LRU algorithm is described fully in the Storage System PLM. One important thing to be checked in a dump analysis is that the CMEs are all threaded together correctly. In Release 2.2 and later systems, CMEs for out-of-service pages and RWS (read-write sequence) buffers are <u>not</u> threaded in.

Each CME holds the device address for the page that resides in the core block represented by that CME. A device address consists of an 18-bit record number and a 4-bit device identification. (The first bit of this device ID indicates the paging device.) The one exception is when the page occupying the core block associated with the CME is a new page and has not yet been assigned a disk address. In this case, a null device address is stored as the device address. Null device addresses may also appear in PTWs. Null device addresses are coded for debugging purposes to be able to tell which program created the null address. Listed below are the null addresses (any address greater than 777000 octal is considered to be a null address):

```
777777  created by append
777001  created by pc$truncate
777002  created by pc$truncate
777003  created by salv_check_map
777004  created by salv_check_map
777005  created by salv_truncate
777006  created when page is zero
777007  created by pc$move_page_table
777010  created by pc$move_page_table
777011  created by get_aste
777012  created by make_sdw
777013  created by deactivate
777014  created by move_file_map
777015  created when page is bad on paging device
```

Listed below are the Multics device ID numbers:

```
1  Bulk Store
2  D191
3  E191
4  D190
5  E190
6  D181
```

If the paging device indicator is not on, then the device address is a disk address. The only consistency check one can make in this case is to look at the PTW pointed to by the PTW pointer in the CME and make sure that the core address in the PTW corresponds to the core block represented by the CME. During a read/write sequence, the PTW pointer is replaced by a pointer to a Paging Device Map Entry (PDME). A simple algorithm to do this is:

```
Let x = ptw.add      (18 bit 0 mod 64 core address)
Let y = sst.cmp      (y is offset of core map in sst)
Then address of CME = y + x/4
```

If this relationship is not true, then either a read/write sequence is in progress (in which case the PTW pointer no longer points to a PTW, but to a PDME), or there is an inconsistency in the SST. It can easily be determined if a read/write sequence is in progress since there is a flag in the CME (cme.rws as defined in cmp.incl.pl1) that indicates this.

If the paging device indicator bit is on, then the record address is an address on the paging device. As described below, there is a 4-word PDME for each record on the paging device. It is possible to find the PDME associated with a particular CME by taking the paging device record number, multiplying it by four, and adding in the offset portion of the pointer in sst.pdmap. It is important to note that this can be a negative offset. This is true, for example, when Multics is only using the last 1000 pages of a 2000 page paging device. Rather than having 1000 empty PDMEs for pages 0-999, the pointer in sst.pdmap is backed up so that the first PDME in the Paging Device Map represents record 1000. Once the PDME is located, several consistency checks can be made on it. Figure 5-3 depicts the format of a PDME. The PDME is defined in cmp.incl.pl1.

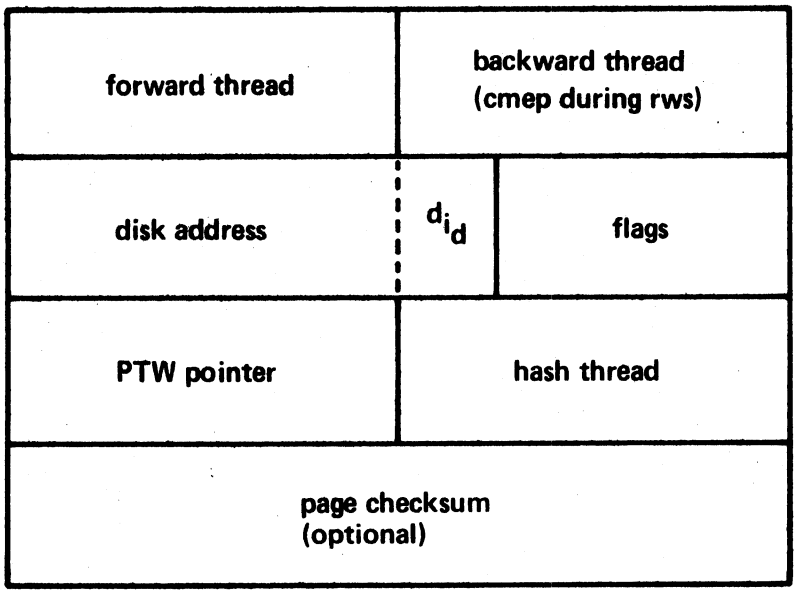


Figure 5-3. Paging Device Map Entry

One check to be made is to make sure that the PTW pointer points to the same PTW as the CME. Another check is to see if the device address is for a disk address. If not, there is an error. Other checks are listed below.

Paging Device Map

The Paging Device Map directly follows the core map in the SST. It has a very similar function in that it is used to manage the 1024-word pages on the paging device in such a manner that the least recently used page on the paging device is the one selected for removal when a new page must be placed on the paging device. This removal process is called a Read-Write Sequence (RWS). It involves reading a page from the paging device and writing it to its secondary storage (disk) address. It is presumed that the reader is familiar with the use of the paging device as described in the Storage System PLM. There are various consistency checks that can be made on the Paging Device Map. First, all PDMEs must be correctly forward and backward threaded. The thread starts with the PDME pointed to by sst.pdusedp. There is one exception to this rule. When a RWS is in progress for a page, its PDME has a zero forward pointer and its back pointer contains the address of the associated CME. Both the CME and the PDME should have the RWS flag on (cme.rws and pdme.rws in cmp.incl.pl1).

Another consistency check one can make is to see if the secondary storage address stored in the PDME is incorrect. One can do this by applying the paging device hashing algorithm to that secondary storage address to see if the PDME in question is on the hash thread. As described in the Storage System PLM, when the paging device hashing algorithm is applied to a disk address, a PDME address is produced. If the disk address in question is not stored in that PDME, the value in pdme.ht is the address of another PDME to look at. Thus, there exists a thread of PDMEs all of which hold disk addresses that produce the same value when the paging device hashing algorithm is applied. To perform the consistency check, take the 18-bit secondary storage record address stored in the PDME, perform a logical AND with the value stored in sst.pd_hash_mask (which is a function of the paging device size), and divide the result by 2. The quotient gives the offset from the base of the hash table (as pointed to by sst.pdhtp) of a pair of hash table addresses. If the remainder of the previous division is 0, use the upper address, otherwise use the lower address. The selected address is either zero (no secondary storage addresses have copies on the paging device) or it is the address of the first PDME in a chain of one or more PDMEs. By following the chain (pdme.ht), the secondary storage address in question should be found or the consistency check has failed. Of course, if the selected hash table address is zero, the check has failed.

Another useful consistency check is to confirm the correctness of the PDME, PTW, and CME association if the page is in core or of the PDME and PTW association if the page is not in core (as determined by the setting of the PDME flag pdme.incore). If the page is not in core then look at the PTW pointed to by pdme.ptwp (if pdme.ptwp is zero, the segment containing that page is not active and hence has no active PTWs). The device address in the PTW must be for the paging device or there is an error. To determine if it is the correct paging device address, multiply the 18-bit paging device record number by 4 (the size of a PDME) and add the offset portion of the pointer stored in sst.pdmap. This should yield as a result the offset of the associated PDME in the SST. If the page is in core, compute the CME address from the PTW pointed by pdme.ptwp as described earlier. The device address in the CME must be for the paging device and the address of the associated PDME can be computed as just described.

For any PTW that has ptw.df on, the PTW must, of necessity, contain a core address. If ptw.df is off, it always contains a device address for all systems earlier than Release 2.2. In the case that this page is being read in (ptw.df = "0"b, ptw.os = "1"b), there is always a CME associated with the PTW which, in systems prior to Release 2.2, must be searched for. In Release 2.2 and later systems, a PTW for a page being read in contains a core address, which allows quick location of the CME. In all other cases, the PTW contains a device address.

Another quick consistency check is that all PDMEs that are free (last three words are zero) must be at the head of the used list. The used list is traced by following forward pointers. The address of the first PDME is stored in sst.pdusedp. Also, the number of free PDMEs in the used list plus the number of PDMEs that have an RWS active (stored in sst.pd_wtct) should equal the value in sst.pd_free.

The last type of check that can be made is really more of a heuristic one. The pdme.abort, pdme.truncated, and pdme.notify_requested flags are rarely on and may be symptomatic when looking for the reason for a crash. Also, the pdme.removing flag should only be on when the associated paging device record is being explicitly deleted by the operator.

Active Segment Table

The Active Segment Table (AST) described earlier contains a number of Active Segment Table Entries (ASTE) and associated page tables. The ASTE is eight words long and basically contains copies of some pieces of directory information about a segment. This information, which can change quite rapidly, may be updated in the ASTE rather than paging in the directory to do the updating each time. Figure 5-4 below shows the format of an ASTE.

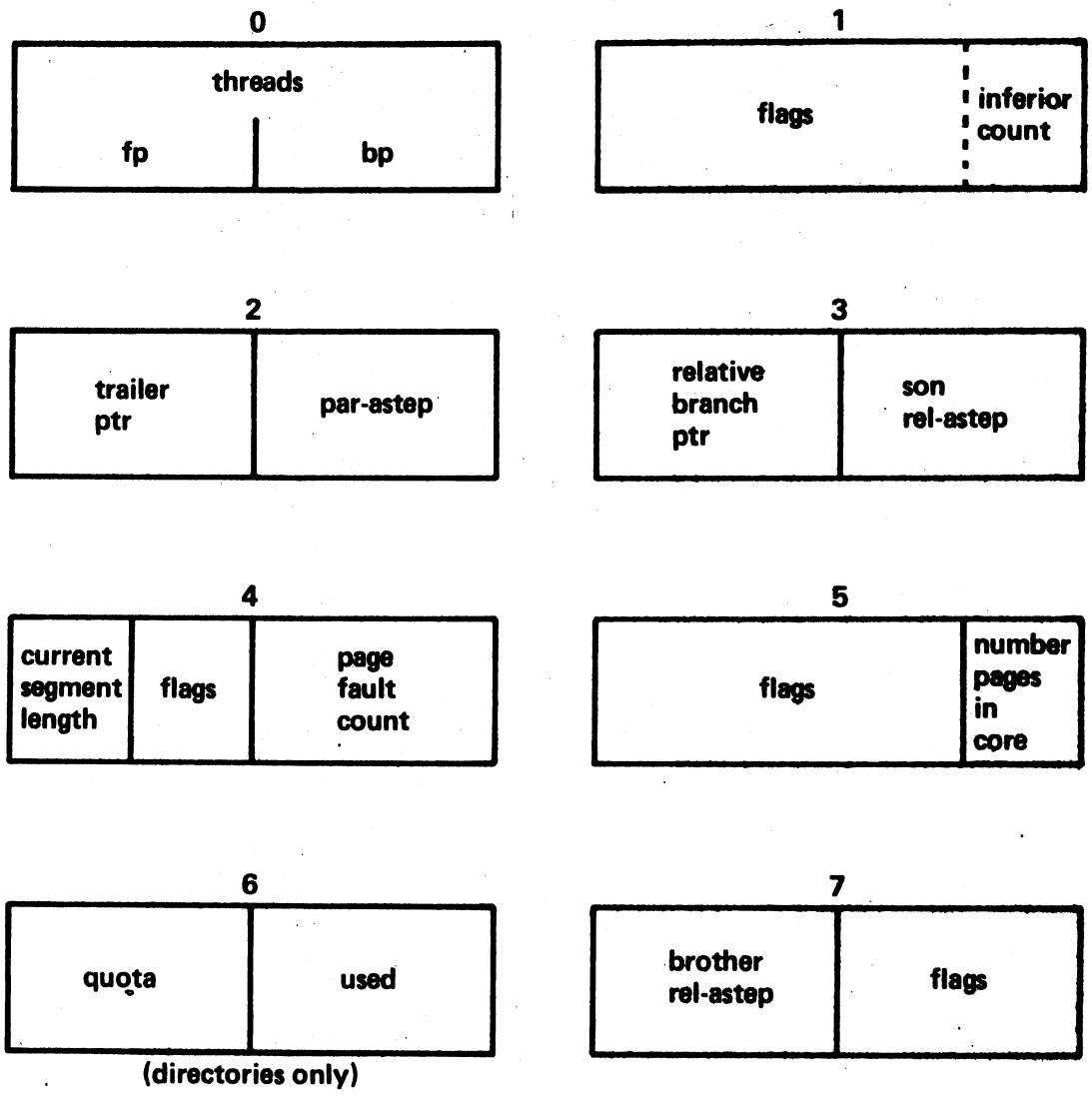


Figure 5-4. Active Segment Table Entry

There are not a large number of consistency checks that one can make on an ASTE. One thing that can be checked is the consistency of the last word of the ASTE. The field aste.marker (as defined in ast.incl.pl1) must have the value 02 in it. Also, aste.ptsi must contain the page table size index. An index of 0 means a page table size of 4 PTWs. An index of 1 is a size of 16, 2 is a size of 64 and 3 is a size of 256. (These indices are used for the array of page table sizes in sst.pts.) Another useful check is to compare the value in aste.np (the number of pages in core) with the PTWs associated with that ASTE. The number of PTWs with the directed fault bit on (in core) should be equal to the value in aste.np.

Another item of interest is that an ASTE with the flag aste.gtms on is almost always an ASTE for a directory. Since the Backup Facility uses this flag, this is not a foolproof indicator. Of course, if aste.ic is nonzero, then that ASTE is guaranteed to be for a directory. Only a directory can have inferior entries. Another check one might want to perform is to see how the information in an ASTE compares with the branch information in the directory (e.g., to compare secondary storage addresses in the PTWs with those in the filemap for the segment). To do this, one must find the ASTE for the containing directory using aste.par_ring. Then the descriptor segments that are dumped must be searched for an SDW whose page table address is equal to the address of the first PTW following the original ASTE in question. If this SDW can be found (if that process wasn't dumped, it can't be found), then the directory pathname is printed in the dump and the branch information in that directory can be found using the value in aste.rep in the original ASTE.

SST Analysis Tools

There are two commands, dump_pdmap and check_sst (described in Section VIII), that perform many of the checks mentioned above. A copy of the SST in an fdump should be extracted using the extract command (described in Section VIII). Then the commands should be run.

ACTIVE PROCESS TABLE

The Active Process Table (APT) is a variable size (via configuration card) data base. It is contained in the unpaged segment tc_data (for traffic controller data). It holds the control blocks called Active Process Table Entries (APTE) for each process in the system as well as some interprocess communication control blocks. Figure 5-5 below gives the general layout of tc_data.

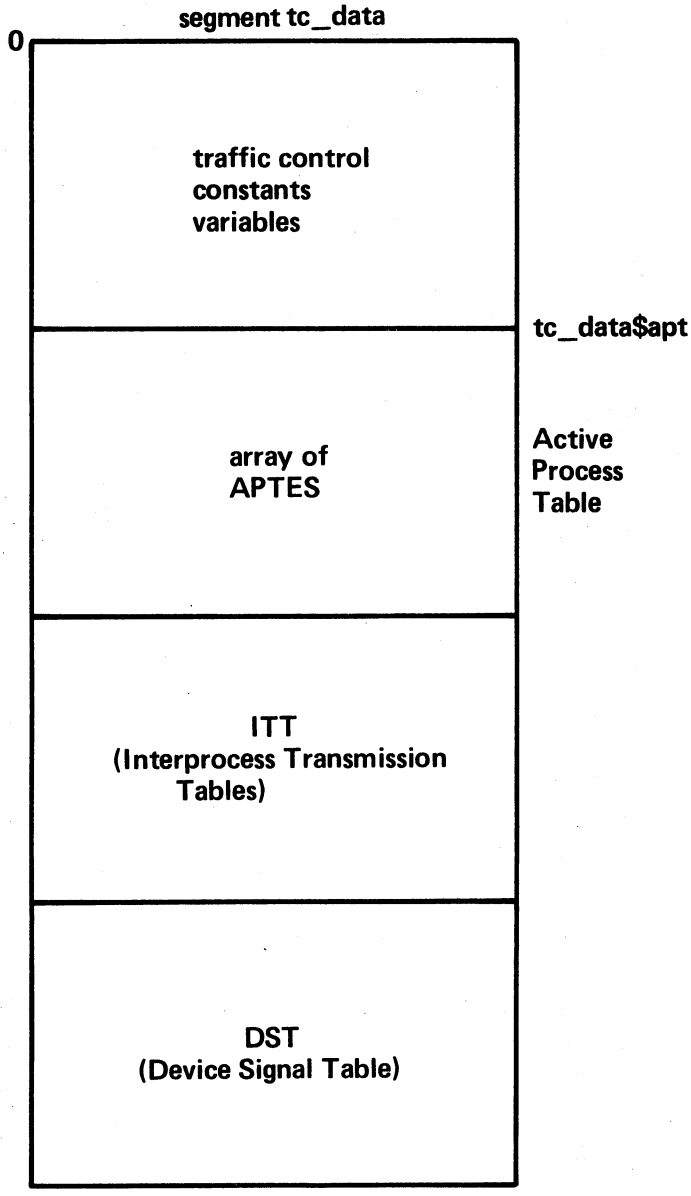


Figure 5-5. tc_data

The header contains a number of meters and variables needed by the traffic controller. This information is given extensive coverage in the Multiprogramming PLM and is not discussed further other than to point out the variable tcm1.ready_q_head (defined in tcm.incl.pl1). Using this variable (also segdefed at tc_data$ready_q_head), it is possible to trace through the ready list finding the APTEs for all processes that are running, eligible to run, ready to run, or waiting. Figure 5-6 below describes the ready list. All other APTEs in the APT that are not threaded into the ready list are in the blocked state or unused state.

The APTE

Generally, when analyzing a crash involving some type of loop, the APT is examined. One usually looks for APTEs waiting for strange events, APTEs in inconsistent states, etc. The format of an APTE is defined in apte.incl.pl1. Generally the thing one looks at first in an APTE is the flags and state word. The states are:

0 Empty (not in use)
1 Running
2 Ready
3 Waiting
4 Blocked
5 Stopped

The flags are covered in the Multiprogramming PLM. In the field apte.processid is stored the process ID for that user. Processids consist of two 18-bit items. The left item is the offset in the APT of the user's APTE. The right most 18 bits hold the last value of a number maintained by the answering service that is incremented each time a user logs in and rolls over at 262144. The next item of general interest in the APTE is the word apte.ipc_pointers. The upper half of this word, if nonzero, is the address of the first of one or more event messages waiting for the process. Event messages are stored in the ITT area of the APT. The format is shown in Figure 5-7 below.

Directly after the ipc thread is a word called apte.ips.message. This word holds 1-bit flags for each of the system-defined ips signals. (These system events are stored in sys_info$ips_mask_data.) There are three types of ips signals. Bit 0 of apte.ips_message is used for the ips signal QUIT. Bit 1 is for CPUT (cpu timer runout). Bit 2 is for ALRM (real-time timer runout).

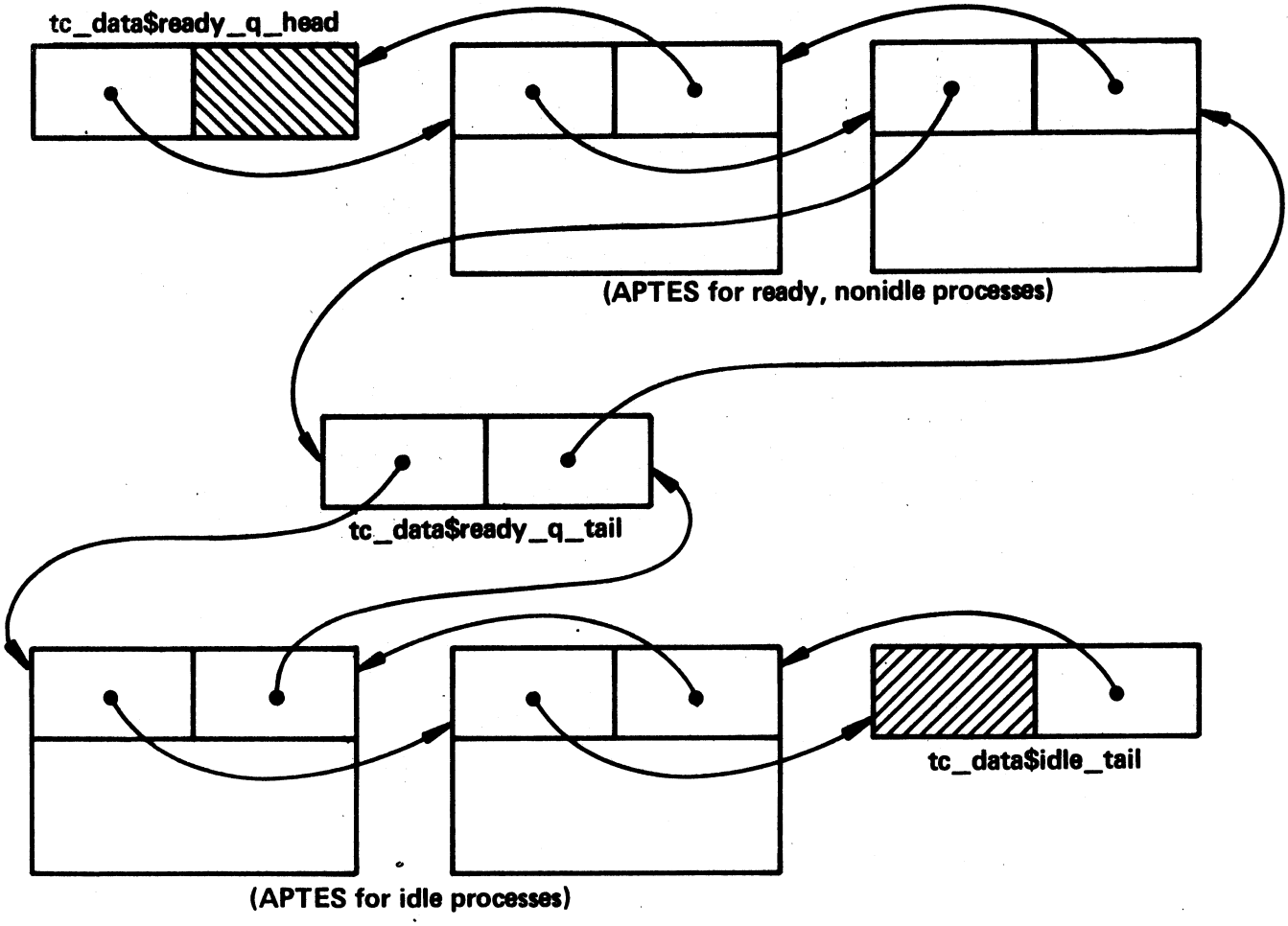


Figure 5-6. Ready List Format

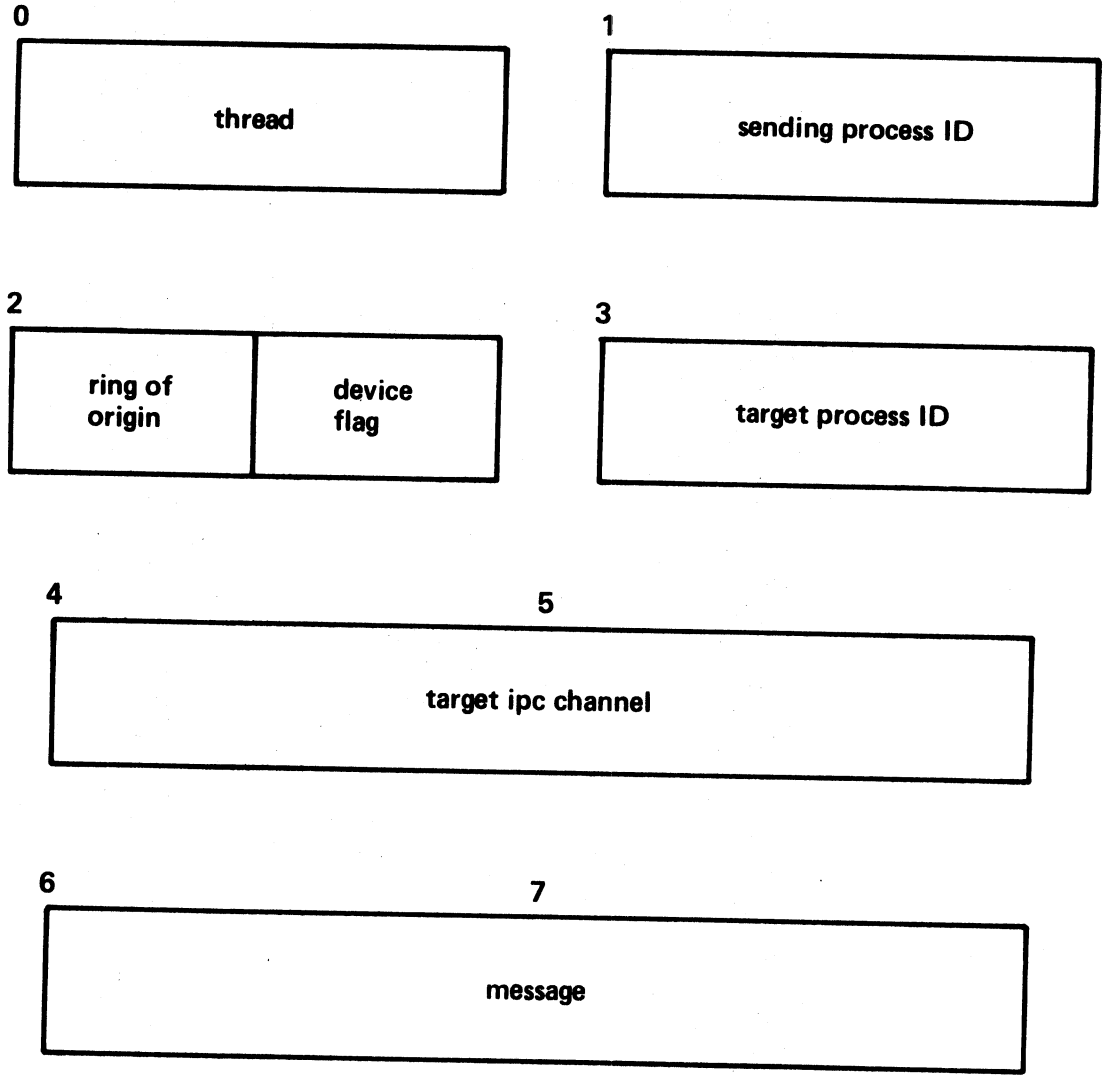


Figure 5-7. Format of ITT Message

The next field of interest in the APTE is labeled apte.asteps. This two-word field holds the relative offset in the SST of the ASTE for the PDS of this process, the offset of the ASTE for the descriptor segment of this process, and the offset in pxss (the traffic controller) of the last call (a TSX7 instruction) to the getwork subroutine. The getwork subroutine of pxss is called when a process must give up the processor it is running on to some other process. By seeing what other subroutine in pxss called the getwork subroutine, it is possible to tell what event caused that process to give up the processor (e.g., end of time quantum, process going blocked, page fault, etc.).

Another item of interest in the APTE is the wait event (apte.wait_event). Listed below in Table 5-1 is the current set of wait events:

Table 5-1. Wait Events

Event	Meaning
000000000071	ttydim waiting for per channel lock
400000000000 (octal)	waiting on AST lock.
"dtm_"	Disk metering waiting on lock.
"free"	Waiting on system_free_seg lock.
"ioat"	Wait on ioat lock.
777777777776 (octal)	Online salvager waiting on lock in salv_data.
777777777777(octal)	Waiting on lock in the root.
000000xxxxxx	If xxxxxx>sst.astap+1, then wait event is offset in SST of PTW for which I/O has been started. Otherwise, the wait event is the offset in the SST of a PDME for which an RWS has been started.
Valid processid	Loop wait in ttydim.
xxxxxxxxxxxx	Directory unique ID.

The last two items of general interest in an APTE are the Descriptor Base Register (DBR) value in apte.dbr and the clock reading in apte.state_change_time. This is the clock reading taken at the last time a process changed its execution state (see the Multiprogramming PLM). Hence, it may be an indicator of trouble if it has been a long time (current time minus apte.state_change_time) since a currently waiting or blocked process ran.

FAULT VECTOR

While the fault vector is not a data base of general interest, one would look at the fault vector if a trouble fault occurred since a typical reason for a trouble fault is a bad instruction pair in a fault or interrupt vector. The fault vector actually consists of interrupt and fault vectors. It begins at absolute location zero. There are 32-double word interrupt vectors followed immediately by 32-double word fault vectors. Each vector consists of an SCU instruction indirect through an ITS pointer and a TRA instruction indirect through an ITS pointer. The ITS pointers come directly after the fault vectors. They are ordered in the following way: ITS pointers for TRA in interrupt vectors, ITS pointers for SCU in interrupt vectors, ITS pointers for TRA in fault vectors, and ITS pointers for SCU in fault vectors.

KNOWN SEGMENT TABLE

The Known Segment Table (KST) is described in detail in the Storage System PLM and so is not described here. About the only issue one would have in the KST when analyzing a crash is finding out the names associated with a segment number when that segment has been deactivated. In the case of active segments, the BOS DUMP program and the online_dump program both print the names associated with each SDW in the descriptor segment. To find out the names associated with a given nonhardcore segment number (hardcore segment names are not in the KST and hardcore segments are never deactivated anyway), one can use the following algorithm to find the address of the Known Segment Table Entry (KSTE) for the segment:

1. Assume y is the segment number in question.
2. Let x = y-kst.hcscnt-1 (kst.incl.pl1)
3. Let i = x/kst.acount
4. Let j = mod(x,kst.acount)
5. Let kstarrayaddress = kstarrayaddress+j*4

Step 2 above subtracts the highest hardcore segment number from the original segment number since hardcore segments are not represented in the KST. Step 3 divides the result of step 2 by the size of a KST array (currently = 200 octal) to find out which KST array holds the KSTE. Step 4 finds the number of the KSTE

within the KST array. The KST array address is found by indexing into kst.kstap in step 5 with the result of step 3. Finally, the KSTE is found by adding the KSTE number times the size of a KSTE (4) to the address of the KST array in step 6. Given a KSTE address now, use the name address in the KSTE to find the list of names associated with the segment number. Figure 5-8 below gives the format of KSTE (see kste.incl.pl1) while Figure 5-9 gives the format of a name (see kst_util.incl.pl1).

One should be aware that the reference names resulting from this algorithm are only a heuristic help in identifying the segment. The branch pointer in the KSTE, identifying a directory entry, can be of help too.

LINKAGE SECTION

Quite often when analyzing a dump, it is necessary to examine internal static or to find the name associated with an unsnapped link. In the user rings, all linkage information usually exists in one combined linkage segment. In general, at the base of the linkage segment is a Linkage Offset Table (LOT). (The stack header contains a pointer to the LOT.) To find the linkage section for a given segment number, that segment number is used as an index into the LOT, which is an array of packed pointers. The packed pointer, if nonzero, points to the base of the linkage section for that segment. Usually this linkage section is somewhere within a combined linkage segment. Once this address is known, internal static can be located by using the linkage section offset given to the internal static variable by the translator or binder. Within the hardcore, there are two combined linkage segments and a separate segment to hold the LOT. One combined linkage segment, wired_sup_linkage, holds the linkage segments for most wired segments while the other combined linkage segment, active_sup_linkage, holds the linkage segments of most paged segments. The exceptions to the wired and paged segments having their linkage sections in wired_sup_linkage or active_sup_linkage are due purely to reasons of antiquity. In any case, the LOT entries for these special cases point to the correct segment. (For example, the LOT entry for the fim points to a segment called fim.link.)

The following discussion describes the procedure involved in associating a segment name and entryname with an unsnapped link. This is expanded upon in the Binding, Linking, and Namespace Management PLM, Order No. AN81. The reader should refer to Figure 5-10 while reading this material. Assume you are presented with machine conditions indicating a fault tag 2 (linkage fault). The TSR contains the segment number of the linkage segment and the computed address holds the offset in the linkage segment of the unsnapped link. The fault/interrupt code in the SCU data is 61, octal. To find the names involves a 4-step process.

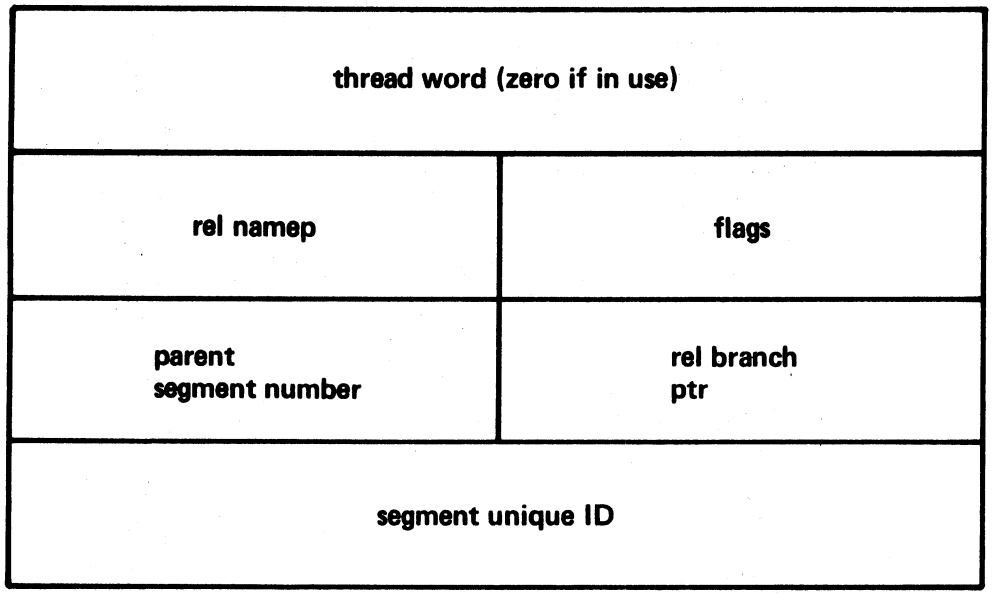


Figure 5-8. Format of KSTE

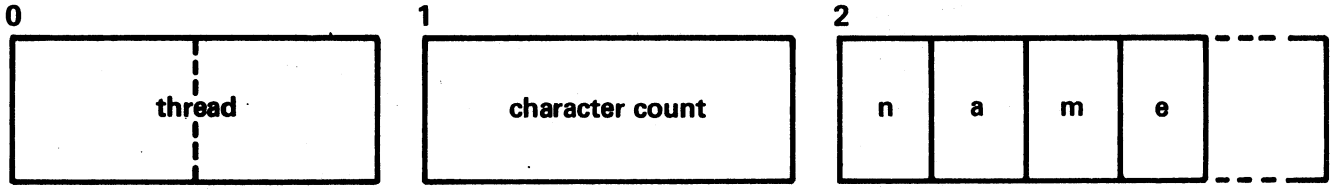


Figure 5-9. Format of name in KST

Step 1 is to find the linkage section header for the linkage section. This can be done in two ways. The most common way is to add the value labeled "header relp" in the link to the address of the link. This value is a negative number that is the negative of the link's offset from the linkage section header. The other way to find the linkage section header is to index into the LOT using the value of the PSR in the machine conditions. This is a packed pointer pointing to the linkage section header. The item of interest in the header is an ITS pointer to the definitions section of the object segment that took the linkage fault. This pointer occupies the first two words of the linkage section header.

Step 2 is to add the 18-bit "expression relp" in the link to the offset portion of the definitions pointer. This produces a pointer to an expression word.

Step 3 is to add the 18-bit "type pair relp" of the expression word to the offset portion of the definitions pointer. This produces a pointer to a double word type pair block.

The last step is to add the 18-bit "segname relp" to the offset portion of the definitions pointer to produce a pointer to an ACC segname string. Also add the 18-bit "offsetname relp" to the offset portion of the definitions pointer to produce a pointer to an ACC entryname string. ACC strings consist of a 9-bit length field followed by 9-bit ASCII characters.

As a final aid in understanding this, listed below is a PL/I program fragment that encodes the algorithm just described.

```
/* Assume linkp points to unsnapped link */
headerp = addrel (linkp, linkp->link.header_relp);
     /*point to link sect hdr*/
defp = headerp->header.def_pointer;
     /*copy definition section pointer*/
expp = addrel (defp, linkp->link.exp_word_relp);
     /*point to expression word*/
typrp = addrel (defp, expp->exp_word.type_pr_relp);
     /*point to type pair block*/
     /*point to ACC segname*/
entryp = addrel (defp, typrp->ty_pr.entryname_relp);
     /*point to ACC entryname*/
```

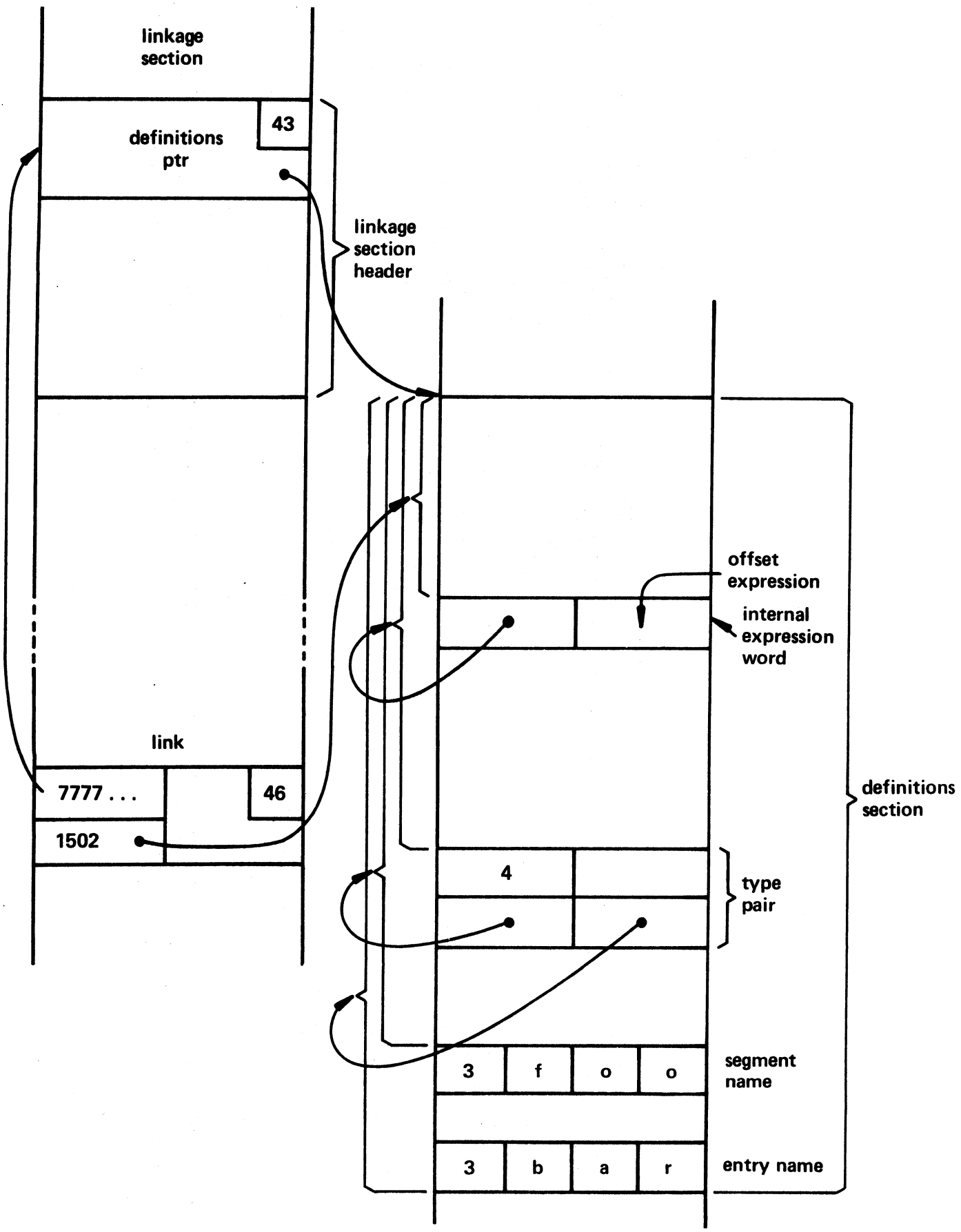


Figure 5-10. Association of Name with Link

LOCK_SEG

One data base to examine when analyzing crashes related to locking problems (deadly embrace, idle loop) is lock_seg. This segment is a wrap-around history queue of all attempts to lock wait type locks in ring 0. The segment consists of an index into the wrap-around queue and a 127-entry queue. The index is in the eighth word of the segment (the first seven are zero) and is the index of the oldest entry in the wrap-around queue. Indexing is from 0 - 126. Figure 5-11 shows the format of one eight-word entry in the array.

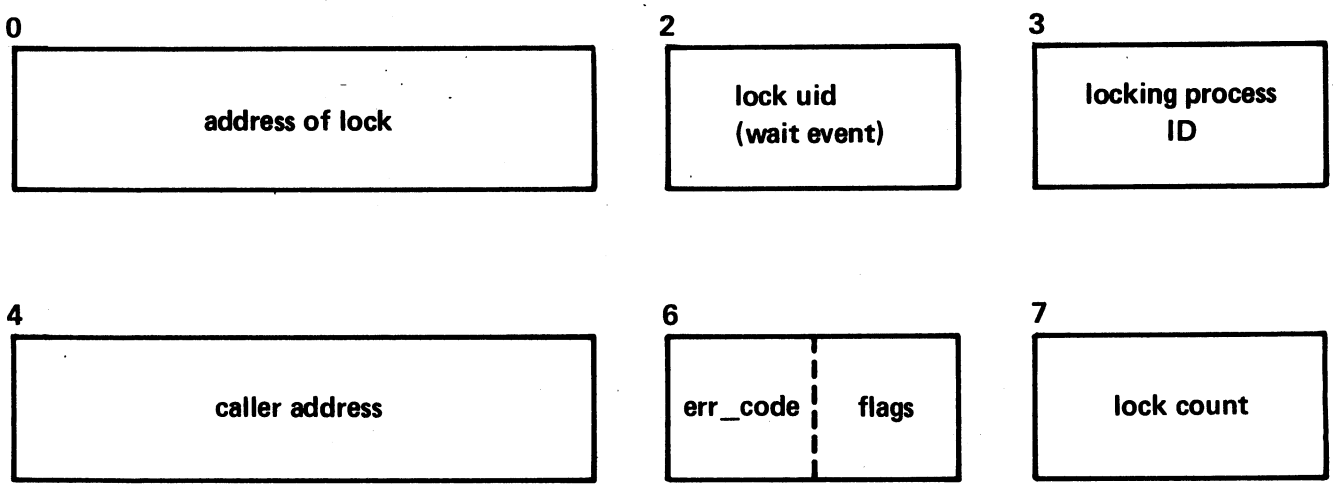


Figure 5-11. Format of Lock Entry in lock_seg

The wait event is one of those listed previously (a directory unique ID, "ioat", etc.). The error code is the rightmost 18 bits of any error_table_ code that lock returned to its caller. The call type occupies bits 21-26 of the word and is the type of entry into lock. (These can be determined by examining the listing of lock.pl1.) The fail switch occupies bit 35 of the word and is on if a lock try failed. The last word holds the value of sst.total_locks_set at the time the entry was made. The value in sst.total_locks_set is the total number of locks being waited on by all processes in the system. It is checked for zero when the system is shut down and a warning is printed on the operator's console if it is nonzero. This is done to indicate that perhaps a directory lock still remains locked, and so the salvager should be run.

PDS

Another important data base is the PDS. This per-process segment is used as a ring 0 stack and has a number of per-process items in the header that are useful in dump analysis. All of the data items are defined by segdefs and are referenced by name in this discussion. Of course, various sets of machine conditions are stored on the PDS. These have already been discussed. For most faults handled by the fim, the history registers are stored in pds$history_reg_data and the associative memories are stored in pds$am_data. This information should be examined if a fault occurs and it looks like a hardware error may be involved. (In some releases, these may be on the PRDS, however.)

The PDS holds the process ID in pds$processid. Given this, the APTE can be located as described previously (pds$apt_ptr also locates the APTE directly). The PDS also holds the process group ID (e.g., Jones.Project_id.a) in the variable pds$process_group_id so that the name of the user may be associated with the PDS being examined. Another useful variable is pds$lock_id. This is the unique 36-bit value used in locking outer ring locks. It is also kept in the APTE for the process so that if an outer ring lock is locked, a call can be made to ring 0 to look at all the APTEs to discover if the process associated with the lock ID still exists. If it does not exist, then that outer ring lock may be zeroed.

When investigating locking problems, pds$lock_array should be examined. This is a 20-entry array that contains information about each ring 0 wait-type lock currently locked by that process. Each eight-word lock array entry holds a pointer to the lock, the 36-bit wait event associated with the lock (e.g., unique ID for a directory), the fixed binary (35) type of data base being locked (directory = 1), a pointer to the caller of lock, a fixed bin (2) number that if 0 means the locked data base (a directory) is being read and if 1 means it is being written, and a fixed bin (2) modify switch that if 1 means the locked data base (a directory) was modified while locked. Figure 5-12 shows the format of a lock array entry.

The main use of the lock array besides debugging is to provide a record of any directories that were locked and modified by a process. If the process attempts to crawl out of ring 0, the program verify_lock can determine whether or not to call the online salvager. The online salvager uses the lock array to discover what directories to salvage. If no directories in the lock array have the modify switch on, then verify_lock does not call the online salvager.

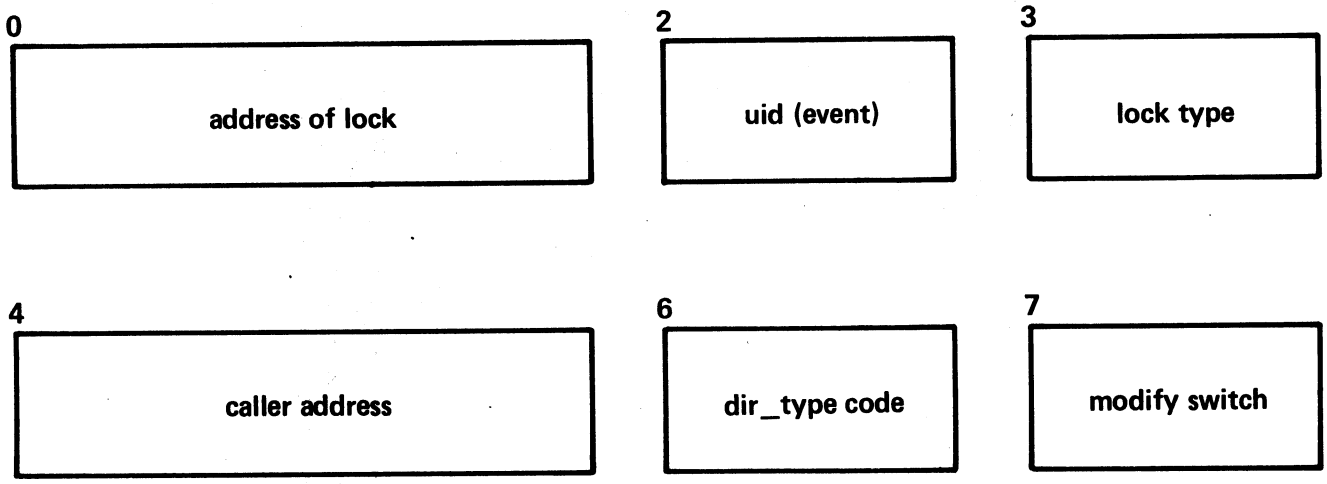


Figure 5-12. Format of Lock Array Entry

Also of general interest in performing dump analysis is the trace table (pds$trace) that is maintained by the system trace facility. The system trace facility was originally put in the system to both "trace" page faults in a process as well as to provide data to the post purging mechanisms of the system. Since the original implementation, the trace has been extended to include many events, other than page faults, that occur in a process. A complete list of traced events occurs below with the corresponding trace type code.

The actual trace data is stored in a wrap-around buffer in the PDS at the segdefed location pds$trace. The data is stored into the trace buffer by the primitive page$enter_data (part of the machine language kernel of page control for efficiency reasons) for nonpage fault entries and by the subroutine page$enter for page fault entries.

The buffer is managed by several indices enabling the system (and the user) to determine the beginning and end of a quantum. A bound of the trace buffer is kept in the header of the buffer enabling correct wrap-around handling. The actual buffer must be wired down (it is referenced at page fault time) and hence must be limited in size. In fact, the size of the buffer is a function of other variables in the PDS and the buffer is assembled so that it fills out the remainder of the first page of the PDS not needed by other wired variables in the PDS.

Data Structure

The following declarations describe the structure of the trace buffer.

```
declare 1 pds$trace aligned ext like trace_data;

dcl 1 trace_data aligned,
    2 next_usable fixed bin (16) unal,
    2 pad1 bit (19) unal,
    2 first_unusable fixed bin (16) unal,
    2 pad2 bit (19) unal,
    2 time fixed bin (71),
    2 first_used_in_quantum fixed bin (16) unal,
    2 pad3 bit (19) unal,
    2 pad4 (3) fixed bin (35),
    2 entry (0:1 refer (trace_data.first_unusable -1)),
      3 code_word,
        4 astep bit (18) unal,
        4 ring_number bit (3) unal,
        4 segment_number bit (15) unal,
      3 trace_word,
        4 type bit (6) unal,
        4 page_number bit (12) unal,
        4 time fixed bit (17) unal;
```

The following descriptions refer to the items declared above:

next_usable	is an index into the trace array (trace_data.entry) to the next entry that is used by page$enter or page$enter_data. This value may be zero but can never equal first_unusable.
first_unusable	is the number of entries in the trace array. Since the trace array is indexed from 0, first_unusable cannot be used as a valid index into the array. This item should be used when implementing a wrap-around technique for looking at the data.

time

is the system clock reading at the time the most recent trace data was entered. It is used to determine the (real) time difference between trace entries.

first_used_in_quantum

is an index to the first entry in the list for the quantum in which it occurs. The variable only gets set if post purging is being done and usually points to the first entry after a rescheduling entry.

astep

is used by the post purging (and process swapping) software to help locate the PTW associated with the page fault that caused this trace entry. This is only set if type = "000000"b, i.e., for page fault entries.

ring number

is the validation level at the time of reference that caused the traced page fault, i.e., scu.trr. This is only valid for page fault entries.

segment_number

is the segment number of the segment referenced at the time of the fault that caused the trace entry.

type

is the coded type of the trace entries. The following types are currently defined:

Type	Binary Value	Decimal Value
page fault	000000	0
seg fault start	000010	2
seg fault end	000011	3
link fault start	000100	4
link fault end	000101	5
bound fault start	000110	6
bound fault end	000111	7
*signaller	001000	8
restart fault	001001	9
reschedule	001010	10
*user marker	001011	11

page_number

is the page number of the page faulted upon. This value is only filled in for page fault entries.

time — is the real (wall-clock) time that elapsed between this and the previous entry. The time is in units of 64 microseconds. If the time difference is too large (about 2**24 microseconds or 15 seconds) a value of 0 is placed in the entry.

For the starred trace types above, the code word should be declared as follows:

```
3 code_word char (4) aligned,
```

The trace data can be looked at (and modified in a limited way) from the user ring. To read the data the following code should be used:

```
declare hcs_$get_page_trace entry (ptr);

call hcs_$get_page_trace (addr (trace_data));
```

trace_data is declared as above. (Output)

The primitive returns the entire trace structure and the caller must provide enough storage. Although the caller cannot know initially how much storage is required, he can depend on it being less than 1024 words. (On subsequent calls the size of the structure is known.)

An interface exists to allow the user to place an entry in the trace structure as a marker to delineate events. This primitive is used as follows:

```
declare hcs_$trace_marker entry (char (4) aligned);

call hcs_$trace_marker (message);
```

where message is a user-specified character string that is placed in the trace structure.

Trace Entry Types

The following is a list of defined entry types and a description of the events they represent.

page fault — means that a page fault occurred on the indicated page of the indicated segment. This entry is filled in for all page faults that occur in the process.

seg fault start — indicates the start of handling a segment fault. The code word of the

	trace entry is filled in with the segment number of the segment faulted on.
seg fault end	indicates the end of handling a segment fault. The code word is filled in with the segment number of the segment faulted on.
link fault start	indicates the start of handling a linkage fault on the start of processing or an hcs_$make_ptr call. For linkage faults the code word contains the segment number of the procedure causing the fault. For make_ptr entries the code word is set to 0.
link fault end	indicates the end of handling a linkage fault or the end of processing an hcs_$make_ptr call. For linkage faults the code word is filled in with a packed pointer value equal to the snapped link. For make_ptr calls the code word is set to 0.
bound fault start	indicates the start of processing of a bounds fault. The code word is filled in with the segment number of the segment faulted on.
bounds fault end	indicates the end of processing for a bounds fault. The code word is filled in with the segment number of the segment faulted on.
signaller	indicates the occurrence of an event signalled by the supervisor. The first four characters of the condition name of the signalled event are placed in the code word.
restart fault	indicates an attempt to restart or continue a signalled event.
reschedule	indicates that the process was rescheduled. It indicates the time the process's quantum expired.

user marker	is one generated by the user with the hcs_$trace_marker primitive. The code word is specified (and interpreted) by the user but is generally a character string four characters long.

SECTION VI

TYPES OF CRASHES

This section describes some heuristics one might use when faced with various types of crashes. It is impossible to provide a complete list since much of dump reading involves an intuitive process that is not easily defined.

LOOPS

The type of crash termed as a loop generally covers two cases. The first is an "idle loop" where all processors are just idling even though there is work to be done. One can recognize when a processor is running an idle process because an idle process displays a pattern that alternates in the A and Q registers. The pattern is an octal 777777000000 in the A register and an octal 000000777777 in the Q. In later releases, a "rotating chain" pattern may be observed. This pattern is flipped periodically (whenever an interrupt occurs) so that if the processor display is set up to show the AQ, the idle loop situation is easily verified. The other type of loop is a loop within ring 0. This, of course, ties up the processor so that no useful work can be done. In both cases, an execute fault is the usual way to crash the system so that a dump can be taken.

In the case of a ring 0 loop, little can be said here to allow one to discover the cause of the loop. What must be done is to see what program was running at the time of the execute fault, see what the value of the stack pointer was (pointer register 6 in the execute fault machine conditions), and using this information examine the stack history and see what the program was doing when it was stopped.

In the case of an idle loop, the execute fault data is probably of little use. What one should look at first is the APT to discover why no process is running. Of interest are the wait events of those processes in wait state. If the wait events are unique IDs (uid) of directories, one must find the directory associated with the unique ID. There are three ways to find this out. The first is to scan the lock array in the PDS of one of the processes waiting on that event. If the uid is found, then

that lock array entry contains a pointer to the lock and given the pointer, one can of course find the lock to get the process ID of the process that did the locking. The second way is to scan lock_seg looking for an entry with a wait event equal to the uid. If such an entry is found, the lock pointer in the entry can be used to examine the lock. The final method is to search the KST. There is a hashing algorithm to produce the address of the KSTE for a segment given its uid. However, the algorithm is difficult to do by hand and so the best alternative is to scan the KST looking for a KSTE containing the uid (last word of the four word KSTE). There is sufficient information in the KSTE to find the lock and to extract the process ID of the process that locked that lock. Once the process ID has been learned, the APTE for that process as well as the PDS can be examined. Hopefully, the PDS contains enough history in its stack to allow one to determine the reason why that process did not unlock the lock. In a crash of this sort, a normal FDUMP that dumps only the running processes is not sufficient since the running processes are idle processes. Hence, one should use the SHORT option to dump the descriptor segment, PDS, and KST of <u>all</u> processes.

PAGE CONTROL CRASHES

In general, when there is a page control problem a syserr message is printed although there are a few cases where page control loops (TRA *) when an error condition is encountered. This causes a lockup fault that results in a crash with no message. A common syserr message is "Fatal error in page_fault at location n" where n is the octal location within bound_page_control where some error was noticed. A listing provides further information as to the cause of the crash.

There are several conventions one should be aware of when analyzing page control problems. The first is the coding conventions used in that portion of page control that is written in ALM. Subroutines are not called using the normal call/push/return sequences due to the overhead involved. Instead, all "calls" are done via a TSX7 instruction and all subroutines within the ALM portion of page control share the same stack frame. In addition, page$done calls pxss to perform a NOTIFY when a page has been read in and so pxss also shares this stack frame. The stack frame is defined by pxss_page_stack.incl.alm. There is a small save stack for use by page control and one for use by pxss. These stacks are used to push and pop values of x7. The stack variable "stackp" is a tally word that points to the next place to store a value of x7 on the page control x7 save stack, save_stack. Hence, when reading a dump, the value in stackp is the upper bound on what x7 values are valid in save_stack. The value in the word of save_stack just before the word pointed to by stackp is the address of the last subroutine that was "called" via a TSX7. The same conventions are true of the pxss x7 save stack, pxss_save_stack, and its stack pointer, pxss_stackp.

Also of interest in the stack frame are save areas for four sets of index registers. The index registers saved in notify_regs are stored when the done_ subroutine of page_fault calls pxss to do a NOTIFY when a page read is complete. Index registers are also saved in notify_regs when meter_disk is called even if no disk metering is going on.) The registers stored in bulk_reg are stored there when bulk_store_control calls page$done after a bulk store read (or write in some cases) has completed. The registers stored in page_reg_bs are stored when bulk_store_control is entered. The final items of general interest in the stack frame are the variables did (the device ID), devadd (the device address), and ptp_astep, which holds the address of the PTW for a page being read (upper half) and the address of the ASTE for the page being read (lower half). This cell is also used by the RWS code.

Certain conventions have been established for the use of index registers by page control. The following register assignments are used when running in all parts of page control written in ALM except for bulk_store_control:

x0	temporary (may be used at any time)
x1	pointer to PDME and temporary
x2, pr2	pointer to PTW
x3	pointer to ASTE
x4	pointer to CME, also used in PDME hashing code.
x5	temporary
x6	temporary
x7	used for subroutine calls
pr3	pointer to the base of the SST segment

The register conventions for bulk_store_control are not of general interest and are documented in the listing.

ATTEMPT TO TERMINATE INITIALIZER PROCESS

Whenever a process takes a fatal error (e.g., runs off its stack), it is terminated by the signaller. The signaller accomplishes this by referencing through a pointer with a segment number of -2 and a word offset that identifies the reason for the termination. A word offset of -4 means that the signaller or restart_fault faulted while processing a fault. A word offset of -8 means the user's stack is in an inconsistent state. Other values can be found in the listing of terminate_proc, which is the program called by the fim when an attempt is made to reference through a pointer with a segment number of -2. If terminate_proc is called to terminate the initializer process, it crashes the system. If this happens, examine the fault data on the PDS of the initializer. The computed address in the fault data is the word offset of the pointer with the -2 segment number. This is an indication of why the initializer was terminated. The other way to discover this is to find the stack frame of terminate_proc. It takes one argument that is either a

standard error_table_ error code or the negative word offset from the fault caused by the attempt to reference through the pointer described above. Quite often, the initializer is terminated due to an overflow of its stack caused by recursive faults. In this case, the first set of fault conditions on the ring 4 initializer stack is for the original fault that caused the problems. These fault conditions can be located by tracing the stack in a forward direction (starting at the stack location pointed to by the stack begin pointer in the stack header), looking for a frame with a return pointer for the program return_to_ring_0_.

It is unfortunate that the data from the fault that resulted from the use of the pointer to segment -2 overwrites pds$fim_data. Thus, the machine conditions for the original fault cannot be found there. With luck, a heuristic search of the PDS for old stack frames owned by fim or return_to_ring_0_, or data that appears to be fault data may be of use. The 30000 stack words skipped by syserr when crashing help facilitate this search. It may also prove useful to inspect pds$signal_data as well.

Once the original cause for the termination of the initializer process has been determined, there remains the task of discovering why the fault occurred in the first place. Since it would be impossible to list all the possible causes, the initializer's data bases are described so that they may be checked for consistency in a dump. It is assumed that the reader has previously read the material in the System Administration PLM, Order No. AN72.

TELETYPE DIM PROBLEMS

Teletype DIM (ttydim) problems usually fall in one of two classes. Either the FNP crashes or the ttydim within the Multics Processor notices a problem and crashes. In the former case, a syserr message of the form "Emergency Interrupt from 355 A" or "dn355: mailbox timeout, please dump the 355." indicates that the FNP has crashed. In the case of the ttydim, the most common errors are of the form "tty_free error n" or "tty_inter error n" where n is an error number. It is not the intent of this material to describe the internal structure of the FNP software or the ttydim software since it is described in the Supervisor Input/Output PLM, Order No. AN65. What is given here are a few hints to offer some direction to the crash analyzer.

When an FNP fdump is printed (using od_355), the dump is broken up into three sections. The first section gives the cause for the crash and the registers at the time of the crash. Registers are listed as IC (instruction counter), IR (indicator register), A (A register), Q (Q register), X1 (index register 1), X2 (index register 2), X3 (index register 3), ER (interrupt enable register) and ET (elapsed timer register). Possible crash causes are:

power off	
power on	
mem parity	memory parity error
ill opcode	illegal (invalid) operation code
overflow	
ill mem op	illegal (invalid) memory operation
divide chk	
ill prg in	illegal (invalid) program interrupt
unexp int	unexpected interrupt
iom ch flt	iom channel fault
console	

Many of these faults are self-explanatory. The fault "ill mem op" refers to the fact that one of the following conditions has occurred:

1. The memory controller on the FNP timed out (hardware error).

2. There was an invalid command to the memory controller (hardware error).

3. Out of bounds address.

4. Attempt to alter storage in a protected region (protection not used by Multics currently).

5. A character address of seven.

The fault "ill prg in" refers to a hardware error in which the processor attempts to answer an interrupt when there was no interrupt present or a valid interrupt occurred but the interrupt sublevel word for that interrupt was all zeros. The error "unexp int" means that an interrupt from an unconfigured device occurred. The message "iom ch flt" indicates that an iom channel fault occurred. The fault word can be found in locations 420-437 of the FNP. The location selected is 420 (8) + iom channel number (0-17). Figure 6-1 below depicts the format of the fault status word. Finally, the message "console" indicates that an ABORT command was typed on the control console for the FNP (see the Supervisor I/O PLM).

The next section of the dump is the formatted contents of the internal trace table. This table contains the last fifty or so events printed in ascending chronological order. Each trace entry consists of a type and the value of the elapsed timer at the time the trace entry was made. The elapsed timer increments every millisecond. Listed below in Table 6-1 are the items printed for each trace type.

The final section consists of the FNP dump itself. The dump is formatted eight words per line. Preceding the octal dump of the eight words is the absolute location being dumped, the module (if any) being dumped, and the relative location being dumped within that module. The Supervisor I/O PLM describes the internal logic of the software within the FNP.

As far as ttydim problems within the Multics Processor go, the two problems most commonly found are syserr crashes of the form "tty_free error n" or "tty inter error n". Tables 6-3 and 6-4 list the meanings of these errors. A brief statement should be made at this point about the layout of the buffer pool. The unpaged segment tty_buf is used to hold several data bases necessary to the ttydim. Figure 6-4 depicts the format of tty_buf. The include file tty.incl.pl1 describes, among other things, the header area of tty_buf. The bleft variable contains the number of free buffers remaining in the buffer pool and the free variable contains the address of the first free buffer. All free buffers are threaded together in a forward direction only by an address in the first 18 bits of each buffer. This address is relative to the base of tty_buf. All free buffers are marked by a 36-bit pattern of alternating binary 1's and 0's in the last word of the 16-word buffer. When a buffer is allocated, this 36-bit pattern is changed to alternating binary 0's and 1's. The program tty_free makes certain checks whenever a buffer is allocated or freed. Table 6-3 lists the possible errors that can be found.

The other common form of ttydim crashes is a message of the form "tty inter error n". Table 6-4 lists all the error codes from the program tty_inter.

Data Commands	Interrupt Commands
000 None	000 None
001 Load	001 Unconditional
010 Store	010 Conditional or TRO
011 Add	011 Conditional or PTRO
100 Subtract	100 Conditional or Data Neg.
101 Add	101 Conditional or Zero
110 Or	110 Conditional or Overflow
111 Fault	111 Fault

Fault Types

0000	None
0100	All other Memory Illegal actions
1000	Memory Parity
1100	Illegal command to IOM
1101	IOM bus channel parity error
1110	IOM adder parity error
1111	IOM priority break

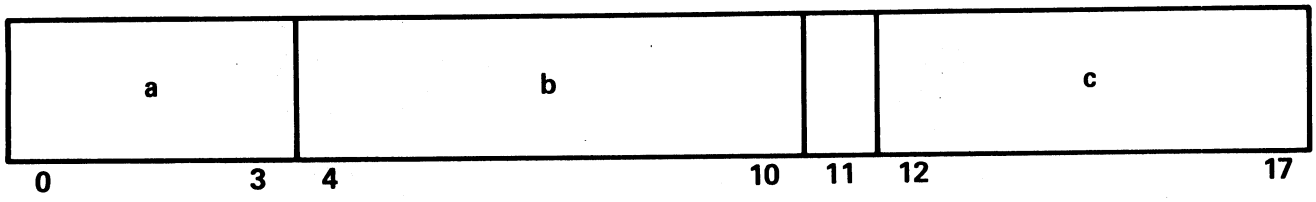
Figure 6-1. Format of 6600 FNP IOM Fault Word

Table 6-1. Trace Types

Trace Type	Data
1 (I/O Interrupt)	Instruction counter at time of interrupt and a coded word indicating what device interrupted (see Figure 6-2).
2 (Idle)	Interrupt enable mask at time processor went idle.
3 (HSLA Activity)	See Table 6-2.
4 (DIA Activity)	Subtype and varying data. For subtype 1, the data is the transaction control word (see Figure 6-3). For subtype 2, the data is either a Multics Processor address and line number, or an error code and transaction control word. This is seen only when the FNP is crashing due to a bad opcode in a mailbox.
7 (LSLA Activity)	The data consists of a subtype (1 only), a Terminal Information Block (TIB) address, the value of the flag word in the TIB, the value of the line status word in the TIB, the value of the LSLA status character which occasioned the trace entry, and the value of the flag word in the RMX table entry for the terminal.
10 (System Crash)	The data consists of a coded word, the rightmost 15 bits of which are the instruction counter at the time of crash, and the value of the indicator register at the time of the crash.

Table 6-2. HSLA Trace Subtypes

Subtype (8)	HSLA Subroutine	Data
1	nsintp	Software Com Region (SFCM) address and coded interrupt word (see Figure 6-2).
2	hscnct	SFCM address.
4	snpcw1	SFCM, HSLA PCW.
5	sttprc	SFCM, HSLA status word.
6	ccwint	SFCM, CCW, and right half of HSLA PCW.
7	sttchk	SFCM and value of status word in TIB.
10	dcwprc	SFCM and value of status word in TIB.
12	diadun	SFCM and DIA action (see Figure 6-3).
13	hsstop	SFCM
14	spchn	SFCM
15	conect	SFCM and CCW.
16	conect (delayed stop)	SFCM
21	dcwprc (terminate)	SFCM and value of status word in TIB.
22	ckio	SFCM and flag word of SFCM.
100	dcwmic	SFCM, word 2 of LPW, and DCW.
101	dcwccd	SFCM, word 2 of LPW, and DCW.
103	dcwxfr	SFCM, word 2 of LPW, and DCW.
104	dcwcom	SFCM, word 2 of LPW, and DCW.
105	dcwlit	SFCM, word 2 of LPW, and DCW.



where:

a is 4 bits representing the iom channel number.

b is a 7-bit device number, coded for specific devices, i.e., hslas and lslas.

c is a 6-bit module number, indicating which module should handle this interrupt.

Figure 6-2. Format of Coded Interrupt Word

DIA Action Flags

Bit Number	Meaning
0	Performed list service.
1	Read data.
2	Wrote data.
3	Sent special.
4	Read mailbox.
5	Processed a get status command.
6	Processed a connect.

Format of Line Number

Bits	Meaning
8	0=LSLA line, 1=HSLA line
9-11	LSLA number (0-5) or HSLA number (0-2)
12-17	HSLA subchannel (0-31) or LSLA slot (0-52)

Figure 6-3. Format of Transaction Control Word for DIA

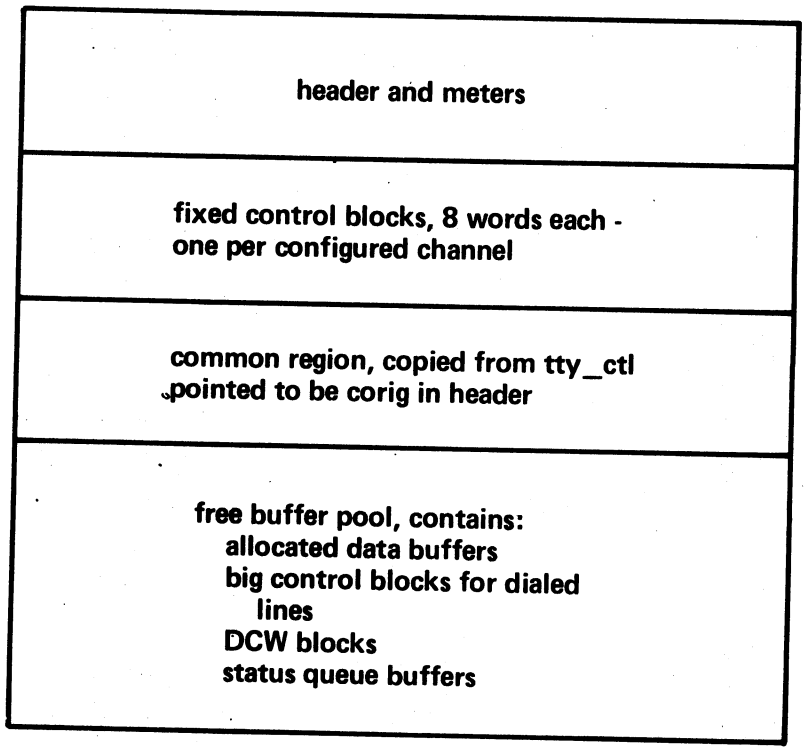


Figure 6-4. Format of tty_buf

Table 6-3. Errors from tty_free

Error Number (8)	Cause
1	Out of buffers.
2	Free buffer does not have the free pattern in it.
3	Thread in a buffer is not 0 mod 16 (buffer size).
4	Buffer being freed is not 0 mod 16.
5	Buffer being freed has the free pattern in it (already free).
6	The thread in one buffer of a chain being freed is not 0 mod 16.
7	A buffer in a chain being freed has the free pattern in it (already free).
10	The thread in a free buffer is not 0 mod 16.
11	The address of a buffer being freed is not in the buffer pool area in tty_buf.
12	The address of the first buffer in a chain being freed is not in the buffer pool area in tty_buf.
13	The address in the transfer dcw of one buffer in a chain being freed is 0.

Table 6-4. Errors from tty_inter

Error Number	Cause
1	Global lock (tty_buf.slock) not locked prior to causing time out.
2	Global lock not locked prior to clearing it.
3	Per tty lock (fctl.lock) not locked prior to clearing it.
4	FNP still processing read dcws when request made to free the read buffer chain.
5	FNP still processing write dcws when request made to free the write buffer chain.
6	Transfer dcw in read buffer does not point correctly to another read buffer.
9	There is an extra read block in the read chain.
10	Attempt to put more than eight dcws in a dcw block.
11	Global lock prior to clearing it in order to process a status word.
12	Apparent bad fixed control block (fctl).

HARDWARE PROBLEMS

This section cannot hope to describe all possible hardware problems but is instead offered as a first step to be taken by the person performing system problem analysis prior to consulting with engineering support personnel.

Bulk Store Problems

Bulk store problems usually can be designated as hardware status errors, data errors, or lack of response to a connect. In the last case, the message "page: bulk store timeout" is printed and the operation is retried. If it fails three times, a fatal read or write error is reported. In the case of a hardware status error, a message is printed of the form:

```
"bulk store err, status = x x x"
"csb status = y, addr = n"
```

The first line of the message prints the bulk store status, next address of data transfer, tally residue, and hardware indicators (the first three words of the status block). The second line of the message prints the status bits in the Current Status Block (CSB), followed by the bulk store address that was accessed. In Release 2.2 and later systems, only one word of DCB status is given.

Another type of bulk store error is a checksum error. If the proper option is specified on a DEBG configuration card, the bulk store software performs checksum calcuation and checking for every bulk store operation. If an error is detected, a message of the form "bulk store cksm err,addr = x, core = y" is printed where x is the bulk store address and y is the 24-bit absolute memory address.

The final type of bulk store error to be described is the nonfatal Error Detection and Correction (EDAC) error. This is a 1-bit error detected and corrected by the bulk store. All such errors are counted by the bulk store software and are reported by the metering program file_system_meters that is described in the Tools PLM. Further information is kept by the bulk store software on which Core Storage Module (CSM) is getting the EDAC errors. Currently, this information is attainable only by dumping the words (one per CSM) starting at the symbol mbx.edac_buckets as defined in the include file bulk_store_mailbox.incl.alm. There is one word (EDAC corrected error counter) for each 256k CSM of bulk store.

IOM Problems

No attempt is made to try to list possible IOM problems. Instead, all that is presented is the format of the various control words used by the IOM and the format of an IOM status word and IOM system fault word. More information may be found in the Supervisor I/O PLM.

Disk Problems

Disk errors are reported by a syserr message in the following format:

```
"dnnn error:     ch=c, cmd=cm, stat=s"
"                area=a, sect=sc, cyl=cy, hd=h, addr=ad"
"dnnn_control:   detailed status=xxxxxxxxx"
```

If the major status is "device attention," the message "dnnn_control:device attention -- please check disk unit" is printed following the area, sector, etc., information. In the rest of the message, c is IOM channel number, cm is IOM command, s is IOM status, a is area or logical drive number, sc is sector, cy is cylinder, h is head, and ad is the Multics address. In each message, dnnn is actually d190, d191, etc. depending on which type of disk got the error. The last line of the message prints nine bits of detailed status. Information about these nine bits may be found in the documentation for each type of disk subsystem. Figure 5-1 (DSS181 Extended Status), Figure 5-2 (DSS190A Extended Status), and Figure 5-3 (DSS190B Extended Status) in the Debuggers' Handbook PLM describe the meanings of the various major status and substatus bits to be found in a disk status word. More information about the disk software data bases, etc. may be found in the Supervisor I/O PLM.

Memory Parity Errors

Multics tries to recover from parity errors in almost all cases. If a parity error occurs while "running" on the PRDS however, (using the PRDS as a stack) Multics crashes. Parity errors are reported on the operator's console after first reading the locations of the instruction and operand to see which location (if any) had the parity error. The results are printed in a message as follows:

```
"parity fault in process-group-id."
"xxxx"
"xxxx"
"abs tsr loc:  n, contents; c"
"abs psr loc:  n, contents; c"
```

Where process-group-id is the Person_id.Project_id (e.g., Greenberg.Multics.a), the eight x's are the eight words of scu

data, n is the absolute 24-bit address of the instruction or the operand, and c is the contents of that 24-bit address. If no parity error occurred on the retry of the instruction fetch, the message "no parity error repeating psr memory access" is printed. If no parity error occurred on the retry of the operand fetch, the message "no parity error repeating tsr memory access" is printed. If both messages are printed, this is a good indication that the parity error occurred on the processor/memory interface rather than in the actual memory. In addition, if the parity error occurred in the user ring, the pathname of the user ring program is printed along with the instruction that got the parity error, and the instruction after. This is done so that if the parity errors occurred on the instruction fetch, one can see which bits are bad by comparing the instructions printed from the user program and the instruction printed. Further information about scanning a memory for parity errors using the system controller maintenance panel may be found in the Multics Operator's Handbook, Order No. AM81.

SECTION VII

SYSTEM PERFORMANCE DEGRADATION

This section is intended to give some direction to the reader who is trying to discover why Multics is running, but is running with very poor response. As is the case with crashes, the possible causes for poor system performance are myriad but it is often possible to discover what is wrong (if not why) using one or more of the Multics metering tools. All of these tools are described either within this PLM, the System Metering PLM, Order No. AN52, or within the Tools PLM. A generally useful tool is total_time_meters (ttm). This metering command indicates how the processor(s) is being allocated in terms of what percentage of total processor time is being used for interrupt processing, the time for page fault processing, etc. Hence, if some device is generating excessive interrupts, this time will show up in the output produced by ttm. If ttm shows that an excessive paging percentage is probably what is causing the performance degradation, then the metering command page_multilevel_meters (pmlm) should be used to check to see what percentage of page faults is being satisfied by a page on the paging device. If this number is abnormally low, the paging device map is probably inconsistent. If the speed of the paging device or disks is suspect, the device_meters (dvm) command should be invoked. This command indicates if excessive device errors are occurring, if the paging device or disks are being overloaded, or if the paging device or disks are running abnormally slow.

If ttm shows that most of the system is being tied up in the process of interrupts, the interrupt_meters (intm) command should be used to see what IOM channel appears to be tying up the system. If ttm shows that the system is spending much of its time in an idle condition, this is probably an indication of a poorly tuned system. The print_tuning_parameters (ptp) command prints out all the generally setable tuning parameters. If max_eligible is too low, this can cause excessive idling (similarly, when it is too high, it can cause excessive thrashing).

If the working set factor is too large, the system idles and again the converse is true. If the double write switch is on, this means that all pages are being written to disk as well as the paging device which will, of course, slow down the system.

If the output of ttm is not conclusive, one can try running the system_performance_graph (spg) tool to get a graph of system performance in an attempt to pick up patterns over a period of time. If this leads to the conclusion that one user has managed to "steal" much of the system, try running the traffic_control_queve (tcq) command several times. This shows up any user who is getting an inordinately high percentage of the available processor time. The command print_apt_entry (pae) (see Section VIII) may be used to print the APT entry of that user for further examination.

If much of the system time seems to be spent in overhead activities, the file_system_meters (fsm) command indicates if that overhead is due to the thrashing caused by too many pages being wired or incorrect allocation of AST entries (as indicated by the AST grace time). If this proves fruitless, run the meter_gate (mg) command to see if some ring 0 gate entry appears to be using a vast portion of the processor time. If there are still no indicators, perhaps a processor itself is running incorrectly. The set_processor_required (sprq) command can be used to force execution on only a particular processor. Then, execution of the instr_speed command shows if that processor is running below normal performance levels. It should be noted parenthetically that the EIS tester program, et, and the test_cpu program can be used to discover if a processor is in fact working correctly if not slowly. (See Hardware Diagnostic Aids, Order No. AR97, for the use of these programs.)

There are several other tools available to the investigator of system problems. If the paging device map is suspected of being in an inconsistent state, it can be copied out of ring 0 using the copy_out (cpo) command (see Section VIII) and then the dump_pdmap command can be used to confirm or deny these suspicions. Another command, check_sst, can be used to perform consistency checks on the core map, and the various AST pools. The command ring_zero_dump (rzd) (see Section VIII) can be used to dump various data bases in octal format for quick examination. If a patch to a ring 0 database will restore the system to proper operation, the patch_ring_zero (prz) command (see Section VIII) can be used as long as the user process has access to the hphcs_ gate.

One last note should be made here about another type of system problem. When Multics crashes and the Salvager is run, some key system directories may be partially destroyed so that it is impossible to bring Multics up again. If the system can be brought up to command level in the initializer process, the command comp_dir_info (described in the Tools PLM) can be used to see what is missing from certain critical directories. This, of

course, presumes that the command save_dir_info (described in the Tools PLM) is run regularly on these critical directories. If a directory has been changed, then the command rebuild_dir (described in the Tools PLM) may be used to reconstruct the directory, preventing a large amount of system down time for a restore or a reload.

SECTION VIII

COMMAND AND SUBROUTINE DESCRIPTIONS

This section contains the command and subroutine descriptions needed to analyze dumps. Some of these commands and subroutines have been referenced in previous sections.

The command and subroutine descriptions are arranged alphabetically in two groups. Command descriptions precede subroutine descriptions.

Name: check_sst

The check_sst command performs a large number of consistency checks on page control data bases in a copy of the System Segment Table (SST). Such a copy may be obtained from an fdump (see the extract command) or ring 0 (see the copy_out command).

The Core Map, Paging Device Map, and Active Segment Table (AST) are scanned, and inconsistencies reported. In addition, some meters on page and segment usage gleaned from these scans are printed out.

Usage

check_sst path

where path is the pathname of the copy of the SST segment to be analyzed.

Notes

Copies of the SST copied out of ring 0 are likely to be inconsistent unless special care is taken to minimize page faults and other system paging activity while such a copy is made.

The check_sst command makes its own copy of the SST provided. In it, it sets pad fields in CMEs and PDMEs, and ptw.processed bits as a form of marking. The presence of these bits in printouts of these data items should be understood as originating in this manner.

Name: copy_dump

The copy_dump command is used to copy an fdump image taken by BOS out of the dump partition into segments in the Multics hierarchy. The main entry point copies dumps into segments in the directory >dumps.

Usage

copy_dump

There are no arguments

Entry: copy_dump$set_fdump_num, copy_dump$sfdn

This entry point is used to set the error report form (ERF) number for the next fdump to be taken.

Usage

copy_dump$set_fdump_num erfno

where erfno is the ERF number for the next fdump to be taken.

Note

This command does not allow a particular dump to be copied twice. It also does not allow the ERF number to be set if the dump currently in the dump partition has not been copied.

Name: copy_out, cpo

The copy_out command copies a segment from the supervisor ring into a user-ring segment.

Usage

copy_out segname -path-

where:

1. segname is the SLT name or the octal number of the segment to be copied.

2. newname is the pathname of the copy created from segname.

Notes

If path is not specified the segment is copied into the working directory with the entryname segname. However, if an octal number is given, the correct SLT name, if one exists, for the segment is used.

If path already exists, it is truncated prior to the copy. The ring_zero_peek_ subroutine is used to copy the segment out.

copy_salvager_output

Name: copy_salvager_output

The copy_salvager_output command is used to copy the segment >online_salvager_output into a user-ring segment.

Usage

copy_salvager_output path

where path is the pathname of the user-ring segment into which the copy of >online_salvager_output is placed. The segment is created if it does not already exist. If the segment already exists, its previous contents are destroyed.

Notes

The privileged entry point phcs_$ring_0_peek is used to copy the data.

The number of words copied is calculated from the bitcount of >online_salvager_output. Upon successful completion of the command the same bit count is placed on the user-ring segment.

dump_pdmap

Name: dump_pdmap

The dump_pdmap command is used to check a copy of the System Segment Table (SST) segment for consistency. Its primary concern is with the paging device map but other checks are also made.

Usage

dump_pdmap path -control_arg-

where:

1. path — is the pathname of a copy of the SST to examine.
2. control_arg — can be one of the following:

   -long, -lg — prints out each Paging Device Map Entry (PDME) as it is scanned.

   -brief, -bf — prints only summary information (default).

Note

Most of the output is self explanatory. The user should try to get as consistent a copy of the SST as possible, since any inconsistencies found are reported. The copy_out command can be used to get a copy of the SST from a currently running system.

extract

Name: extract

The extract command is used to extract a segment from an fdump and leaves a copy of the segment in the working directory.

Usage

extract erfno segname

where:

1. erfno — is the error report form number of the fdump from which the segment is to be extracted.

2. segname — is the name or octal number of the segment to be extracted.

Note

Only the first process in a fdump is searched. The created segment has the name segname.erfno, where segname and erfno are the command arguments.

ol_dump

Name: ol_dump

The ol_dump command can be used to look at selected parts of an online dump created by the BOS FDUMP command and copied into the Multics hierarchy by the copy_fdump command. The command is designed to aid system programmers in the task of crash analysis. The command assumes all dumps of interest are found in the directory >dumps.

Usage

```
ol_dump -erfno-
```

where erfno is an optional error report form number given in decimal. If erfno is not specified, the ol_dump command enters its request loop described below. If an erfno is given, the ol_dump command searches the directory >dumps for a copy of the dump and if it finds the dump, it initializes itself to be able to process the given dump. If the dump is not found, the user is told and the request loop is entered.

Request Loop

Once the ol_dump command has processed the erfno argument it enters a loop reading requests from user_input. The requests allow the user to look at selected regions of the dump currently under analysis or to choose another dump (erfno) for analysis. The following requests are implemented (letters in parentheses are abbreviations):

Request	Function
erf no	selects another dump (the one whose erfno is no) for immediate analysis.
quit (q)	returns.
command (c)	passes the rest of the request line onto the current command processor.
list (l)	lists the dumps in >dumps by showing the name of the first component of the dump. The names of dumps tell when the dump was taken and what the erfno is.

ol_dump

help (?)	lists the requests of the ol_dump command.
dump (seg, d)	prints selected words of the specified segment. The format of the request is: dump name first count mode where name may be an SLT name or segment number, first and count are octal and mode is a one character output mode. The output modes are just those used by the debug command. If no mode is given octal is assumed. If no count is given, 1 is assumed.
dbr value	switches to another process (in the same dump) by specifying the dbr value for the new process.
art (pt) name	prints out the AST entry and page table for the given segment. Name may be an SLT name or a segment number.
name (n) segno	prints out the SLT name for the given segment number.
proc (p) No	prints out some APT data for the process specified. If proc -all is typed, all processes are dumped.
queue (tcq)	prints out the scheduler's priority queue in order of priority.
stack (s) name offset	traces the queue stack from the offset specified. If offset is not queue, the stack is traced from the base.

If the request line is none of the above, the entire line is passed directly to the current command processor.

online_dump

Name: online_dump

The online_dump command is used to create a printable dump from an fdump created by BOS. The fdump must have previously been copied into the hierarchy by the copy_dump command. The printable dump image is output through the ios_ subroutine. Optional control arguments may specify the segments that are to be dumped, that the online_dump is to be restarted, and the device and DIM to which output is attached.

Usage

online_dump erfno -control_args-

where:

1. erfno — is the error report form number that is used to access the segments of the dump image.

2. control_args — are optional and can be chosen from the following:

   -dim dimname — is the name of the ios_ DIM through which the stream od_output is to be directed. The default is prtdim, unless a different DIM has been specified earlier in the process.

   -dev devname — is the name of the device or ios_ stream to be attached to od_output. The default is prta, unless a different device has been specified earlier in the process.

   -restart procno segno — is the process number and segment number (both octal) at which the dump is to be restarted. The process number is the position of a dumped process relative to other dumped processes.

-segs indicates that input is read from the following lines that specify which segments are to be dumped by the online_dump command. Any number of segments may be specified on each line. When the word "quit" is reached, the online_dump command ceases reading input and begins the dump.

Notes

If the -restart control argument is present then the online_dump command skips over the machine registers and all segments of the dump until process number "procno" is read and a segment number greater than or equal to "segno" is found. From that point, the dump proceeds normally subject to the -segs control argument, if present.

If the -segs control argument is present, the segment identifications are interpreted in the following manner. If the seg_id is "regs" then the machine registers are dumped. Otherwise the seg_id is assumed to be the octal segment number of a segment to be dumped.

If the seg_id is not an octal number, then the seg_id is checked against the first name of each segment given in the Segment Loading Table (SLT) present in the dump.

If the seg_id is not found in the SLT then the user is warned that the segment cannot be found.

In addition, the user is warned if the SLT or certain other segments that are used to interpret the dump cannot be found.

Examples

If the printing of the entire dump image of erfno 45 was being done on a line printer and was interrupted at segment 100 of process 1 then the following command line would enable the rest of the dump to be printed:

```
online_dump 45 -restart 1 100
```

If the user wished to merely inspect the registers at his console:

```
online_dump 45 -dim syn -dev user_io -segs
regs
quit
```

The following exec_com causes the dumping of certain segments upon every execution, and the specification of others as needed:

```
& ec to extract useful info from a dump and dprint the dumped
& segments.
& command_line of
&attach
&input_line off
od &1 -dev dump_output.&1 -dim file -segs
regs dseg fault_vector iom_mailbox
 . . . .
kst_seg lock_seg str_seg
201 204 &2 &3 &3 quit
dp -dl -h "ERFNO &1" dump_output.&1
&quit
```

Entry: od_cleanup

This entry point may be called when printing of a dump is to be suspended so that the currently attached device may be detached.

Usage

```
od_cleanup
```

There are no arguments.

Entry: online_dump_355, od_355

This command is used to format a dump of a DATANET 6600 Front-End Network Processor (FNP) core image that has been created by the BOS FD355 command and copied into the Multics hierarchy.

Usage

online_dump_355 erfno -control_args-

where:

1. erfno — is the same as the online_dump command above.

2. control_args — can be chosen from the following:

   -dim dimname — is the same as the online_dump command above.

   -dev devname — is the same as the online_dump command above.

patch_ring_zero

Name: patch_ring_zero, prz

The patch_ring_zero command is used to change specified locations of ring 0. It requires access to hphcs_ by the user.

Usage

patch_ring_zero segment offset values

where:

1. segment is the octal segment number or segment name of a ring 0 segment.

2. offset is the relative offset (in octal) of the first of n consecutive words to be changed.

3. values are the values for the specified locations in ring 0.

Note

The call to the patch_ring_zero command first prints out the changes that are performed and then asks the user if the changes are correct. The user must respond with "yes" for the changes to be made. The user may patch read-only segments in ring 0 without explictly changing the access, as this is done by the command itself.

Example

```
patch_ring_zero sst 120  0  0

120  001761101001  to  000000000000

121  011376143210  to  000000000000

Type "yes" if patches are correct:  yes
```

print_apt_entry

Name: print_apt_entry, pae

The print_apt_entry command dumps, in octal, the contents of the specified user's Active Process Table (APT) entry. The command searches the Answer Table to find the process ID of the specified user and extracts the APT entry specified by the process ID.

Usage

print_apt_entry user_name

where user_name is either the name of the user or the name of the teletype channel assigned to the user, e.g., ttyxxx (or caaxxx) where xxx specifies some channel number.

print_aste_ptp

Name: print_aste_ptp, pap

The print_aste_ptp command prints out the Active Segment Table Entry (ASTE) and page table of the specified segment. If any pages are in core, the device address is extracted from the appropriate Core Map Entry (CME) and printed out as well.

Usage

print_aste_ptp segment

where segment is either the pathname or the segment number of the segment whose ASTE is to be printed. If the argument is an octal number, it is taken to be the segment number of the segment to be printed. If the argument is a pathname, the specified segment is printed. If the segment cannot be found, the ring 0 segments are searched to see if the segment name given specifies a ring 0 segment.

print_dump_tape

Name: print_dump_tape, pdt

The print_dump_tape command is used to print dump tapes produced by BOS. These tapes are written as unblocked BCD records.

Usage

print_dump_tape -control_args-

where the control arguments are optional and can be the following:

tape_number	is the number of the dump tape to be printed. If this control argument is not specified, "*** dump tape ***" is used in the mount message.
printer	is the name of the printer to be used. This name must begin with "prt". If this control argument is not specified, "prtb34" is used.
-file pathname	is used to direct the output into a file instead of printing it online.
-page page_no	is used to start printing the tape at the specified page number.

Note

The following I/O streams are used:

dump_tape_in	input tape stream
dump_to_printer	output stream (usually to a printer but possibly to a file)

ring_zero_dump

Name: ring_zero_dump, rzd

The ring_zero_dump command prints the locations of the specified ring 0 or user-ring segment in full word octal format. This command does not require access to phcs_ for those segments accessible through the ring_zero_peek_ subroutine.

Usage

ring_zero_dump segname -control_args-

where:

1. segname — is either an octal segment number or the name of a ring 0 segment. To specify a segment name that consists entirely of octal digits the name must be preceded by the -name (-nm) control argument.

2. control_args — can be one of the following:

   -first — is the octal location of the first word to dump. If the first and count arguments are omitted, the entire segment is dumped starting with location zero.

   -count — is the octal number of words to dump. If count is omitted, count is set to one. When the count argument is supplied, the first argument must also be supplied.

Examples

```
ring_zero_dump sst 200 10

ring_zero_dump -nm 400 0 100

ring_zero_dump 0 212
```

Notes

If the specified segment is not found in ring 0, the expand_path_ subroutine (described in MPM Subroutine, Order No. AG93) is used for an additional search.

The -first control argument is verified to be a legitimate address.

ring_zero_dump

When the combination of the -first and -count control arguments specify an address beyond the last page of the segment, the segment is dumped only through the last page.

copy_dump_seg_

Name: copy_dump_seg_

The copy_dump_seg_ subroutine is called by the online_dump command to copy a segment from the dump image into a separate segment so that it can be randomly accessed at a later time.

Usage

```
declare copy_dump_seg_ entry (fixed bin, fixed bin, (0:9)
     ptr, (0:9) fixed bin, ptr, fixed bin);

call copy_dump_seg_ (segno, cur_proc_index, ptr_array,
     len_array, outptr, outlen);
```

where:

1.	segno	is the segment number that is looked for in the dump image. (Input)
2.	cur_proc_index	is the index in the array ptr_array of segment 0 of the process for which segno is to be found. (Input)
3.	ptr_array	is the array of pointers to successive segments of the dump image. (Input)
4.	len_array	is the array of current lengths of the image segments. (Input)
5.	outptr	is a pointer to the segment into which the copy is to be made. (Input)
6.	outlen	is the number of words copied. If the segment could not be found, the value is 0. (Output)

Note

The segment pointed to by outptr is truncated before the copy is made.

format_355_dump_line_

Name: format_355_dump_line_

The format_355_dump_line_ subroutine is an ALM procedure that is called by the online_355_dump_ subroutine to produce an octal representation of one or more FNP words. In addition, there is another entry point, format_355_dump_line_$line, which produces an octal represention of one or more FNP words as well as an octal representation of two fixed binary numbers that are absolute and relative location counters. These are printed on a dump line by the online_355_dump_ subroutine.

Entry: format_355_dump_line_

This entry point is called to convert one or more FNP words to their octal representation.

Usage

```
declare format_355_dump_line_ entry (ptr, fixed bin, ptr);

call format_355_dump_line_ (input, count, output);
```

where:

1. input — points to the first of the FNP words to be dumped. This pointer must be pointing to an 18-bit aligned item. (Input)

2. count — is the number of FNP words to convert to octal. (Input)

3. output — points to the area in which to place octal representation of 355 words. This pointer must be pointing to a 9-bit aligned item. (Input)

Entry: format_355_dump_line_$line

This entry point is called to convert one or more FNP words to their octal representation. In addition, it converts two fixed bin numbers to octal. These numbers are absolute and relative location counters.

format_355_dump_line_

Usage

```
declare format_355_dump_line_$line entry (ptr, fixed bin,
     ptr, ptr, fixed bin, ptr, fixed bin);

call format_355_dump_line_$line (input, count, output, absp,
     absloc, relp, relloc);
```

where:

1. input — same as above. (Input)
2. count — same as above. (Input)
3. output — same as above. (Input)
4. absp — is a pointer to the area in which to place octal representation of absolute location counter (argument 5). (Input)
5. absloc — is the absolute location currently being printed in dump. (Input)
6. relp — is a pointer to the area in which to place octal representation of relative location counter (argument 7). (Input)
7. relloc — is the relative location currently being printed in dump. (Input)

get_ast_name_

Name: get_ast_name_

The get_ast_name_ subroutine is called by the online_dump command to obtain the pathname of a segment from copies of the SST and SST name table segments.

The get_ast_name_ subroutine assumes that the SST name table supplied was validly filled (by either Multics or BOS). The get_ast_name_ subroutine tries to fit the full primary_name pathname of the specified segment in the supplied return string. If it cannot fit, components of the pathname are removed (recognizable as ">>" in the output string) towards the left-hand end of the pathname. The get_ast_name_ subroutine never truncates a pathname, and thus, the entryname always appears intact. If the get_ast_name_ subroutine cannot obtain the pathname, the message "CANNOT GET PATHNAME" is returned.

Usage

```
declare get_ast_name_ entry (ptr, ptr, ptr, char(*));

call get_ast_name_ (astep, sstp, sstnp, retstr);
```

where:

1. astep is a pointer to the Active Segment Table Entry (ASTE) of the segment whose pathname is desired. The ASTE must be in the segment pointed to by sstp. (Input)

2. sstp is a pointer to the copy of the SST segment to be used to determine the pathname. sstp must point to the base of a segment. (Input)

3. sstnp is a pointer to the copy of the SST name table segment to be used to determine the pathname. sstnp must point to the base of a segment. (Input)

4. retstr is the pathname of the segment whose ASTE is pointed to by astep. (Output)

get_dump_ptrs_

Name: get_dump_ptrs_

The get_dump_ptrs_ subroutine returns pointers to the component segments of a fdump, given the ASCII representation of the error report form number for the fdump.

Usage

```
declare get_dump_ptrs_ entry (char(*), (0:9) ptr, (0:9)
     fixed bin, fixed bin, char(32) aligned);

call get_dump_ptrs_ (erfno, ptr_array, len_array, nsegs,
     primary_name);
```

where:

1.	erfno	is the ASCII representation of the error report form number of the fdump. (Input)
2.	ptr_array	is filled in with pointers to the component segments of the fdump. (Output)
3.	len_array	is an array of the current lengths of the component segments of the fdump. (Output)
4.	nsegs	is the number of segments that make up the fdump, i.e., the number of pointers returned. If this number is 0, there was some trouble initiating the specified fdump segments. (Output)
5.	primary_name	is the entryname of the first segment of the fdump. (Output)

Note

The format of standard fdump names is as follows:

mmddyy.hhmm.i.erfno

where:

mmddyy	is the date of the fdump.
hhmm	is the time of the fdump.

get_dump_ptrs_

i	is an integer from 0 to 9 indicating which fdump segment it is.
erfno	is the error report form number of the fdump.

od_print_

Name: od_print_

The od_print_ subroutine provides a page and line formatting capability for the online_dump command.

Entry: od_print_

This entry point provides a general formatting capability for the online_dump command equivalent to that of the ioa_ subroutine (described in the MPM Subroutines, Order No. AG93).

Usage

```
declare od_print_ entry;

call od_print_ (nlines, fmt_string, arg1, ..., argn);
```

where:

1. nlines — is an integer value (fixed bin) denoting the number of lines to be generated during formatting. (Input)

2. fmt_string — is the control string (char(*)) used to produce the desired output. Formatting control characters are identical to those of the ioa_ subroutine. (Input)

3. argi — are the arguments required by the fmt_string argument. (Input)

Entry: od_print_$op_fmt_line

This entry point is called to format and print eight words in octal with their associated location field.

Usage

```
declare od_print_$op_fmt_line entry (fixed bin, fixed bin,
     (0:7) fixed bin);

call od_print_$op_fmt_line (abs_loc, loc, arr);
```

where:

1. abs_loc — is the absolute location the data occupied. (Input)
2. loc — is the integer value to be printed as the offset for the line. (Input)
3. arr — is the array of words to be printed. (Input)

Entry: od_print_$op_finish

This entry point is called when dumping is finished to transfer the last buffer of formatted characters to the I/O switch. (See "Notes" below.)

Usage

```
declare od_print_$op_finish entry;

call od_print_$op_finish;
```

There are no arguments.

Entry: od_print_$op_new_seg

This entry point is used to inform the od_print_ subroutine that a new segment is being printed. This is done so that a new page may be started with the new segment number included in the page header.

Usage

```
declare od_print_$op_new_seg entry (fixed bin);

call od_print_$op_new_seg (segno);
```

where segno is the number of the segment to be printed next. (Input)

Entry: od_print_$op_init

This entry point is called to initialize the od_print_ subroutine and provide certain constant information for the page header.

Usage

```
declare od_print_$op_init entry (fixed bin, fixed bin(71));

call od_print_$op_init (erfno, time);
```

where:

1. erfno — is the error report form number associated with the dump. (Input)
2. time — is the time at which the dump was created. (Input)

Entry: od_print_$op_new_page

This entry point may be called when the next line of output should appear on a new page.

Usage

```
declare od_print_$op_new_page entry;

call od_print_$op_new_page;
```

There are no arguments.

Notes

Formatted data is internally buffered so that the I/O switch is called less often.

In order to speed up the online dumper's operation, formatted data as passed to the I/O switch contains ASCII NUL characters, (000)8.

The size of the formatted string may not exceed 256 characters.

A new page header is printed before the currently requested line is printed, if the number of lines currently formatted on the page, plus the number of lines for the current request, exceeds the number of lines per page.

od_stack_

Name: od_stack_

The od_stack_ subroutine is used by the online_dump command to format and print stack segments. Its primary purpose is to break the stack into frames and to number them in sequence.

Usage

```
declare od_stack_ entry (ptr, fixed bin, ptr, ptr, ptr);

call od_stack_ (stkp, stklen, sltp, namp, sstp, sstnp);
```

where:

1. stkp — is a pointer to the stack segment. (Input)
2. stklen — is the length of the stack in works. (Input)
3. sltp — is a pointer to a Segment Loading Table (SLT) to be used to determine names of hardcore segments. (Input)
4. namp — is a pointer to a name_seg to be used to determine the names of hardcore segments. (Input)
5. sstp — is a pointer to an image of the SST segment from the fdump. With the pointer sstnp, it is used to determine the names of nonhardcore segments. (Input)
6. sstnp — is a pointer to an image of the SST name table segment in the fdump. (Input)

Notes

The frames are numbered with the lowest number being at the head of the stack. If in some frame stack_frame.next_sp is equal to stack_header.stack_end_ptr, then that frame is numbered zero. If that is not the case for any frame, then the stack header is given number zero. The stack trace continues beyond stack_end_ptr so long as the back pointers are good.

If any stack frames have been jumped over by syserr in its attempt to preserve the stack history, then these frames are also broken out and numbered XX.

Name: online_355_dump_

The online_355_dump_ subroutine is called by the od_355 command (see the online_dump command). It is passed a pointer to an FNP fdump. It processes the dump producing an octal memory dump, a print out of the FNP registers, and an interpretation of the FNP software trace table. This data is written using the ios_ subroutine on the stream od_output, which must be attached before calling the online_355_dump_ subroutine.

Usage

```
declare online_355_dump_ entry (ptr);

call online_355_dump_ (dumpp);
```

where dumpp points to a FNP fdump. (Input)

Note

The dump output begins with register values, trace table and memory contents. Memory is dumped eight words per line. Included on the line are the absolute location, module name, relative location in that module, and memory contents. Duplicate lines are not printed; instead an asterisk is put at the beginning of the next line.

Name: print_dump_seg_name_

The print_dump_seg_name_ subroutine is called to print the Segment Descriptor Word (SDW), pathname, and reference names for a segment. It is used by the online segment dumper.

Usage

```
declare print_dump_seg_name_ entry (fixed bin, fixed
     bin(71), ptr, ptr);

call print_dump_seg_name_ (segno, sdw, sstp, sstnp);
```

where:

1. segno is the segment number to be used. (Input)

2. sdw is the SDW that is to be printed. (Input)

3. sstp is a pointer to an image of the System Segment Table (SST) segment from the fdump. With the pointer sstnp, sstp is used to determine the names of nonhardcore segments. (Input)

4. sstnp is a pointer to an image of the SST name table segment in the fdump. (Input)

Note

If sstp or sstnp is null or if the SST name table pointed to by sstnp is not valid, only the SDW breakout is printed.

Entry: print_dump_seg_name_$hard

This entry point is called to print the SDW and name of a hardcore segment. The name printed is the first name given the segment in the Segment Loading Table (SLT) Name Table.

Usage

```
declare print_dump_seg_name_$hard entry (fixed bin, fixed
     bin(71), ptr, ptr);

call print_dump_seg_name_$hard (segno, sdw, sltp, namep);
```

where:

1. segno — is the segment number to be used. (Input)
2. sdw — is the SDW that is to be printed. (Input)
3. sltp — is a pointer to the SLT to be used to find the segment's name. (Input)
4. namep — is a pointer to the SLT names segment to be used to find the segment's name. (Input)

Note

If sltp or namep is null or segno is outside the limits found in the SLT, only the SDW breakout is printed.

Entry: print_dump_seg_name_$get_ptr

This entry point returns a pointer to an online copy of a segment to be examined by the online_dump command or one of its associated subroutines. It is useful because copies of nonwritable (i.e., procedure) segments are generally not dumped by BOS and are not present in the dump itself.

Usage

```
declare print_dump_seg_name_$get_ptr entry (fixed bin, ptr,
     ptr);

call print_dump_seg_name_$get_ptr (segno, sstp, sstnp,
     segptr);
```

where:

1. segno — is the segment number that was associated with some procedure segment in a dumped process. (Input)
2. sstp — is the same as for the print_dump_seg_name_ entry point. (Input)

print_dump_seg_name_

3. sstnp is the same as for the print_dump_seg_name_ entry point. (Input)

4. segptr is a pointer to a copy of the procedure to be examined. (Output)

The Other Computer Company:
Honeywell

HONEYWELL INFORMATION SYSTEMS

PREPRINT EDITION

PREPRINT EDITION