EY-5315E-P0-0001

VAX/VMS Troubleshooting

Ruth Goldenberg

Revision 0.C

CONTENTS

MESSAGES 1
STARTING POINTS 4
ACCESSING PROCESS CONTEXT WITH SDA 5
The PCB And JIB 6
The PHD 6
BUGCHECKS 11
CPU-SPECIFIC INTERRUPTS 16
CPU-SPECIFIC INTERRUPTS - VAX-11/780 AND VAX-11/785 17
SBI Silo Compare 17
CRD/RDS 17
SBI Alert 18
SBI Fault 19
Cpu Timeout 19
CRASHDUMP REQUIREMENTS 21
Crashdump File 21
Bugcheck Mechanism 22
SYSINIT Processing 24
EXCEPTIONS 25
Software Exceptions 25
Hardware Exceptions 25
Exception Dispatching 26
Some Common Exception Types 27
Access Violation Fault 28
Reserved Opcode Faults 30
Reserved Addressing Mode Fault 30
Reserved Operand Exception 31
FATALEXCPT BUGCHECK 33
FORCED CRASHES 43
HALTS - VAX-11/780 AND VAX-11/785 46
Likely Halt Indications 47
HALTED AT Xxxxxxxx 47
HALT INST EXECUTED 48
?CHM ERR 50
?CLOCK PHASE ERROR 51
?CPU DBLE-ERR HALT 51
?ILL I/E VEC 52
?INT-STK INVALID 53
?NO USR WCS 54
Pathological Halts 54
RESTAR.CMD Command Procedure 57
VAX-11/780 And VAX-11/785 Restart Mechanism . 57
Editing RESTAR.CMD 59
HANGS 65
System Hangs 65
Process Hangs 75
INVEXCEPTN BUGCHECK 84
KERNEL STACK LOCATIONS 87
KRNLSTAKNV BUGCHECK 89
LOCATING I/O REQUESTS 92
MACHINE CHECKS 97
MACHINE CHECKS - VAX-11/780 AND VAX-11/785 101
Read Data Substitute Error 102

Translation Buffer Parity Error 104
Cache Parity Error 105
Control Store Parity Error 106
Microcode Not Supposed To Be Here 107
Read Timeout Or Error Confirm Error 108
PGFIPLHI BUGCHECK 113
RELATED REFERENCE MATERIAL 116
Call Frame Layout 116
PSL Layout 116
RESOURCE WAITS 117
Mutex Wait 119
RWAST Resource Wait 120
RWMBX Resource Wait 123
RWNPG Resource Wait 124
RWPFF Resource Wait 126
RWPAG Resource Wait 126
RWBRK Resource Wait 126
RWIMG Resource Wait 129
RWQUO Resource Wait 129
RWLCK Resource Wait 129
RWSWP Resource Wait 130
RWMPE Resource Wait 131
RWMPB Resource Wait 132
RWSCS Resource Wait 132
RWCLU Resource Wait 133
RESTART BUGCHECKS 135
IVLISTK Bugcheck 137
DBLERR Bugcheck 138
HALT Bugcheck 139
ILLVEC Bugcheck 140
NOUSRWCS Bugcheck 141
ERRHALT Bugcheck 141
CHMONIS Bugcheck 141
CHMVEC Bugcheck 143
SCBRDERR Bugcheck 144
WCSCORR Bugcheck 144
CPUCEASED Bugcheck 144
OUTOFSYNC Bugcheck 144
ACCVIOMCHK Bugcheck 144
ACCVIOKSTK Bugcheck 145
SSRVEXCEPT BUGCHECK 146
STACK PATTERNS 149
STACK PATTERNS - EXEC MODE STACK 152
Exec Mode Stack Patterns 152
STACK PATTERNS - INTERRUPT STACK 156
Interrupt Stack Priority Level Usage Table . . . 157
STACK PATTERNS - KERNEL MODE STACK 161
Kernel Mode Stack Patterns 162
SYSTEM SERVICE VECTORS 166
System Service Vector Addresses 166
System Service Vector Contents 166
System Service Vector Stack Footprints 167
Resolving System Service Vector Addresses 168
UNXSIGNAL BUGCHECK 170

VIRTUAL ADDRESSES 174
VIRTUAL ADDRESSES - P0 SPACE 176
VIRTUAL ADDRESSES - P1 SPACE 178
V3 P1 Space Organization 180
V4 P1 Space Organization 182
User Stack 185
Extra User Stack Pages 185
Image I/O Segment 185
Per-Process Message Section 186
CLI Symbol Table 186
CLI Command Table 186
CLI Image 186
Files-11 XQP Regions 186
Process I/O Segment 187
Process Allocation Region 188
Channel Control Block Table 188
P1 Window To Process Header 189
RMS Process Context Area 190
RMS Tracepoint Page 190
RMS Directory Cache 191
RMS IFAB/IRAB Tables 191
Per-Process Common Regions 191
Compatibility Mode Data Pages 192
User Mode Data Page 192
Security Audit Data Pages 192
Image Activator Context Page 192
CLI Data Page 192
Image Activator Scratch Pages 193
Debugger Context Pages 193
Dispatch Vectors For User-Written System
Services And Messages 193
Image Header Buffer 193
KRP Lookaside List 194
Inner Access Mode Stacks 194
System Service Vector Pages 195
P1 Pointer Page 195
Debugger Symbol Table 195
VIRTUAL ADDRESSES - SYSTEM SPACE 196
V3 System Space Organization 198
V4 System Space Organization 201
SYS.EXE 205
Allocatable System Space 205
Adapter I/O Space 205
CONINTERR SPTEs 207
Black Hole Page 207
Mount Verify Page 207
Erase Pattern Buffer 208
Erase Pattern Pseudo Page Table 208
RMS.EXE 208
SYSMSG.EXE 208
Device Driver SVPNs 209
MSCP.EXE 209
Restart Parameter Block (RPB) 209
Page Frame Number (PFN) Data Base 209

Paged Pool 210
Nonpaged Pool Variable Length List 210
Device Driver Images 211
MP.EXE 212
VAXEMUL.EXE 212
FPEMUL.EXE 212
CLUSTRLOA.EXE 213
SCSLOA.EXE 213
SYSLOAxxx.EXE 214
TTDRIVER.EXE 214
System Disk Boot Driver 215
CI Microcode 216
Large Request Packet (LRP) Lookaside List . . . 216
I/O Request Packet (IRP) Lookaside List 217
Small Request Packet (SRP) Lookaside List . . . 217
Interrupt Stack 218
System Control Block 218
Balance Set Slots 219
System Header 219
System Page Table 219
Global Page Table 220
INDEX

PREFACE

This document is a step by step approach to help you analyze VAX/VMS crashdumps, processor halts, and process or system hangs.

The approach documented for analyzing crashdumps will not work for all crashdumps, but it should be helpful for many. The sequence of steps documented is not necessarily the only valid sequence, merely one believed to be useful in many circumstances, helping you more quickly to identify the point of failure.

The approach documented for analyzing processor halts should help you determine whether a halt is due to hardware or software failure.

Process and system hangs may be due to user error, hardware problems, software error, or inadequate system resources. The approach documented for analyzing hangs should help you determine the nature of some hangs and perhaps restore normal operations.

Please note that this document is currently FOR INTERNAL USE ONLY.

Note to Reviewers

If you are reading this, please consider yourself a reviewer.

This document is available on the following Easynet systems:
VAXWRK::SYS$NOTES:BUGCHECK.MEM,
| VMS SWE cluster DOCD$:[V4LIBRARY.MISC]BUGCHECK.MEM,
ISOLA::BUGCHECK.MEM

| This book may also be ordered from Educational Services/Software
| Services Training. See below for further information.

This is a draft of a larger undertaking. I would appreciate comments on the format, on errors of omission or commission, on your own favorite tricks, hints, folklore, and other useful information. This document is the beginning of a tree-like set of procedures, and I'm interested in your reaction to the basic idea.

Please send review comments, suggestions, and omissions from the Index via engineering net mail to VAXWRK::GOLDENBERG or interoffice mail to Ruth Goldenberg PK02/M21.

Change bars mark the changes between this version, Revision 0.C, and
| the previous one. Revision 0.C renames several sections. In
| particular, IDENTIFYING VIRTUAL ADDRESSES is now titled VIRTUAL
| ADDRESSES; DECIPHERING STACKS is now STACK PATTERNS; CPU HALTS is now
| HALTS. Future enhanced versions of BUGCHECK.MEM will be announced in VAXWRK::SYS$NOTES:VMSNOTES.NOT, in the VMS SWE cluster SYSNOTES, and in VAXworks' VMSnews.

Intended Audience

This document is intended for Software Specialists, Field Service engineers, and others who must troubleshoot VAX/VMS problems. You are presumed to have knowledge of VMS internals and some familiarity with its sources and with the System Dump Analyzer (SDA) utility.

How to Use This Document

This manual is organized into sections and is not intended to be read straight through from the first page to the last.

Each section begins on a new page and is titled in the top left hand corner. Except for the first two sections, the sections are in alphabetic sequence by section title.

The first section, MESSAGES, is an index to messages described elsewhere in the document. These include console halt messages, fatal bugcheck messages, and SDA error messages.

The second section is STARTING POINTS. If you are trying to troubleshoot a crash, halt, or hang, begin with STARTING POINTS.

If you are looking for information on a particular topic, the index at the back of the document may be helpful.

Conventions Used in This Document

The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/Z.

Command examples include underlines for all output lines or prompting characters that the system prints or displays.

Angle brackets ("<" and ">"), enclosing a descriptive name, are used to indicate information which you must supply as part of a command, and should not themselves be included in the command. For example,

```
$EXIT %X<exception_type>
```

means that you supply the actual exception type value when you issue this DCL command.

The expression A(symbol) means address of symbol. For example, A(SRVEXIT) means the address of the symbol SRVEXIT.

For consistency with the VAX/VMS Internals and Data Structures manual, the term "SYSBOOT parameter" is used rather than "SYSGEN parameter", to describe any of the adjustable parameters used by the secondary bootstrap program SYSBOOT to configure the system.

All addresses in the text are hexadecimal. All other numbers in the text are decimal unless otherwise identified.

Stack layouts show positional information, for example, which longword in a signal array is the PC of the exception. Where appropriate, stack layouts show actual hexadecimal numeric information, for example, 0000000C as the exception type in an access violation signal array. Variable contents of a stack longword are expressed as some number of unknown hexadecimal digits, such as "xxxxxxxx" for the value of the exception PC in a signal array.

All addresses in the text are virtual unless identified as physical.

Unidentified global names are defined in SYS.EXE.

System modules are identified as [<facility_name>]<module_name>, for example, [SYS]ASTDEL or [DRIVER]LPDRIVER. The source listing microfiche is organized by facility name. The last sheet of the source fiche contains an index to the rest of the fiche. The facilities are ordered alphabetically in the fiche with link maps and source listings for the components of the facility.

The term <cr> means the RETURN key. It is shown only in examples of relatively unfamiliar utilities, such as MicroODT on the console LSI-11 of a VAX-11/780. Most examples in this document do not explicitly show the RETURN key being pressed following commands. Assume that all command lines shown end with a <cr> unless stated otherwise.

[TBS] means "to be supplied".

VMS manuals referenced in the text are V4 manuals unless otherwise noted.

Associated Documents

The following documents are necessary accompaniments to this document:

- o VAX/VMS Internals and Data Structures Manual
- o VAX/VMS System Dump Analyzer Reference Manual
- o VAX Architecture Standard (DEC Standard 032) or VAX-11 Architecture Reference Manual
- o VAX/VMS Source List Microfiche

Ordering Information

Ordering information for employees wishing self-paced training material and information documents produced by Educational Services/Software Services Training appears below. Note that this is not necessarily the same procedure as for enrolling in lecture courses in your particular geography.

U.S.

US Software Specialists should order from their Software Services Training Registrar. Other US employees should order directly from the Educational Services Bookroom in Billerica, Massachusetts.

When ordering directly from the Bookroom, specify title, order number, quantity, full ship-to address (not just a mailstop), if partial shipments are allowed, name, badge number, and cost center. Send order by VAXmail to CECILE::MAILPO, by DECmail to MAILPO @BKO, by interoffice mail to mailstop BKO, by postoffice to Digital Equipment Corporation/12A Esquire Road/N. Billerica, MA 01862/USA, or telephone 800-343-8321.

Europe

European Software Specialists should order from their logistics contact. Other European employees should order from their logistics contact, or directly from the Educational Services Bookroom in Billerica, Massachusetts (see instructions under US, above).

GIA

All GIA employees should order from their logistic contact.

MESSAGES

%%
VAX-11/750, 66
?CHM ERR
VAX-11/780, 50
VAX-11/785, 50
?CLOCK PHASE ERROR
VAX-11/785, 51
?CPU DBLE-ERR HALT
VAX-11/780, 51
VAX-11/785, 51
?ILL I/E VEC
VAX-11/780, 52
VAX-11/785, 52
?INT-STK INVALID
VAX-11/780, 53
VAX-11/785, 53
?NO USR WCS
VAX-11/780, 54
VAX-11/785, 54

@ console prompt
VAX-11/780, 56
VAX-11/785, 56
ATTEMPTING WARM RESTART
VAX-11/780, 58
VAX-11/785, 58

FATAL BUG CHECK, VERSION = Vn.n
ASYNCWRTER, 18, 19
CHMONIS, 141
CHMVEC, 143
DBLERR, 138
FATALEXCPT, 33
HALT, 139
ILLVEC, 140
INVEXCEPTN, 84
IVLISTK, 137
KRNLSTAKNV, 89
MACHINECHK, 98
NOUSRWCS, 141
OPERATOR, 13
OUTOFSYNC, 51, 144
PGFIPLHI, 113
SCBRDERR, 144
SSRVEXCEPT, 146
STATENTSVD, 135
UNKRSTRT, 51, 59, 136
UNXINTEXC, 17
UNXSIGNAL, 170

FATAL BUG CHECK, VERSION = Vn.n, 11

HALT INST EXECUTED
  VAX-11/780, 48
  VAX-11/785, 48
HALTED AT 00000000
  VAX-11/780, 48
  VAX-11/785, 48
HALTED AT 200034F9
  VAX-11/780, 49
HALTED AT 2000350A
  VAX-11/780, 48, 58
HALTED AT 20003552
  VAX-11/780, 49
  VAX-11/785, 49
HALTED AT 20003563
  VAX-11/780, 58
  VAX-11/785, 58
HALTED AT 20003564
  VAX-11/780, 48
  VAX-11/785, 48
HALTED AT 8xxxxxxx
  VAX-11/780, 49
  VAX-11/785, 49
HALTED AT xxxxxxxx
  VAX-11/780, 47
  VAX-11/785, 47

OPCOM
  mount verification in progress, 68

RMS-F-DME, 188

SDA
  Remaining registers not available, 145
  SDA-E-DUMPEMPTY, 13
  SDA-W-NOREAD, 12
  SDA-W-NOREQ, 12
  SDA-E-NOSYMBOLS, 197
  SDA-W-SHORTDUMP, 22
  SYSTEM-E-SPTNOTFND, 22
  SYSTEM-F-VASFULL, 12
  unable to access location, 5
SYSTEM SHUTDOWN COMPLETE - USE CONSOLE TO HALT THE SYSTEM, 23
SYSTEM-F-MCHECK, 98
SYSTEM-W-PAGECRIT, 66
SYSTEM-W-PAGEFRAG, 66
SYSTEM-F-INSFMEM, 188, 210, 211
SYSTEM-W-POOLEXPF, 125

VAX-11/750
  %%, 66

VAX-11/780
@ console prompt, 56
?CHM ERR, 50
?CPU DBLE-ERR HALT, 51
?ILL I/E VEC, 52
?INT-STK INVALID, 53
?NO USR WCS, 54
ATTEMPTING WARM RESTART, 58
HALT INST EXECUTED, 48
HALTED AT 00000000, 48
HALTED AT 200034F9, 49
HALTED AT 2000350A, 48, 58
HALTED AT 20003552, 49
HALTED AT 20003563, 48, 58
HALTED AT 8xxxxxxx, 49
HALTED AT xxxxxxxx, 47
VAX-11/785
@ console prompt, 56
?CHM ERR, 50
?CLOCK PHASE ERROR, 51
?CPU DBLE-ERR HALT, 51
?ILL I/E VEC, 52
?INT-STK INVALID, 53
?NO USR WCS, 54
ATTEMPTING WARM RESTART, 58
HALT INST EXECUTED, 48
HALTED AT 00000000, 48
HALTED AT 20003552, 49
HALTED AT 20003563, 48, 58
HALTED AT 8xxxxxxx, 49
HALTED AT xxxxxxxx, 47

STARTING POINTS

The initial starting point for troubleshooting crashes is the section BUGCHECKS. Follow the directions in that section and in any sections to which the text directs you.

The initial starting point for troubleshooting processor halts is the section HALTS - VAX-11/780 and VAX-11/785. (Other VAX processors' halts will be documented in future revisions of this document.) Follow the directions in that section and in any sections to which the text directs you.

The initial starting point for troubleshooting suspected hangs, process or system, is the section HANGS. Follow the directions in that section and in any sections to which the text directs you.

ACCESSING PROCESS CONTEXT WITH SDA

Process context means the per-process virtual address space, general registers, processor registers, and system data structures associated with a particular process. The major system data structures that describe a process are the software Process Control Block (PCB), Job Information Block (JIB), and the Process Header (PHD). Examining these data structures, whether in a crashdump or on the running system, is usually straightforward. Type the following SDA command

```
SDA> SHOW SUMMARY          !to get process id and pix
```

Under V4, the column labeled "Indx" contains the process index, or "pix". Under V3, the pix is the low word of the process id. Type the following SDA commands

```
SDA> SET PROCESS/INDEX=<pix>
SDA> READ SYS$SYSTEM:SYSDEF.STB !if you haven't
SDA> SHOW PROCESS/PCB      !display PCB and get JIB address
SDA> FORMAT <jib_address> !format JIB
SDA> SHOW PROCESS/PHD      !display PHD fixed part
SDA> SHOW PROCESS/REG      !...process registers
SDA> SHOW PROCESS/WORK     !...working set list
SDA> SHOW PROCESS/PROC     !...process section table
SDA> SHOW PROCESS/PAGE     !...page tables
SDA> SHOW STACK/USER       !user stack
SDA> SHOW STACK/SUP        !sup. stack
SDA> SHOW STACK/EXEC       !exec stack
SDA> SHOW STACK/KERNEL     !kernel stack
```

Issue EXAMINE commands to examine whatever per-process addresses you want to see.

Under some circumstances, SDA is unable to execute any of the commands above and outputs the error

```
unable to access location <x>
```

This error means that SDA, analyzing the running system, is unable to access the target process.

When you are analyzing the running system, SDA queues a special kernel AST to the process of interest in order to examine its address space, its PCB, or its PHD. The special kernel AST collects information, running in the context of the target process, and requeues itself to the process running SDA. SDA waits three seconds for the special kernel AST to complete this. Even if the target process is outswapped, the AST enqueuing makes the process computable and usually causes it to be inswapped. If, however, the process's priority is very low with respect to other computable processes and/or the system is heavily loaded, SDA's three second timeout may expire. Furthermore, if the process is in a lengthy wait at IPL 2, the AST cannot be delivered, and SDA's timeout will expire. When the timeout expires, SDA issues the error message above.

If you suspect that the process's priority is the problem, first display the system with either the SDA SHOW SUMMARY command or the DCL SHOW SYSTEM command for confirmation. Increase the process's priority and then re-issue the SDA command. Remember to lower the process's priority when you are done. From an account with ALTPRI and WORLD privileges, type the DCL command

```
$ SET PROCESS/PRIORITY=<new_priority>/ID=<pid>
```

You may still be unable to examine the process's context after altering its priority, particularly if the process is being waited at IPL 2. The sections below discuss possible alternative ways to examine the PCB, JIB, and PHD. There is no way to examine the per-process address space of a process on the running system which is being waited at IPL 2.

The PCB And JIB

The software PCB and JIB are in nonpaged pool and always accessible to SDA. However, SDA's usual method of obtaining these for a process on the current system fails under the circumstances described above.

The alternative is the following SDA commands

```
SDA> READ SYS$SYSTEM:SYSDEF.STB
SDA> DEF PCB=@(@SCH$GL_PCBVEC+(4*<pix>))
SDA> FORMAT PCB                     !display PCB
SDA> FORMAT @(PCB + PCB$L_JIB)      !display JIB
```

The PHD

The PHD contains information which is generally not needed unless the process is resident: the hardware PCB, the working set list, the process section table, the per-process page tables, and accounting and quota information. The hardware PCB is the area used to record the process's general registers when the process's context is saved. The hardware PCB may be of particular interest if you are trying to determine why a process is in a lengthy wait. The PHD is nonpageable, except for the per-process page tables.

The PHD is in a region of system space called the balance set slots. When a process is inswapped a balance set slot is allocated for its PHD. When a process is outswapped, its PHD may be outswapped also, and the balance set slot virtual address space set to no access or allocated to another process. When the process is inswapped again, its PHD is likely to be in a different balance set slot. Because the PHD is not permanently resident, SDA always accesses it from the context of its process. However, if the process is being waited at IPL 2, SDA's special kernel AST is unable to run in that process's context.

The program in subsection Hints and Kinks is an alternative way to access the nonpageable part of the process header. Run it from a process with CMKRNL privilege. Given the pix of a target process, the program makes various checks on the validity of the process header and displays its hardware PCB if possible. Modification of the program to display the working set list, process section table, or accounting and quota information is left as an exercise to the reader.

If you are trying to learn more about a problem involving a hung process which can be reproduced at will, you should lock the process into the balance set before reproducing the problem so that its PHD is always resident. The user must be granted the privilege PSWAPM and must issue the DCL command SET PROCESS/NOSWAP before running the program that causes the process to hang.

Hints And Kinks

```
     .title READPHD
;
; This program, run from an account with CMKRNL privilege, displays the
;   hardware pcb from the process header of a specified process. Its
;   intended use is to display the general registers of a resident process
;   being waited at IPL 2 and thus inaccessible to online SDA.
;
; Build it using the following commands:
;    MACRO READPHD + SYS$LIBRARY:LIB/LIB
;    LINK READPHD + SYS$SYSTEM:SYS.STB/SEL
;
; When you run it, it will prompt for the process index, or "pix" of the
;   target process. Under V3, the pix is the low word of the process
;   id. Under V4, SDA SHOW SUMMARY has a column "Indx" which contains the
;   pix of each process. If the input pix contains any non-hex digits, the
;   program will reprompt. If the pix isn't valid or the PHD unavailable,
;   the program will exit with status nonexistent process.
;
     $IPLDEF                        ;define IPL symbols
     $PCBDEF                        ;define process control block offsets
     $PHDDEF                        ;define process header offsets
start::      .word   0
;
;  get target process index
;
10$: pushaw  pixinput               ;gets input size from lib$get_input
     pushaq  pixprompt              ;prompt arg
     pushaq  pixinput               ;input arg
     calls   #3,g^lib$get_input     ;get target pix
     blbc    r0,20$                 ;branch if error
;
;  convert pix to binary
;
     clrl    -(sp)                  ;zero flags arg
     pushl   #4                     ;value-size arg = longword
     pushal  pixarg                 ;binary result arg
     pushaq  pixinput               ;input string address arg
     calls   #4,g^ots$cvt_tz_l      ;convert pix to binary
     blbc    r0,10$                 ;branch if non-hex digit
;
;  call kernel mode procedure to copy hardware pcb of target process
;
     $cmkrnl_s getphd,arglist       ;read hardware pcb of target
     blbc    r0,20$                 ;branch if failure to access phd
;
;  format and display hardware pcb
;
     $faol_s faoctr,outsize,output,phdbuffer ;format hardware pcb
     pushaq  output                 ;output arg
     calls   #1,g^lib$put_output    ;display hardware pcb
20$: $exit_s r0
;
```

```
;  kernel mode procedure to read target's process header if available
;
        .enable lsb
getphd: .word           ^m<r2,r3,r4,r5>
        movl    4(ap),r2                ;get pix of target
        cmpl    r2,sch$gl_maxpix        ;is pix too large?
        bgtr    20$                     ;branch if yes
5$:     setipl  synch                   ;raise ipl and lock pages touched
                                        ; at high ipl
        movl    @sch$gl_pcbvec[r2],r4   ;get pcb address of target
        cmpl    r4,#sch$gl_nullpcb      ;has target process been deleted?
        beql    20$                     ;branch if yes
        bbc     #pcb$v_phdres,pcb$l_sts(r4),20$
                                        ;branch if  phd not resident
        movl    pcb$l_phd(r4),r3        ;get phd address of target
        bgeq    20$                     ;additional sanity check
                                        ;branch if phd not system space addre
ss
        movc3   #24*4,phd$l_ksp(r3),phdbuffer ;copy hardware pcb fields only
        movzwl  #ss$_normal,r0          ;return success
10$:    setipl  #0                      ;restore ipl
        ret
20$:    movzwl  #ss$_nonexpr,r0         ;proc. outswapped, pix incorrect, etc
.
        brb     10$
;
;  data
;
phdbuffer:
        .blkl   24
synch:          .long   ipl$_synch
        assume <.-5$> le 512
        .disable lsb
;
; end of data accessed at high ipl
;
arglist:.long           1               ;for getphd procedure
pixarg:         .blkl   1               ;process index of target phd
;
pixprompt:
        .ascid  \enter hex pix:\
pixinput:
        .ascid  \           \
faoctr: .ascid          \KSP=!XL  ESP=!XL  SSP=!XL  USP=!XL!/\-
\ R0=!XL    R1=!XL    R2=!XL    R3=!XL!/\-
\ R4=!XL    R5=!XL    R6=!XL    R7=!XL!/\-
\ R8=!XL    R9=!XL   R10=!XL   R11=!XL!/\-
\ AP=!XL    FP=!XL    PC=!XL   PSL=!XL!/\-
\P0BR=!XL  P0LRASTL=!XL  P1BR=!XL  P1LR=!XL\
output:         .long   400
        .address 10$
10$: .blkb   400                        ;buffer for fao output
outsize: .blkw          1               ;length of fao output
        .end    start
```

Additional References

V3 VAX/VMS Internals and Data Structure Manual, Chapter 14, Memory Management Data Structures

BUGCHECKS

The main steps of initial bugcheck analysis follow.

- o Find the dump file and determine the bugcheck type. If you are uncertain about how to do this or experience problems, see the following pages in this section for more detailed directions.

- o

FOR BUGCHECK TYPE:	GO TO SECTION:
ASYNCWRTER	CPU-SPECIFIC INTERRUPTS
CHMONIS	RESTART BUGCHECKS
CHMVEC	RESTART BUGCHECKS
DBLERR	RESTART BUGCHECKS
FATALEXCEPT	FATALEXCEPT BUGCHECK
HALT	RESTART BUGCHECKS
ILLVEC	RESTART BUGCHECKS
INVEXCEPTN	INVEXCEPTN BUGCHECK
IVLISTK	RESTART BUGCHECKS
KRNLSTAKNV	KRNLSTAKNV BUGCHECK
MACHINECHK	MACHINE CHECKS
OPERATOR	uninteresting dump
OUTOFSYNC	RESTART BUGCHECKS
PGFIPLHI	PGFIPLHI BUGCHECK
SCBRDERR	RESTART BUGCHECKS
SSRVEXCEPT	SSRVEXCEPT BUGCHECK
STATENTSVD	RESTART BUGCHECKS
UNKRSTRT	RESTART BUGCHECKS
UNXSIGNAL	UNXSIGNAL BUGCHECK

- o If the bugcheck type is not one of those listed above, see the following pages in this section.

All crashes result from a system software decision that system integrity is compromised. Following this decision, the system software bugchecks by executing the code generated by the BUG_CHECK macro. An example of a system software decision to crash the system is in the IPL 3 interrupt service routine; the code which selects a new process to place into execution bugchecks if the data structure it removed from a compute queue is not a software PCB.

The BUG_CHECK macro has one required argument, the bugcheck name, and an optional argument, the keyword FATAL. The macro generates the two-byte opcode FEFF followed by an immediate operand generated from the macro arguments. Execution of the bugcheck opcode results in an Opcode Reserved to Digital exception. The exception service routine special cases the bugcheck opcode by dispatching to EXE$BUG_CHECK. For a fatal exec or kernel mode bugcheck, EXE$BUG_CHECK crashes the system, writing a register and stack display to the console terminal and, if conditions allow (see section CRASHDUMP REQUIREMENTS), writing a dump of physical memory to the system dump file.

1. First, find the dump file. Initially, the dump should be in SYS$SYSTEM:SYSDUMP.DMP or SYS$SYSTEM:PAGEFILE.SYS. Often the dump will have been copied elsewhere. If it hasn't been, do so before proceeding further to avoid its loss following another crash or a normal system shutdown. Note that the dump file by default has the NOBACKUP attribute. This means that if you copy the dump with BACKUP, you must use the /IGNORE=NOBACKUP qualifier.

2. Run SDA to determine whether the dump file is valid. From an account with file access to the dump, type the DCL command

   $ ANALYZE/CRASH <dump_filespec>

   SDA should respond with the date the dump was taken and the bugcheck type and message text.

3. If SDA reports the error SDA-W-NOREQ, symbol "<x>" not found in system symbol table, or the error SDA-W-NOREAD, unable to access location <x>, then most likely there is an incompatibility among the version of SDA, the version of SYS.STB SDA reads as part of its initialization, and the version of the crashed system. In order for you successfully to analyze a dump, the dump, SDA, and the SYS.STB must be from the same major release of VMS.

   If you don't specify the /SYMBOL qualifier to the ANALYZE command, SDA looks for SYS.STB first in the directory containing the dump and then in SYS$SYSTEM.

4. If SDA reports the error SYSTEM-F-VASFULL, virtual address space is full, then the SYSBOOT parameter VIRTUALPAGCNT is too small. Analyzing a dump requires a SYSBOOT VIRTUALPAGCNT parameter of some 2000 pages plus the size of the dump file, whether it is SYSDUMP.DMP or PAGEFILE.SYS. (Actual requirements may vary as a

function of SYSBOOT parameters.)

5. If SDA reports the error SDA-E-DUMPEMPTY, dump file contains no valid dump, first check whether a BACKUP save and restore performed without the /IGNORE=NOBACKUP could have restored the size and other attributes of a NOBACKUP dump file without its contents. If that is not the problem, see the section on CRASHDUMP REQUIREMENTS and take appropriate action to get a valid dump the next time the system crashes.

   If you do not have a valid dump file, but do have console bugcheck output, analysis of the crash may be possible. You should attempt it, following the directions below to the extent possible. The console bugcheck output is similar to the result of issuing the SDA commands

   ```
   SDA> SHOW CRASH
   SDA> SHOW STACK
   ```

   The console output omits the processor register contents. However, the bugcheck errorlog entry from the time of the crash, which may have been written to SYS$ERRORLOG:ERRLOG.SYS, would contain the processor register contents. Under V3, run SYE and specify S in response to the "OPTIONS" prompt and /BU in response to the "DEVICE NAME" prompt to limit the display to bugcheck entries. Under V4, type the DCL command ANALYZE/ERROR/INCLUDE=BUGCHECKS <file_spec>. Locate the entry corresponding to the date and time the system crashed and read its processor register contents.

6. If the bugcheck type is OPERATOR, this crashdump resulted from an operator requested shutdown of the system. These always occur as the last step of shutdown and are generally of no interest. If you or someone else mistakenly shut down the system because you wanted to examine a hung or slow system, next time use the CRASH command procedure documented in Section 4.1 of the Guide to VAX/VMS System Management and Daily Operations.

7. Type the SDA command

   ```
   SDA> SHOW CRASH
   ```

   to display the system version and register contents. The registers displayed are the register values at the time the BUG_CHECK was requested. Subtract 4 from the displayed PC to determine the address of the BUG_CHECK macro.

8. Type the SDA command SHOW STACK to display the stack current at the time of crash. Note that for each V3 SDA COPY command used to copy the dump, the SP will be 8 bytes greater than its actual value; that is, SDA will show the SP pointing to a stack address 8 bytes higher than it should. This V3 bug has been corrected in V4.

9. If the bugcheck type is in the list below, continue with the steps in the specified section.

FOR BUGCHECK TYPE:	GO TO SECTION:
ASYNCWRTER	CPU-SPECIFIC INTERRUPTS
CHMONIS	RESTART BUGCHECKS
CHMVEC	RESTART BUGCHECKS
DBLERR	RESTART BUGCHECKS
FATALEXCEPT	FATALEXCEPT BUGCHECK
HALT	RESTART BUGCHECKS
ILLVEC	RESTART BUGCHECKS
INVEXCEPTN	INVEXCEPTN BUGCHECK
IVLISTK	RESTART BUGCHECKS
KRNLSTAKNV	KRNLSTAKNV BUGCHECK
MACHINECHK	MACHINE CHECKS
OPERATOR	uninteresting dump
OUTOFSYNC	RESTART BUGCHECKS
PGFIPLHI	PGFIPLHI BUGCHECK
SCBRDERR	RESTART BUGCHECKS
SSRVEXCEPT	SSRVEXCEPT BUGCHECK
STATENTSVD	RESTART BUGCHECKS
UNKRSTRT	RESTART BUGCHECKS
UNXSIGNAL	UNXSIGNAL BUGCHECK

10. If the bugcheck type is not one of those listed above, identify in what source module the PC is, using directions in section VIRTUAL ADDRESSES. Locate and read the source code to determine what anomaly the system detected and the significance of the general registers and relevant data structure contents.

11. Decipher the current stack to trace control flow up to the point of error. See the section STACK PATTERNS.

Hints And Kinks

1. Although "deciphering stacks" and "identifying virtual addresses" are listed as single and separate steps, in practice, they are usually repetitive and intertwined. For example, that a particular longword can be interpreted as a particular address should be confirmed in the context of what code was executing and manipulating that longword. Usually this requires that some piece of the stack be deciphered. Another example is that identifying a particular footprint on the stack may require or result in the identification of addresses within that footprint.

2. When SDA examines the process current at the time of an interrupt stack bugcheck, SDA assumes the bugcheck PC and PSL and all the general registers are part of that process's context and displays

them in response to the SHOW PROCESS/REGISTER command.

3. Whenever you modify SYSBOOT parameters, remember to make AUTOGEN aware of your changes so that they propagate across AUTOGENs. Include any parameter changes you make in V3 SYS$SYSTEM:PARAMS.DAT or in V4 SYS$SYSTEM:MODPARAMS.DAT. See Chapter 11 in the Guide to VAX/VMS System Management and Daily Operations for further information on AUTOGEN.

Additional References

V3 VAX/VMS Internals and Data Structures Manual, Section 8.2, System Crashes

VAX/VMS System Dump Analyzer Reference Manual, for use of SDA

CPU-SPECIFIC INTERRUPTS

Five vectors in the System Control Block, at hex offsets 50 through 60, are reserved for cpu-specific system bus and memory errors. These interrupts occur at cpu-specific IPLs within the range hex 18 through 1D.

VMS services these interrupts in a cpu-specific image loaded into nonpaged pool during system initialization. The image name is of the form SYSLOAxxx.EXE, where xxx designates the cpu type.

CPU	IMAGE NAME
MicroVAX I	SYSLOAUV1.EXE
MicroVAX II	SYSLOAUV2.EXE
VAX-11/730	SYSLOA730.EXE
VAX-11/750	SYSLOA750.EXE
VAX-11/780	SYSLOA780.EXE
VAX-11/785	SYSLOA780.EXE
VAX 8600	SYSLOA790.EXE
VAXstation I	SYSLOAWS1.EXE
VAXstation II	SYSLOAWS2.EXE

See the subsection SYSLOAXXX.EXE in the section VIRTUAL ADDRESSES - SYSTEM SPACE for more information on the mechanism for dispatching into SYSLOAxxx.EXE.

In general, VMS servicing of these interrupts is done at IPL 31 and includes logging an error to the error log.

For more information, see the section corresponding to the cpu of interest

CPU-SPECIFIC INTERRUPTS - VAX-11/780 AND VAX-11/785
[others TBS]

CPU-SPECIFIC INTERRUPTS - VAX-11/780 AND VAX-11/785

The VAX-11/780 and VAX-11/785 have five cpu-specific interrupts.

Hex SCB Offset	Hex IPL	Interrupt Name
50	19	SBI Silo Compare
54	1A	CRD/RDS
58	1B	SBI Alert
5C	1C	SBI Fault
60	1D	CPU Timeout

SBI Silo Compare

This interrupt occurs when a match is detected on particular signal fields of the SBI bus. The signal fields being checked can be program-selected by control bits in the SILO COMPARATOR register. The previous sixteen cycles on the SBI bus are latched in the silo register for interrogation by diagnostic software.

This interrupt can occur only if it is enabled in the SILO COMPARATOR register. It is very unlikely to occur and documented here for completeness more than anything else. VMS does not enable it and handles this interrupt as an unexpected interrupt. That is, VMS signals the nonfatal bugcheck UNXINTEXC. If BUGCHECKFATAL is 0, its default value, the result is a bugcheck error log entry.

This interrupt can be used as an ad hoc troubleshooting tool for particular kinds of hardware problems by someone who understands the SBI protocol, who can interpret the silo contents, and who can load a service routine for the interrupt into nonpaged pool.

CRD/RDS

The Corrected Read Data (CRD) interrupt occurs when the processor receives read data which has been error-corrected by memory. The Read Data Substitute (RDS) interrupt may occur when the processor receives bad data which cannot be error-corrected. If the cpu attempts to use bad data in instruction execution, a machine check occurs. If instruction execution alters control flow so that bad data in the instruction prefetch buffer is unused, the RDS interrupt occurs.

This interrupt cannot be generated through software error. It can be caused by hardware problems in the memory controllers or their memory arrays.

VMS's interrupt service routine logs this in the error log with type SE (soft memory error) and increments EXE$GL_MEMERRS. The contents

of EXE$GL_MEMERRS are displayed in the output from the DCL command SHOW ERROR as MEMORY errors. VMS reads the memory controller status registers to include them in the errorlog entry. If there is a MS780E controller indicating either SBI interface write data parity error or microsequencer parity error, VMS signals the fatal bugcheck ASYNCWRTER. (In contrast to the ASYNCWRTER bugcheck signaled by the cpu write timeout interrupt service routine, this crashdump does not have a faked machine check logout on the stack.) Otherwise, it dismisses the interrupt.

If you think an ASYNCWRTER crash was caused by this problem, the best way to learn more is through analyzing the error log, because the crashdump does not contain the contents of all the interesting processor and memory controller registers. Under V3, run SYE and specify S in response to the "OPTIONS" prompt and /CP to the "DEVICE NAME" prompt. Under V4, type the DCL command ANALYZE/ERROR/INCLUDE=CPU <filespec>. The error log report displays and interprets the contents of the SBITA and SBIER registers. The SBITA register contains the physical SBI address (of a longword) that timed out. The error log report shows this as the address (of a byte) following "TIMEOUT CONSOLE ADDR =". If this address is less than 20000000, then it is a memory address. Otherwise, it is a nexus register address. The physical address corresponding to the start of nexus N's registers is 200xx000, where xx equals 2 times N in hex. For example, the registers for nexus 4 begin at 20008000. Compute the nexus number. To find out what is present at that nexus, look at the error log report. It displays the configuration/status register for each nexus, along with the nexus number and type.

SBI Alert

This interrupt occurs when an SBI adapter or controller asserts the SBI Alert line. Adapters or controllers which have no other means of requesting SBI interrupts can assert this signal to request an interrupt. Currently, only MS780C and MS780E memory controllers without ISP ROMs assert this line to report memory power failure or recovery. (A memory controller with an ISP ROM is considered more critical to system functioning and is usually jumpered to assert a different SBI signal to report power problems.)

This interrupt cannot be generated through software error. It can be caused by hardware problems in memory controllers or their power supplies.

VMS logs this in the error log with type SA (SBI Alert) and increments EXE$GL_MEMERRS. The contents of EXE$GL_MEMERRS are displayed in the output from the DCL command SHOW ERROR as MEMORY errors. VMS reads the memory controller status registers to include them in the errorlog entry. If there is a MS780E controller indicating either SBI interface write data parity error or microsequencer parity error, VMS signals the fatal bugcheck ASYNCWRTER. (In contrast to the ASYNCWRTER bugcheck signaled by the

cpu write timeout interrupt service routine, this crashdump does not have a faked machine check logout on the stack.)

If you think an ASYNCWRTER crash was caused by this problem, the best way to learn more is through analyzing the error log, because the crashdump does not contain the contents of all the interesting processor and memory controller registers.

SBI Fault

This interrupt occurs if an SBI bus error was detected by any adapter or controller on the SBI, including the cpu. Possible bus errors include SBI parity error, write sequence fault, interlock sequence fault, and multiple SBI transmitter fault. If the cpu detects a fault condition preventing completion of a read cycle for the cpu, the cpu also generates a machine check, typically a read timeout machine check. See section MACHINE CHECKS - VAX-11/780 and VAX-11/785.

This interrupt cannot be generated through software error. It can be caused by hardware problems in the SBI, memory controllers, and SBI nexus.

VMS's interrupt service routine logs this in the error log with type BE (bus error), increments EXE$GL_MCHKERRS, and dismisses the interrupt. The contents of EXE$GL_MCHKERRS are displayed in the output from the DCL command SHOW ERROR as CPU errors. If you see errors of this sort in the error log, contact Field Service.

Cpu Timeout

This interrupt occurs if the processor receives an error confirmation from an SBI nexus for the second longword of an extended read operation or does not receive SBI command completion within 512 SBI cycles.

This interrupt cannot be generated through software error. It can be caused by hardware problems in the SBI, memory controllers, and SBI nexus.

VMS's interrupt service routine logs the error in the error log with an entry type of AW (asynchronous write) and increments EXE$GL_MCHKERRS. The contents of EXE$GL_MCHKERRS are displayed in the output from the DCL command SHOW ERROR as CPU errors. VMS then tests whether the address reference was made from user/supervisor mode or from exec/kernel mode. If the reference was made from exec or kernel mode, VMS signals the fatal bugcheck ASYNCWRTER. If the reference was made from user or supervisor mode, VMS signals a machine check exception to the access mode active at the time the interrupt occurred. If you see errors of this sort in the error log,

contact Field Service.

The error log from the time of the crash is very important in analyzing this error, because the crashdump does not contain the contents of all the interesting processor and memory controller registers. The interrupt PC and PSL are possibly irrelevant to the error, since these interrupts do not necessarily occur during the instruction which caused them; the processor is allowed to continue execution while an SBI write cycle is pending.

The crashdump interrupt stack contains a faked microcode machine check error logout, beginning with a hex byte count of 28. This error logout on the stack is present for convenience in executing a common code path, is meaningless, and should be ignored.

Additional References

VAX-11/780 TB/CACHE/SBI Control Technical Description

CRASHDUMP REQUIREMENTS

Following is a list of requirements that must be met for VAX/VMS to write a complete crashdump. Most of these are discussed in more detail below.

- o There must be a crashdump file in SYS$SYSTEM named either SYSDUMP.DMP or PAGEFILE.SYS. If the dump file is SYSDUMP.DMP, it must be four blocks bigger than physical memory. If SYSDUMP.DMP is not present, VMS will write crashdumps to PAGEFILE.SYS; it must be at least 1004 blocks bigger than physical memory, and the SYSBOOT parameter SAVEDUMP must be 1 (default is 0).
- o The data fields in system space describing the dump file's extents must be intact when the system crashes.
- o The resident part of EXE$BUG_CHECK must be able to read in the non-resident code, using the boot driver.
- o The SYSBOOT parameter DUMPBUG must be 1 (default is 1).
- o Physical memory must contain no pages with unrecoverable parity errors.
- o The boot driver must be able to write to the dump file.
- o The user must not halt the system via the console terminal until after the console dump messages have been printed in their entirety and memory contents have been written to the crashdump file.

Crashdump File

During system initialization, SYSBOOT first looks up the crashdump file as the highest version of SYS$SYSTEM:SYSDUMP.DMP and records the location of its extent(s) in a data structure called the Boot Control Block. If there is no SYSDUMP.DMP, SYSBOOT maps the extent(s) of SYS$SYSTEM:PAGEFILE.SYS instead. The contents of the Boot Control Block are checksummed at system initialization and again during fatal bugcheck processing. A crash dump is written only if the two checksums are equal. When the system crashes, VMS does not do a further lookup of the dump file. Note that you run a serious risk of corrupting your system disk if you delete the dump file whose extent(s) VMS mapped at system initialization.

For a complete dump to be written, the dump file must be at least as big as local physical memory (unless the SYSBOOT parameter PHYSICALPAGES is less than this) plus any multiport memory plus four blocks. For example, a 2 mb VAX system requires a SYSDUMP.DMP file of 4100 blocks; a 1mb system requires 2052 blocks. If the dump is to be written to PAGEFILE.SYS, increase these numbers by 1000 blocks.

The first block of the dump file is used for a formatted error log message with bugcheck information and the contents of the processor registers. The second and third blocks of the file are used to save the contents of the error log message block buffers (error log messages not yet written to SYS$ERRORLOG:ERRLOG.SYS) The fourth block is reserved and currently unused.

Fatal bugcheck processing code writes physical memory contents to the crashdump file following the reserved blocks of the file. If the dump does not include all of physical memory, SDA outputs the error SDA-W-SHORTDUMP, the dump only contains x out of y pages of phyical memory. If you have set the SYSBOOT parameter PHYSICALPAGES so as not to use all of physical memory, and the dump file is PHYSICALPAGES plus 4, ignore this warning message.

If, however, the dump file is too small to include the System Page Table (SPT), SDA cannot analyze it and outputs the error SDA-E-SPTNOTFND, system page table not found in dump file. The System Page Table (SPT), the key to translating physical addresses to virtual addresses, is usually allocated in the highest physical memory.

You can alter the dump file size with the command procedure SYS$UPDATE:SWAPFILES.COM. For size increases, SYSGEN (invoked by SWAPFILES.COM) extends the current dump file by default. Use of a new or extended dump file, whether it is SYSDUMP.DMP or PAGEFILE.SYS, requires a system reboot. SWAPFILES.COM is documented in Section 11.7 of the Guide to VAX/VMS System Management and Daily Operations.

If you have insufficient free disk space to extend SYSDUMP.DMP large enough, a larger PAGEFILE.SYS and no SYSDUMP.DMP may solve the problem. First, rename SYSDUMP.DMP so that system initialization code cannot find it. Do not delete the renamed SYSDUMP.DMP until after you have rebooted the system, because its extents will be used for a dump when you shut down the system. Shutdown the system and then reboot the system. Delete the renamed SYSDUMP.DMP, and extend PAGEFILE.SYS to the necessary size. Shutdown and reboot again in order to use the extended portion of PAGEFILE.SYS.

Bugcheck Mechanism

When the service routine for Opcode Reserved to Digital exceptions detects either of the two bugcheck special opcodes, ^XFEFF (BUGW) or ^XFDFF (BUGL, currently unused), it transfers control to EXE$BUG_CHECK. Use of the exception mechanism automatically changes access mode to kernel and allows code running in other than kernel mode to report bugchecks.

To determine what to do, EXE$BUG_CHECK looks at the PSL previous mode, that is, the access mode in which the exception occurred; the word or longword of bugcheck information that follows the bugcheck opcode; and several SYSBOOT parameters.

If the previous mode was kernel or exec, EXE$BUG_CHECK determines whether the bugcheck was fatal or continuable. If the bug severity is greater or equal to ERROR, the bugcheck is considered fatal. Also, if SYSBOOT parameter BUGCHECKFATAL is 1, all kernel and exec mode bugchecks become fatal. BUGCHECKFATAL is 0 in the V3 and V4 default parameters.

For non-fatal exec and kernel mode bugchecks, EXE$BUG_CHECK fills in an error log entry with bugcheck information and REIs.

For fatal exec and kernel mode bugchecks, EXE$BUG_CHECK creates an error log entry with bugcheck information and displays bugcheck information on the system console. It should write bugcheck information, error log message block buffers, and memory contents to the crashdump file.

The code which processes fatal bugchecks is not part of the resident executive. EXE$BUG_CHECK reads it from the system image into system space, over read-only non-paged executive, at global symbol BUG$FATAL. EXE$BUG_CHECK first initializes the system disk's adapter and then calls any unit initialization routine specified by the boot driver. If the unit initialization routine fails, EXE$BUG_CHECK sends a reboot message to the console and halts. If the unit initialization succeeds, EXE$BUG_CHECK calls the bootstrap driver to read the fatal bugcheck processing code. If the bootstrap driver gets a fatal I/O error or exceeds its retry count for recoverable errors, it returns an error status code.

If an error occurs, EXE$BUG_CHECK loops, re-initializing the adapter and attempting to read the fatal bugcheck code. Eventually, its I/O should succeed, and fatal bugcheck processing will continue.

After EXE$BUG_CHECK's fatal bugcheck overlay creates the bugcheck error log entry in the page of memory preceding BUG$FATAL and displays bugcheck information on the console, it tests the SYSBOOT parameter BUGREBOOT. If BUGREBOOT is 0, EXE$BUG_CHECK dispatches to code which causes an XDELTA breakpoint if XDELTA is resident. The user then has an opportunity to examine system data structures of interest. When the user types ";p" to terminate the breakpoint, control returns to EXE$BUG_CHECK. If SYSBOOT parameter DUMPBUG is 1, EXE$BUG_CHECK then attempts to write information to the crash dump file. DUMPBUG is 1 in the V3 and V4 default parameters.

After fatal bugcheck processing is complete, EXE$BUG_CHECK concludes either by halting or by looping endlessly to avoid automatic restart. If BUGREBOOT is 1, EXE$BUG_CHECK sends a reboot message to the console and halts. BUGREBOOT is 1 in the V3 and V4 default parameters. If BUGREBOOT is 0, EXE$BUG_CHECK prints the following message on the console terminal and loops.

SYSTEM SHUTDOWN COMPLETE - USE CONSOLE TO HALT THE SYSTEM

VMS V3 systems loop at PC 80007D3C. VMS V4 systems loop at PC 80008D7E.

SYSINIT Processing

When the system is first rebooted after a crash, SYSINIT uses the second and third blocks of the dump file to restore the error log message block buffers, so that error messages from the time of the crash will be written to the error log file by ERRFMT's normal processing.

By default SYSINIT enables the use of PAGEFILE.SYS (other than the first four blocks) as pagefile. As modified pages are written to PAGEFILE.SYS, any dump in it is overwritten. To prevent this when PAGEFILE.SYS is used as a dump file, set parameter SAVEDUMP to 1 and ensure that PAGEFILE.SYS is as big as physical memory (including any MA780 memory) plus 4 blocks plus at least 1000 blocks. SAVEDUMP is 0 in the V3 and V4 default parameters. After a crash run SDA from an account with CMKRNL privilege and access to the dump. Use SDA to copy the dump elsewhere to enable the pages of PAGEFILE.SYS occupied by the dump to be used as normal pagefile.

Hints And Kinks

1. Whenever you modify SYSBOOT parameters, remember to make AUTOGEN aware of your changes so that they propagate across AUTOGENs. Include any parameter changes you make in V3 SYS$SYSTEM:PARAMS.DAT or in V4 SYS$SYSTEM:MODPARAMS.DAT. See Chapter 11 in the Guide to VAX/VMS System Management and Daily Operations for further information on AUTOGEN.

Additional References

V3 VAX/VMS Internals and Data Structures Manual, Section 8.2, System Crashes

Guide to VAX/VMS System Management and Daily Operations, Section 3.4.3, on dumps in PAGEFILE.SYS

EXCEPTIONS

An exception is an unusual event encountered in the flow of instruction execution that alters that normal flow. Many exceptions are detected by the hardware (including microcode), but some types of exceptions are detected by software. Most exceptions that cause system crashes are detected by the hardware.

An exception may simply mean, for example, that an arithmetic operation overflowed, that a programming error resulted in an attempt to execute an illegal or invalid instruction, that a virtual page referenced needs to be read into memory, or that a program is requesting a system service.

Exceptions do not directly cause crashes. However, VMS assumes that certain kinds of exceptions in inner access modes (kernel and exec) mean that system integrity is compromised, that the system should be crashed or the current process deleted. A fairly large percentage of system crashes occur as the result of inner access mode exceptions. Most often the exception indicates that an earlier corruption or error has occurred.

In order to analyze these crashes, you must examine the stack footprints left by the exception to learn what exception occurred, and then examine the relevant code and data structures to infer what error(s) led up to the exception.

Software Exceptions

Software-detected exceptions are errors detected by software and signaled in such a way that they can be processed analogously to hardware exceptions. An example of a software exception is the condition AST fault. An AST fault means that an AST could not be delivered to a particular access mode in a particular process because its stack was invalid. Another example is the condition system service failure, which means that a system or RMS service completed with an error or a severe error.

In general, software exceptions are implemented to allow application level software to signal errors, although VMS makes some use of the mechanism. Most software exceptions occur in outer access modes and do not result in system crashes.

Hardware Exceptions

A hardware exception is synchronous with and caused by the execution of an instruction. Hardware exceptions include arithmetic overflow, access violation, translation not valid (known as pagefault), trace fault, and change mode traps.

When the processor detects an exception, it pushes on the stack the PC and PSL at which the exception occurred. It also pushes on the stack any exception dependent information, for example, the address whose attempted reference caused an access violation.

A hardware exception is either a fault, a trap, or an abort. The exception type depends on the individual exception; for example, an access violation exception is a fault. The distinction among these that is key to troubleshooting is the significance of the exception PC saved on the stack.

A trap is an exception that occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would have been executed had the exception not occurred. This means that you must examine the instruction before the exception PC to analyze the exception.

A fault is an exception that occurs during an instruction. The microcode leaves the registers and memory in a consistent state such that elimination of the fault condition and restart of the instruction will give correct results. The PC saved on the stack is the address of the faulting instruction. This means that you must examine the instruction at the exception PC to analyze the exception.

An abort is an exception that occurs during an instruction, leaving the registers and memory unpredictable, such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone. After an abort, the PC saved on the stack is the address of the aborted instruction. This means that you must examine the instruction at the exception PC to analyze the exception.

Exception Dispatching

After saving on the stack the PC, PSL, and any exception dependent information, the processor transfers control to the service routine specified in the System Control Block vector for that particular exception. Most of these exception service routines run in kernel mode.

Exceptions can be divided into two categories: ones which VMS will pass on to process-declared condition handlers and ones which VMS uses to perform its normal work (such as CHME and CHMK traps, pagefault).

The service routines for exceptions that are passed on to condition handlers are very simple and very similar. The service routines push more information on the stack: a system status code indicating what type of exception occurred (for example, SS$_ACCVIO) and a count of how many longwords of exception information are now on the stack. The exception information on the stack now comprises a signal argument list (also called a signal array). The exception service routines all converge to a common dispatching routine called

EXE$EXCEPTION.

EXE$EXCEPTION builds another argument list on the stack called a mechanism argument list (or mechanism array) and then checks whether this exception has occurred in a legal context; that is, it checks that the processor is currently not running on the interrupt stack and is running at an IPL no higher than two. If either of these checks fails, EXE$EXCEPTION signals the fatal bugcheck INVEXCEPTN. See section INVEXCEPTN BUGCHECK for a detailed stack layout.

If the checks pass, EXE$EXCEPTION builds one more argument list on the stack, the condition handler argument list, which contains the addresses of the signal and mechanism arrays. It then makes several more checks to prevent possible loops in exception servicing; for example, it checks whether the exception occurred calling a last chance handler or an AST procedure. If either of these checks fails, EXE$EXCEPTION signals a FATALEXCEPT bugcheck. See section FATALEXCEPT BUGCHECK for a detailed stack layout.

If the checks pass, EXE$EXCEPTION moves the three argument lists to the stack of the access mode that incurred the exception and REIs to that mode.

Executing in the access mode of the exception, EXE$EXCEPTION searches for a process-declared condition handler to handle the exception. It checks the primary and secondary exception vectors for that access mode. The primary and secondary exception vectors are in the first and second longword of CTL$AQ_EXCVEC, postindexed by access mode. EXE$EXCEPTION then traverses the the current stack, following nested call frames, looking for a call frame condition handler. The last place EXE$EXCEPTION looks is the last chance vector, CTL$AL_FINALEXC postindexed by access mode.

EXE$EXCEPTION calls any condition handler it finds with the condition handler argument list. A condition handler typically examines the signal array to decide whether it can handle that exception type.

If a condition handler returns a status indicating it cannot handle that particular exception type, EXE$EXCEPTION continues its search for a condition handler.

There are two common bugcheck types that are signaled by condition handlers: SSRVEXCEPT and UNXSIGNAL. SSRVEXCEPT is signaled by the default last chance handlers for kernel and exec mode. See section SSRVEXCEPT BUGCHECK for a detailed stack layout. UNXSIGNAL is signaled by call frame condition handlers used by several ACPs and the Files-11 XQP. See section UNXSIGNAL BUGCHECK for a detailed stack layout.

Some Common Exception Types

Some of the more common hardware exceptions that cause INVEXCEPTN,

SSRVEXCEPT, and UNXSIGNAL bugchecks are access violation fault, opcode reserved to customers and opcode reserved to Digital faults, reserved addressing mode fault, and reserved operand fault.

The subsections below describe each of these exceptions in slightly more detail with a layout of its signal array. The intent is to to show how the information in the documentation listed below can be applied to analyze these and other hardware exceptions which cause system crashes.

Access Violation Fault

An access violation fault means that an instruction has tried to reference a virtual address whose page table entry protection field prohibits that reference from that access mode. An access violation can also result from attempting access to an address beyond the range mapped by its respective page table; this is called a length violation.

The microcode pushes two longwords of exception dependent information on the stack: the address whose attempted reference caused the fault and a reason mask. The VAX architectures specifies that the faulting virtual address may be some other address in the same virtual page as the actual faulting operand address, but this rarely happens.

The signal array for this exception follows.

```
+-------------------+
|          00000005 | argument count
|          0000000C | SS$_ACCVIO signal type
|          0000000x | reason mask
|          xxxxxxxx | faulting virtual address
|          xxxxxxxx | exception PC
|          xxxxxxxx | exception PSL
+-------------------+
```

The reason mask longword contains 3 bits of information.

1. Bit 0 - the type of access violation

   0 means the PTE protection code prohibits the intended access
   1 means the reference was a length violation

2. Bit 1 - page table entry reference

   0 means the virtual address itself was not accessible
   1 means the PTE mapping the virtual address was not accessible

3. Bit 2 - intended access

   0 means the intended access was a read
   1 means the intended access was a write

First, examine the reason mask to determine whether this is a length violation, whether the PTE mapping the virtual address was not accessible, or whether the PTE's protection simply prohibited the intended access. A length violation occurs when the virtual page number of a P0 or System virtual address is greater than the contents of the P0 or System length register; a P1 space length violation occurs when the virtual page number is less than the contents of the P1 length register. Length violations are among the more common kinds of access violation and often easy to spot because the faulting virtual address looks "strange". Incorrect references to location 0, or any address in virtual page 0, are another common cause of access violations. Inaccessible PTEs are less common.

Examine the instruction that incurred the fault and its operands to determine which operand reference caused the fault.

Many crash-causing access violations result from software errors. Software errors that can cause access violations include:

1. use of a corrupted pointer in one data structure to reference another data structure
2. use of a corrupted register as a pointer to a data structure
3. use of an invalid input argument, such as size, in an address computation
4. corruption of code in memory (or code on disk)
5. erroneous transfer of control into the middle of random data or code
6. corruption in a page table resulting in incorrect protection information.

An access violation can also be signaled by VMS memory management code; that is, this exception is not always detected by hardware. The pagefault exception service routine, MMG$PAGEFAULT, may signal an access violation if a process incurs a pagefault for a page in another process's process header. Although a process header is in system space, not per-process space, it is paged in the working set list of the process whose header it is. The system cannot allow one process to fault a page which belongs to another process.

You can check whether the access violation might be an attempt to touch another process's process header by comparing the faulting virtual address to the address range reserved for the balance set slots. Type the following SDA commands.

```
SDA> DEF BALBASE = @SWP$GL_BALBASE       !define symbol
SDA> EVAL BALBASE                        !start address
SDA> EVAL BALBASE + (@SGN$GL_BALSETCT*@SWP$GL_BSLOTSZ*200)
SDA>                                     !end address
```

If the address is not within that range, then it is a hardware detected access violation. If the address is within that range, then see whether the address is within the process's own header by typing the following SDA commands.

```
SDA> SHOW PROCESS                       !PHD start address
SDA> EVAL <phd> + (@SWP$GL_BSLOTSZ*200)
SDA>                                    !end address
```

If the address is within the process's own header, then most likely this is a hardware detected access violation.

Reserved Opcode Faults

An opcode reserved to Digital fault (SS$_OPCDEC) means that an attempt was made to execute an undefined opcode or, from an outer mode, an instruction which requires the process to be in kernel mode. Examples of instructions that may only be executed in kernel mode are SVPCTX and MTPR.

An opcode reserved to customer fault (SS$_OPCCUS) means that an attempt was made to execute an instruction starting with the hex opcode FC.

For either of these faults, no extra exception information is pushed on the stack.

The signal array for these exceptions follows.

```
+-------------------+
|          00000003 | argument count
|0000043C/00000434  | SS$_OPCDEC/SS$_OPCCUS signal type
|          xxxxxxxx | exception PC
|          xxxxxxxx | exception PSL
+-------------------+
```

Many crash-causing reserved operand faults result from software errors. Software errors that can cause reserved opcode faults include corruption of code in memory (or code on disk) and erroneous transfer of control into the middle of random data or code.

Reserved Addressing Mode Fault

A reserved addressing mode fault means that an instruction contains an operand specifier for an addressing mode that is not allowed in the context in which it occurs. No extra exception information is pushed on the stack. An example of a reserved addressing mode is the use of a short literal as a destination operand. See the VAX-11 Architecture Reference Manual or System Reference Manual, section 6.4.3, for a list of illegal addressing modes.

The signal array for this exception follows.

```
+--------------------+
|          00000003 | argument count
|          0000044C | SS$_RADRMOD signal type
|          xxxxxxxx | exception PC
|          xxxxxxxx | exception PSL
+--------------------+
```

Many crash-causing reserved operand faults result from software errors. Software errors that can cause reserved addressing mode faults include corruption of code in memory (or code on disk) and erroneous transfer of control into the middle of random data or code.

Reserved Operand Exception

A reserved operand exception means that an attempt was made to execute an instruction with an operand that has an invalid format. One example of a reserved operand fault is a CALLS/G to a procedure with an invalid entry mask. Another example is an attempt to REI with an invalid saved PSL. Whether the exception is an abort or a fault depends upon the cause. See the VAX-11 Architecture Reference Manual or System Reference Manual, section 6.4.3, for a list of causes of reserved operand exceptions and their exception types. No extra exception information is pushed on the stack.

The signal array for this exception follows.

```
+--------------------+
|          00000003 | argument count
|          00000454 | SS$_ROPRAND, signal type
|          xxxxxxxx | exception PC
|          xxxxxxxx | exception PSL
+--------------------+
```

Examine the instruction that incurred the fault and its operands to determine which operand reference caused the fault.

If the exception PC is the address of an REI instruction, then the two longwords on the stack at higher addresses than the signal array should be a PC-PSL pair to be restored with the REI. The REI microcode makes numerous integrity checks on the saved PSL before restoring it. (See the VAX-11 Architecture Reference Manal or System Reference Manual, section 6.9, for a list of these checks.) Examine the PSL to see which test failed and to evaluate whether the PSL has been corrupted.

Many crash-causing reserved operand faults result from software errors. Possible software errors that result in reserved operand faults include corruption of code in memory (or code on disk) and erroneous transfer of control into the middle of random data or code.

For REI reserved operand faults some more specific possibilities are:

1. stack corruption overwriting the PSL
2. incorrect stack usage popping too many or too few longwords prior to an REI
3. attempts to run a compatibility mode image on a MicroVAX
4. incorrect lowering of IPL in an interrupt service routine or system service.

Additional References

V3 VAX/VMS Internals and Data Structure Manual, Chapter 4, Condition Handling, for details of VMS exception dispatching

VAX/VMS Run-Time Library Routines Reference Manual, Chapter 7, Condition Handling Procedures, for information on writing condition handlers and using related RTL routines and condition handlers

VAX/VMS System Services Reference Manual, Chapter 10, Condition-Handling Services, for a list of hardware detected and VMS-signaled software exceptions and their exception-dependent information and use of system services related to condition handling

VAX-11 Architecture Reference Manual or System Reference Manual, Chapter 6, Exceptions and Interrupts, for the architectural definition of hardware detected exceptions and the details of interrupt/exception initiation and the REI instruction

Introduction to VAX/VMS System Routines, Chapter 2, VAX Procedure Calling and Condition Handling Standard, for background on the goals of the condition handling design

FATALEXCPT BUGCHECK

The FATALEXCPT bugcheck is signaled by the common exception dispatching code when it is unable to dispatch to a condition handler for a kernel or exec mode exception. In kernel mode, this bugcheck is fatal. In exec mode, this bugcheck is fatal only if the SYSBOOT parameter BUGCHECKFATAL is 1; by default, BUGCHECKFATAL is 0.

There are several sets of circumstances under which the common exception dispatching code signals this bugcheck.

1. EXE$ASTDEL, the AST delivery code, incurs an exception trying to call a kernel or exec mode AST procedure, and there is no last chance handler declared for that access mode.

2. The common exception dispatching code incurs an exception trying to call the kernel or exec mode last chance handler.

3. The common exception dispatching code is unable to copy the condition handler argument list, signal and mechanism arrays to the exec mode stack using the current exec mode stack pointer, and there is no exec mode last chance handler.

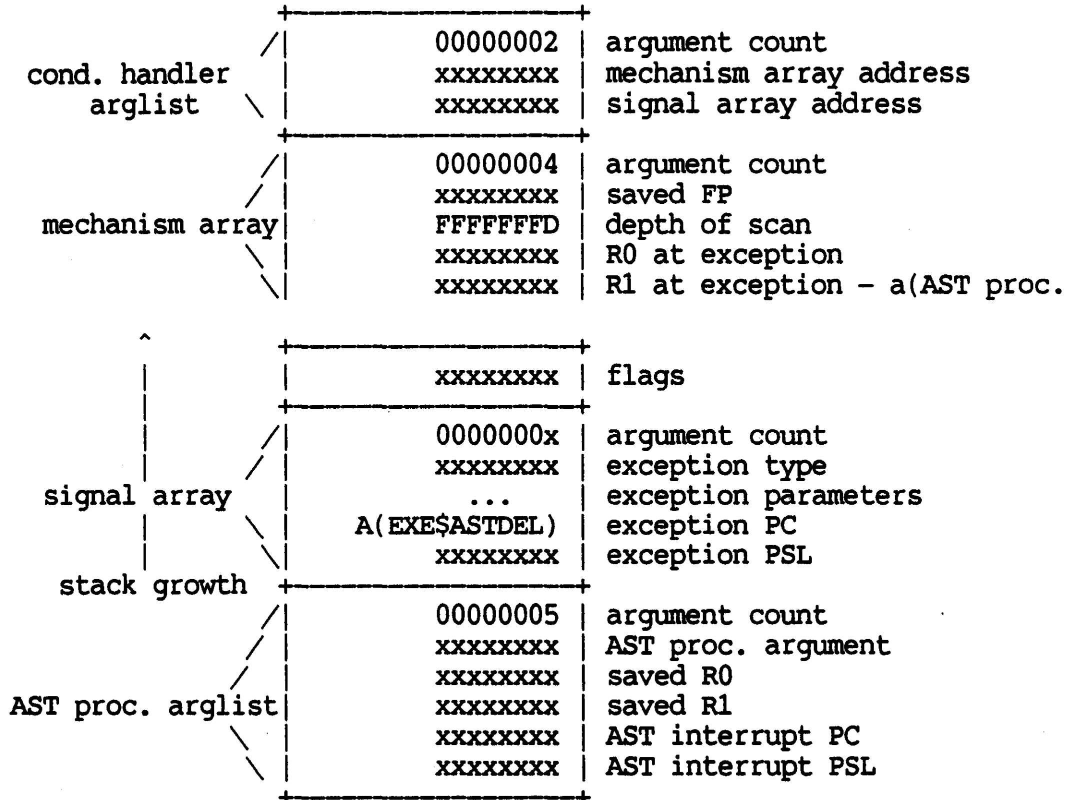
4. A kernel or exec mode exception occurs, and no condition handler for that access mode handles that conditon.

In practice, these circumstances are rare. VMS always declares last chance condition handlers for kernel and exec mode. Only inner access mode code can override those declarations or overwrite the P1 space locations which contain the last chance handler addresses. The address of the kernel mode last chance handler is stored in CTL$AL_FINALEXC; the address of the exec mode handler, in CTL$AL_FINALEXC+4.

The stack layout varies, depending on which set of circumstances triggered the FATALEXCPT bugcheck, although, in all cases, there should be at least one signal and mechanism array on the current stack visible among the newer stack longwords (i.e., lower addresses).

Select the stack layout that matches your crash from among the following ones. The EXCEPTION PC in the (newer) signal array on your stack is a good clue for most of them. Follow the directions in the text associated with the appropriate stack layout.

1. EXE$ASTDEL Exception



If the exception PC is the address EXE$ASTDEL, then an exception occurred at the call to a kernel or exec mode AST procedure, and there was no last chance handler for that mode. This means that there are at least two anomalies to be explained:

- o the exception calling the AST procedure;
- o zero contents in CTL$AL_FINALEXC (kernel mode) or CTL$AL_FINALEXC+4 (exec mode).

a. Locate the mechanism array. Saved R0 and saved R1 are the registers' values at the time the exception occurred. Saved R1 is the address of the AST procedure EXE$ASTDEL tried to call.

b. Skip 1 longword, the flags longword.

c. The next longword, the beginning of the signal array, contains an argument count, the number of longwords that follow. Use the count to identify all entries in the signal array. The number of exception parameters present is a function of exception type and can be 0, 1, or 2 longwords.

d. The exception type is a status value, e.g., C (hex) or SS$_ACCVIO. The DCL command

```
$ EXIT %X<exception_type>
```

writes the message text associated with the exception type status value. The V4 SDA command

```
SDA> EVAL/CONDITION <exception_type>
```

writes the message text associated with the exception type status value.

Typically, the exception is one generated by "hardware" (or microcode), for example, access violation. "Hardware" generated exceptions are listed with a description of their associated exception parameters in Section 10.1 of the VAX/VMS System Services Reference Manual. See section EXCEPTIONS for information about the more common hardware exceptions.

e. The exception PC in the signal array is the instruction whose [attempted] execution resulted in the unexpected exec or kernel mode exception. In this case, the instruction at EXE$ASTDEL is a CALLG (SP),(R1).

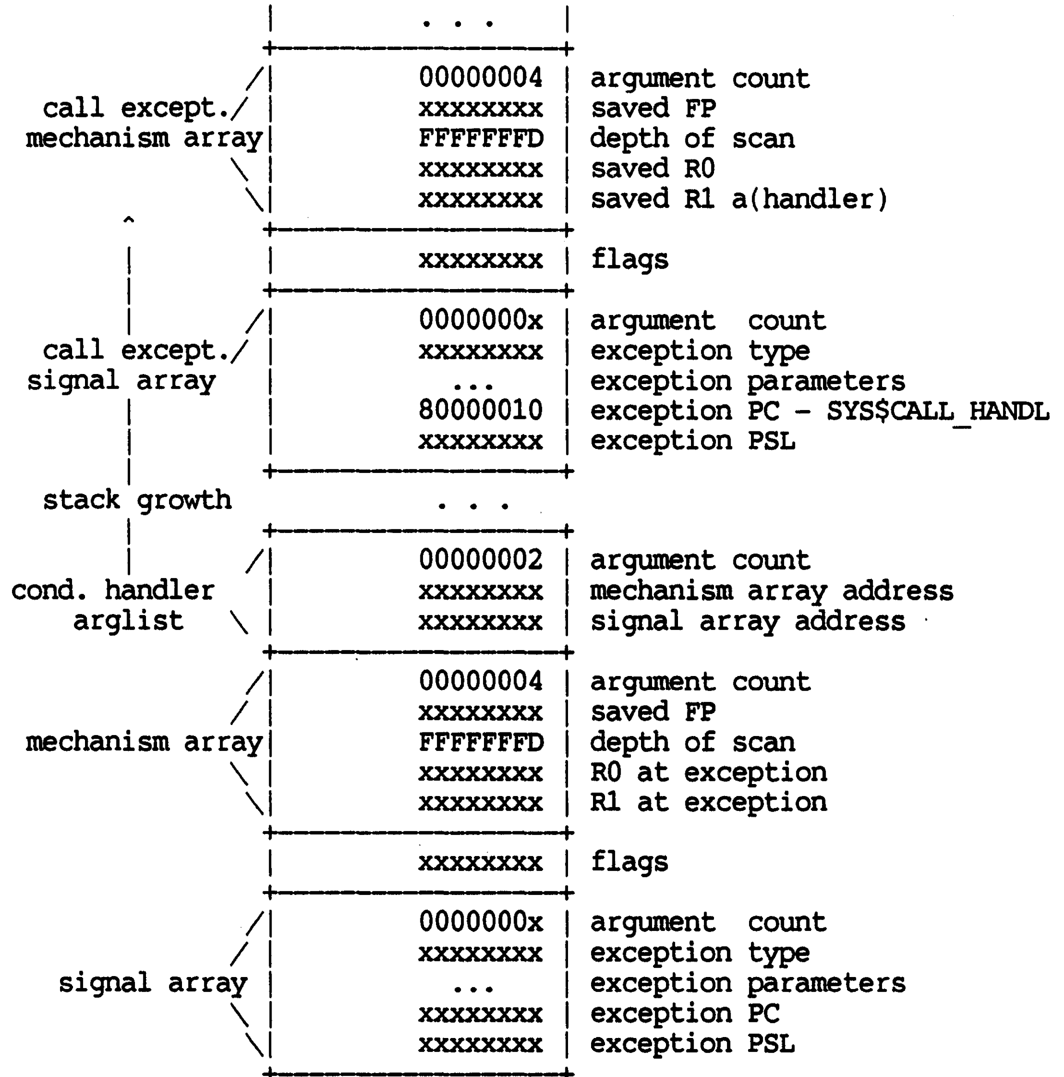
Figure out why the CALLG generated an exception. Use saved R1 in the mechanism array: determine whether it points to a valid AST procedure, whether that address exists and has suitable protection, etc.

f. The argument list built by AST delivery code contains the PC and PSL that describe the thread of execution interrupted by AST delivery and the contents of R0 and R1 at the time of the interrupt. These may be important in explaining both anomalies, the exception at EXE$ASTDEL and the clearing of the last chance handler address.

g. Examine P1 space around CTL$AL_FINALEXC, comparing it to that of other processes, to determine if there is any other corruption. If not, it is more likely that the current image issued a $SETEXV system service request from an inner mode to clear the handler address.

h. Decipher anything earlier on the current stack to trace control flow, in case there are clues about what led to the current situation. See section STACK PATTERNS.

2. SYS$CALL_HANDL Exception



If the exception PC is the addresss SYS$CALL_HANDL, then an exception occurred at the call to a kernel or exec mode last chance handler. This means that there are at least two anomalies to be explained:

- o the exception calling the last chance handler;
- o the original exception.

a. Locate the newer mechanism array. Saved R0 and saved R1 are the registers' values at the time the exception occurred. Saved R1 is the address of the last chance handler SYS$CALL_HANDL tried to call.

b. Skip 1 longword, the flags longword.

c. The next longword, the beginning of the signal array, contains an argument count, the number of longwords that follow. Use the count to identify all entries in the signal array. The number of exception parameters present is a function of exception type and can be 0, 1, or 2 longwords.

d. The exception type is a status value, e.g., C (hex) or SS$_ACCVIO. The DCL command

```
$ EXIT %X<exception_type>
```

writes the message text associated with the exception type status value. The V4 SDA command

```
SDA> EVAL/CONDITION <exception_type>
```

writes the message text associated with the exception type status value.

Typically, the exception is one generated by "hardware" (or microcode), for example, access violation. "Hardware" generated exceptions are listed with a description of their associated exception parameters in Section 10.1 of the VAX/VMS System Services Reference Manual. See section EXCEPTIONS for information about the more common hardware exceptions.

e. The exception PC in the signal array is the instruction whose [attempted] execution resulted in the unexpected exec or kernel mode exception. In this case, the instruction at SYS$CALL_HANDL is a CALLG 4(SP),(R1).

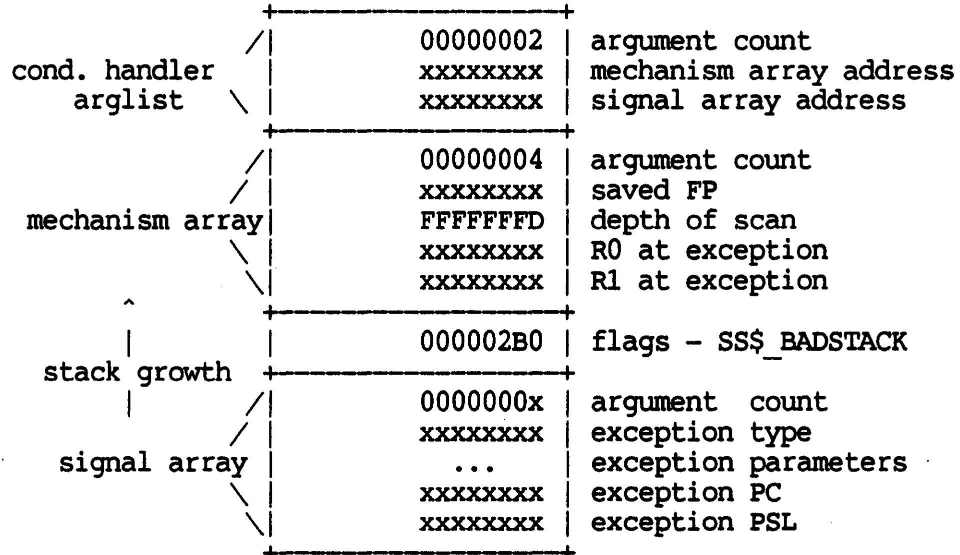
Figure out why the CALLG generated an exception. Use saved R1 in the mechanism array: determine whether it points to a valid last chance procedure, whether that address exists and has suitable protection, etc. Compare saved R1 to the contents of CTL$AL_FINALEXC (kernel mode) or CTL$AL_FINALEXC + 4 (exec mode); they should be the same.

f. Examine P1 space around CTL$AL_FINALEXC, comparing it to that of other processes, to determine if there is any other corruption. If not, it is more likely that the current image issued a $SETEXV system service request from an inner mode with an invalid handler address.

g. The older signal and mechanism arrays describe the original exception, for which the common exception dispatching code was trying to locate a handler. Analyze that exception to see whether there might be a common cause for both exceptions.

h. Decipher anything earlier on the current stack to trace control flow, in case there are clues about what led to the current situation. See section STACK PATTERNS.

3. Stack Problem



If the exception PC is neither EXE$ASTDEL nor SYS$CALL_HANDL and the bugcheck stack is the exec stack, then the common exception dispatching code was unable to copy information to the exec stack to dispatch to a condition handler. It therefore reset the exec mode stack pointer, recreated the stack address space if necessary, and copied the exception information to the stack, before it signaled the FATALEXCPT bugcheck.

a. The PC displayed by the SDA SHOW CRASH command reflects the common exception dispatching code rather than the location of the exception(s). R0 and R1 in the SHOW CRASH display have been altered by the exception dispatching code. The PC, R0, and R1 at the time of the exception(s) can be obtained as described below.

b. Locate the mechanism array. Saved R0 and saved R1 are the registers' values at the time the exception occurred.

c. Skip 1 longword, the flags longword, which should contain SS$_BADSTACK.

d. The next longword, the beginning of the signal array, contains an argument count, the number of longwords that follow. Use the count to identify all entries in the signal array. The number of exception parameters present is a function of exception type and can be 0, 1, or 2 longwords.

e. The exception type is a status value, e.g., C (hex) or SS$_ACCVIO. The DCL command

```
$ EXIT %X<exception_type>
```

writes the message text associated with the exception type status value. The V4 SDA command

```
SDA> EVAL/CONDITION <exception_type>
```

writes the message text associated with the exception type status value.

Typically, the exception is one generated by "hardware" (or microcode), for example, access violation. "Hardware" generated exceptions are listed with a description of their associated exception parameters in Section 10.1 of the VAX/VMS System Services Reference Manual. See section EXCEPTIONS for information about the more common hardware exceptions.

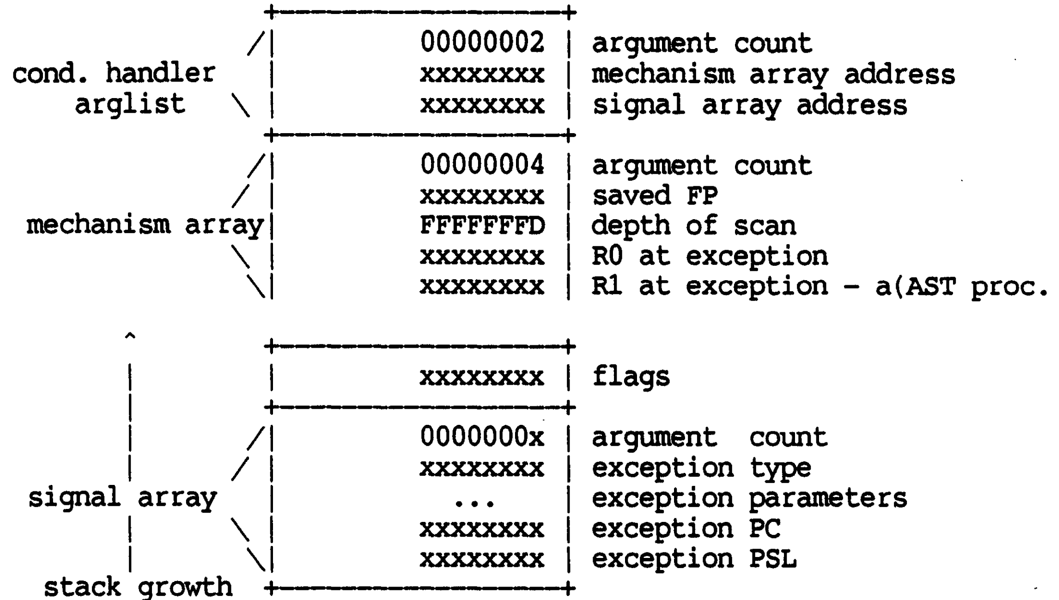
f. The PC in the signal array is the instruction whose [attempted] execution resulted in the unexpected exec or kernel mode exception. Whether the PC points to the beginning of the instruction or the end depends on whether the exception was a trap (end), fault (beginning), or abort (beginning). The reference above specifies whether each exception is a trap, fault, or abort. Identify in what source module the PC is. See section VIRTUAL ADDRESSES. Often examining instructions around the PC is helpful enough to eliminate a microfiche search. Try the SDA command

```
SDA> EXAMINE/INSTRUCTION <exception_pc>-20;30
```

Figure out why the instruction generated an exception. For example, if an access violation occurred, look at the operands to see which access was in error.

g. In this particular case, all stack footprints are gone. The exception PC and other register and data structure contents are the only clues you have as to what the thread of execution was doing.

4. No Condition Handler



. . .

If none of the previous cases applies, then the common dispatching code signaled this bugcheck because it was unable to find any condition handler to dispatch. This means that there are at least two anomlies to be explained:

- the exception;
- zero contents, for kernel mode, in the longwords at CTL$AQ_EXCVEC and CTL$AL_FINALEXC or, for exec mode, in the longwords at CTL$AQ_EXCVEC+8 and CTL$AL_FINALEXC+4.

a. The PC displayed by the SDA SHOW CRASH command reflects the common exception dispatching code rather than the location of the exception(s). R0 and R1 in the SHOW CRASH display have been altered by the exception dispatching code. The PC, R0, and R1 at the time of the exception(s) can be obtained as described below.

b. Locate the mechanism array. Saved R0 and Saved R1 are the registers' values at the time the exception occurred.

c. Skip 1 longword, the flags longword.

d. The next longword, the beginning of the signal array, contains an argument count, the number of longwords that follow. Use the count to identify all entries in the signal array. The number of exception parameters present is a function of exception type and can be 0, 1, or 2 longwords.

e. The exception type is a status value, e.g., C (hex) or SS$_ACCVIO. The DCL command

```
$ EXIT %X<exception_type>
```

writes the message text associated with the exception type status value. The V4 SDA command

```
SDA> EVAL/CONDITION <exception_type>
```

writes the message text associated with the exception type status value.

Typically, the exception is one generated by "hardware" (or microcode), for example, access violation. "Hardware" generated exceptions are listed with a description of their associated exception parameters in Section 10.1 of the VAX/VMS System Services Reference Manual. See section EXCEPTIONS for information about the more common hardware exceptions.

f. The PC in the signal array is the instruction whose [attempted] execution resulted in the unexpected exec or kernel mode exception. Whether the PC points to the beginning of the instruction or the end depends on whether the exception was a trap (end), fault (beginning), or abort (beginning). The reference above specifies whether each exception is a trap, fault, or abort. Identify in what source module the PC is. See section VIRTUAL ADDRESSES. Often examining instructions around the PC is helpful enough to eliminate a microfiche search. Try the SDA command

```
SDA> EXAMINE/INSTRUCTION <exception_pc>-20;30
```

Figure out why the instruction generated an exception. For example, if an access violation occurred, look at the operands to see which access was in error.

Hints And Kinks

1. The FATALEXCPT bugcheck may also be signaled by software other than the VMS exec. In particular, the REMACP signals this bugcheck fatally from its kernel-mode condition handler if any unexpected exceptions occur. Under those circumstances the AP register should point to a condition handler argument list containing the addresses of the signal and mechanism arrays, and the newer stack should resemble that in section SSRVEXCEPT BUGCHECK.

2. Not all access violations are signaled by microcode. The pagefault exception service routine, MMG$PAGEFAULT, may signal an access violation if a process incurs a pagefault for a page in

another process's process header.

3. Note that for each V3 SDA COPY command used to copy the dump, the SP will be 8 bytes greater than its actual value; that is, SDA will show the SP pointing to a stack address 8 bytes higher than it should. This V3 bug has been corrected in V4.

4. The VAX instruction set is sufficiently rich that most random data can be interpreted as instructions. Most system code deals with binary integer and character data. This means that if an EXAMINE/INSTRUCTION display includes many packed decimal and/or floating point instructions, you are probably examining a data area or using a start address which is not an instruction boundary.

   One common error that results in a nonsensical display is to examine instructions in the bugcheck overlay area. During a crash, fatal bugcheck code and message text overlay resident system image code, beginning one page before label BUG$FATAL, for a length of about 12000 decimal or 3000 hex bytes.

Additional References

V3 VAX/VMS Internals and Data Structure Manual, Chapter 4, for general exception dispatching and details of exceptions signaled by VMS system software

VAX Architure Standard (DEC Standard 032) or VAX-11 Architecture Reference Manual, Chapter 6, Exceptions and Interrupts

VAX/VMS System Services Reference Manual, Chapter 10, Condition-Handling Services

FORCED CRASHES

Forced crashes cause INVEXCEPTN bugchecks. If the signal array shows the PC as approximately FFFFFFFF and the PSL as kernel mode and IPL 31, the system was probably crashed through the console CRASH procedure, as documented in the Guide to VAX/VMS System Management and Daily Operations Section 4.1. On a VAX-11/780, VAX-11/785, and MicroVAX II, the faulting virtual address from a forced crash is FFFFFFFC; on a VAX-11/730, VAX-11/750, MicroVAX I, and VAX 8600, the faulting virtual address from a forced crash is FFFFFFFF. This sequence of commands writes to the console terminal the PC, PSL, and the 5 stack pointer registers and then deposits into the PC and PSL to cause a crash. The PC is loaded with FFFFFFFF, a nonexistent address, and the PSL is set to IPL 31 and kernel mode. When the processor is continued, attempted execution at location FFFFFFFF causes an access violation. With the processor running at IPL 31, the access violation causes an INVEXCEPTN bugcheck.

There are two important differences between forced crashes and other crashes. The first difference is that some human decided to crash the system. Thus, it is important to find out why s/he crashed the system, what s/he thought was wrong. The second difference is that the CRASH procedure alters control flow and possibly access mode and stack. Thus, the values of the processor registers written to the console terminal are critical in determining what the system was doing prior to the crash.

Whether or not it is important to know what the system was doing prior to the crash depends on why the system was crashed. For example, some crashes are forced to record a scheduler data base or memory management data base believed to be corrupted. In such a case, you should ignore the directions below, since examining the stack may not be very useful; instead, examine the data structures having to do with the reason for the forced crash. If the system was crashed because it was hung, see sections HANGS and RESOURCE WAITS for hints on what to look for in the dump. Some crashes are forced because the system is believed to be looping at high IPL. In such a case, examining the stack is important.

1. Read the PSL in the console terminal output from the CRASH procedure (written by one of the following console commands: VAX-11/780 and VAX-11/785 EXAMINE PSL; VAX-11/750 and MicroVAX I E P; VAX-11/730 and MicroVAX II E PSL). It specifies what IPL and what access mode the processor was in prior to the crash, and whether the system was running on the interrupt stack. Decode the PSL using the layout in the section RELATED REFERENCE MATERIAL or with the V4 SDA command EXAMINE/PSL.

2. Select a number between 0 and 4 using the decoded PSL access mode and interrupt stack (IS) fields as follows

- o if IS is 1, the number is 4
- o if IS is 0, the number is the access mode (e.g., kernel is 0, exec is 1, etc.).

This is the number of the processor register which records the stack pointer current at the time of the console halt.

3. Read the console terminal output from the CRASH procedure and locate the display of the processor register whose number you selected in the previous step. Read its value. These process registers are written to the console terminal by one of the following console commands depending on cpu type.

   - o VAX-11/780, VAX-11/785 - EXAMINE/INTERN/NEXT:4 0
   - o VAX-11/750, MicroVAX I, MicroVAX II - E/I 0, E/I 1, E/I 2, E/I 3, E/I 4
   - o VAX-11/725, VAX-11/730 - E/I/N:4 0

   The value displayed is the lowest end of that stack, the address of the newest valid stack contents.

4. Invoke SDA and determine the high end of that stack. If the stack was the interrupt stack, type the following command to determine its high end (that is, the initial address loaded into the stack pointer register)

   ```
   SDA> EXAMINE EXE$GL_INTSTK
   ```

   If the stack was not the interrupt or kernel stack, then type the following command to determine its high end

   ```
   SDA> EXAMINE (4 * <access_mode_number>) + CTL$AL_STACK
   ```

   If the access mode at the time the crash was forced was kernel, see the section KERNEL STACK LOCATIONS to determine the high (oldest) limit of the kernel stack.

5. Display the stack just prior to the crash by using the values you determined above. SDA writes this range in "stack" format, with attempted symbolic interpretation, in response to the command

   ```
   SDA> SHOW STACK <low_address>:<high_address>
   ```

6. Read the PC in the console terminal output from the CRASH procedure (written in response to HALTing a 780 and 785, typing CTRL/P on a 750 and 730, or depressing the halt button on a MicroVAX I and II). value is the address of the instruction that was about to be executed. If appropriate, identify in what source module the PC is. See the section VIRTUAL ADDRESSES. Often examining instructions around the PC is helpful enough to eliminate a microfiche search. Try the SDA command

   ```
   SDA> EXAMINE/INSTRUCTION <halt_pc>-20;30
   ```

7. If appropriate, decipher the stack to trace control flow. See the section STACK PATTERNS. If the access mode just prior to the execution of the crash procedure was kernel, you can ignore the signal and mechanism arrays from the access violation and any stack contents newer than they are, that is, at lower addresses.

Hints And Kinks

1. The VAX instruction set is sufficiently rich that most random data can be interpreted as instructions. Most system code deals with binary integer and character data. This means that if an EXAMINE/INSTRUCTION display includes many packed decimal and/or floating point instructions, you are probably examining a data area or using a start address which is not an instruction boundary.

   One common error that results in a nonsensical display is to examine instructions in the bugcheck overlay area. During a crash, fatal bugcheck code and message text overlay resident system image code, beginning one page before label BUG$FATAL, for a length of about 12000 decimal or 3000 hex bytes.

HALTS - VAX-11/780 AND VAX-11/785

VAX-11/780s and VAX-11/785s halt in response to halt instructions, console HALT commands, and various error conditions. The VAX-11/780 and VAX-11/785 halt behaviors are identical, except that the VAX-11/785 has one unique error halt, clock phase error.

The error conditions that cause halts are severe enough to interfere with the normal exception/interrupt mechanism; for example, if the interrupt stack is invalid, the cpu cannot write a microcode machine check error logout on the stack. The LSI-11 console software periodically polls the state of the VAX cpu, testing to see if the cpu has halted, and has access to information about the halt PC, PSL, reason for the halt, and the setting of the auto restart switch.

If auto restart is disabled, the console prompts, leaving the VAX cpu halted, and accepts commands if the cpu key is in the LOCAL ENABLE position. If auto restart is enabled, the console restarts VMS, using the floppy command procedure RESTAR.CMD. Following a powerfail recovery, the console reloads writable control store on the VAX cpu and, if auto restart is enabled, executes RESTAR.CMD, which passes control to the instruction-level ROM (ISP ROM) in the memory controller. The ISP ROM passes control to a restart routine in VMS.

Restarting VMS for any reason other than power fail recovery causes a crash. The system is crashed to preserve pending error log messages and to provide information that might be useful in troubleshooting the halt. As a result of the VAX-11/780 and VAX-11/785 restart mechanism, these crashdumps do not contain the contents of R0 - R5, R10, R11, AP, FP, and SP at the time of the halt. See below the subsection VAX-11/780 and VAX-11/785 Restart Mechanism for further details. If analyzing a particular halt's dump is recommended in the subsections below, see section RESTART BUGCHECKS for any additional crashdump analysis suggestions.

Many of these halts are caused by hardware problems. Unfortunately, the default restart mechanism sometimes provides insufficient hardware status. The subsection Editing RESTAR.CMD recommends editing RESTAR.CMD to display various internal registers at the time of a halt. Each of the subsections below describing a particular halt indicates which internal registers are of interest and how to display them. If you have edited RESTAR.CMD on the halting system, the information will be displayed automatically. Otherwise, if the system is still halted, enter the recommended commands to display that information. If the system has already restarted, it is too late to obtain any desirable additional information.

For all of these halt conditions, carefully examine the system errorlog file, SYS$ERRORLOG:ERRLOG.SYS for any errors or anomalies that occurred before or at the time of the halt that might be associated with the halt or provide a clue about possible hardware problems.

Likely Halt Indications

The console control panel is at the top right of the CPU cabinet. When the VAX cpu is halted, normally the red leftmost light, labeled ATTN, is lit; the green RUN light is not lit; and the green POWER light is lit. See below subsection Pathological Halts for other possibilities.

Normally the LSI-11 console software prints a message on the console terminal indicating the nature of the halt and the PC at the halt. In the case of a power failure, the message is printed after power is restored.

Console Halt Message	Meaning
HALTED AT xxxxxxxx	xxxxxxxx is updated PC at halt; see below
HALT INST EXECUTED	Halt instruction executed in kernel mode
?CHM ERR	CHMx instruction executed on the interrupt stack
?CLOCK PHASE ERROR	VAX-11/785 cpu and SBI clocks out of phase
?CPU DBLE-ERR HLT	Machine check occurred during machine check servicing
?ILL I/E VEC	Illegal value in low 2 bits of SCB vector
?INT-STK INVALID	ISP points to invalid page or one without write access
?NO USR WCS	Attempt to jump to nonexistent user WCS

If there are unexpected (i.e., not the result of someone's typing CTRL/P) LSI-11 console software prompts (>>>) without any halt messages, then there may be a problem in the console interface board, cpu power supplies, or the LSI-11. Contact Field Service.

If you see an LSI-11 MicroODT prompt (@), see below the subsection Pathological Halts.

If the system seems to be halted but there is no message, see below the subsection Pathological Halts.

The subsections below contain more information about each halt message listed above.

HALTED AT Xxxxxxxx

Normally this message follows a message describing the reason for the halt. If a user types CTRL/P and HALT on the console terminal, this message is printed with no other message. xxxxxxxx is normally the PC at which the cpu was halted. For example, xxxxxxxx is the address of a halt instruction plus 1; xxxxxxxx is the address of a CHMx opcode the system tried to execute while on the interrupt stack; xxxxxxxx is the offset of an SCB vector containing illegal values for the low two bits.

When xxxxxxxx is an address (rather than an SCB vector offset), the address is a physical address if memory management is disabled or a virtual address if memory management is enabled.

Conceivably, this message may appear without any other message and any user intervention. This could be due to intermittent very brief power problems that don't cause a power fail sequence or due to problems in the console interface board (CIB).

After a power fail and recovery, the cpu is halted at physical location 0. This halt can usually be ignored. This message is printed along with other text

```
CPU HALTED,SOMM CLEAR,....
RAD=HEX,ADD=PHYS,...
INIT SEQ DONE
HALTED AT 00000000

(RELOADING WCS)
...
```

A power failure recovery on a system without battery backup and with auto restart enabled should result in another halt from the ISP ROM and a reboot from the default system disk. The ISP ROM checks whether memory contents are valid. Without battery backup, the memory contents are not valid after a power failure. The ISP ROM for the MS780-C memory controllers halts at 2000350A if memory is invalid. The newer ISP ROM for the MS780-E memory controllers halts at 20003563 if memory is invalid. These ISP ROM addresses are physical addresses. The system also exhibits that behavior if the battery backup is faulty or doesn't have enough charge to power the memory for the duration of the power failure.

If you see these messages without a true electrical failure (e.g., the room lights are still on), then there may be a problem in the VAX or LSI-11 power supplies.

HALT INST EXECUTED

This halt usually means that some kernel mode code halted. (The HALT instruction can only be executed from kernel mode.) The PC following the HALT instruction is displayed in the console's "HALTED AT xxxxxxxx" message. If memory management is enabled (normal state while VMS is running), xxxxxxxx is a virtual address; if memory management is disabled, xxxxxxxx is a physical address. The cpu may execute a byte of 0 as a HALT instruction following corruption of code or erroneous dispatch into random code or data. VMS contains various HALTs that are executed in extreme circumstances where no recovery is possible.

One such extreme circumstance is a failure in fatal bugcheck processing. If EXE$BUG_CHECK cannot initialize the system disk or

finds the boot control block corrupted, EXE$BUG_CHECK writes a reboot message to the console and halts. When the console sees a reboot message and a halt, it reboots VMS using DEFBOO.CMD, regardless of the setting of the auto restart switch. The following console messages are printed. 8xxxxxxx is an address within the console terminal driver in SYSLOA780.EXE.

```
HALT INST EXECUTED
HALTED AT 8xxxxxxx

(BOOTING)
CPU HALTED
INIT SEQUENCE DONE
...
```

Note that this message is also printed as the result of bootstrap operations (the ISP ROM for the MS780-C memory controllers halts at physical 200034F9; the ISP ROM for the newer MS780-E memory controllers halts at physical 20003552), normal VMS shutdown operations with reboot requested, and, if SYSBOOT parameter BUGREBOOT is 1, fatal bugcheck processing. You should ignore it under those circumstances.

If this halt is the result of a software error or a deliberately executed HALT instruction, analyzing the dump is the best way to troubleshoot the problem. You should analyze the dump to rule out a software caused problem before contacting Field Service.

Possible hardware causes of this halt include problems in the memory controller(s), instruction buffer, datapath, adapters (particularly in their map registers), and, less frequently, cache.

If you have already edited RESTAR.CMD, look at the output in response to the examine commands below. If you have not edited RESTAR.CMD, but the system is still halted, type the following commands.

```
>>>E IR             !examine opcode just executed
>>>E/N:E R0         !examine R0 - SP
>>>E/ID/N:3F 0      !dump the ID registers
>>>E PC             !PC points 1 byte past opcode
>>>E/V @            !examine @PC
>>>E/V -            !re-examine previous longword
>>>D/ID 1D 18000    !turn off cache
>>>E PC
>>>E/V @            !examine @PC again
>>>E/V -            !examine previous longword
>>>DEP AP 6         !set code for halt executed
>>>@RESTAR.CMD      !invoke normal restart command procedure
```

If the displayed contents of the memory at PC-1 change after you turned off cache, then probably cache is at fault. If the byte at PC-1 is not a 0 (halt opcode), then you may have a problem in either the IDP or the IRC board. Examining IR results in a display of three numbers, the first of which is the opcode just executed. If this is a 0, but the byte at PC-1 is not a 0, there may be faulty shifting in

the instruction buffer or an intermittent addressing problem during instruction decode. In any of these cases, contact Field Service.

?CHM ERR

This halt means that while the system was running on the interrupt stack, an attempt was made to execute one of the change mode instructions (CHMU, CHMS, CHME, or CHMK). The PC of the CHMx instruction is displayed in the console's "HALTED AT xxxxxxxx" message.

This halt might occur as the result of software error; for example, some process context code's executing in system context, a user-written driver's erroneously requesting system services while executing on the interrupt stack, erroneous transfer of control to data or the middle of an instruction, etc.

If this halt is the result of a software error, analyzing the dump is the best way to troubleshoot the problem. You should analyze the dump to rule out a software caused problem before contacting Field Service.

This halt has rarely, if ever, been seen as a result of hardware error. Conceivable hardware causes of this halt include problems in the datapath boards or the interrupt and exception logic (CEH and ICL boards).

If you have already edited RESTAR.CMD, look at the output in response to the examine commands below. If you have not edited RESTAR.CMD, but the system is still halted, type the following commands.

```
>>>E IR              !examine opcode just executed
>>>E/N:E R0          !examine R0 - SP
>>>E/ID/N:3F 0       !dump the ID registers
>>>E PC              !PC points at opcode
>>>E/V @             !examine @PC
>>>D/ID 1D 18000     !turn off cache
>>>E PC
>>>E/V @             !examine @PC again
>>>DEP AP 6          !set code for halt executed
>>>@RESTAR.CMD       !invoke normal restart command procedure
```

If the displayed contents of the memory at PC change after you turned off cache, then probably cache is at fault. If the byte at PC is not a hex BC, BD, BE, or BF (CHMx opcode), then you may have a problem in either the IDP or the IRC board. Examining IR results in a display of three numbers, the first of which is the opcode just executed. If this is a hex BC, BD, BE, or BF, but the byte at PC is not the same, there may be faulty shifting in the instruction buffer or an intermittent addressing problem during instruction decode. In any of these cases, contact Field Service.

?CLOCK PHASE ERROR

This error is unique to the VAX-11/785. When the LSI-11 console software detects this error during its periodic polling of the VAX-11/785 state, the console initializes the VAX cpu to reset it to a known state. This is a serious error that cannot be caused by software problems. Under VMS V3, if auto restart is enabled, this halt results in the fatal bugcheck UNKRSTRT. Under VMS V4, if auto restart is enabled, this halt results in the fatal bugcheck OUTOFSYNC. Analyzing the resulting crashdump is not recommended. There is no useful state saved in the crashdump. Contact Field Service.

The VAX-11/785 cpu runs at 133 nanoseconds per cycle, and the SBI at 200 nanoseconds per cycle. The cpu and SBI are in synch in a 3:2 ratio. If either clock shifts with respect to the other or if the SBI clock stops, the clocks become out of phase, and the SBI freezes to prevent corruption of data on mass storage devices. Since the SBI is therefore inaccessible, the cpu stalls the next time it sends out an SBI command, for example, to fetch data from memory.

?CPU DBLE-ERR HALT

This halt usually means that while the cpu was trying to write the microcode machine check logout onto the stack, another machine check occurred. Actually, the microcode sets a flag called EFP at entry to its error handling routine and clears it at exit. If the flag is already set, the microcode halts with a double error halt. Hardware problems are usually responsible for this halt. One of the few ways software can cause a double error halt is a corrupted interrupt stack pointer that points to UNIBUS I/O space or nonexistent memory or nonexistent I/O space.

This problem can occur during the boot or warm restart sequences if the boot or restart command procedure deposits an incorrect value into R1. The ISP ROM in the memory controller uses the map registers of the adapter whose nexus number is in R1 as a stack. If R1 points to a non-existent nexus or one without map registers, the ISP ROM's stack manipulations cause a double error halt.

For this halt, the auto restart actions destroy critical information in internal processor registers. If you have already edited RESTAR.CMD, look at the output in response to the examine commands below. If you have not edited RESTAR.CMD, but the system is still halted, type the following commands.

```
>>>! display information about 1st machine check
>>>E/ID/N:9 30          !ID 30 = SUMMARY PARAMETER
                        !ID 31 = CES
                        !ID 32 = TRAPPED UPC
                        !ID 33 = VA/VIBA
                        !ID 34 = D-REG
```

```
                    !ID 35 = TB REG 0
                    !ID 36 = TB REG 1
                    !ID 37 = TIMEOUT ADDR
                    !ID 38 = PARITY
                    !ID 39 = SBI ERROR
>>>! display information about 2nd machine check
>>>E/ID C           !display CES
>>>E/ID 20          !display TRAPPED UPC
>>>E/ID 8           !display D-REG
>>>E/ID 12          !display TB REG 0
>>>E/ID 13          !display TB REG 1
>>>E/ID 1A          !display TIMEOUT ADDR
>>>E/ID 1E          !display PARITY
>>>E/ID 19          !display SBI ERROR
>>>E/N:D R0         !examine general registers
>>>E/I/N:4 0        !examine stack pointer registers
>>>DEP AP 5         !set DBL-ERR halt code
>>>@RESTAR.CMD      !invoke usual restart
```

Use the references listed in ADDITIONAL REFERENCES to decode the processor register contents in an attempt to identify what kinds of machine checks occurred to rule out a software caused problem before contacting Field Service.

See section MACHINE CHECKS - VAX-11/780 AND VAX-11/785 for further information on specific types of machine checks.

Note that occasionally system software, such as the ISP ROM or VMS system initialization code, executes instructions anticipated to cause cpu timeout machine checks in an attempt to determine what hardware is present on the system. If there is a problem such as a control store parity error while the microcode is processing the initial cpu timeout, a double error halt results. Therefore, it is recommended that you be somewhat cautious in drawing conclusions from any first machine check which is a cpu timeout; try to correct the cause of the second machine check so that system software can service the initial cpu timeout.

?ILL I/E VEC

This halt means that an interrupt or exception dispatch was attempted through a System Control Block (SCB) vector whose low two bits contained an illegal value; that is, the low two bits were either binary 10 on a machine without user optional Writable Control Store (WCS) or binary 11. The offset from the beginning of the SCB of the vector containing the illegal value is displayed in the console's "HALTED AT xxxxxxxx" message and is passed in R10 to VMS restart code when auto restart is enabled. The PSL contains an accurate IPL and is passed to VMS restart code in R11 when auto restart is enabled.

This halt can be caused by software corruption of a System Control Block vector or of the PR$_SCBB register.

Troubleshooters sometimes alter an SCB vector as a "trap catcher" so that an interrupt or exception through a particular vector causes a halt rather than execution of the usual service routine. Therefore, when you see this halt, check to see whether it might have been caused deliberately by human intervention.

Possible hardware causes of this halt include problems in the datapath boards, ICL or CEH boards, or vector PROM. This halt can also be caused by hardware corruption of an SCB vector resulting from memory problems or adapter map register problems. Analyze the dump, checking the relevant vector and surrounding vectors to rule out a software caused problem before contacting Field Service.

?INT-STK INVALID

This halt means that an attempted cpu read or write reference to the interrupt stack during interrupt or exception processing would have resulted in a translation not valid or access violation.

This halt can be caused by stack overflow, stack underflow, corruption of the interrupt SP, or corruption in the System Page Table Entries (SPTEs) that map the interrupt stack. SPTE corruption can be due to software problems or hardware problems. A frequent hardware cause is corruption in the memory array containing the SPT.

If this halt is the result of a software error or an insufficiently large interrupt stack, analyzing the dump is the best way to troubleshoot the problem. You should analyze the dump to rule out a software problem before contacting Field Service. Check for user-written drivers or other kernel-mode code that may have corrupted the interrupt stack pointer or the SPTEs that map it.

If the stack pointer at the time of the halt contains a valid interrupt stack address (the interrupt stack pointer low and high boundaries are stored in EXE$GL_INTSTKLM and EXE$GL_INTSTK), there may be a hardware problem in the translation buffer or cache or corruption in the interrupt stack's SPTEs.

If the stack pointer at the time of the halt contains an address below the low boundary of the interrupt stack, the stack is likely to have overflowed. Look for recurring machine check frames or other recurring exceptions on the stack that may have caused it to overflow.

If the SP contains a random address, there may be a memory problem or a bad instruction decode. If you have already edited RESTAR.CMD, look at the output in response to the examine commands below. If you haven't edited RESTAR.CMD, but the system is still halted, type the following commands.

```
>>>E SP          !examine stack pointer register
>>>E/I 4         !examine PR$_ISP
```

```
>>>E/N:D R0        !examine R0 - FP
>>>DEP AP 4        !load inv. int. stk. halt code
>>>@RESTAR.CMD     !invoke usual restart
```

The contents of SP and PR$_ISP should be the same. If they are not, there may be a hardware problem in one of the datapath boards.

?NO USR WCS

Although this halt code is defined, the microcode on a VAX-11/780 cpu revision level 7 or later never generates it. If you see this message on a rev 7 or later VAX-11/780, you may have hardware problems in the console interface board.

Pathological Halts

If the system seems to be halted, but there are no console halt messages, first look at the cpu front panel lights. There are four indicators; from left to right they are ATTN, RUN, POWER, and REMOTE.

When the red ATTN light is lit, the VAX cpu is halted; that is, it is executing the console wait loop microcode, waiting for a command from the console. When the green RUN indicator is lit, the cpu is strobing for hardware interrupts regularly. When the green POWER indicator is lit, the +5 volt power supply is on. When the red REMOTE indicator is lit, remote console access is enabled through the cpu key rotary switch. When the cpu is operating normally, running VMS, for example, the ATTN light is off, and the RUN and POWER lights on.

Find the subsection below corresponding to the state of the lights you see.

If the console terminal has a @ prompt, also see the subsection below @ Prompt on Console Terminal.

ATTN Lit, POWER Lit, RUN Off

The cpu is halted. If there are no console messages, there may be a simple console terminal problem preventing output, such as a blown fuse, paper fault, or terminal left in local mode. If none of these is the case, then, in all likelihood, there is a hardware problem. Contact Field Service.

There may be a VAX power supply problem, console interface board problem, or LSI-11 problem that prevents a halt message from being output.

ATTN Off, POWER Lit, RUN Off

If the POWER light is lit, but RUN and ATTN are off, either the cpu clock is stopped or the cpu is hung in a microcode loop. You can determine which it is by opening the cpu cabinet and looking at the leds on the microsequencer (USC) and clock (CLK) boards.

These boards are on the left side of the cabinet. Each board has a small tab in the middle with its module number. On the VAX-11/780 the USC board is M8235, and the CLK board is M8232. On the VAX-11/785, the USC board is M7476, and the CLK board is M7474. In addition, most cpu cabinets have stickers to the left of the door showing what modules are in what slots. Look for a sticker titled KA780 Module Utilization on the VAX-11/780 or KA785 Module Utilization on the VAX-11/785. The sticker lists the cpu modules and their board slot numbers (for example, 21). The board slots are identified by stickers that run horizontally beneath the boards and that have a number under each board slot.

If the clock is running, four leds on the VAX-11/780 clock board (M8232) or eight leds on the VAX-11/785 clock board (M7474) are solidly lit. If they are dimly lit or if only one is lit, the cpu clock is stopped. Sometimes in response to console commands, the LSI-11 console software stops the clock, but this is generally a temporary state which you should not see under normal circumstances.

The microsequencer board has fourteen leds: one halfway up the board, and thirteen leds below that one and separate from it. The lower thirteen leds on the microsequencer board display the micropc and, thus, normally flash on and off. If the system is very busy, the flashes may be quick enough to make the leds glow dimly. If the microsequencer is caught in a loop, the leds glow more brightly and appear solidly lit.

You can determine the micropc by reading the leds on the microsequencer board or by using the console commands below

```
CTRL/P
>>> H
    NO CPU RESPONSE !if cpu really hung
>>> SET STEP STATE
    CLK STOPPED
    CPT0    UPC <xxxx>
>>> N
    CPT1    <space bar>
    CPT2    <space bar>
    CPT3    APC <yyyy>    <space bar>
    CPT0    UPC <xxxx>    <CR>
```

In the console output at CTP0, <xxxx> is the micropc. If the output at both times is the same, then there may be a clock problem or the microcode may be branching back to itself waiting for some condition to be completed. If the output is different, the microcode is caught in a loop.

Read the micropc from the top of the thirteen leds to the bottom. A lit led indicates a 1 and an unlit led a 0. The most significant bit is the top led. The micropc is read as 4 hexadecimal digits, with the most significant digit either a 1 or 0 depending on the top led.

If the VAX-11/780 or VAX-11/785 micropc is at 0100, the cpu is executing its INIT sequence. This indicates a likely power problem. If the VAX-11/780 micropc is at 00FF, the cpu is executing its console wait loop, and the ATTN light would normally be lit. If the VAX-11/785 micropc is at 0E13, the cpu is executing its console wait loop. (The VAX-11/785 init micropc is [TBS].)

The top led on the microsequencer board lights when the cpu is in a "cache stall", typically performing a read from memory of data not in cache. This led should normally flash on and off. If it is solidly lit, that indicates a problem in the SBI or one of its nexus.

One possibility is a power problem locking up the SBI. The voltages and connectors on the power supplies and backplanes should be checked.

In any of these cases, contact Field Service.

ATTN Off, POWER Lit, RUN Lit

If the RUN and POWER lights are lit and ATTN is off, that means that the cpu has power and is strobing interrupts. If the leds on the USC board and the CLK board are flashing, then the cpu is indeed running and not hung in a microloop. See the subsection above, ATTN Off, POWER Lit, RUN Off, for information on how to find these boards.

If the system is in this state and you thought it was halted because of abnormal or nonexistent response to users, see section HANGS.

@ Prompt On Console Terminal

If the console terminal has a @ prompt, that means the LSI-11 is executing MicroODT, rather than the console software.

On some systems, when the console terminal sends a <BREAK>, the LSI-11 executes MicroODT. This means that an interactive user on that terminal can accidentally hit the <BREAK> key and invoke MicroODT. If this has happened, type P to the @ prompt to resume execution of the console software.

This behavior can be disabled through the FEH jumper on the DLV-11 that interfaces the console terminal to the LSI-11. Contact Field Service to change the jumper.

Additionally, if the console terminal is an LA120, it may have been

set up to generate a break signal in response to paper out, head jam, or cover open. Put the terminal into SET-UP mode and then type U to see the current setting of the LA120 break action. A 1 means that the LA120 automatically sends a break signal in response to paper out, head jam, or cover open. If 1 is the current setting, type U again to disable the sending of the break.

If no one typed <BREAK> and if nothing is wrong with the console terminal, then the @ prompt may be caused by a problem in the LSI-11 or its power supplies. Type the following commands (without the comments following the exclamation points) on the console terminal to gather information about the state of the LSI-11 console software and to restart it.

```
@Mxxxxxn                    !M must be typed uppercase
@R0/xxxxxx<LF>              !exam R0
R1/xxxxxx<LF>               !exam R1
R2/xxxxxx<LF>               !exam R2
R3/xxxxxx<LF>               !exam R3
R4/xxxxxx<LF>               !exam R4
R5/xxxxxx<LF>               !exam R5
R6/xxxxxx<LF>               !exam R6, the SP
R7/xxxxxx<CR>               !PC at halt
@R6/xxxxxx@                 !examine SP
   xxxxxx/yyyyyy<LF>        !examine (SP) old PC
   xxxxxx+2/zzzzzz<CR>      !examine (SP)+2 old PSW
@R7/xxxxxx 141330<CR>       !load PC
@P                          !restart console software
>>>SET TERMINAL PROGRAM
```

This should restart either a VAX-11/780 or VAX-11/785 console without rebooting the VAX cpu. Save the console printout, and contact Field Service. The VAX Maintenance Handbook VAX-11/780 (August 1982 edition) pages 97-98 describe the LSI-11 MicroODT commands and the meanings of the value displayed in response to the M command.

RESTAR.CMD Command Procedure

The console floppy contains a file called RESTAR.CMD. This command procedure is used when the LSI-11 restarts a VAX-11/780 or VAX-11/785 following a power failure recovery or cpu halt. The subsections below describe the VAX-11/780 and VAX-11/785 restart mechanism and discuss editing RESTAR.CMD to gather more information about a cpu halt.

VAX-11/780 And VAX-11/785 Restart Mechanism

The auto restart toggle switch at the left of the console control panel determines the system's response to a power failure recovery or cpu halt. The usual recommendation is that the switch be set to ON.

This means that after a power fail recovery or halt, the LSI-11 console subsystem attempts a warm restart by copying the halt PC, halt PSL, and reason for the halt to R10, R11, and AP, and by executing the command procedure RESTAR.CMD from the console floppy.

The distributed RESTAR.CMD procedure initializes the cpu and nexus adapters and controllers, deposits into R0 - R5 and FP, and passes control to the restart entry point of the VAX instruction-level ROM (ISP ROM) present as part of the (first) local memory controller. The combination of console actions and execution of RESTAR.CMD initializes many processor registers and overwrites most of the general registers.

The ISP ROM tests that memory contents are valid (that is, that battery backup during a power failure was sufficient), that local memory (MS780) is configured correctly, and tries to locate the Restart Parameter Block (RPB) built during system initialization.

The ISP ROM reports errors on the console terminal. Fatal errors result in a loop in the ISP ROM to prevent infinite attempts at warm restart. Following are the possible messages from the ISP ROM. For further information on the error messages, see the [TBS] section BOOT FAILURES - VAX-11/780 AND VAX-11/785.

```
ATTEMPTING WARM RESTART
FATAL ERROR-- CPU ERROR, R7 INDICATES FAILING SUBTEST
FATAL ERROR-- MEMORY ADDRESS SPACE OVERLAPS
FATAL ERROR-- MEMORY(IES) IMPROPERLY INTERLEAVED
FATAL ERROR-- MEMORY NOT INITIALIZED
FATAL ERROR-- MIX OF 64K AND 256K ARRAY CARDS
FATAL ERROR-- NEXUS HAS BAD MAPS
FATAL ERROR-- NO WORKING MEMORY
FATAL ERROR-- UNEXPECTED MACHINE CHECK
WARNING-- FAULT DETECTED ON SBI, CONTINUING
```

If the ISP ROM locates the RPB and validates its contents, the ISP ROM types the message "ATTEMPTING WARM RESTART" on the console terminal and jumps to the address contained in RPB offset RPB$L_RESTART. This longword should contain the physical address of the VMS routine EXE$RESTART.

If the ISP ROM finds that memory contents are not valid, if it is unable to locate the RPB, or if the RPB has been corrupted, the ISP ROM sends a reboot message to the console and halts. The console reboots the system from the default system disk. The MS780-C ISP ROM halts at physical 2000350A; the newer MS780-E ISP ROM halts at physical 20003563.

EXE$RESTART's responsibility is either to restart the system following a power fail or to crash the system following a halt. The system is crashed to preserve error log messages that have not yet been written to SYS$ERRORLOG:ERRLOG.SYS and to provide information useful for troubleshooting the halt. (SYSINIT locates the error log messages in the dump during the next reboot and causes them to be

written to the error log file.) EXE$RESTART crashes the system with a bugcheck whose type is a function of the halt code passed in AP. The halt PC and halt PSL are in R10 and R11 when EXE$RESTART bugchecks. See the section RESTART BUGCHECKS for more information on EXE$RESTART.

For certain kinds of processor halts, the auto restart actions above destroy information likely to be needed in troubleshooting the problem. In such cases, there are two possibilities: set the auto restart toggle switch OFF, use the console to display what information you need, deposit a halt code into AP, and manually invoke RESTAR.CMD; or alter the RESTAR.CMD procedure to display needed information automatically. See below the subsection Editing RESTAR.CMD for further details.

If the auto restart toggle switch is off, the cpu remains halted until some human intervenes. If the cpu key is in local enable, the console accepts commands in response to its >>> prompt. You may enter commands to display various registers. Afterwards, you should manually invoke RESTAR.CMD by typing @RESTAR.CMD. Unless you load AP with a halt code before you invoke RESTAR.CMD, EXE$RESTART will signal the bugcheck UNKRSTRT. See the section RESTART BUGCHECKS for a table of possible halt codes and their meaning.

The advantage to disabling auto restart and invoking RESTAR.CMD manually is that you can obtain the contents of R10, R11, and AP at the time of the halt. Occasionally, these may be important in troubleshooting a problem. The disadvantage is the required human intervention. For this reason, editing RESTAR.CMD is preferred and is recommended for any systems experiencing intermittent problems or frequent halts.

Editing RESTAR.CMD

The basic sequence is to

1. copy the console floppy

2. determine adapters and controllers present on the system

3. create a console command file named DISPLA.CMD that examines various internal, processor, and nexus registers and includes the commands from RESTAR.CMD

4. create a new RESTAR.CMD that only invokes DISPLA.CMD

5. copy the new and altered files to the new console floppy

When auto restart is enabled, the console automatically invokes RESTAR.CMD after a halt (unless it has received a reboot message). RESTAR.CMD invokes DISPLA.CMD, which displays various registers and

restarts the system.

The reason for not placing the contents of DISPLA.CMD into RESTAR.CMD is that invoking DISPLA.CMD explicitly as a command procedure (rather than the console's invoking it automatically as RESTAR.CMD) results in the echoing of the commands in the procedure. This makes the output somewhat easier to interpret. If you are indifferent to the echoing of the DISPLA.CMD commands, then simply edit RESTAR.CMD to include the commands shown below as the contents of DISPLA.CMD.

First, log into the SYSTEM account or one with CMKRNL, SYSPRV, and SYSNAM privileges. Make a copy of the console floppy using CONSCOPY, as documented in section 2.81 of the Guide to VAX/VMS System Management and Daily Operations. Ensure that the original copy of the console floppy is in the drive whenever VMS maintenance updates are installed to prevent failure of any updates to the floppy as a result of insufficient free space on it. Ensure that your altered console floppy reflects any console floppy changes made by VMS maintenance updates, VAX FCO installations, or system management actions.

Next, determine the adapters and controllers present at each nexus, using the following commands

```
$ MC SYSGEN
SYSGEN> SHOW /ADAPTER
SYSGEN> EXIT
```

Note that SYSGEN displays decimal nexus numbers.

Next, using your favorite editor, create a file called DISPLA.CMD containing the commands below.

```
HALT            !halt cpu
E/ID/N:17 0     !examine ID 0 - 17
E/ID 18         !examine 16 entries in SBI silo
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID 18
E/ID/N:6 19     !examine ID 19 - 1F
E/ID 20         !pop microstack
E/ID 20
E/ID 20
E/ID 20
E/ID/N:1E 21    !examine ID 21 - 3F
E/N:D R0        !examine R0-FP
E PSL
E SP
E/V/N:10 @      !11 longwords of  current stack
E/I 4           !examine PR$_ISP
E/V/N:10 @      !11 longwords of interrupt stack
E/I/N:3 0       !PR$_KSP - PR$_USP
E IR            !examine instruction register
E PC            !examine PC
E/V @           !examine (PC)
E/V -           !examine (PC)-4
D/ID 1D 18000   !turn off cache
E PC
E/V @           !examine (PC)
E/V -           !examine (PC)-4
! THE ADDRESSES IN THE FOLLOWING COMMANDS DISPLAY NEXUS REGISTERS FOR A
!  POSSIBLE CONFIGURATION. THEY SHOULD BE ALTERED AND/OR COMMANDS ADDED
!  TO REFLECT THE ACTUAL HARDWARE CONFIGURATION.
E 20002000/N:3  !MS780E MEMORY TR1
E 20004000/N:2  !MS780C MEMORY TR2
E 20006000/N:7  !UNIBUS ADAPTER   TR3
E 20010000/N:7  !MASSBUS ADAPTER TR8
E 20012000/N:7  !MASSBUS ADAPTER TR9
E 20004000/N:9  !MA780 MEMORY TR 2
E 20014000/N:1  !DR780 TR10
E 2001C000/N:1  !CI780 TR14
```

Tailor the command procedure to reflect the hardware configuration you determined with SYSGEN. Tailor it by deleting, adding, or modifying the commands that examine the nexus registers. The physical address corresponding to nexus N's registers is 200xx000, where xx equals 2 times N in hex. For example, the registers for nexus 4 begin at 20008000.

You might also want to include commands to examine the registers of the system disk drive, if it is not a DSA-style disk.

Then, enable access to the console floppy with the following commands

```
$ ! load the console driver if it hasn't been
$ MC SYSGEN
SYSGEN> CONNECT CONSOLE
SYSGEN> EXIT
```

Under V3, type the following commands

```
$ ! mount the console floppy if it hasn't been
$ MOUNT/FOR/SYS/PROT=(SY:RWLP) CSA1: CONSOLE
$ !
$ ! copy restar.cmd to your default disk, directory
$ MC FLX
FLX> /RS=CS1:RESTAR.CMD/RT
FLX> CTRL/Z
$ !
$ ! include restart procedure in DISPLA.CMD
$ APPEND RESTAR.CMD DISPLA.CMD
$ !
$ ! here use EDT to create a new RESTAR.CMD
$ !  to contain only the command @DISPLA.CMD
$ !
$ ! replace console RESTAR.CMD with edited version and
$ !  copy DISPLA.CMD to console floppy
$ MC FLX
FLX> CS1:/RT=DISPLA.CMD/RS
FLX> CS1:RESTAR.CMD/DE/RT
FLX> CS1:/RT=RESTAR.CMD/RS
FLX> CTRL/Z
$ ! clean up default directory
$ DELETE RESTAR.CMD;*,DISPLA.CMD;*
```

Under V4, type the following commands

```
$ ! copy original RESTAR.CMD
$ EXCHANGE COPY CS1:RESTAR.CMD RESTAR.CMD
$ !
$ ! include RESTAR.CMD in DISPLA.CMD
$ APPEND RESTAR.CMD DISPLA.CMD
$ !
$ ! here use EDT to create RESTAR.CMD
$ !   to contain only the command @DISPLA.CMD
$ !
$ ! copy DISPLA.CMD and RESTAR.CMD to console floppy
$ EXCHANGE
EXCHANGE> COPY RESTAR.CMD CS1:RESTAR.CMD
EXCHANGE> COPY DISPLA.CMD CS1:DISPLA.CMD
EXCHANGE> EXIT
$ ! clean up default directory
$ DELETE RESTAR.CMD;*,DISPLA.CMD;*
```

Hints And Kinks

1. If you have auto restart enabled and set the VAX cpu into single instruction step mode and erroneously continue it via the console commands

   ```
   >>>SET STEP INSTRUCTION
   >>>CONTINUE
   ```

   when the cpu halts after executing the next instruction, the console restarts the VAX, probably resulting in a crash.

   If you are trying to single step the VAX through the console, use the following commands instead.

   ```
   >>>SET STEP INSTRUCTION
   >>>NEXT
   <space>
   ```

   Each time you depress the space bar, the cpu will execute one instruction, and the console will not auto restart the VAX cpu.

2. The LSI-11 console software is case-sensitive. Ensure that all invocations of chained command procedures are upper case.

3. The console block storage medium has an RT-11 file structure. The RT-11 file structure implements three different record formats: stream ASCII, formatted binary, and fixed-length record. Under VMS you use the V3 FLX utility or the V4 EXCHANGE utility to transfer files to and from the console.

Both FLX and EXCHANGE select a default record transfer mode based on file extension type. For example, extensions of OBJ and BIN default as EXCHANGE /RECORD=BINARY and FLX /FB transfer modes.

Occasionally the default based on file extension type is inconsistent with the file's record format. In particular, CI780.BIN, the CI microcode; WCSxxx.PAT, the VAX-11/780 microcode; and PCS750.BIN, the VAX-11/750 microcode, will not be copied correctly unless you override the default transfer mode.

If you are not sure what the transfer mode should be, you can use the EXCHANGE qualifier /RECORD_FORMAT=STREAM or the FLX switch /FA for all text files (e.g. command files). Use the EXCHANGE qualifier /RECORD_FORMAT=FIXED (or /TRANSFER_MODE=BLOCK) or the FLX switch /IM for all other files (binary files such as images, microcode files, patch files). The VMS console contains no formatted binary files, so you will never want /RECORD_FORMAT=BINARY or FLX's /FB.

Additional References

VAX Architecture Standard (DEC Standard 032), Section 12.7 Halts

VAX-11/780 Console Interface Board Technical Description

VAX-11/780 Datapath Description, Section 5.3 Machine Halts

VAX-11/780 Hardware User's Guide, Chapter 3 Console Operator/Program Communication

VAX-11/780 VAX Maintenance Handbook for processor register layouts (pp. 133-154 of the 8/82 edition)

VAX-11/780 VAX Maintenance Handbook for additional information on interpreting the contents of the SBI silo (pp. 156-158 and pp. 185-188 of the 8/82 edition)

VAX-11/780 VAX Maintenance Handbook for the location of the DLV-11 FEH jumper (p. 103 of the 8/82 edition)

Sheet ESOAD-1, VAX Hardware Documentation Microfiche Library, MS780-C ISP ROM listing

VAX/VMS Error Log Utility Reference Manual

Microcomputers and Memories (EB-20912-20) for more information on LSI-11 MicroODT

HANGS

There are many possible reasons for lack of system response to users, ranging from a blown fuse on a terminal to a compute-bound user program to a halted processor. This section describes procedures and suggestions to help you determine why the system is not interacting with one or more users after it has been running normally. If the system has only just been booted and seems not to have completed system initialization successfully, see [TBS] section BOOT FAILURES.

First, find out whether or not the whole system seems hung. Ask the users. Try a terminal other than the one(s) in use by the affected user(s) to issue the DCL command SHOW SYSTEM. If the system seems not to be responding to any users, follow the directions in subsection System Hangs. Otherwise, follow the directions in Process Hangs.

System Hangs

1. First, look at the console terminal for any messages.

2. If there are no messages, put the cpu key in the local enable position and type CTRL/P on the console terminal. If the console terminal and its cpu connection are working, you should get a console subsystem prompt >>>. On a VAX-11/780 or VAX-11/785, type SET TERMINAL PROGRAM in response to the prompt to continue. On other processors, type a C in response to the console prompt to continue. Then go on to item 8.

   If you don't get the prompt, put the terminal into local mode and check for problems such as a blown fuse or paper out. If the terminal works in local mode, but you are unable to communicate with the console subsystem, then the system may have some sort of hardware hang. Additionally, you may have missed an important message concerning a software problem. See the section HALTS - <cpu_type> for information about processor specific hardware hangs and halts.

   On a VAX-11/750 without remote diagnosis, cpu microcode implements the console interface. If the UNIBUS is hung in such a way as to hang the cpu microsequencer, the console subsystem is uncommunicative. This is a known symptom, in response to which you should probably power sequence the processor, by pushing the white RESET button on the front of the machine. This may clear the UNIBUS hang. (If you don't have memory battery backup, this causes a reboot.) If this doesn't clear the hang, call Field Service.

3. If there is a halt code or other console software message on the console terminal and/or you think the system may be halted, see section HALTS - <cpu_type>.

   One possibility on a VAX-11/750 is a reboot following a halt restart. After a processor halt, the console checks the auto restart switch to determine whether to leave the processor halted, to reboot it, or to restart it. If a VAX-11/750 auto restart switch is in the reboot position, the console reboots the system. For example, if a power fail recovery occurs and the auto restart switch is in the reboot position, the system will reboot. The only console message printed is %%, followed by the VMS announcement message. If you suspect that is happening, put the auto-restart switch into the halt position, so that the system will halt instead. This will enable you to distinguish between power recovery restarts and software initiated reboots.

4. If there is a fatal bugcheck message on the console terminal, the system is crashing or has crashed. If SYSBOOT parameter BUGREBOOT is 0, the fatal bugcheck code prints the following message on the console terminal and loops.

   ```
   SYSTEM SHUTDOWN COMPLETE - USE CONSOLE TO HALT THE SYSTEM
   ```

   If parameter BUGREBOOT is 1, the fatal bugcheck code reboots the system from the default system disk. After the system reboots, analyze the crashdump, following the directions in section BUGCHECKS.

5. Under V3 there are two possible messages about the page file you may see

   ```
   SYSTEM-W-PAGEFRAG, Pagefile 65% full, system continuing
   SYSTEM-W-PAGECRIT, Pagefile 90% full, system trying to continue
   ```

   Under V4, the text of these messages is changed to

   ```
   SYSTEM-W-PAGEFRAG, Pagefile badly fragmented,system continuing
   SYSTEM-W-PAGECRIT, Pagefile space critical, system trying to conti
   nue
   ```

   These messages are each issued only once during a boot of the system, no matter how many page files you have installed. When one page file becomes badly fragmented or fairly full, the first message is output; when this or another page file becomes very full, the second message is output. These messages may be an indication that the system requires an(other) alternate page file, although they may also merely mean that one particular file has become full. Furthermore, because of the nature of the system checks, it is possible for the system to run out of page file space without any message's having been printed.

   When the page file(s) become full, the SWAPPER process may be unable to write the modified page list. As the modified page list grows and reaches the size of the SYSBOOT parameter

MPW_WAITLIM, processes faulting modified pages out of their working sets are placed into resource wait RSN$_MPWBUSY. If your system is swapping to page file(s) that have become full, processes whose working sets are being expanded may be placed into resource wait RSN$_SWPFILE. See section RESOURCE WAITS for more information on these wait states.

To find out which file(s) are becoming full, you may be able to issue the DCL command SHOW MEMORY/FILES/FULL. If a heavily used page file is full or almost full, you may be able to install another one in an attempt to prevent a system hang. Because a process is assigned for its lifetime to a particular page file, installing a new page file will not necessarily clear up the problem. It may be necessary to remove user processes, whether through LOGOUT, explicit STOP/ID commands, or a system shutdown.

While you are trying to take these steps, temporarily raise the parameter MPW_WAITLIMIT so that your own process is not placed into resource wait. Raising MPW_WAITLIMIT is a temporary workaround that works until the system runs out of free pages.

From an account with CMKRNL privilege, type the following DCL commands.

```
$ SHOW MEMORY/FILES/FULL
$ MC SYSGEN
SYSGEN> SHOW MPW_WAITLIMIT
Parameter Name   Current
MPW_WAITLIMIT     xxxx
SYSGEN> SET MPW_WAITLIMIT 16384
SYSGEN> WRITE ACTIVE
SYSGEN> SHOW PAGFILCNT
SYSGEN> EXIT
```

If the setting of parameter PAGFILCNT is too low to allow installation of another page file, have users log out, alter the parameter, and shut down the system. After it reboots, create and install another page file.

If the setting of PAGFILCNT permits installation of another page file, from the SYSTEM account or one set to a SYSTEM UIC, type the following DCL commands to install another page file and reset MPW_WAITLIMIT.

```
$ SET PROT=(SY:RWED,OW:RWED)/DEFAULT
$ MC SYSGEN
! specify a unique page file name
SYSGEN> CREATE <file_spec>/SIZE=<size>
SYSGEN> INSTALL <file_spec>/PAGEFILE
SYSGEN> SET MPW_WAITLIMIT <xxxx> !previous value
SYSGEN> WRITE ACTIVE
SYSGEN> EXIT
```

If your own process goes into RSN$_MPWBUSY wait and you are

unable to issue any DCL commands, then you must alter MPW_WAITLIMIT from the console terminal. With the key in local enable, type the following commands, filling in the global values from the table below. On a MicroVAX I, depress the HALT button on the front panel instead of typing CTRL/P.

```
CTRL/P
>>>H
>>>!         examine MPW_WAITLIMIT
>>>E/V/L <A(MPW$GL_WAITLIM)>
>>>!         read number of pages on modified list
>>>E/V/L <A(SCH$GL_MFYCNT)>
>>>!         examine PSL
>>>E P
        xxxxxxxx
>>>!         put processor into kernel mode
>>>D P 0
>>>!         raise MPW_WAITLIMIT
>>>D/V/L <A(MPW$GL_WAITLIM)> 3FFF
>>>!         trigger modified page writer
>>>D/V/L <A(SCH$GL_MFYLIM)> 0
>>>!         restore previous PSL
>>>D P <xxxxxxxx>
>>>!         continue processor
>>>C
```

GLOBAL NAME	V3 VALUE	V4 VALUE
MPW$GL_WAITLIM	8000328C	80003C90
SCH$GL_MFYCNT	80001DF4	80001EF0
SCH$GL_MFYLIM	80001E04	80001F00

6. The following message on the console terminal indicates that nonpaged pool could not be expanded.
   %SYSTEM-W-POOLEXPF, Pool expansion failure
   The system may hang as a result, if it runs out of nonpaged pool. See section RESOURCE WAITS, subsection RWNPG for more information.

7. If you see messages indicating that a disk is undergoing mount verification, then user, system, and Files-11 I/O requests to that disk are being stalled. Users may be able to type CTRL/Y and STOP to abort their images and thus cancel their outstanding I/O requests. Files-11 I/O, however, cannot be canceled. Under V3, Files-11 is implemented as a separate process, an "ACP", which does one I/O request at a time. While the ACP is processing one request, other I/O requests queued to it must wait. This means that if a Files-11 ACP I/O request is queued to a disk in mount verification, the ACP's I/O request is stalled, the ACP itself is stalled, and all I/O requests queued to the ACP are stalled. How noticeable this effect might be is partially a function of how many disk volumes this ACP is managing. If the system is set up with very few ACPs and lots of caching, then the effect is likely to be quite noticeable. See the Guide to

VAX/VMS System Management and Daily Operations Guide, section 7.6, for more information on the possible causes of a disk's going into mount verification and actions to take.

Under V4, Files-11 ODS-2, the default file structure, is implemented with procedure-based routines, the Files-11 XQP, that run in process context. If a process's XQP I/O request is stalled, then any further Files-11 I/O requests (for example, IO$_ACCESS, IO$_DEACCESS, window turns) from that process are also stalled. Files-11 ODS-1, the RSX-11 file structure, is still implemented with a separate ACP.

8. If you see messages on a V4 member of a cluster indicating that connections have been lost or timed out and that quorum has been lost, see below item 11.

9. If there are no messages and the cpu does not appear to be halted, try to log in on the console terminal. If you are able to, then possibly there is a hardware problem preventing access to all the other terminals, such as no power to the UNIBUS or UNIBUS adapter or perhaps the terminal controller is not working. (Note that on a VAX-11/750, if the UNIBUS has no power or is broken, you will probably not be able to use the console terminal in any way, other than terminal local mode.)

10. If you are unable to login on the console terminal, the system may be rebooting without having been shutdown. This can happen following an aborted fatal bugcheck. Fatal bugcheck processing may be aborted due to corruption in the boot control block that maps the extents of the system dump file or a failure to initialize the system disk. Depending on cpu type, you may see console messages indicating a halt instruction excecuted and a reboot initiated.

    These conditions are rare, but if you hear a lot of console medium and/or system disk activity, wait for several minutes to see whether the system is booting before continuing with these directions. If the system doesn't reboot, continue with the next item.

    If the system does reboot following an aborted bugcheck, there is no way to find out why it crashed; the only thing you can do is prevent the immediate reboot next time by booting the system with XDELTA. If XDELTA is present, the bugcheck routine breakpoints before rebooting, allowing you to examine the general registers and current stack from the console terminal.

11. If you are unable to login on the console terminal and the system is not rebooting, it may be hung in a high IPL loop, it may be continually servicing interrupts from a malfunctioning device, or there may be a compute bound realtime process preventing any normal processes from being scheduled.

    You should single step the system through enough instructions to

capture the addresses of any loop it may be caught in. In fact, it's generally a good idea to go through the loop, continue the processor, and then single step it at least once more through the loop to be certain that the addresses you obtained are representative of any loop causing the hang. Afterwards, if appropriate, crash the system and examine the dump. First make sure that the cpu key is in the local enable position. Then type CTRL/P and the console commands below applicable to the cpu type. On a MicroVAX I and MicroVAX II, depress the HALT button on the front panel instead of typing CTRL/P.

o For a VAX 8600

```
                    CPU HALT, csm code: 11
                    PC: xxxxxxxx        !xxxxxxxx is PC
>>>D/I 18 0                             !disable timer
>>>E PSL                                !examine PSL
>>>NEXT
                     U PSL xxxxxxxx
                    PC: xxxxxxxx>>><space_bar> !xxxxxxxx is PC
                    PC: xxxxxxxx>>><space_bar> !each space is 1 step
                    PC: xxxxxxxx>>><CR>        !exit step mode
>>>E PSL                                !exam PSL every several steps
                     U PSL xxxxxxxx
>>>NEXT
                    PC: xxxxxxxx>>><space_bar> !each space is 1 step
                    PC: xxxxxxxx>>><CR>        !exit step mode
                     . . .
>>>D/I 18 80000051                      !re-enable timer
>>>E/N:E R0                             !examine registers
>>>C                                    !continue the cpu
```

o For a VAX-11/780 or VAX-11/785

```
>>>H                                    !halt cpu
HALTED AT xxxxxxxx                      !xxxxxxxx is PC
>>>D/I 18 0                             !disable timer
>>>E PSL                                !examine PSL
               xxxxxxxx
>>>N
HALTED AT xxxxxxxx                      !xxxxxxxx is PC
>>><space_bar>                          !each space is 1 step
HALTED AT xxxxxxxx                      !xxxxxxxx is PC
<space_bar>                             !each space is 1 step
<cr>                                    !exit space bar step mode
>>>E PSL                                !exam PSL every several steps
>>>N
>>><space_bar>                          !each space is 1 step
 . . .
<cr>                                    !exit space bar step mode
>>>D/I 18 80000051                      !re-enable timer
>>>E/N:E R0                             !examine registers
>>>C                                    !continue the cpu
```

o For a VAX-11/750

```
xxxxxxxx  02                  !xxxxxxxx is PC
>>>D/I 18 0                   !disable timer
>>>E P                        !examine PSL
          xxxxxxxx
>>>N                          !each N is 1 step
xxxxxxxx  02
>>>E P                        !exam PSL every several steps
>>>N                          !each N is 1 step
 . . .
>>>D/I 18 8000005             !re-enable timer
>>>E/G 0                      !examine R0
>>>E/G 1                      !examine R1
>>>E/G 2                      !examine R2
     . . .
>>>E/G E                      !examine SP
>>>C                          !continue the cpu
```

o For a VAX-11/730 (it is not necessary to disable timer interrupts)

```
?02  PC=xxxxxxxx              !xxxxxxxx is PC
>>>E PSL                      !examine PSL
M 00000000 xxxxxxxx
>>>N
?02  PC=xxxxxxxx              !xxxxxxxx is PC
>>><space_bar>                !each space is 1 step
?02  PC=xxxxxxxx              !xxxxxxxx is PC
>>>E PSL                      !exam PSL every several steps
>>>N
>>><space_bar>                !each space is 1 step
 . . .
>>>E/G/N:E R0                 !examine registers
>>>C                          !continue the cpu
```

o For a MicroVAX I

```
xxxxxxxx  02                  !xxxxxxxx is PC
>>>D/I 18 0                   !disable timer
>>>E P                        !examine PSL
P  xxxxxxxx
>>>N xxxxxxxx/yyyyyyyy        !xxxxxxxx is PC, and yyyyyyyy
>>>                           !  is contents
>>>N                          !each N is 1 step
>>>E P                        !exam PSL every several steps
>>>N                          !each N is 1 step
 . . .
>>>D/I 18 80000040            !re-enable timer
>>>E/G 0                      !examine R0
>>>E/G 1                      !examine R1
>>>E/G 2                      !examine R2
     . . .
>>>E/G E                      !examine SP
>>>C                          !continue the cpu
```

o For a MicroVAX II

```
?02 EXT HLT
    PC = xxxxxxxx              !xxxxxxxx is PC
>>>D/I 18 0                    !disable timer
>>>E PSL                       !examine PSL
    M 00000000 yyyyyyyy        !yyyyyy is PSL
>>>S                           !each S is one step
?02 EXT HLT
    PC = xxxxxxxx              !xxxxxxxx is PC
>>>S                           !each S is one step
?02 EXT HLT
    PC = xxxxxxxx              !xxxxxxxx is PC
>>>E PSL                       !examine PSL
>>>S                           !each S is one step
 . . .
>>>E/G/N:E R0                  !examine registers
>>>D/I 18 80000040             !re-enable timer
>>>C                           !continue the cpu
```

a. If the halt PC and the single step PC are the address EXE$NULLPROC, that means the system is running the null job. The V3 address of EXE$NULLPROC is 80007B06. The V4 address of EXE$NULLPROC is 80008B1F. The null job is scheduled when there are no other resident computable processes. This happens frequently during normal system operation for relatively brief intervals, but should not persist. One possible cause of this is a modified page list larger than the SYSBOOT parameter MPW_WAITLIMIT. You can confirm this by using the console to examine MPW$GL_WAITLIM and SCH$GL_MFYCNT. See item 5 above for directions and more information. If too large a modified page list is not the problem, crash the system and look at the scheduling state of processes to try to determine why they were not computable. Follow the directions in subsection Process Hangs to learn more about various scheduling states.

b. If the system is not looping at EXE$NULLPROC, then decode the PSL using the layout in the section RELATED REFERENCE MATERIAL. If it shows interrupt stack execution at device IPLs (hex 14 to 17), then the system may be continuously executing a device interrupt service routine because of some hardware problem. Once you have the loop addresses, crash the system. If you are sufficiently familiar with the hardware configuration, also examine device registers before crashing the system. See section HALTS - <cpu_type> for typical console commands used to do this. Follow the directions in section FORCED CRASH to examine the interrupt stack and determine in what code the system was looping. Read the code to figure out which general registers point to data structures that identify which device or controller is causing the interrupts.

c. Under V4, a system which is a member of a cluster that has just lost quorum loops until quorum is regained. The other members of the cluster are hung in similar loops. There should be console messages indicating that connections have been lost or timed out and that quorum has been lost. The PSL should show interrupt stack execution at IPL 8 and, occasionally, IPL 4. The loop includes addresses within CLUSTRLOA.EXE and in the vicinity of IOC$IOPOST. The V4 value of IOC$IOPOST is 80004910. CLUSTRLOA.EXE is loaded during system initialization; the address at which it is loaded is recorded in system global CLU$GL_LOA_ADDR. The console messages and interrupt stack execution at IPLs 8 and 4 should be sufficient to identify this loop.

If a member of the cluster has crashed and cannot be rebooted, it is possible that there may not be enough votes among the remaining nodes to make up a quorum. You can force quorum to be recalculated as a function of the votes of only the remaining members by typing the following console commands on one of the remaining cluster nodes.

```
CTRL/P
>>>H          !halt cpu
>>>D/I 14 C   !request IPL C interrupt
>>>C          !continue cpu
IPC> Q        !request quorum calculation
IPC> CTRL/Z   !exit IPL C service routine
! type the following commands for VMS V4.0 only
CTRL/P
>>>H          !halt cpu
>>>D/I 20 40 !enable console receive interrupts
>>>D/I 22 40 !enable console transmit interrupts
```

The V4.0 IPL C interrupt service routine leaves console interrupts disabled, preventing further VMS use of the console terminal. This has been corrected in V4.1.

d. If IPL is 1F, the system may be looping trying to bugcheck fatally. The system may or may not be running on the interrupt stack, depending on what code signaled the fatal bugcheck. The general sequence for a fatal bugcheck is to initialize the adapter, controller, and unit of the system disk and use the minimal boot driver to read in the fatal bugcheck overlay. If there are hardware errors, the system generally repeats this sequence. This loop is near the global MPH$BUGCHKHK and includes subroutine calls to a routine in SYSLOAxxx.EXE and to routines in the system disk boot driver. (See the section VIRTUAL ADDRESSES - SYSTEM SPACE for information on locating SYSLOAxxx.EXE and the system disk boot driver.) The V3 value of MPH$BUGCHKHK is 8000397A. The V4 value of MPH$BUGCHKHK is 8000436A.

If the system is hung in this loop, try to spin down the system disk, power it off and on, and spin it up again. This

may clear the problem. If the system remains hung in this loop, call Field Service.

e. If the PSL interrupt stack bit is clear and IPL is 0, the system may be executing some compute bound process whose base priority is higher than that of the other processes on the system. This is particularly likely if many of the PCs are in process space (although note that the image may request system services that cause execution of system space code and a non-user mode PSL). You can confirm this by examining the global SCH$GL_CURPCB, which contains the address of the software PCB of the current process. If the contents seem to remain the same for several minutes, then the system may be running the same compute bound process. The V3 address of SCH$GL_CURPCB is 8000210C. The V4 address of SCH$GL_CURPCB is 800021F8. Type the following console command, periodically halting the processor and then continuing it.

```
>>>E/V/L <A(SCH$GL_CURPCB)>
```

If a compute bound process is the problem, you can lower its priority and thus allow other processes to be scheduled by altering SCH$GB_PRI and the priority fields in the process's software PCB. Type the following commands, replacing the expressions in angle brackets with the actual values in the table below.

```
>>>E P                        !examine PSL
          xxxxxxxx
>>>D P 0                      !put processor into kernel mode
>>>E/V/L <A(SCH$GL_CURPCB)>
>>>D/V/B <A(SCH$GB_PRI)> 1D !priority 2
>>>D/V/B <<contents of SCH$GL_CURPCB> + <PCB$B_PRIB>> 1D
>>>D/V/B <<contents of SCH$GL_CURPCB> + <PCB$B_PRI>> 1D
>>>D P <xxxxxxxx>             !restore previous PSL
>>>C
```

GLOBAL NAME	V3 VALUE	V4 VALUE
SCH$GB_PRI	80002158	80002244
SCH$GL_CURPCB	8000210C	800021F8
PCB$B_PRI	B	B
PCB$B_PRIB	2F	2F

Once you've issued these commands, the process should be running at external priority 2, and the system should be scheduling other processes. You might suspend the process for later examination with the DCL command SET PROCESS/SUSPEND. If the problem is an intermittent one and likely to occur again, create an interactive, nonswappable, realtime process with a higher priority in order to stop the runaway realtime process without having to repeat the above or crash the system.

f. If none of the above possibilities applies, once you have the loop addresses and general register contents, crash the system. Follow the directions in section FORCED CRASH to examine the stack current at the time of the loop and to determine in what code the system was looping.

Process Hangs

1. First, look at the SHOW SYSTEM output to see if the process(es) of interest still exist(s).

2. If the process does not exist, then it was unexpectedly deleted. This could occur as the result of a nonfatal bugcheck from kernel or exec mode, a malicious or mistaken user with enough privilege to delete the affected process, or some problem that occurred in the process in an inner mode. If process accounting is enabled for the system and was not disabled for the deleted process, there should be an entry written in the accounting log at the time the process was deleted. It may contain an informative final status. From an account with access to SYS$MANAGER:ACCOUNTNG.DAT, type the following DCL command to see the accounting entries for that user.

   ```
   $ ACCOUNTING/SINCE:TODAY/FULL/USER=<username>
   ```

   Also, check the error log to see if the process was deleted after a nonfatal bugcheck. From an account with SYSPRV privilege or access to SYS$ERRORLOG:ERRLOG.SYS, type the following V4 DCL command to see any bugcheck entries made today.

   ```
   $ ANALYZE/ERROR/INCLUDE=BUGCHECKS/SINCE:TODAY
   ```

   Under V3, run SYE and specify S in response to the "OPTIONS" prompt, /BU in response to the "DEVICE NAME" prompt, and "-- 08:00" in response to the "AFTER DATE" prompt to limit the display to bugcheck entries made since 8 a.m. today.

   If the process was deleted after a nonfatal bugcheck, the bugcheck errorlog entry may have enough information to identify a known problem but probably not have enough to troubleshoot an unknown problem. You may want to set the SYSBOOT parameter BUGCHECKFATAL to 1 and try to reproduce the problem to cause a system crash so a full crashdump is available for analysis.

3. If the process does exist, and under V3, its scheduling state is MWAIT, see the section RESOURCE WAITS; under V4, if its scheduling state is displayed as a resource wait (e.g., RWAST), see the section RESOURCE WAITS.

4. If the process's scheduling state is COMO (computable and outswapped), perhaps the system has no more balance set slots to inswap the process. Issue the DCL command SHOW MEMORY/SLOTS to see if there are free balance set slots. If there are no free balance set slots, probably the SYSBOOT parameter BALSETCNT is too small. Try increasing it, shutting down the system, and rebooting.

5. If the process's scheduling state is COM (computable) or COMO (computable and outswapped) and you suspect that process priority is the problem, first look at the SHOW SYSTEM output. Compare the process's priority to that of other computable processes. It may be that the system is heavily loaded and/or that this process is very low priority with respect to other computable processes. To alter the process's priority, type the following DCL command from an account with ALTPRI and WORLD privileges.

   ```
   $ SET PROCESS/PRIORITY<new_priority> /ID=<pid>
   ```

6. If the process is COM and its priority is not the reason for lack of system response, it is possible that the process is in an infinite loop. Issue the DCL command SHOW SYSTEM or MONITOR PROCESSES to see if the process's cpu time increases. (If its cpu time doesn't increase, there may be a higher priority compute bound job.)If this is an interactive process and the user has enabled CTRL/T, have the user type CTRL/T several times to see whether his cpu time increases. (This will work only if the process is looping in user or supervisor mode.) If the process is looping in user mode, the user may be able to type CTRL/C DEBUG and use the Debugger to trace the loop. (This will not work if the image has been linked /NOTRACE or installed with privilege.)

   If the process is looping below IPL 2, you should be able to obtain some information through the DCL command SHOW PROCESS/CONTINUOUS. Write down PC and PSL values as possible clues to its loop. You may suspend the process for later examination with the DCL command SET PROCESS/SUSPEND. You may want to attempt further investigation through SDA, either on the current system or a crashdump.

   If the process is looping at IPL 2, you may be able to read its registers (including PC and PSL) by running the program GETPHD, listed in section ACCESSING PROCESS CONTEXT WITH SDA.

7. If the process does exist and is in LEF or LEFO, it is possible, though quite difficult in many cases, to determine what the process is waiting for; it may not be possible to take the process out of its wait, although in many cases you should be able to abort the image or delete the process.

   A process in LEF or LEFO may be waiting for any number of different things; I/O completion, a lock grant, or timer expiration are the likeliest. Failure of an I/O request to complete can be due to a lost device interrupt, device failure,

disk undergoing mount verification, software error, etc. A process waiting for a lock grant may be blocked by another process which has taken out an incompatible lock. A process waiting a long time for timer expiration may have specified an incorrect expiration time.

a. If the process is an interactive process, first type CTRL/Q on the user's terminal in case the problem is merely that terminal output is blocked by a previous CTRL/S. Also, check the console terminal or any disk operator terminals for a mount verification message concerning a disk to which the process is doing I/O. If the process is doing I/O to a disk in mount verification, its I/O requests to the disk are stalled. See subsection System Hangs, item 7, for more information.

b. If you can examine the process's registers, see if the wait PC is informative. It should fall within the system service vector area. Follow the directions in the section SYSTEM SERVICE VECTORS to determine whether the process is waiting after having issued a compound system service request. If it is, determine which service and, if possible, examine the argument list and read the code of that service to determine the exact nature of the process's request. If the service was a $QIOW or $ENQW, see below for information on locating I/O requests and lock blocks.

c. If the wait PC is within SYS$WAITFR, SYS$WFLAND, SYS$WFLOR, or, under V4, SYS$SYNCH, then determining what the process is waiting for is much more difficult. Basically, you must determine which flag(s) the process is waiting for; try to locate all the process's outstanding I/O requests, lock requests, and timer requests; identify the request associated with this flag; and determine why the request has not completed. Note that under V4, there are several other asynchronous system services whose completion is signaled by the setting of an event flag. A V4 process in LEF or LEFO could be waiting for any of these.

   Several fields in the PCB specify which event flag(s) the process is waiting for: PCBL_EFWM, PCBB_WEFC, and the wait all bit PCB$V_WALL in PCB$L_STS. In SDA's SHOW PROCESS output, these are called "Event flag wait mask", "Waiting EF cluster", and "WALL" in "Process status". PCB$L_EFWM is the one's complement of the flags in a particular cluster, with a 0 bit indicating a flag being waited for. PCB$B_WEFC specifies whether the mask applies to event flag cluster 0 or 1. (Clusters 2 and 3 are common event flag clusters; a process waiting on common event flags would be in state CEF.) PCB$V_WALL, when set, indicates that the process is waiting for all the flags described by PCB$L_EFWM; a zero PCB$V_WALL indicates that the process issued a $WFLOR system service.

   Determine the flag number(s) by first writing the contents of

PCB$L_EFWM in binary to determine the position of each zero bit (bit 0 means flag 0; bit 1, flag 1; etc.) Then, if PCB$B_WEFC is a 1, add 20 hex or 32 decimal to each flag number. For example, if PCB$B_WEFC contains 0, PCB$L_EFWM contains F7FFFFFF, then the zero bit and the flag being waited for is decimal number 27. If this flag is associated with an I/O request, for example, then you should be able to find an IRP for this process with IRP$B_EFN equal to hex 1B, or decimal 27.

d. Determine whether the process has any outstanding I/O requests by examining the output of the SDA command SHOW PROCESS. If the displays for Buffered I/O count/limit and Direct I/O count/limit show the same numbers (e.g., 6/6), then the process has no outstanding I/O. Go to the next item to determine whether the process has outstanding lock requests or timer queue requests.

   If the count/limit numbers are different, then the process does have outstanding I/O requests, approximately the difference between the numbers in each of the two displays. (The number may not be precise because certain ACP requests are charged as both buffered and direct I/O.)

   If the process has outstanding I/O requests, then you must locate them to see which, if any, are associated with the event flag(s) being waited for. Follow the directions in section LOCATING I/O REQUESTS to locate the request(s) associated with a particular event flag number. Note that when SDA displays a request (IRP or CDRP) as part of the SHOW DEVICE display, it converts the flag number to decimal; if you format an IRP or CDRP, all numbers are displayed in hex.

   For an IRP that specifies a flag for which the process is waiting, read the driver code and try to figure out the device state in order to determine why the request has not completed.

   If the I/O request is for a terminal, see item 9 below for hints specific to terminals.

   If the flag is one for which the process is waiting and the request is queued to an ACP or, under V4, Files-11 XQP, then you must determine what is blocking the ACP or XQP.

   An ACP may be too low a priority to get enough cpu time; it may be in LEF/O waiting for an I/O request of its own to complete; etc. You may have to look through the ACP's CCBs to locate its IRP and determine why that request hasn't completed.

e. One way to find out whether a process has outstanding locks or timer requests is to examine the JIB. (Under V4, the easier way to find out if the process has outstanding locks

is the SDA command SHOW PROCESS/LOCKS.) The JIB contains the count/limit information for timer queue requests and lock requests. The names of these fields are JIBW_TQCNT, JIBW_TQLM, JIB$W_ENQCNT, and JIB$W_ENQLM. The number of outstanding timer requests is the difference between JIB$W_TQLM and JIB$W_TQCNT. The number of outstanding locks is the difference between JIB$W_ENQLM and JIB$W_ENQCNT. To display the JIB, type the following SDA commands.

```
SDA> SHOW PROCESS                  !read address of JIB
SDA> READ SYS$SYSTEM:SYSDEF.STB!if you haven't already
SDA> FORMAT <jib_address>
```

Remember, though, that the JIB describes the job as a whole, the main process and any of its subprocesses. If this process is the only one in the job, then JIB$W_PRCCNT will be zero, and any outstanding locks or timer requests belong to this process. If there are multiple processes in the job and outstanding timer requests or lock requests, then you must further examine the LEF/O process to determine whether it really has outstanding timer requests or locks.

f. If you think a V4 process is waiting for a lock request to complete, type the following SDA commands to display those locks and, possibly, get information about the process blocking the ungranted lock request.

```
SDA> ! under V4
SDA> SHOW PROCESS/LOCK          !display all locks
SDA> ! For each lock displayed as "Waiting for..."
SDA> !   get Lock ID and LKB address
SDA> DEF LKB=<lkb_address>
SDA> EXAM LKB+LKB$B_EFN         !examine associated EFN
SDA> ! If EFN is the one in question, then display RSB
SDA> SHOW RESOURCE/LOCK=<lock_id>
SDA> ! If CSID is 0, then issue following commands;
SDA> !   else, resource is mastered on other node
SDA> ! Examine the Grant Queue to get the Lock Id of the
SDA> !  granted lock blocking this one
SDA> ! Display that lock to get PID of owner
SDA> SHOW LOCK <blocking_lock_id>
SDA> SHOW PROCESS/INDEX=<pix>
```

If the process is waiting for a blocked lock request, examine the process(es) with incompatible granted locks and try to determine if they themselves are blocked for some reason. See the Lock Management System Services chapter in the VAX/VMS System Services Reference Manual for a table showing compatibility among the various lock modes.

When RSB$L_CSID is nonzero, the resource is being mastered on another node of the cluster. [more information TBS]

V3 VMS's use of locks is minimal; it uses them only for RMS

shared files. There are no SDA commands to make finding lock requests easy. If you find a V3 process in LEF/O that you think is waiting for a blocked lock request, read Chapter 10 in the V3 VAX/VMS Internals and Data Structures Manual to see how PCBs, LBKs, and RSBs are connected and then issue appropriate SDA EXAMINE and FORMAT commands.

g. If the process has outstanding timer request(s), locate them and compare their expiration times to the system time by typing the following SDA commands.

```
SDA> EVAL EXE$GL_TQFL          !timer queue listhead
SDA> EXAM EXE$GL_TQFL          !address 1st TQE
SDA> ! Repeat next command til back at EXE$GL_TQFL or
SDA> !     you have located the TQE(s)
SDA> FORMAT @.                 !display one TQE
SDA> ! If matching TQE$L_PID, then check RQTYPE and EFN
SDA> !     if TQE$B_RQTYPE = 1
SDA> !     and TQE$B_EFN matches, you've found the request
SDA> !     TQE$Q_TIME is expiration time
SDA> EXAM EXE$GQ_SYSTIME       !read system time
SDA> EXAM EXE$GQ_SYSTIME+4
```

8. If the process is waiting for terminal I/O to complete and the user has not disabled all broadcasts, then, under V4, from a process with SHARE privilege, first try the DCL command

```
$ SET TERM/XON <terminal>
```

a. If that doesn't work, try doing a broadcast to the user's terminal to see if the problem lies in his terminal connection. Although V4 broadcast is done with a $QIO, it may work in some cases even though the terminal seems hung. From an account with OPER privilege, type one of the following DCL commands.

```
$ REPL/URGENT/TERM=<term> <some_message> !V4
$ REPL/TERM=<term> <some_message>        !V3
```

If you see the broadcast message on the user's terminal, then his physical terminal connection is fine.

b. If the user hasn't disabled broadcasts and the broadcast message does not appear, then there may be a problem in the user's terminal connection. You might try any or all of the following:

- Type the DCL command SHOW ERROR to see if the terminal is listed in the display. Terminal hardware errors are not logged; however, that hardware errors have occurred is recorded. Any errors indicate some sort of hardware problem.

o Check whether a keyboard locked light is on. The terminal could have been put into a strange state as a result of random binary data output interpreted as escape sequence(s). If the light is on, press the SETUP key twice to see if that fixes the problem. If the light is still lit, press SETUP and then RESET.

o Put the terminal in local mode and try typing to determine whether the terminal works at all. Check for a blown fuse, paper out, etc.

o If the terminal works, check that its baud rate, type, and other changeable characteristics are consistent with what the system believes them to be.

o If the terminal is connected through a patch panel or switch, check that its connections are intact.

o If all else fails, try a SETUP/RESET if the terminal supports that feature.

o Replace the terminal with one known to work to see if that makes a difference.

c. If you suspect a software problem, type the following DCL commands. Under V4, your process needs SHARE, PHY_IO, and SYSPRV or READALL. Under V3, your process needs PHY_IO and SYSPRV.

```
$ SHO TERM <terminal>
$ SHO TERM/PERM <terminal>
```

Save the output, along with a display of the IRP that describes the uncompleted I/O request. Delete the process and try to reproduce the problem in the simplest way possible. If you can reproduce the problem, get the SHOW TERM output and IRP display again. If you can't reproduce the problem in a simpler way, try to find out what kinds of I/O requests were made to the terminal with what modifiers. Report the problem with that information, the SHOW TERM output, and the IRP display.

9. For a process in any other wait state, read Chapter 12 of the V3 VAX/VMS System Management and Operations Guide and/or the Internals and Data Structures Manual to learn more about the wait state and possible reasons for its length.

Hints And Kinks

1. Under V4, you must be careful about single stepping with the console or XDELTA on a system which is a member of a cluster. The CI port has a 99 second sanity timer which, if enabled, must be reset by software every 99 seconds. Whenever the CI driver initializes the CI port, the driver uses the value of the dynamic SYSBOOT parameter PASANITY to determine whether or not to enable the sanity timer. The default value of PASANITY, which is 1, causes the sanity timer to be enabled.

   If the timer is enabled and not reset by software every 99 seconds, the CI port places itself into maintenance mode, breaking SCS connections to other nodes on the cluster. The other nodes, as a result, reconfigure the cluster to exclude this node.

   If you know you're going to be single stepping a system, boot interactively and use SYSBOOT to change PASANITY to 0. If the system is already running when you decide to single step, run SYSGEN first to alter PASANITY. When you're done using the console or XDELTA, reset PASANITY.

   Use the following commands from an account with CMKRNL privilege

   ```
   $ MC SYSGEN
   SYSGEN> SHOW PASANITY          !display current setting
   SYSGEN> SET PASANITY 0         !to disable timer
                   or
   SYSGEN> SET PASANITY 1         !to enable timer
   SYSGEN> WRITE ACTIVE
   SYSGEN> EXIT
   ```

   Then, if you're disabling the sanity timer, halt the cpu for 100 seconds and continue it. The CI port will reinit due to sanity timer expiration. When the CI port reinitializes, it does so with the sanity timer disabled.

2. Whenever you modify SYSBOOT parameters, remember to make AUTOGEN aware of your changes so that they propagate across AUTOGENs. Include any parameter changes you make in V3 SYS$SYSTEM:PARAMS.DAT or in V4 SYS$SYSTEM:MODPARAMS.DAT. See Chapter 11 in the Guide to VAX/VMS System Management and Daily Operations for further information on AUTOGEN.

Additional References

V3 VAX/VMS Internals and Data Structures Manual, Chapter 10, Scheduling; Chapter 12, Process Control and Communication; Chapter 13, VAX/VMS Lock Manager; Section 14.5, Data Structures that

Describe the Page and Swap Files; Section 15.5.2, Modified Page Writing; Chapter 18, I/O System Services

Guide to VAX/VMS System Management and Daily Operations, Section 4.1, Shutting Down the Operating System

Guide to VAX/VMS Performance Management

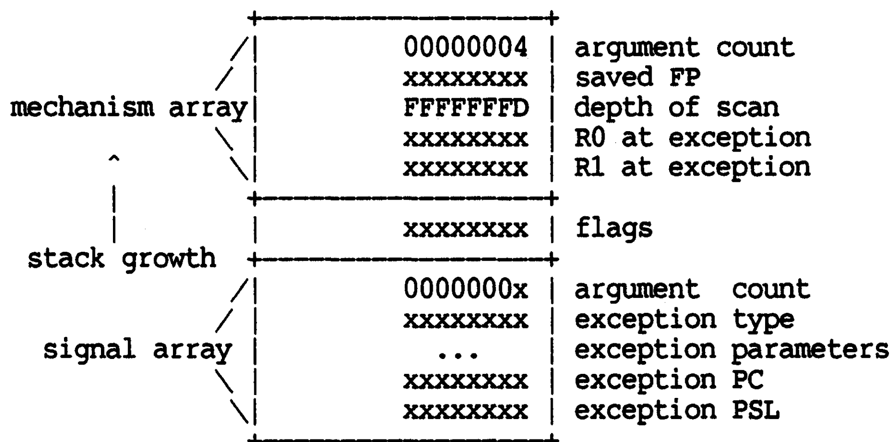
INVEXCEPTN BUGCHECK

The INVEXCEPTN bugcheck is signaled by the common exception dispatching code when it detects that an exception occurred above IPL 2 or on the interrupt stack. Somewhat simplistically, this bugcheck means that while the processor was running in system context, an exception occurred which VMS would normally handle by dispatching to a condition handler established by a process.

This bugcheck is also signaled if the common dispatching code determines that the current process's CTL$AL_STACK array is inaccessible. This is taken to imply that the current process has no P1 space and is thus either the SWAPPER or the NULL process, neither of which should incur exceptions of this sort.

The PC displayed by the SDA SHOW CRASH command reflects the exception dispatching code rather than the location of the exception. R0 and R1 in the SHOW CRASH display have been altered by the exception dispatching code. The PC, R0, and R1 at the time of the exception can be obtained as described below.

When this bugcheck is signaled, signal and mechanism arrays have already been built on the current stack and should be visible among the newest (i.e., lowest addresses) entries on the stack.



1. Locate the mechanism array and identify all its entries. Saved R0 and saved R1 are the registers' values at the time the exception occurred.

2. Skip 1 longword, the flags longword.

3. The next longword, the beginning of the signal array, contains an argument count, the number of longwords that follow. Use the count to identify all entries in the signal array. The number of exception parameters present is a function of exception type and can be 0, 1, or 2 longwords.

4. The exception type is a status value, e.g., C (hex) or SS$_ACCVIO. The DCL command

   ```
   $ EXIT %X<exception_type>
   ```

   writes the message text associated with the exception type status value. The V4 SDA command

   ```
   SDA> EVAL/CONDITION <exception_type>
   ```

   writes the message text associated with the exception type status value.

   Typically, the exception is one generated by "hardware" (or microcode), for example, access violation. "Hardware" generated exceptions are listed with a description of their associated exception parameters in Section 10.1 of the VAX/VMS System Services Reference Manual. See section EXCEPTIONS for information about the more common hardware exceptions.

5. Forced crashes cause INVEXCEPTN bugchecks. If the signal array shows the exception type as SS$_ACCVIO, the PC as approximately FFFFFFFF, the faulting virtual address as approximately FFFFFFFF, and the PSL as kernel mode and IPL 31, the system was crashed using the console CRASH procedure. Continue with the section FORCED CRASHES.

6. The exception PC in the signal array is the instruction whose [attempted] execution resulted in the unexpected exec or kernel mode exception. Whether the PC points to the beginning of the instruction or the end depends on whether the exception was a trap (end), fault (beginning), or abort (beginning). The reference above specifies whether each exception is a trap, fault, or abort. Identify in what source module the PC is. See the section VIRTUAL ADDRESSES. Often examining instructions around the PC is helpful enough to eliminate a microfiche search. Try the SDA command

   ```
   SDA> EXAMINE/INSTRUCTION <exception_pc>-20;30
   ```

   Figure out why the instruction generated an exception. For example, if an access violation occurred, look at the operands to see which access was in error.

7. Look at the IPL and IS values in the exception PSL to determine if the IPL was above 2 and/or the exception occurred on the interrupt stack. Decode the PSL using the PSL layout in the section RELATED REFERENCE MATERIAL or with the V4 SDA command EXAMINE/PSL.

8. If the exception occurred on the interrupt stack, the saved FP in the mechanism array is probably from the current process and not that relevant in analyzing the crash. If the exception occurred on the kernel stack, the saved FP is likely to be the address of

the previous call frame, which contains the previous saved FP. If the stack is intact, these saved FPs can be used to trace the sequence of calls that occurred prior to the crash.

9. Decipher the current stack to trace control flow. See the section STACK PATTERNS.

Hints And Kinks

1. Note that for each V3 SDA COPY command used to copy the dump, the SP will be 8 bytes greater than its actual value; that is, SDA will show the SP pointing to a stack address 8 bytes higher than it should. This V3 bug has been corrected in V4.

2. When SDA examines the process current at the time of an interrupt stack bugcheck, SDA assumes the bugcheck PC and PSL and all the general registers are part of that process's context and displays them in response to the SHOW PROCESS/REGISTER command.

3. The VAX instruction set is sufficiently rich that most random data can be interpreted as instructions. Most system code deals with binary integer and character data. This means that if an EXAMINE/INSTRUCTION display includes many packed decimal and/or floating point instructions, you are probably examining a data area or using a start address which is not an instruction boundary.

   One common error that results in a nonsensical display is to examine instructions in the bugcheck overlay area. During a crash, fatal bugcheck code and message text overlay resident system image code, beginning one page before label BUG$FATAL, for a length of about 12000 decimal or 3000 hex bytes.

Additional References

V3 VAX/VMS Internals and Data Structure Manual, Chapter 4, for general exception dispatching and details of exceptions signaled by VMS system software

VAX Architure Standard (DEC Standard 032) or VAX-11 Architecture Reference Manual, Chapter 6, Exceptions and Interrupts

VAX/VMS System Services Reference Manual, Chapter 10, Condition-Handling Services

KERNEL STACK LOCATIONS

1. If the process is the Swapper, determine the high and low ends of its stack by typing the SDA commands

```
SDA> EVAL SWP$A_KSTK-(4*SWP$K_KSTKSZ)!low end
SDA> EVAL SWP$A_KSTK                 !high end
```

2. If the process is the Null Job, determine the high and low ends of its stack by typing the SDA commands

```
SDA> EVAL SWP$A_KSTK-(4*SWP$K_KSTKSZ)-80 !low end
SDA> EVAL SWP$A_KSTK-(4*SWP$K_KSTKSZ)    !high end
```

3. If the process is not the Null Job or the Swapper, determine the current high and low ends of its kernel stack by typing the SDA commands

```
SDA> EXAM CTL$AL_STACKLIM !low end
SDA> EXAM CTL$AL_STACK    !high end
```

The usual kernel mode stack is in a fixed position in P1 space. Its size and location varies from major release to major release.

For V3, the high and low ends are 7FFEAE00 and 7FFEA800. These are the values stored at CTL$AL_STACK and CTL$AL_STACKLIM. No distributed VMS code moves the kernel mode stack.

With V4, the usual high and low ends are 7FFE7E00 and 7FFE7800. If the kernel stack is expanded, CTL$AL_STACKLIM is altered. If CTL$AL_STACK contains 7FFE7E00, then the process is running on the usual kernel stack, and you have determined its limits.

4. If CTL$AL_STACK and CTL$AL_STACKLIM do not contain the limits of the usual kernel stack, then the process may be executing Files-11 XQP code, which runs on a private kernel stack. To locate the high and low ends of this stack, type the following SDA commands

```
SDA> DEF XQP = @CTL$GL_F11BXQP!P1 space location of XQP
SDA> EXAM XQP + 2C   !read XQP_STKLIM+4 = low end
SDA> EXAM XQP + 28   !read XQP_STKLIM = high end
```

If CTL$AL_STACK and CTL$AL_STACKLIM contain these values, then the process is executing XQP code on the XQP stack, and you have determined its limits.

When the XQP is started up via a call from kernel mode, it saves the contents of CTLAL_STACK, CTLAL_STACKLIM, and the FP before switching to its private stack. You can determine the saved values and display the process's usual kernel stack contents by typing the following SDA commands.

```
SDA> DEF XQP = @CTL$GL_F11BXQP !P1 space location of XQP
SDA> EXAM XQP + 24 !read PREV_STKLIM+4=old CTL$AL_STACKLIM
SDA> EXAM XQP + 20 !read PREV_STKLIM = old CTL$AL_STACK
SDA> EXAM XQP + 1C !read PREV_FP = former FP, on KSP
SDA> SHO STACK @(XQP+1C):@(XQP+20) !display krnl stack
```

KRNLSTAKNV BUGCHECK

The KRNLSTAKNV bugcheck is signaled by the service routine for the exception kernel stack not valid. The microcode generates this exception when it is unable to push information onto the kernel stack during the initiation of an exception or interrupt. This exception is serviced on the interrupt stack at IPL 31. The service routine immediately signals a fatal bugcheck. The bugcheck means that while the processor was running in process context, the current process's kernel mode stack was found invalid by the microcode.

An invalid kernel stack can result from stack overflow, stack underflow, corruption in the P1 page table entries that map the kernel stack, some other kind of software error, or an obscure hardware problem.

V3 VMS defines the kernel stack to be three pages which are locked into the process's working set and which should thus always be valid. V4 VMS defines the process's usual kernel stack to be four pages which are locked into the process's working set and which should thus always be valid.

The PC and PSL displayed by the SDA SHOW CRASH command reflect the exception service routine rather than the location of the exception. The PC and PSL at the time of the exception can be obtained as described below.

1. To display the interrupt stack, type the SDA command SHOW STACK. Its format follows

```
                ^           +-------------------+
                |           |          xxxxxxxx | exception PC
                |           |          xxxxxxxx | exception PSL
          stack growth      +-------------------+
```

2. The KSP value displayed by SDA in response to SHOW CRASH is the address at which the microcode tried to write, after autodecrementing the stack.

3. See the section KERNEL STACK LOCATIONS to determine its high and low limits.

4. Display the pages allocated for the process's kernel stack by using the stack limits you determined. SDA writes this range in "stack" format, with attempted symbolic interpretation, in response to the command

   <u>SDA></u> SHOW STACK <low_address>:<high_address>

   If the KSP value is at or just above its expected high end, then it is possible that some kernel mode code popped more off the stack than it should have. One way this might occur is a RET back to the change mode dispatcher from a kernel mode system service with a corrupted FP. A value of KSP lower than its expected low end generally indicates an overflowed kernel stack.

5. If the KSP is within the limits you determined above, check to see if the page table entries that map it have been corrupted. The page table entries that map it should include the valid bit, the protection as SRKW, and the owner access mode as kernel

   V3 SDA displays only entire process page tables. First convert the low limit virtual address to a virtual page number by issuing the following DCL commands.

   ```
   $ VPN = (%X<faulting_address> .AND. %X3FFFFE00) / 512
   $ SHOW SYMBOL VPN
   ```

   To see 10 P1PTEs (10*4 bytes per PTE = 28 hex bytes), issue the SDA command

   ```
   SDA> EXAM @P1BR + (4*<virtual_page_number>);28
   ```

   See the Internals and Data Structure Manual reference below for information to enable you to decode the PTE.

   V4 SDA can display a range of process PTEs. Type the command

   ```
   SDA> SHOW PROC/PAGE <low_limit>:<high_limit>
   ```

6. Identify in what source module the EXCEPTION PC is. See the section VIRTUAL ADDRESSES.

7. Start at the highest end of the kernel stack and decipher as much of the stack pages as seems to make sense. Without knowing the lowest valid address, this may be difficult since many previous kernel mode threads of execution have used the stack. See the section STACK PATTERNS.

Hints And Kinks

1. When SDA examines the process current at the time of a KRNLSTKNV bugcheck, SDA displays the bugcheck PC and PSL as part of that process's context in response to the SHOW PROCESS/REGISTER command. The process's most recent PC and PSL are those pushed on the interrupt stack as the exception PC and PSL.

2. Note that for each V3 SDA COPY command used to copy the dump, the SP will be 8 bytes greater than its actual value; that is, SDA will show the SP pointing to a stack address 8 bytes higher than it should. This V3 bug has been corrected in V4.

3. The VAX instruction set is sufficiently rich that most random data can be interpreted as instructions. Most system code deals with binary integer and character data. This means that if an EXAMINE/INSTRUCTION display includes many packed decimal and/or

floating point instructions, you are probably examining a data area or using a start address which is not an instruction boundary.

One common error that results in a nonsensical display is to examine instructions in the bugcheck overlay area. During a crash, fatal bugcheck code and message text overlay resident system image code, beginning one page before label BUG$FATAL, for a length of about 12000 decimal or 3000 hex bytes.

LOCATING I/O REQUESTS

Locating a process's outstanding I/O requests is difficult, although the process's Channel Control Blocks (CCBs) include a count of outstanding I/O requests on the channel and the address of the Unit Control Block (UCB) to which the channel has been assigned. This, however, is just the beginning; I/O requests may be queued to a number of different places. Possibilities include (but are absolutely NOT limited to) the following:

1. An I/O request is generally described by an I/O Request Packet (IRP). IRPs may be queued to a list of pending IRPs for a particular unit. For a conventional device, this queue is at UCB$L_IOQFL and UCB$L_IOQBL.

2. If the IRP is the current request of a conventional device, its address is in UCB$L_IRP.

3. The IRP may be queued to the ACP or, under V4, the Files-11 XQP servicing a volume mounted on the unit.

4. The IRP may be the current request of the ACP or, under V4, the Files-11 XQP servicing a volume mounted on the unit.

5. An IRP for a file that spans a multi-volume set may be queued to a UCB describing another disk in the multi-volume set.

6. An I/O request to an SCS device (e..g, UDA disk, HSC disk, MSCP-served disk, etc.) is described by a Class Driver Request Packet (CDRP). A CDRP includes an IRP, a fork block, and space for SCS parameters. The driver is multi-threaded and handles multiple requests concurrently. Requests do not necessarily complete in the order they were queued. CDRPs in progress are queued to a Class Driver Data Block (CDDB).

7. CDRPs waiting for an unavailable resource may be in any number of different wait queues.

8. A write request to a full duplex terminal is described by a Terminal Write Packet (TWP) and queued to a different UCB queue than UCB$L_IOQFL.

9. An IRP for a V4 virtual terminal is queued to the UCB describing the physical terminal to which the virtual terminal is connected.

The directions that follow address some, but not all of the possibilities listed above. In order to locate other I/O request queues, you may have to read the relevant driver, ACP, SCS code, etc. [More information TBS]

1. First, look through the process's CCBs for any with non-zero I/O count. CCB$W_IOC is incremented by EXE$QIO and decremented by the IOC$IOPOST special kernel AST that completes post-processing

of the request. Note that CCB$W_IOC is incremented whether the request has been queued to the driver, to the ACP servicing the volume mounted on that unit, or to the Files-11 XQP servicing a V4 Files-11 ODS-2 volume mounted on that unit.

For each CCB with non-zero CCB$W_IOC, determine to which device and unit the channel is assigned. To look through the CCBs, follow the directions in subsection Channel Control Block Table in section VIRTUAL ADDRESSES - P1 SPACE.

Using the results from these commands, issue the SDA command SHOW DEVICE, specifying the device name and the unit number converted to decimal.

For some devices, SDA's SHOW DEVICE output device will be sufficient to locate the process's request. That is, the request may be the unit's current request, it may be queued to the unit's pending request list, or it may be queued to the ACP Queue Block (AQB) for the ACP servicing the volume mounted on that unit.

2. Under V4, Files-11 ODS-2 is implemented by procedure-based routines called the Files-11 XQP that run in the context of the process issuing the Files-11 request. The XQP services one request at a time. Pending IRPs are generally queued to its queue in P1 space. To determine whether the XQP is active, to format its pending I/O request queue and its current request, type the following SDA commands.

```
SDA> SHOW PROCESS              !get PCB address
SDA> READ SYS$SYSTEM:SYSDEF.STB
SDA> EXAM <PCB_address>+PCB$B_DPC
SDA> ! zero PCB$B_DPC implies no XQP activity
SDA> EXAM CTL$GL_F11BXQP       !address of XQP_QUEUE
SDA> EXAM @.                   !contents of XQP_QUEUE
SDA> ! if XQP_QUEUE doesn't contain its own address,
SDA> !   then repeat next command til back at list head
SDA> FORMAT @.                 !format each pending IRP
SDA> FORMAT @(@CTL$GL_F11BXQP+50) !format current IRP
```

For relatively short periods of time during the XQP's processing of a request, the IRP is queued on the AQB queue. At that point, the process is the only process allowed to deal with the file system cache. This is done to interlock searching the cache for a given buffer, to modify the description of what a buffer contains, etc. This interlock is only held while the buffer descriptors are modified, not while the buffer is read/written to disk, or while any of the rest of the file system manipulates the contents of the buffer.

3. The IRP in question may be the current request of an ACP servicing the volume. In both the Files-11 ODS-1 and ODS-2 ACPs there is a global IO_PACKET which contains the address of the current IRP. Look at the maps ([F11A]F11AACP.MAP and [F11B]F11BACP.MAP) in the source fiche to determine the value of

this symbol.

4. For SCS devices, such as units of a disk class driver, you will have to look further. A single device-unit may be servicing many CDRPs concurrently, which do not necessarily complete in order. CDRPs in progress are queued to the Class Driver Data Block (CDDB). A CDRP that requires an unavailable SCS resource before it can be serviced may be queued to any number of places: the Response Descriptor Table (RDT), if a response ID is needed; the Connection Descriptor Table, if a send credit or message buffer is needed; the Port Descriptor Table, if nonpaged pool is needed. [others TBS]

   To try to locate a process's CDRPs, type one of the following sets of SDA commands.

```
SDA> ! Under V3
SDA> READ SYS$SYSTEM:SYSDEF.STB
SDA> READ SYS$SYSTEM:SCSDEF.STB
SDA> SHOW DEVICE <device_name>
SDA> DEFINE UCB = <UCB address>.
SDA> DEFINE CDDB=@(<CRB address>+CRB$L_AUXSTRUC)
SDA> EXAM CDDB+CDDB$L_CDRPQFL
SDA> ! Repeat next command til back at listhead
SDA> FORMAT @./TYP=CDRP
SDA> ! Examine RDT queue, at beginning of RDT
SDA> EXAM (@SCS$GL_RDT)-18
SDA> ! If this doesn't contain its own address,
SDA> !    then repeat next command til back at listhead
SDA> FORMAT @./TYP=CDRP
SDA> ! Examine PDT queue
SDA> DEF PDT= @(UCB+UCB$L_PDT)
SDA> EXAM PDT+PDT$L_WAITQFL
SDA> ! If this doesn't contain its own address,
SDA> !    then repeat next command til back at listhead
SDA> FORMAT @./TYP=CDRP
SDA> ! Examine CDT queues
SDA> DEF UCB$L_CDT=B8
SDA> DEF CDT=@(UCB+UCB$L_CDT)
SDA> EXAM CDT+CDT$L_WAITQFL
SDA> ! If this doesn't contain its own address,
SDA> !    then repeat next command til back at listhead
SDA> FORMAT @./TYP=CDRP
SDA> EXAM CDT+CDT$L_CRWAITQFL
SDA> ! If this doesn't contain its own address,
SDA> !    then repeat next command til back at listhead
SDA> FORMAT @./TYP=CDRP
SDA>
SDA>
SDA> ! Under V4
SDA> READ SYS$SYSTEM:SYSDEF.STB
SDA> READ SYS$SYSTEM:SCSDEF.STB
SDA> !  SHOW DEVICE displays CDDB CDRP queue
SDA> SHOW DEVICE <device_name>
```

```
SDA> DEFINE UCB = <UCB_address>
SDA> !
SDA> ! Examine RDT queue, at beginning of RDT
SDA> EXAM @SCS$GL_RDT
SDA> ! If this doesn't contain its own address,
SDA> !    then repeat next command til back at listhead
SDA> FORMAT @./TYP=CDRP
SDA> ! Examine PDT queue
SDA> DEF PDT= @(UCB+UCB$L_PDT)
SDA> EXAM PDT+PDT$L_WAITQFL
SDA> ! If this doesn't contain its own address,
SDA> !    then repeat next command til back at listhead
SDA> FORMAT @./TYP=CDRP
SDA> ! Examine CDT queues
SDA> DEF CDT=@(UCB+UCB$L_CDT)
SDA> EXAM CDT+CDT$L_WAITQFL
SDA> ! If this doesn't contain its own address,
SDA> !    then repeat next command til back at listhead
SDA> FORMAT @./TYP=CDRP
SDA> EXAM CDT+CDT$L_CRWAITQFL
SDA> ! If this doesn't contain its own address,
SDA> !    then repeat next command til back at listhead
SDA> FORMAT @./TYP=CDRP
```

If you find CDRPs queued to the RDT, this may mean that SYSBOOT parameter SCSRESPCNT is set too low. It could also mean that, because of some other problem, previously issued SCS requests are not completing. You may want to alter SCSRESPCNT and reboot the system. If you find CDRPs queued to PDT$L_WAITQFL, this means that there isn't enough nonpaged pool. Check with the DCL command SHOW MEMORY, alter parameters as necessary, and reboot. [more information TBS]

5. If the device is a V4 virtual terminal, VTA, look at the SHOW DEVICE display to see whether it is connected. The second line of the characteristics will include "det" if the virtual terminal is disconnected from a physical terminal. Any pending IRPs to a disconnected virtual terminal are queued to the virtual terminal pending I/O queue, at UCB$L_IOQFL. If the virtual terminal is connected to a physical terminal, UCB$L_TL_PHYUCB contains its address, and I/O requests are queued to the physical UCB.

6. If the device is a terminal device set to full duplex, the IRP will not be queued to the usual UCB I/O pending queue. Instead, the IRP will be pointed to by a terminal write packet (TWP). The current write request is pointed to by UCB$L_TT_WRTBUF; pending write requests are queued to the UCB at UCB$L_TT_WFLINK. UCB$L_TT_WRTBUF is not cleared at I/O completion; it may, therefore, contain a stale TWP address. If the write state (TTY$V_ST_WRITE) is set in the terminal state quadword (UCB$Q_TTSTATE), then the contents of UCB$L_TT_WRTBUF are valid. Type the following commands to examine these fields.

```
SDA> CTRL/Y
```

```
$ SPAWN
$ MACRO/OBJ=SYS$LOGIN:TTYDEF  SYS$INPUT: -
$_ + SYS$LIBRARY:LIB/LIB
    $TTYDEF GLOBAL
    .END
CTRL/Z
$ LO
$ CONT
SDA> READ SYS$LOGIN:TTYDEF.OBJ
SDA> !If device is a V4 connected VTA, then
SDA> DEF UCB=@(<ucb_address>+UCB$L_TL_PHYUCB)
SDA> !  else, do next command
SDA> DEF UCB=<ucb_address>
SDA> EXAM UCB+UCB$Q_TT_STATE
SDA> EVAL TTY$V_ST_WRITE
SDA> !If TTY$V_ST_WRITE bit clear in UCB$Q_TT_STATE,
SDA> !  then there are no write commands queued to
SDA> !  the terminal, and you're done
SDA> DEF TWP= @(<ucb_address> + UCB$L_TT_WRTBUF)
SDA> !See if this TWP is a broadcast or normal write
SDA> EXAM TWP+TTY$L_WB_IRP   !if zero, then ignore
SDA> FORMAT @.               !  else, format IRP
SDA> DEF TWP=@( <ucb_address>+UCB$L_TT_WFLINK)
SDA> EXAM TWP+TTY$L_WB_IRP !if zero, then ignore
SDA> FORMAT @.               !  else, format IRP
SDA> DEF TWP=@TWP            !flink to next TWP...
SDA> !  Continue til back at listhead
```

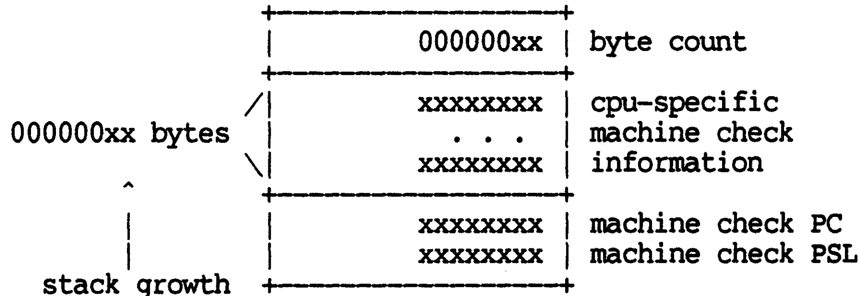
MACHINE CHECKS

A machine check is an exception reported when the cpu microcode detects an internal error. Machine check errors are cpu-specific; possible machine check errors include cache parity error, translation buffer parity error, and cpu timeout. Almost all machine checks represent potentially serious problems. Many machine checks are caused by some type of hardware problem; some can be caused by errors in user-written software or VMS.

The following external symptoms indicate that a system is incurring machine checks

- o the error message SYSTEM-F-MCHECK
- o the fatal bugcheck MACHINECHK
- o a console halt code of 5, double error halt (or equivalent message, depending on cpu type)
- o machine check entries in the error log
- o non-zero CPU errors as displayed by the DCL command SHOW ERROR.

During machine check exceptions, the microcode logs information, called the microcode machine check logout, on the interrupt stack. The machine check logout identifies the type of machine check and includes the contents of relevant cpu registers. The exact layout and contents of the logout are cpu-specific. The generic microcode machine check error logout is shown below. The byte count describes the size of the cpu-specific information and does not include the PC, the PSL, and the longword containing the byte count.



The microcode then vectors through the System Control Block (SCB) vector at offset 4 to a VMS machine check exception service routine. VMS sets the low order bit in the SCB vector, specifying that the exception be serviced on the interrupt stack and at IPL 31. The VMS machine check exception service routine is in the cpu-specific module [SYSLOA]MCHECKxxx, which is loaded into nonpaged pool as part of a cpu-specific image named SYSLOAxxx.EXE

See the subsection SYSLOAXXX.EXE in the section VIRTUAL ADDRESSES - SYSTEM SPACE for more information on the mechanism for dispatching into SYSLOAxxx.EXE and on the names of the SYSLOAxxx.EXE images.

VMS first determines which type of machine check occurred from the

machine check logout. Although VMS treats each type of machine check somewhat differently, its general response is to log a machine check error and increment EXE$GL_MCHKERRS. The DCL command SHOW ERROR displays the contents of EXE$GL_MCHKERRS as CPU errors.

VMS then determines whether the error is recoverable or not. Recoverability is minimally a function of two things: whether the machine check exception was a fault or an abort and whether the instruction is resumable. Whether an instruction is resumable or not is a function of the microcode that implements it.

If the machine check is recoverable, VMS dismisses the exception, resuming the thread of execution that incurred the machine check.

If the machine check is nonrecoverable and occurred in either of the outer modes, VMS signals a machine check exception to that mode. If the process has declared no condition handler for machine checks, the last chance handler generates the following error message

```
SYSTEM-F-MCHECK, detected hardware error, PC=xxxxxxxx, PSL=xxxxxxxx
```

If the machine check is nonrecoverable and occurred in kernel or exec mode, VMS signals a fatal MACHINECHK bugcheck.

Typically, machine checks are caused by hardware faults, but they may also occur as the result of software error. Follow the directions below to analyze a MACHINECHK bugcheck.

1. If you have the ERRORLOG.SYS file in use at the time of the crash, use SYE to decode the machine check information. With V3, specify S in response to SYE's "OPTIONS" prompt and /CP in response to the "DEVICE NAME" prompt. With V4, type the DCL command ANALYZE/ERROR/INCL=MACHINE_CHECK <file_spec>.

2. If you don't have ERRORLOG.SYS, you must decode the machine check information yourself. If you don't know the cpu type, type the SDA command EXAMINE EXE$GB_CPUDATA to display the contents of the processor ID register. The high-order byte displayed is processor type. If the high-order byte is a 1, the processor may be a VAX-11/780 or a VAX-11/785. Bit 23 (decimal) of the displayed longword specifies which processor type it is: for a VAX-11/780 bit 23 is 0; for a VAX-11/785 bit 23 is 1. The VAX-11/785 machine check logout is identical to that of the VAX-11/780. See the table below to interpret the other processor type values and to determine the corresponding machine check error logout size. Note that a "VAX-11/725" is really a VAX-11/730 cpu and that a VAX-11/782 is two VAX-11/780s.

EXE$GB_CPUDATA	CPU TYPE	HEX BYTE COUNT
1	VAX-11/780 & VAX-11/785	28
2	VAX-11/750	28
3	VAX-11/730	C
4	VAX 8600	58

7	MicroVAX I	C
8	MicroVAX II	C

3. The summary parameter, the first longword of cpu-specific information, is the key to an intelligent guess about whether the problem is hardware related. Likely hardware problems are values such as "translation buffer parity error fault", "cache parity error fault", "control store parity fault", "read data substitute fault", "microcode not supposed to get here abort". "Read timeouts" are sometimes caused by software specifying erroneous I/O space addresses.

   See section MACHINE CHECKS - <cpu_type> for further information on specific types of machine checks, or see below Additional References for documentation on the cpu-specific microcode machine check logout. Using the machine check logout documentation and processor register layouts, decode the machine check logout.

4. The PC displayed by the SDA SHOW CRASH command reflects machine check exception processing rather than the location of the machine check, and the PSL displayed has been altered by the machine check exception.

   In the stack layout above, the machine check PC is the address of the instruction whose [attempted] execution resulted in the machine check exception. This is of particular interest if you suspect a software-induced machine check. Identify in what source module the PC is. See section VIRTUAL ADDRESSES. Often examining instructions around the PC is helpful enough to eliminate a microfiche search. Try the SDA command

   ```
   SDA> EXAMINE/INSTRUCTION <machine_check_pc>-20;30
   ```

5. If you suspect a software problem, decipher the stack of the access mode that incurred the machine check to trace control flow. Decode the machine check PSL to get the current mode field. Use the layout in the section RELATED REFERENCE MATERIAL or the V4 SDA command EXAMINE/PSL. Then, enter the SDA command SHOW STACK/<current_mode>. See the section STACK PATTERNS.

6. Call Field Service about machine checks which seem to be hardware related.

Hints And Kinks

1. Note that for each V3 SDA COPY command used to copy the dump, the SP will be 8 bytes greater than its actual value; that is, SDA will show the SP pointing to a stack address 8 bytes higher than it should. This V3 bug has been corrected in V4.

2. The VAX instruction set is sufficiently rich that most random data can be interpreted as instructions. Most system code deals with binary integer and character data. This means that if an EXAMINE/INSTRUCTION display includes many packed decimal and/or floating point instructions, you are probably examining a data area or using a start address which is not an instruction boundary.

   One common error that results in a nonsensical display is to examine instructions in the bugcheck overlay area. During a crash, fatal bugcheck code and message text overlay resident system image code, beginning one page before label BUG$FATAL, for a length of about 12000 decimal or 3000 hex bytes.

3. When SDA examines the process current at the time of an interrupt stack bugcheck, SDA assumes the bugcheck PC and PSL and all the general registers are part of that process's context and displays them in response to the SHOW PROCESS/REGISTER command.

4. If you're looking at a dump with more than one machine check logout on the stack and the newest one doesn't make sense, examine the earlier ones for a clue about what the problem is.

Additional References

V3 VAX/VMS Internals and Data Structure Manual, Section 8.3, Machine Check Mechanism

VAX/VMS Error Log Utility Reference Manual

VAX-11/750 Self Maintenance Diagnostic Guide booklet, for VAX-11/750 microcode machine check error logout and processor register layout. Also, VAX-11/750 VAX Maintenance Handbook (pp. 267-269 of the 3/83 edition) for microcode machine check logout and processor registers and (pp. 239 - 242 of the 3/83 edition) for a discussion of evalutating the microcode logout. Also, see VMS module [SYSLOA]MCHECK750.

VAX Architecture Standard, Rev. 7 (DEC Standard 032) p. 12-26, for VAX-11/730 microcode machine check error logout. Also, see VMS module [SYSLOA]MCHECK730.

MicroVAX I CPU Technical Description (pp. 2-62 through 2-66 of the 8/84 edition). Also, see VMS module [SYSLOA]MCHECKUV1.

KA630-A CPU Module User's Guide (pages 5-11 through 5-12 of the 4/53 edition). Also, see VMS module [SYSLOA]MCHECKUV2.

MACHINE CHECKS - VAX-11/780 AND VAX-11/785

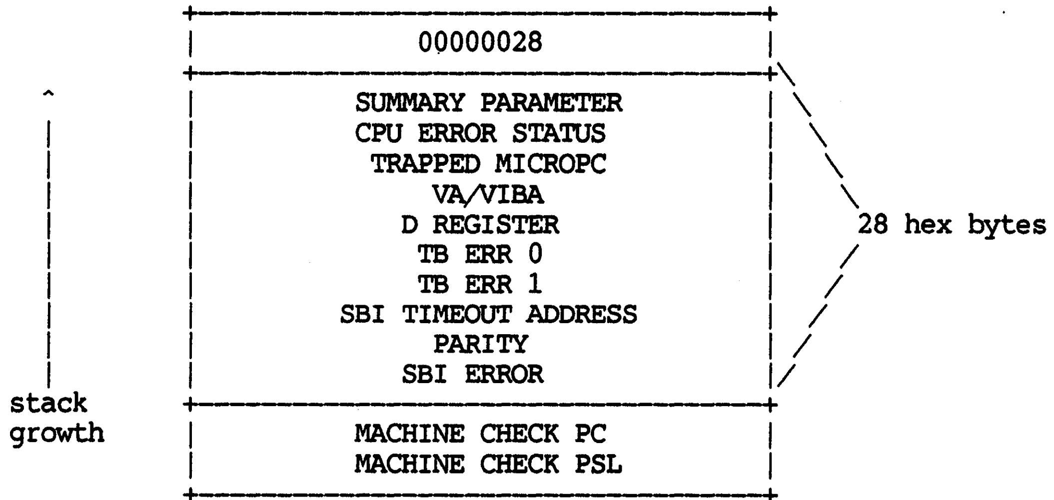
During machine check exceptions, the microcode logs information, called the microcode machine check logout, on the interrupt stack. The machine check logout identifies the type of machine check and includes the contents of relevant cpu registers.

If another machine check occurs before the microcode has serviced the first, the cpu is halted with a double error halt. You should see the following halt message printed on the console terminal

?CPU DBLE-ERR HALT

The second error may occur before or during the time when the microcode writes the machine check logout on the stack; therefore, you must obtain the type of machine check and register contents from ID bus temporaries and various internal registers rather than from the stack. See section CPU HALTS - VAX-11/780 AND VAX-11/785 for more information.

The microcode machine check logout follows



The summary parameter, the first longword of cpu-specific information, describes the type of machine check. Bytes 0 and 1 of the summary parameter are stored by the machine check microcode. VMS stores information into bytes 2 and 3 on certain kinds of errors. The summary parameter layout follows

Byte 0	Error code (See the table below)
Byte 1	A non-zero value means there was a cpu timeout or cpu error confirm pending at the time of the machine check
Byte 2	During control store parity errors and "microcode not supposed to be here" errors, this contains the opcode of the instruction executing at the time of error
Byte 3	During cache parity errors, this contains the cache disable flag (1=group 0, 2=group 1)

Examine the summary parameter on the stack or in the error log entry, locate its low order byte in the table below, and read the subsection below on that error for more information. For a description of general VMS machine check servicing, see section MACHINE CHECKS if you haven't already. See the VAX-11/780 Maintenance Handbook reference listed in subsection Additional References for information on decoding the rest of the microcode machine check logout.

In the table below "CP" refers to memory references explicitly requested by microcode, and "IB" refers to memory reads generated by the instruction buffer in the process of prefetching the instruction stream. In the table below, a "fault" is an error that may be retriable; an "abort" is an error that is not retriable.

Summary Parameter Error Code Values

Byte 0	Error Name
00	CP Read Timeout/Error Confirm Fault
02	CP Translation Buffer Parity Error Fault
03	CP Cache Parity Error Fault
05	CP Read Data Substitute Fault
0A	IB Translation Buffer Parity Error Fault
0C	IB Read Data Substitute Fault
0D	IB Read Timeout/Error Confirm Fault
0F	IB Cache Parity Error Fault
F0	CP Read Timeout/Error Confirm Abort
F1	Control Store Parity Error Abort
F2	CP Translation Buffer Parity Error Abort
F3	CP Cache Parity Error Abort
F5	CP Read Data Substitute Abort
F6	Microcode not Supposed to Be Here Abort

Read Data Substitute Error

A read data substitute (RDS) error occurs when the processor is performing a read or an interlocked read on the SBI and the memory controller returns uncorrected read data. That is, a multiple-bit error in memory contents occurs which cannot be corrected by the memory controller's error checking and correction logic. These errors are never the result of software problems (except for the rare possibility that a privileged user forces them through manipulation of the memory controller registers). These errors can be caused by hardware problems in memory array boards or in the memory controller.

A read data substitute error can be reported through either a machine check or an interrupt. If the cpu attempts to use uncorrected read data in instruction execution, a machine check occurs. If instruction execution alters control flow so that uncorrected read data in the instruction buffer is unused, an RDS interrupt occurs. (See section CPU-SPECIFIC INTERRUPTS - VAX-11/780 AND VAX-11/785 for information on RDS and other cpu-specific interrupts.)

In response to an RDS machine check, VMS increments EXE$GL_MEMERRS and logs the read data substitute error in the error log. The DCL command SHOW ERROR displays the contents of EXE$GL_MEMERRS as MEMORY errors. The error log entry includes the memory controller registers, one of which contains the physical address in error. The error log entry is type HE, hard memory error. These error log entry types are in the V3 SYE display from the /ME qualifier or in the V4 display from ANALYZE/ERRORLOG/INCLUDE=MEMORY. The error log entry also includes any previously unrecorded errors from all the other memory controllers on the system.

Whenever VMS discovers an error on a particular memory controller, it increments a counter, local to SYSLOA780, for that controller. The counter records the number of corrected read data (CRD) and RDS errors on that controller during the current fifteen-minute period. When a particular controller has more than 3 CRD and RDS errors within a fifteen-minute period, VMS disables CRD interrupts for that controller for the remainder of the fifteen-minute period.

Once a minute, VMS scans all memory controllers for unreported CRD errors so that (some) errors on controllers with disabled CRD interrupts can still be logged. If, however, a particular controller has more than than 6 CRD and RDS errors within fifteen-minute period, any CRD errors it has during the remainder of the fifteen-minute period are not logged. This prevents filling the error log with CRD error entries. Every fifteen minutes all the counters are reset to zero.

To examine the array of these counters (which is post-indexed by memory controller nexus number), type the following SDA commands under VMS V4

```
SDA> EXAM EXE$MCHK_ERRCNT
SDA> EXAM @.+10;10
```

After logging the memory error, VMS also logs a machine check, increments EXE$GL_MCHKERRS, and determines whether the read data substitute error is recoverable. Because recovery from a read data substitute error requires paging in a fresh copy of the virtual page, recoverability is based on a number of factors: at what IPL the memory reference was made, whether the error was a fault or an abort, whether the page has a PFN database entry, whether the page had been modified, whether the page is global, whether there was I/O in progress to the page, and whether the instruction is resumable.

If the RDS machine check is recoverable, VMS removes the page from whatever working set list it is in and places it on the bad page list. When VMS REIs from the machine check exception, the process pagefaults attempting to execute the instruction that incurred the RDS error. Pagefault servicing reads in a fresh copy of the virtual page.

If the machine check is nonrecoverable and occurred in either of the outer modes, VMS signals a machine check exception to that mode. If

the machine check is nonrecoverable and occurred in kernel or exec mode, VMS signals a fatal MACHINECHK bugcheck. In all cases, the microcode machine check logout is recorded in the error log entry.

To learn more about the read data substitute error, examine the memory error log entry. Also, if you're looking at a dump, you can examine the contents of the VA/VIBA register in the stack to get the approximate virtual address of the memory error. Then follow the directions below in subsection Read Timeout or Error Confirm Error to translate that virtual address into a physical address.

In the case of a double error halt, a second machine check occurs before the first is serviced. Thus, there will be no error log entries for the two errors. However, if one of the machine checks is a read data substitute error, the physical address in error may still be in the memory controller register (Configuration Register C for a MS780C controller, Configuration Register C or D for a MS780E controller) at the time of the halt.

Translation Buffer Parity Error

The translation buffer is a two-way set associative cache of page table entries. The translation buffer on a VAX-11/780 contains 128 entries, half of which are reserved for system space page table entries and half for process space page table entries. A VAX-11/785 translation buffer has 512 entries. Each of the entries contains a tag field with address, parity, protection, modify status, and validity information, and a data field with a page frame number and parity information.

When a virtual address reference from a VAX-11/780 is translated to a physical address, bits 9-13 of the virtual address are used to index entries in the two groups of the translation buffer. Bit 31 selects for system space page table entry or process space. Bits 14-30 (the "tag field") of the page table entries whose contents are stored at those entries are compared to those of the virtual address being translated to determine if there is a translation buffer "hit" or "miss". On a VAX-11/785, bits 9-15 of the virtual address are index field, and bits 16-30 of the address are tag field.

On a hit, the page frame number and protection information are retrieved from the cache; on a miss, the microcode performs the address translation and replaces one of the two entries with the contents of the page table entry it accessed from memory. The system cannot run with both groups of translation buffer disabled. Although it is possible to disable a translation buffer group through console commands, it is not possible to disable it from software running on the VAX. Since translation buffer parity failures are generally random, disabling half of the cache from the console is not a useful workaround.

A translation buffer parity error occurs when the processor is

translating a virtual address to resolve a virtual memory reference and a parity error occurs on data read out of the translation buffer or on the tag field. These errors are never the result of software problems and can only be caused by hardware problems.

When this type of machine check occurs, VMS invalidates all entries in the translation buffer, logs the translation buffer parity error in the error log (with type MC, machine check), and increments EXE$GL_MCHKERRS. VMS then determines whether the error is recoverable. Recoverability is based on whether the error is an abort or a fault, and, if a fault, whether the instruction is resumable.

If the machine check is nonrecoverable and occurred in either of the outer modes, VMS signals a machine check exception to that mode. If the machine check is nonrecoverable and occured in kernel or exec mode, VMS signals a fatal MACHINECHK bugcheck. On a MACHINECHK bugcheck, the microcode machine check logout is visible on the interrupt stack. In all cases, the microcode machine check logout is recorded in the error log entry.

Examine the contents of the TB ERR1 register on the stack or in the error log entry. Bit 8 set means that there has been a parity error. If any of bits 9-14 are set, there has been a tag parity error. Most of the tag parity logic is in the CAM board (M8220 on a VAX-11/780, M7462 on a VAX-11/785). If any of bits 15-20 are set, there has been a data parity error. Most of the data parity logic is in the TBM board (M8222 on a VAX-11/780, M7464 on a VAX-11/785).

Cache Parity Error

Cache is a two-way set associative buffer for the storage of memory contents most likely to be accessed next. Its access time is considerably shorter than that of main memory. Each of the two groups of cache memory locations contains 512 entries on a VAX-11/780. On a VAX-11/785, each group contains 2048 entries. Each of the entries contains a tag field with address, parity, and validity information and a data field with eight bytes of data and eight bits of parity.

When a VAX-11/780 cpu references memory, bits 3-11 (the "index field") of the 30-bit physical address are used to index entries in the two groups of cache. Bits 12-29 (the "tag field") of the physical addresses whose contents are stored at those entries are compared to those of the address being referenced to determine whether this is a cache "hit" or a cache "miss". On a VAX-11/785, bits 3-13 of the physical address are index field, and bits 14-29 of the address are tag field.

On a hit, the contents of the physical address are retrieved from cache; on a miss, one of the two entries is replaced with the contents of the quadword containing the physical address being

referenced by the cpu. Each group of cache can be disabled independently, and the system can run with both groups of cache disabled.

A cache parity error occurs when the processor is performing a read memory reference and a parity error is detected on either data read out of the cache or on the tag field. These errors are never the result of software problems (except for the rare possibility that a privileged user forces them through manipulation of processor registers). These errors can be caused by hardware problems.

When this type of machine check occurs, VMS determines whether there have been three cache parity errors within the last 100 milleseconds. If not, VMS restores the cache to its previous state of enables, logs the cache parity error in the error log (with type MC, machine check), increments EXE$GL_MCHKERRS, and determines whether the error is recoverable. Recoverability is based on whether the error is an abort or a fault and, if a fault, whether the instruction is resumable. If there have been three cache parity errors within the last 100 milleseconds, VMS determines which cache group is currently in error, forces misses in both groups to cause replacement of all entries, and enables replacements only in the cache group which didn't have the last error. This means that the last cache group to have an error is disabled, even if previous errors occurred in the other cache group. VMS records in byte 3 of the machine check summary longword which group of cache, if either, it disabled.

Examine the contents of the PARITY register on the stack or in the error log entry. Bit 15 set means that there has been a parity error. If any of bits 6-13 is clear (note clear, not set!), there has been a data parity error. Most of the cache parity logic is in the CDM board (M8221 on a VAX-11/780, M7463 on a VAX-11/785). If any of bits 0-5 is clear (!), there has been a tag parity error. Most of the tag parity logic is in the CAM board (M8220 on a VAX-11/780, M7462 on a VAX-11/785).

Control Store Parity Error

The basic microprogram of the VAX-11/780 is contained in a 4k 99-bit PROM control store (PCS). The 99-bit control word contains 96 data bits and 3 parity bits. The system also contains a 2K 99-bit writable diagnostic control store (WCS) used for microdiagnostics and updates to the microprogram in PCS and, optionally, another 2K 99-bit writable control store for G and H floating point support and/or customer-written microcode. When the system is powered up, the console subsystem loads the writable control store from a file on the console floppy. Micropc addresses between 0 and FFF select microwords in PCS; micropc addresses between 1000 and 17FF select microwords in WCS, and addresses 1800-1FFF select the optional writable control store. G and H floating point support is between 1800 and 1BFF, if it is present.

The basic microprogram of the VAX-11/785 is contained in a 512 microword PROM control store (PCS) and a 7.5K microword writable control store (WCS).

Parity is checked on each microword read from PCS or WCS. A control store parity error occurs when a parity error is detected on a microword from PCS or WCS. These errors are never the result of software problems and can only be caused by hardware problems in the microstore boards.

When this type of machine check occurs, VMS records the opcode of the instruction being executed in byte 2 of the machine check summary longword.

VMS logs the control store parity error in the error log (with type MC, machine check), increments EXE$GL_MCHKERRS, and determines whether to bugcheck based on the access mode in which the error occurred. If the control store parity error machine check occurred in either of the outer modes, VMS signals a machine check exception to that mode. If the control store parity error machine check occurred in kernel or exec mode, VMS signals a fatal MACHINECHK bugcheck. On a MACHINECHK bugcheck, the microcode machine check logout is visible on the stack. In all cases, the microcode machine check logout is recorded in the error log entry.

Examine the TRAPPED MICROPC and CPU ERROR STATUS registers on the stack or in the error log entry. Bit 15 in the CPU Error Status register indicates that there has been a control store parity error. Bits 12-14 indicate which group is in error. The parity checking logic is part of the PCS board; this means that WCS parity errors can result from hardware problems on the PCS board.

If you see this error, you might power down the LSI to cause the WCS to be reloaded. Open the cpu cabinet and locate the LSI in the lower left hand corner. Its leftmost switch, labeled DC ON/OFF is the power switch. Toggling the switch will powerfail the LSI and the VAX-11/780 or VAX-11/785. If the system still incurs control store parity errors after WCS has been reloaded, you might try replacing the console floppy and reloading WCS from the new floppy in case a floppy disk error has resulted in alterations to WCSxxx.PAT.

Microcode Not Supposed To Be Here

A microcode not supposed to be here error occurs when microcode detects that it has arrived at an illegal microaddress. These errors are never the result of software problems. These errors can be caused by various hardware problems as well as simply a bad copy of WCS from the floppy.

When this type of machine check occurs, VMS records the opcode of the instruction being executed in byte 2 of the machine check summary longword. The opcode might be useful information if it is the same

opcode in most or all of these exceptions on a system incurring many microcode not supposed to be here machine checks.

VMS then logs the microcode not supposed to be here error in the error log (with type MC, machine check), increments EXE$GL_MCHKERRS, and determines whether to bugcheck based on the access mode in which the error occurred. If the machine check is nonrecoverable and occurred in either of the outer modes, VMS signals a machine check exception to that mode. If the machine check is nonrecoverable and occured in kernel or exec mode, VMS signals a fatal MACHINECHK bugcheck. On a MACHINECHK bugcheck, the microcode machine check logout is visible on the stack. In all cases, the microcode machine check logout is recorded in the error log entry.

If you see this error, you might power down the LSI to cause the WCS to be reloaded. Open the cpu cabinet and locate the LSI is on the lower left hand corner. Its leftmost switch, labeled DC ON/OFF is the power switch. Toggling the switch will powerfail the LSI and the VAX-11/780 or VAX-11/785. See if the system will run without machine checks for a while before reloading any customer-written microcode. If the the system still incurs these errors after WCS has been reloaded, you might try replacing the console floppy and reloading WCS in case a floppy disk error has resulted in alterations to WCSxxx.PAT. Locate a known good console floppy, copy its WCSxxx.PAT file to the console floppy, shut down the system, power it off, and power it back on.

If the problem persists, there may be problems in the PCS or WCS boards, the microsequencer logic, the clock board, the CIB board, or the LSI subsystem.

Read Timeout Or Error Confirm Error

A read timeout error occurs when the cpu is performing a read or interlocked read on the SBI and either there is no response to the cpu's command within 512 SBI cycles, the cpu bus control logic could not gain access to the SBI, or the addressed nexus responded with BUSY for 512 cycles. An SBI error confirm occurs when the cpu is performing a read or interlocked read on the SBI, and the target nexus responds with an error confirm because the command is in error. Both read timeouts and error confirms can occur as the result of data path action or instruction buffer prefetch.

These errors can be caused by software problems, for example, word references to MASSBUS adapter or UNIBUS adapter registers, longword references to UNIBUS address space, references to nonexistent physical memory, refences to nonexistent adapter register space, or page table corruption. A read timeout can also be caused by a program which issues an incorrect $CRMPSC system service that does PFN-mapping to a nonexistent PFN and which then references the section. Hardware problems that can cause these errors include problems in nexus, memory controllers, and the translation buffer.

When this type of machine check occurs, VMS logs the error in the error log (with type MC, machine check) and increments EXE$GL_MCHKERRS. It then tests whether the PC of the machine check and virtual address referenced match particular instructions in the UBA interrupt service routines that access the BRRVR registers. These instructions generate UNIBUS bus grants and obtain the vector of the device making the bus request. A timeout on one of these instructions is dismissed, to minimize the possibility that a device hung in interrupt sequence or a lost UNIBUS bus grant causes a system crash.

If the machine check does not match the UBA interrupt service routine test, VMS determines whether this is the second timeout or error confirm within 10 milliseconds. If there have been two timeouts or error confirms within 10 milliseconds, if the error is an abort, or if the instruction is not resumable, VMS determines whether to bugcheck based on the access mode in which the error occurred. If the error occurred in supervisor or user mode, VMS signals a machine check exception to that access mode. If the error occurred in kernel or exec mode, VMS signals a fatal MACHINECHK bugcheck.

Examine the SBI Error Register on the stack or in the error log entry to distinguish between read timeout and error confirm errors. Bit 6 is set if an instruction buffer requested cycle timed out. Bit 12 is set if a cpu requested cycle timed out. Bit 3 of the SBI Error Register is set if an instruction buffer requested cycle received an error confirm to a command. Bit 8 of the register is set if a cpu requested cycle received an error confirm to a command.

Read Timeout Error

If a timeout occurred, examine the contents of the SBI Timeout Address (SBITA) on the stack or in the error log entry to determine the address which caused the timeout. This address may give some indication of the cause of the problem. The SBITA contents are a physical SBI address, the address of a longword, and must be shifted two bits to the left to get the physical address in byte form. Bits 31 and 30 of the SBITA indicate the access mode of the request. Bit 29 is the protection check bit. Bit 27 of the SBITA set to 1 indicates an I/O space address.

Physical addresses below 20000000 (byte address) are memory addresses. The address range above that is reserved for nexus register space and UNIBUS space. If the converted SBITA address is in I/O space, converting it to a particular nexus address may be helpful.

The space for nexus 1 registers begins at 20002000; the space for nexus 2, at 20004000; the space for nexus 3, 20006000; etc. There are four address ranges reserved for UNIBUS space: the first UNIBUS space begins at 20100000; the second, at 20140000; the third, at 20180000; and the fourth, at 201C0000.

To determine what nexus are present on the running system, type the following commands.

```
$ MC SYSGEN
SYSGEN> SHOW /ADAPTER
SYSGEN> EXIT
```

SYSGEN displays nexus numbers as decimal numbers.

To determine what nexus are present on a system represented by a crash dump, examine and interpret the contents of the byte array whose address is in EXE$GL_CONFREG. This array contains one byte of adapter type code, indexed by nexus number. The adapter type codes are defined by the SYS$LIBRARY:LIB.MLB macro $NDTDEF. Display the adapter type codes and the array contents with the following commands.

```
SDA> CTRL/Y
$ SPAWN
$ LIBR/OUT=TT:/EXTRACT=$NDTDEF SYS$LIBRARY:LIB/MACRO
$ LOGOUT
$ CONTINUE
SDA> EXAM @EXE$GL_CONFREG;@EXE$GL_NUMNEXUS
```

Error Confirm Error

If an error confirm occurred, examine the longword VA/VIBA on the stack or in the error log entry to determine the virtual address reference which caused the error confirm.

The VA, the Virtual Address register, contains the address of the memory data referenced by the cpu which is to be read or written into the cpu. The VA generally contains a virtual address which must be translated to physical.

The VIBA, Virtual Instruction Buffer Address, contains the address of the instruction stream data which is to be loaded into the instruction buffer.

If you are analyzing a crash dump, you can translate the VA/VIBA contents on the stack to a physical address. If the VA/VIBA contents are a system space address, type the following SDA command to display and format the PTE corresponding to that address

```
SDA> SHOW PAGE/SYSTEM <virtual_address>;200
```

V3 SDA displays only entire process page tables. If the faulting virtual address is in process space, first convert the faulting virtual address to a virtual page number by issuing the following DCL commands>

```
$ VPN = (%X<virtual_address> .AND. %X3FFFFE00) / 512
$ SHOW SYMBOL VPN
```

Then select the SDA symbol P0BR or P1BR, based on whether the address is in P0 or P1 space. Issue the SDA command

```
SDA> EXAM @<PxBR> + (4*<VPN>)
```

V4 SDA can display a range of process PTEs. Type the command

```
SDA> SHOW PROC/PAGE <virtual_address>;200.
```

The low 21 bits of the page table entry are the page frame number (PFN). Multiply the PFN by 200 hex (bytes per page) and add the low order 9 bits of the virtual address. Type the following commands.

```
SDA> CTRL/Y
$ SPAWN
$ BOFF = (%X<virtual_address> .AND. %X000001FF)
$ LOGOUT
$ CONTINUE
SDA> DEF PHYS_ADDRESS = <pfn>*200 + <BOFF>
SDA> EVAL PHYS_ADDRESS
```

Hints And Kinks

1. If you are looking at a dump with more than one machine check logout on the stack and the newest one makes no sense, examine the earlier ones for a clue about what the problem is.

2. If a machine check occurs during the execution of kernel mode code which has protected itself by declaring a "machine check recovery block", VMS dismisses the exception without logging an error or incrementing any counters. See the VAX/VMS Internals and Data Structures reference below for more information.

3. The console block storage medium has an RT-11 file structure. The RT-11 file structure implements three different record formats: stream ASCII, formatted binary, and fixed-length record. Under VMS you use the V3 FLX utility or the V4 EXCHANGE utility to transfer files to and from the console.

   Both FLX and EXCHANGE select a default record transfer mode based on file extension type. For example, extensions of OBJ and BIN default as EXCHANGE /RECORD=BINARY and FLX /FB transfer modes.

   Occasionally the default based on file extension type is inconsistent with the file's record format. In particular, CI780.BIN, the CI microcode; WCSxxx.PAT, the VAX-11/780 microcode; and PCS750.BIN, the VAX-11/750 microcode, will not be copied correctly unless you override the default transfer mode.

If you are not sure what the transfer mode should be, you can use the EXCHANGE qualifier /RECORD_FORMAT=STREAM or the FLX switch /FA for all text files (e.g. command files). Use the EXCHANGE qualifier /RECORD_FORMAT=FIXED (or /TRANSFER_MODE=BLOCK) or the FLX switch /IM for all other files (binary files such as images, microcode files, patch files). The VMS console contains no formatted binary files, so you will never want /RECORD_FORMAT=BINARY or FLX's /FB.

Additional References

VAX-11/780 VAX Maintenance Handbook (pp. 6-27 through 6-30 of the 12/78 edition, pp. 159-160 of the 8/82 edition), for VAX-11/780 microcode machine check error logout. The processor registers displayed as part of the stack are also documented there (pp. 3-26 through 3-48 in 12/78 edition, pp. 133-154 of the 8/82 edition).

VAX-11/780 VAX Maintenance Handbook (pp. 197-199 of the 8/82 edition) for a description of MS780C memory controller registers

VAX-11/780 Data Path Manual, Chapter 6, Machine Check Abort/Fault/Halt.

VAX-11/780 TB/Cache/SBI Control Technical Description

V3 VAX/VMS Internals and Data Structures, Section 8.3.4, Machine Check Recovery Blocks

VMS module [SYSLOA]MCHECK780.

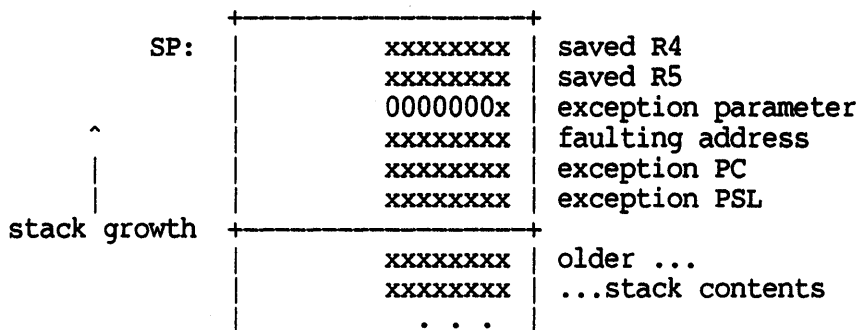
PGFIPLHI BUGCHECK

The PGFIPLHI bugcheck is signaled by MMG$PAGEFAULT, the exception service routine for translation not valid faults (pagefaults). After saving R5 and R4 on the stack, MMG$PAGEFAULT tests whether the pagefault occurred either on the interrupt stack or at an IPL above 2. If either is true, MMG$PAGEFAULT signals this fatal bugcheck.

The PC displayed by the SDA SHOW CRASH command reflects the location of the bugcheck rather than the location of the pagefault. The location of the the pagefault can be obtained as described below.

When this bugcheck is signaled, the translation not valid exception microcode has recorded on the stack the exception PC and PSL, the faulting virtual address, and a longword with more information about the pagefault.

Sometimes this bugcheck is due to a software fault in a rarely-taken error path which erroneously touches nonvalid pages at high ipl. Sometimes this bugcheck occurs when system software uses a data structure field containing an address which has been corrupted by some earlier problem.



1. Using the SP, locate the pagefault exception information.

2. The exception parameter contains 2 bits with additional information about the pagefault. Bit 1 set to 1 means that the fault occurred during the reference to the process page table associated with the faulting virtual address. Bit 2 set to 1 means that the intended access was a modify or write. Bit 2 equal to 0 means the program's intended access was a read.

3. The faulting address is a location in the page whose access caused the pagefault. Identify in what source module it is or what data structure. See section VIRTUAL ADDRESSES. Use bit 1 of the exception parameter to determine whether the PTE for this address is invalid or whether the page that maps the page table containing the PTE is invalid.

4. Sometimes additional useful information can be obtained by examining the invalid PTE. If the faulting virtual address is in system space, SDA formats the PTE if you issue the command

```
SDA> SHOW PAGE/SYSTEM <faulting_address>;200
```

V3 SDA displays only entire process page tables. If the faulting virtual address is in process space, first convert the faulting virtual address to a virtual page number by issuing the following DCL commands

```
$ VPN = (%X<faulting_address> .AND. %X3FFFFE00) / 512
$ SHOW SYMBOL VPN
```

Then select the SDA symbol P0BR or P1BR, based on whether the address is in P0 or P1 space. Issue the SDA command

```
SDA> EXAM @<PxBR> + (4*<VPN>)
```

See the Internals and Data Structure Manual reference below for information to enable you to decode the PTE.

V4 SDA can display a range of process PTEs. Type the command

```
SDA> SHOW PROC/PAGE <faulting_address>;200.
```

5. The exception pc is the instruction whose attempted execution resulted in the pagefault. Identify in what source module the PC is. See section VIRTUAL ADDRESSES. Often examining instructions around the PC is helpful enough to eliminate a microfiche search. Try the SDA command

```
SDA> EXAMINE/INSTRUCTION <exception_pc>-20;30
```

6. Decipher the current stack to trace control flow. See section STACK PATTERNS.

Hints And Kinks

1. Note that for each V3 SDA COPY command used to copy the dump, the SP will be 8 bytes greater than its actual value; that is, SDA will show the SP pointing to a stack address 8 bytes higher than it should. This V3 bug has been corrected in V4.

2. The VAX instruction set is sufficiently rich that most random data can be interpreted as instructions. Most system code deals with binary integer and character data. This means that if an EXAMINE/INSTRUCTION display includes many packed decimal and/or floating point instructions, you are probably examining a data area or using a start address which is not an instruction

boundary.

One common error that results in a nonsensical display is to examine instructions in the bugcheck overlay area. During a crash, fatal bugcheck code and message text overlay resident system image code, beginning one page before label BUG$FATAL, for a length of about 12000 decimal or 3000 hex bytes.

Additional References

V3 VAX/VMS Internals and Data Structure Manual, Section 15.1.2, Initial Pager Action, and Chapter 14, Memory Management Data Structures.

VAX Archititure Standard (DEC Standard 032) or VAX-11 Architecture Reference Manual, Chapter 5, Memory Management

RELATED REFERENCE MATERIAL

Call Frame Layout

```
+--------------------------------------------------+
|            CONDITION HANDLER ADDRESS             |
|--------------------------------------------------|
| REGISTER MASK/CONTROL |  SAVED PSW               |
|--------------------------------------------------|
|            SAVED AP                              |
|            SAVED FP                              |
|            SAVED PC                              |
| SAVED REGISTERS SPECIFIED BY MASK...LOW NUMBERS  |
|                 . . .                            |
|             TO HIGHER REGISTER NUMBERS           |
+--------------------------------------------------+
   (0 TO 3 BYTES AS SPECIFIED BY SPA VALUE)
```

The register mask/control field is broken down as follows

```
3 3 2 2 2                 1
1 0 9 8 7                 6
+------------------------------+
|SPA|S|0| REGISTER MASK <11:0>|
+------------------------------+
  ^   ^
  |   |_______SET IF CALLS; CLEARED IF CALLG
  |___________0-3 BYTES REQUIRED FOR LONGWORD STACK ALIGNMENT
```

PSL Layout

```
3 3      2 2 2 2 2 2   2   1 1                0
1 0      7 6 5 4 3 2   0   6 5                0
+------------------------------------------------+
| | |MBZ | | |   |   |/| IPL |  PSW            |
+------------------------------------------------+
 ^ ^      ^ ^  ^   ^
 | |      | |  |   |____PREVIOUS ACCESS MODE
 | |      | |  |________CURRENT ACCESS MODE
 | |      | |___________INTERRUPT STACK
 | |      |_____________FIRST PART DONE
 | |____________________TRACE FAULT PENDING
 |______________________COMPATIBILITY MODE
```

Additional References

VAX Architure Standard (DEC Standard 032) or VAX-11 Architecture Reference Manual, Section 4.6 Procedure Call Instructions and Section 6.2 Processor Status

RESOURCE WAITS

A process in a resource wait is waiting to allocate some unavailable resource or to acquire ownership of a mutex. In theory, a process is placed in this scheduling state for a short time, until the resource becomes available. Occasionally, however, because of some system problem, a process remains in this state for a long enough time that the user notices the lack of system response.

When a process is placed into any wait state, its context is saved such that when it is placed back into execution, it repeats the code thread which resulted in its wait. The PC and PSL saved in its process header determine what code runs, at what access mode, and at what IPL. For example, the saved PC for a process in PFW (pagefault wait) is the address of the beginning of the instruction that incurred the pagefault. When the process is placed back into execution, the instruction is repeated. If all pages referenced by the instruction are now valid, the instruction completes. If a page referenced by the instruction is not valid, the process is placed back into PFW with the same saved PC.

A process placed into resource wait becomes computable (or computable outswapped) whenever an AST is queued to it and/or whenever system code reports as available the resource the process is waiting for. An enqueued AST cannot be delivered if the process is waiting at IPL 2 or if the process is waiting in a more privileged access mode than that of the AST. (Also, an enqueued normal AST cannot be delivered if AST delivery to that mode has been disabled or if there is already an AST active in that mode.)

Careful specification of the wait PC and PSL simplifies delivery of ASTs to a process whose main thread of execution has just placed it into a wait: after the AST is delivered (if the wait access mode and IPL permit), the REI that dismisses the AST interrupt results in execution at the wait PC with the wait PSL. If the process should still wait, it will, by executing the same code which caused it to wait previously.

Because processes are frequently placed into resource wait in kernel mode, sometimes at IPL 0 and sometimes at IPL 2, user attempts to type CTRL/C, CTRL/Y, or CTRL/T often are in vain. CTRL/C, CTRL/Y, and CTRL/T are implemented with supervisor or user mode ASTs; an outer mode AST cannot be delivered to a process waiting in kernel mode. Since SDA uses special kernel ASTs to examine the context of a process on the running system, you cannot learn much with SDA about a process waiting at IPL 2. (This is not an issue with a crashdump, only the running system.) Furthermore, since process deletion is implemented with ASTs, it is impossible to delete a process waiting at IPL 2.

If you are dealing with a resource wait process on a running system, ideally you would like to learn enough about the problem to take whatever action might be possible to satisfy the process's resource wait. However, you might have to force a crash to learn the reason

for the resource wait. Even if you can identify the reason for the wait, you may be unable to do anything to satisfy it without shutting down and rebooting the system.

There should be fewer IPL 2 MWAITS under V4 than under V3. If you encounter one, you should determine whether it is a known problem before trying to take the process out of its wait using any methods indicated below. If the problem is not a known one, it might be better to crash the system so that the problem can be investigated.

For a process in MWAIT, or resource wait, the software PCB field PCB$L_EFWM contains either the system space address of a mutex or a small positive integer which is a system resource number. Occasionally, this piece of information is sufficient for you to make an intelligent guess about the problem; however, you often must also know the process's wait PC and other general registers, and you may need to examine its kernel stack.

1. First, if you're looking at the running system, follow the directions in section ACCESSING PROCESS CONTEXT WITH SDA to examine the process's PCB and, possibly, process header and kernel stack. If you're looking at a dump, use the SDA command SHOW PROCESS/INDEX=<n>.

2. If PCB$L_EFWM contains a system space address, follow the directions in subsection Mutex Wait below.

3. If PCB$L_EFWM contains a small integer, use the table below to translate that integer to a resource name and follow the directions in the appropriate subsection below. Under V4, the largest resource number defined is hex E. Under V3, it is hex C. If PCB$L_EFWM contains a number larger than the largest defined resource number, then it has been corrupted. The Displayed Name column indicates the process state name used by V4 SHOW SYSTEM and SDA SHOW SUMMARY to describe that particular MWAIT state.

RESOURCE NAME	DISPLAYED NAME SUBSECTION NAME	HEX VALUE
RSN$_ASTWAIT	RWAST	1
RSN$_MAILBOX	RWMBX	2
RSN$_NPDYNMEM	RWNPG	3
RSN$_PGFILE	RWPFF	4
RSN$_PGDYNMEM	RWPAG	5
RSN$_BRKTHRU	RWBRK	6
RSN$_IACLOCK	RWIMG	7
RSN$_JQUOTA	RWQUO	8
RSN$_LOCKID	RWLCK	9
RSN$_SWPFILE	RWSWP	A
RSN$_MPLEMPTY	RWMPE	B
RSN$_MPWBUSY	RWMPB	C
RSN$_SCS	RWSCS	D
RSN$_CLUSTRAN	RWCLU	E

Mutex Wait

If the process is waiting to acquire a mutex and continues to be in that state for more than ten minutes, it is quite possible that the process's state will not change, that something is wrong with the system. In such a case, probably the best thing to do is force a system crash in order to find out what process(es) own(s) the mutex and why it has not been released. See section 7.2.3 in the V3 Systems and Operations Guide for instructions on forcing a crash.

There is no simple way to determine which process(es) currently own(s) the mutex. The mutex data structure specifies only how many owner processes there are, and PCB$W_MTXCNT specifies only how many mutexes that particular process owns. Which mutexes a process owns can only be inferred from following the thread of execution on its kernel stack. A well-behaved process does not leave kernel mode or lower IPL below 2 until it releases all mutexes it owns.

1. First, examine the address pointed to by PCB$L_EFWM, and confirm that it is a mutex and that its owner count indicates at least one owner. The SDA EXAMINE command displays any known symbol name associated with that address. If SDA does not display a symbol name, the address may be that of the mutex associated with each line printer UCB. Use the SHOW DEVICE LP command to get the addresses of the line printer UCBs. See the V3 VAX/VMS Internals and Data Structure Manual, Section 2.3, for a list of V3 mutexes. A list of V4 mutex names follows:

MUTEX NAME	DATA STRUCTURE(S)
LNM$AL_MUTEX	Logical Name Tables
IOC$GL_MUTEX	I/O Data Base
EXE$GL_CEBMTX	Common Event Block List
EXE$GL_PGDYNMTX	Paged Dynamic Memory
EXE$GL_GSDMTX	Global Section Descriptor List
EXE$GL_SHMGSMTX	Shared Memory Global Section Descriptor List
EXE$GL_SHMMBMTX	Shared Memory Mailbox Descriptor Table
EXE$GL_ENQMTX	Enqueue/Dequeue Tables (unused)
EXE$GL_ACLMTX	Access Control Lists
CIA$GL_MUTEX	CIA Queues of Suspected and Known Intruders
EXE$GL_KFIMTX	Known File Table (unused)
UCB$L_LP_MUTEX	Line Printer Unit Control Block, 1 per UCB

A mutex data structure is one longword, with the owner count in the low order word. A count of -1 (FFFF) indicates no owners; a count of 0, one owner; etc. The high word has 1 bit defined, MTX$V_WRT, which indicates that some process has requested write ownership of the mutex. If PCB$L_EFWM is not the address of a mutex, it has been corrupted. If you are looking at the running system and you find that the owner count of the mutex is FFFF, see if the MWAIT process is still in MWAIT. The process should not remain in MWAIT once all owners release the mutex.

2. If the mutex is still owned, look through all the processes on the system, setting SDA process index to each in turn, checking for PCB$W_MTXCNT non-zero.

3. Each process you find that owns a mutex should still be in kernel mode at IPL 2. If you are looking at a dump, you should be able to decipher the process's kernel stack, using sections STACK PATTERNS - KERNEL MODE STACK and VIRTUAL ADDRESSES wherever necessary. Following the kernel thread of execution, determine whether the process owns the mutex for which the MWAIT process is waiting. When you find the mutex owner, try to determine why it hasn't released the mutex and what's happening in its process context.

   A process with PCB$W_MTXCNT non-zero which is not in kernel mode indicates a problem, generally software. Try to find out from the user what image(s) the process was running, whether there were error messages, and any other background information that might be helpful. In such a case, try to reproduce the problem in the simplest way, crash the system, and report the problem with background information, the dump, and the console output.

RWAST Resource Wait

RSN$_ASTWAIT is a general purpose resource wait used primarily when the wait is expected to be satisfied by the delivery and/or enqueueing of an AST to the process. That is, there is no "real" system wide resource ASTWAIT, although there are places in the system where RSN$_ASTWAIT is reported available.

One use of ASTWAIT is in EXE$QIO and the routines it calls to do quota checking, EXE$SNGLEQUOTA, EXE$BUFQUOPRC, etc. EXE$QIO uses these routines to check whether a process is allowed any more outstanding direct or buffered I/O requests; driver-specified FDT routines may also use these routines to see if a process has sufficient byte count quota for a buffered I/O request. When the process can have, for example, no more outstanding direct I/O requests, EXE$SNGLEQUOTA puts the process into RSN$_ASTWAIT wait state. The process leaves the wait state whenever an AST is queued to it. The process is placed back into execution again at EXE$SNGLEQUOTA, which repeats its test for whether the process can issue another direct I/O request. In this particular ASTWAIT case, when IOC$IOPOST post-processes a direct I/O request for this process, IOC$IOPOST increments the count of direct I/O requests that the process can issue and queues a special kernel AST to the process to do process context post-processing. The AST enqueueing results in making the process computable; EXE$SNGLEQUOTA repeats the test for whether the process can issue another direct I/O request, and this time the process can.

Similarly, FDT routines call EXE$BUFFRQUOTA to test for sufficient

buffered I/O byte count quota. If the process has insufficient buffered I/O byte count quota, it may wait in RSN$_ASTWAIT wait state (depending on the state of the process resource wait flag). When IOC$IOPOST post-processes a buffered I/O request for the process, it returns buffered I/O byte count quota and queues a special kernel AST to the process to do process context post-processing. The AST enqueueing results in making the process computable; EXE$BUFFRQUOTA repeats the test for whether the process has sufficient buffered I/O byte count, and this time the process may.

The byte count example above, though, is somewhat misleading in that buffered I/O byte count quota is a quota pooled among all the processes in a job tree. The enqueueing of an AST to one subprocess does not cause another subprocess in that job to become computable. Hence, approximately once a second, RSN$_ASTWAIT is declared available for all processes in the system waiting on it.

RSN$_ASTWAIT is also used by EXE$DASSGN to wait for all the process's outstanding I/O on a channel to complete. It is possible for a process to be hung in RSN$_ASTWAIT when an I/O request is lost, when a driver fails to complete an I/O request.

Another V4 use of RSN$_ASTWAIT is forcing a process about to be deleted or suspended to wait until outstanding Files-11 XQP activity completes.

In order to find out what a process in ASTWAIT is really waiting for, you need to know the PC at which the process is waiting and possibly the contents of other general registers and the process's kernel stack.

1. One common RSN$_ASTWAIT occurs in the quota checking routines used by driver FDT routines, EXE$SNGLEQUOTA, EXE$BUFFRQUOTA, EXE$BUFQUOPRC, and EXE$MULTIQUOTA. First, check whether the process is waiting in that code by comparing its PC to the result of the following SDA command.

   SDA> EVAL EXE$MULTIQUOTA+32

2. If the process's wait PC is the address of EXE$MULTIQUOTA + 32, then R2 contains the address of an insufficient process quota. This quota field could be in the PCB or the JIB. If necessary, re-format the PCB and JIB using the method in ACCESSING PROCESS CONTEXT WITH ONLINE SDA to determine what resource the process is waiting for. In many cases the process is waiting at IPL 2 and thus cannot be deleted. If the process is part of a multi-process job and the resource is one pooled among the job's processes (that is, a quota described by the JIB), you may be able to take the process out of its wait by deleting another process in the job.

   If the process cannot be deleted and deleting other processes in the job doesn't work or is not possible, another possibility is to alter the field containing the quota with DELTA and then issue

the DCL command STOP/ID. The desired effect is to increase the quota so that the process can complete whatever service it is requesting, queue an AST to the process to change its state to COM(O), and delete it before it goes into MWAIT again. This risks crashing the system and should be done only as a last resort. First, issue a SHOW SUMMARY SDA command to get the PID (Internal PID, under V4) of your own process for use in the DELTA deposit command below. Then, from an account with CMKRNL and WORLD privileges, type the following commands.

```
$ ! The combination of 1;M and a pid in the deposit command
$ !  "enables" kernel mode operation and deposits.
$ RUN SYS$LIBRARY:DELTA.EXE
DELTA  Vx.y
1;M<cr>
00000001
[W                  !issue this if quota field is word
<pid>:<A(quota field)>/xxxxxxxx yyyyyyyy<cr>
EXIT<cr>
$ STOP /ID=<n>
```

Longer term solutions might include altering the user's authorization file record to increase his quota, or perhaps recoding the user's application to use less of the resource.

3. If the process's wait PC is not EXE$MULTIQUOTA + 32, then see if it is waiting in EXE$DASSGN. Compare its PC to the result of the following SDA command.

```
SDA> EVAL EXE$DASSGN + 6D        !for V4
SDA> EVAL EXE$DASSGN + 67        !for V3
```

If the process is being waited by EXE$DASSGN, then R6 contains the address of the channel control block with outstanding I/O request(s). Type the following SDA commands to determine to which device-unit this channel is assigned.

```
SDA> READ SYS$SYSTEM:SYSDEF.STB !if you haven't already
SDA> EXAM @R6+CCB$W_IOC         !# outstanding requests
SDA> DEF UCB=@(@R6+CCB$L_UCB)   !UCB address
SDA> EXAM UCB + UCB$W_UNIT      !low word is unit #
SDA> EXAM @(UCB+UCB$L_DDB)+DDB$T_NAME;8 !device name
```

See section LOCATING IRPS for information on locating the outstanding IRP(s).

4. Under V4, if the process's wait PC is not within EXE$MULTIQUOTA or EXE$DASSGN, then see if delete or suspend code is forcing the process to wait until Files-11 XQP activity completes. If the process is being forced to wait under these circumstances, the SDA SHOW PROCESS command displays the Process status as "DELPEN" or "SUSPEN", and PCB$B_DPC is greater than zero. To check for this possibility and to format the XQP's current and pending requests, type the following SDA commands.

```
SDA> READ SYS$SYSTEM:SYSDEF.STB !if you haven't already
SDA> SHOW PROCESS               !get PCB address and see
SDA> !  whether DELPEN or SUSPEN is set in PCB$L_STS
SDA> EXAM <PCB_address>+PCB$B_DPC
SDA> ! zero PCB$B_DPC implies no XQP activity
SDA> EXAM CTL$GL_F11BXQP        !address of XQP_QUEUE
SDA> EXAM @.                    !contents of XQP_QUEUE
SDA> ! if XQP_QUEUE doesn't contain its own address,
SDA> !   then repeat next command til back at list head
SDA> FORMAT @.                  !format each pending IRP
SDA> FORMAT @(@CTL$GL_F11BXQP+50) !format current IRP
```

5. If the process's wait PC is not one of the possibilities listed above, then determine at what offset in what module the wait PC is, using the section VIRTUAL ADDRESSES.

   Read the source code of that module, beginning at the wait PC. The wait PC is always set up to repeat the attempted resource allocation that placed the process into the resource wait. You may need to examine the process's general registers and/or kernel mode stack to follow the code path that would occur were the process to be placed into execution.

RWMBX Resource Wait

This resource wait means that a process is trying to write to a mailbox that is full or has insufficient buffering space. A mailbox is created with some amount of "space" for buffering messages that have been written to the mailbox and not yet read. This quota is specified as the BUFQUO argument to $CREMBX system service. If that argument is omitted, its value defaults to the SYSBOOT parameter DEFMBXBUFQUO.

If process resource wait mode is enabled, a process trying to write to a full mailbox waits transparently at IPL 0 in the access mode which issued the $QIO request until its write can complete. If wait mode is disabled, the I/O is completed immediately with the error SS$_MBFULL. The resource wait mode flag can be toggled through DCL command SET PROCESS/RESOURCE or system service $SETRWM. By default, resource wait mode is enabled. Under V4, there is a new I/O function modifier, IO$M_NORSWAIT, that allows a user to specify that a particular write attempt should not wait, independent of the setting of the process resource wait mode flag.

Resource wait RSN$_MAILBOX is usually caused by application error. One possible error is that the reader process's priority is lower than that of the writers. Another possibility is that the reader process does not read the mailbox often enough or completely enough. One more possibility is that a single reader process also writes to

the mailbox and is put into MWAIT when the mailbox is full, thus deadlocking the application system.

A simple possible workaround is to specify a larger BUFQUO argument to $CREMBX or increase SYSBOOT parameter DEFMBXBUFQUO, although this may just delay the onset of the problem. Another way to avoid the problem is that the reader process always have an outstanding read on the mailbox (without the modifier IO$M_NOW). Another workaround is that the reader process issue a setmode QIO request to ask for AST notification of unsolicited messages placed into the mailbox and read the mailbox whenever the AST is delivered. Another possibility is that the writers (and perhaps the reader) disable resource wait mode.

If you want to determine which mailbox the process is trying to write, type the following SDA commands.

```
SDA> SET PROCESS/INDEX=<pix>
SDA> !  get mailbox channel number
SDA> DEF MBCHAN = @(@AP+8)  !read QIO channel number
SDA> READ SYS$SYSTEM:SYSDEF.STB !if you haven't already
SDA> !  get mailbox UCB address
SDA> DEF MBUCB = @(@CTL$GL_CCBBASE-MBCHAN+CCB$L_UCB)
SDA> !  display unit
SDA> EVAL (@(MBUCB+UCB$W_UNIT)@10)@-10
SDA>  !  display device name
SDA> EXAM @(MBUCB+UCB$L_DDB)+DDB$T_NAME
```

If the process is waiting in user mode, the user can simply type the DCL commands CTRL/Y and STOP. If, however, the user's program has disabled CTRL/Y recognition or is waiting in exec mode (as the result of an RMS write to the mailbox), you must either delete the process with the DCL command STOP/ID or read messages from the mailbox to unblock the process.

To read messages from the mailbox, first display the protection and owner of the mailbox with the DCL command SHOW DEVICE/FULL. From an account with CMKRNL privilege, set your UIC appropriately. Read a message from the mailbox by issuing a DCL COPY command, specifying the input file as the mailbox and the output file as TT:, SYS$OUTPUT, NLA0:, or a file. Convert the unit number of the mailbox to decimal for the DCL commands.

RWNPG Resource Wait

This resource wait means that a process is waiting to acquire nonpaged pool. This resource wait should be rare under normal circumstances, since nonpaged pool is expanded upon demand up to a SYSBOOT parameter specified limit.

On the running system, type the following DCL commands to find out whether any of the nonpaged pool lists has approached its limit. If so, consider altering your parameters and rebooting.

```
$ MC SYSGEN
SYSGEN> USE ACTIVE
SYSGEN> SHOW NPAGEVIR  !expanded limit variable list
SYSGEN> SHOW LRPCOUNTV !expanded limit LRP list
SYSGEN> SHOW IRPCOUNTV !expanded limit IRP list
SYSGEN> SHOW SRPCOUNTV !expanded limit SPR list
SYSGEN> SHOW WSMAX     !largest working set size
SYSGEN> SHOW FREELIM   !minimum free list size
SYSGEN> SHOW MPW_LOWLIMIT !minimum mod. list size
SYSGEN> EXIT
$ SHOW MEMORY/POOL/PHYSICAL
```

If the lists have not expanded to their limits (if the Total column is less than the relevant parameter value), one possible reason is that there is not enough free physical memory left in the system. Before expanding nonpaged pool, the system checks to see that there will be enough pages left on the free list for the sum of the largest working set (parameter WSMAX), the free list low limit (parameter FREELIM), and the modified page low limit (parameter MPW_LOLIMIT). SHOW MEMORY displays the number of pages currently on the free list as Physical Memory Usage (Free column). If the free list is not sufficiently large, VMS does not expand nonpaged pool and prints the following message on the console terminal.

SYSTEM-W-POOLEXPF, Pool expansion failure

It is also possible that a process is in resource wait because it is asking for a piece of pool larger than the largest piece available. SHOW MEMORY displays the largest piece free in the Largest column. The process's R1 may contain the number of bytes of pool requested, depending on which system code tried to allocate pool.

Under V3, a process waits for this resource in kernel mode. Under V4, a process may wait for RSN$_NPDYNMEM in kernel mode or in the access mode from which it called a system service that tried to allocate pool. The process may or may not be deletable, depending on the IPL at which it waits.

If you're looking at a crashdump, type the following SDA commands. to compute and display the number of free pages that must be available for VMS to expand pool

```
SDA> !compute # bytes variable list can be expanded
SDA> EVAL @MMG$GL_NPAGEDYN+@SGN$GL_NPAGEVIR-@MMG$GL_NPAGNEXT
SDA> !compute # bytes LRP list can be expanded
SDA> DEF LRPV=@IOC$GL_LRPSPLIT+(@SGN$GL_LRPCNTV*@IOC$GL_LRPSIZE)
SDA> EVAL LRPV-@MMG$GL_LRPNEXT
SDA> !compute # bytes IRP list can be expanded
SDA> DEF IRPV=@EXE$GL_SPLITADR+(@SGN$GL_IRPCNTV*A0)
SDA> EVAL IRPV-@MMG$GL_IRPNEXT
SDA> !compute # bytes SRP list can be expanded
SDA> DEF SRPV=@IOC$GL_SRPSPLIT*(@SGN$GL_SRPCNTV*@SGN$GL_SRPSIZE)
SDA> EVAL SRPV-@MMG$GL_SRPNEXT
SDA> !evaluate whether expansion is possible
SDA> DEF MEM_NEED=@SGN$GL_MAXWSCNT+@SGN$GL_FREELIM
```

```
SDA> DEF MEM_NEED=MEM_NEED+((@MPW$GW_LOLIM@10)@-10)
SDA> EVAL MEM_NEED    !display pages of memory required
SDA> EXAM SCH$GL_FREECNT    !number of pages on free list
SDA> ! MEM_NEED must be less than SCH$GL_FREECNT for expansion
```

RWPFF Resource Wait

Although this resource name is defined, it is not used by any code in V3 or V4. If a process is waiting for this resource, its PCB$L_EFWM has been corrupted.

RWPAG Resource Wait

Although this resource name is defined in V3, no system code waits a process on this resource. V4 does place processes into wait on this resource.

This resource wait means that a process is waiting to acquire paged pool. Under some circumstances the process waits in kernel mode; under others it waits in the access mode from which it made a system service request that resulted in a failure to allocate paged pool. The allocation failure can happen because there is not enough paged pool left or because there is not a large enough piece left. The process's R1 may contain the number of bytes of pool requested, depending on which system code tried to allocate pool. The process may or may not be deletable, depending on the IPL at which it waits.

If you're examining the running system, type the DCL command SHOW MEMORY/POOL to see the amount of unallocated paged pool (the Free column) and the largest piece available (the Largest column).

If you're looking at a crash dump, type the following SDA commands to locate the free paged pool blocks.

```
SDA> EXAM EXE$GL_PAGED   !address of 1st block
SDA> EXAM @.;8           !size of this block & address of next
SDA> EXAM @.;8           !size of this block & address of next
SDA> EXAM @.;8           !size of this block & address of next
SDA> ...                 !continue til address of next is 0
```

If the system seems to be running out of paged pool, alter the SYSBOOT parameter PAGEDYN and reboot.

RWBRK Resource Wait

This particular resource wait should never occur under V4. The broadcast mechanism is rewritten and uses normal $QIOs, with an optional timeout. If the timeout period expires before the I/O

request completes, a $CANCEL is done. If you find a V4 process in MWAIT on this resource, its PCB$L_EFWM has been corrupted.

Under V3, this resource wait means that the process is waiting for a broadcast to complete. Typically, this wait occurs because the user has issued the DCL command REPLY/ALL and there is a bad terminal or an unterminated EIA line. It can also happen if a terminal controller has an input-only device which has not been set nobroadcast.

A process waiting on RSN$_BRKTHRU waits in kernel mode at IPL 2. This means that the process cannot be deleted and cannot be easily examined through SDA.

It is possible, although awkward, to determine which terminal line(s) is (are) at fault. It is possible, although risky, to end the process's wait.

A broadcast request is described by a broadcast data block (BRD), which includes the pid of the requesting process, the broadcast message, and a count of how many terminals have yet to write the broadcast message. A terminal write request is described by a terminal write packet (TWP). TWPs for broadcast writes have offset TTY$L_WB_IRP equal to 0 and offset TTY$L_WB_RETADDR equal to the address of EXE$BRDCSTCOM. A TWP corresponding to a particular broadcast request contains the address of the broadcast message in the BRD (that is, offset TTY$L_WB_NEXT points into the BRD).

In order to determine the terminal(s) at fault, first locate the BRD and then examine each terminal UCB to see whether this terminal has a corresponding write request queued to it. In order to end the wait, you must also locate the BRD.

Type the following commands.

```
SDA> CTRL/Y
$ SPAWN
$ MACRO/OBJ=SYS$LOGIN:DEFS  SYS$INPUT: -
$_ + SYS$LIBRARY:LIB/LIB
 $BRDDEF GLOBAL
 $TTYDEF GLOBAL
 .END
CTRL/Z
$ LO
$ CONT
SDA> READ SYS$LOGIN:DEFS.OBJ
SDA> ! do following if you can read the process's R7
SDA> DEF BRD = @R7 !define symbol
SDA> FORMAT BRD/TYP=BRD   !display BRD
SDA> !
SDA> ! do following if you can't read the process's R7
SDA> DEF BRD = @IOC$GQ_BRDCST !define symbol
SDA> !do til BRD$L_PID matches that of mwait process
SDA> FORMAT BRD/TYP=BRD  !display broadcast queue entry
```

```
SDA> DEF BRD = @BRD!follow forward link to next BRD
SDA>                !and format it, til matching PID
SDA> !
SDA> !for BRD with matching pid...
SDA> !    BRD$L_DATA is start of broadcast msg.
SDA> !    BRD$W_REFC is # of terminals waited for
SDA> DEF BRDMSG = BRD + BRD$L_DATA
SDA> !
SDA> !search the terminal database, looking at each UCB,
SDA> SHOW DEVICE    !to see list of terminal names,
SDA>                !  usually tta, ttb, txa, etc.
SDA> SHOW DEVICE <terminal_name> !e.g., tta
SDA> !for each UCB, e.g., tta0, tta1, etc.
SDA> DEF TWP= @(<ucb_address> + UCB$L_TT_WRTBUF)
SDA> !see if this TWP is a broadcast
SDA> EXAM TWP+TTY$L_WB_IRP !if zero, then a broadcast
SDA> !you might also look at queued TWPs
SDA> DEF TWP=@( <ucb_address>+UCB$L_TT_WFLINK)
SDA> EXAM TWP+TTY$L_WB_IRP !if zero, then a broadcast
SDA> DEF TWP=@TWP                  !flink to next TWP...
SDA> !
SDA> !for each TWP, compare TTY$L_WB_NEXT to BRDMSG
SDA> EXAM TWP+TTY$L_WB_NEXT
SDA> !if equal, this is a terminal which hasn't completed
SDA> !  the broadcast message
```

If you have a process stuck in RSN$_BRKTHRU that you want to unwait, at the risk of crashing the system, use DELTA to clear the BRD$W_REFC field, and then issue a successful broadcast to several working terminals so that RSN$_BRKTHRU is declared available for any processes waiting for that resource. This should be sufficient to make the MWAIT process computable again at a time when its BRD$W_REFC is zero. If the problem terminal(s) complete the broadcast at a later time, there will be a nonfatal BRDMSGLOST bugcheck entry written to the error log. First, issue a SHOW SUMMARY SDA command to get the PID of your own process for use in the DELTA deposit command below. Type the following commands from an account with CMKRNL and OPER privileges. Replace A(BRD$W_REFC) with the address you determined using SDA.

```
$ MC SYSGEN        !so that nonfatal bugchecks don't crash
SYSGEN> SET BUGCHECKFATAL 0
SYSGEN> WRITE ACTIVE
SYSGEN> EXIT
$ ! The combination of 1;M and a pid in the deposit command
$ !  "enables" kernel mode operation and deposits.
$ RUN SYS$LIBRARY:DELTA.EXE
DELTA Vx.x
1;M<cr>
<pid>:<A(BRD$W_REFC)>/xxxxyyyy xxxx0000<cr>
EXIT<cr>
$ ! pick 3 terminals known to work
$ REPL/TERM=(<term1>,<term2>,<term3>) "<some_message>"
```

RWIMG Resource Wait

This particular resource wait should never occur under V4. The interlocking between the image activator and the INSTALL utility is rewritten and uses the lock management system services. If a process is waiting for this resource, its PCB$L_EFWM has been corrupted.

A V3 mechanism called the image activator lock synchronizes access to the data structures involved in the activation of images made known to the system through the INSTALL utility. The lock synchronizes the activities of the INSTALL utility and its use of the image activator system service with other processes' image activations.

The image activator lock is similar to a mutex in that it allows one and only one writer or multiple readers. Unlike a mutex, the image activator lock can be used by the image activator system service, which runs mostly in exec mode.

If a process is waiting for this resource, then examine the image activator lock

```
SDA> EXAM MMG$GL_IACLOCK
```

If the image activator lock is owned by a writer, MMG$GL_IACLOCK contains the address of the writing process's software PCB. A positive integer in MMG$GL_IACLOCK indicates how many reading processes own the lock. A value of 0 indicates no readers and no writer. A process which has locked MMG$GL_IACLOCK has the flag bit RND$V_IACLOCK set in CTL$GL_RUNDNFLG.

A process deadlocked in this resource wait has never been seen. This information is included for completeness. If you find a V3 process in this resource wait, crash the system and report the problem.

RWQUO Resource Wait

Although this resource name is defined, it is not used by any code in V3 or V4. If you find a process in MWAIT on this resource, its PCB$L_EFWM has been corrupted.

RWLCK Resource Wait

Although this resource name is defined, it is not used by any code in V3 or V4. If a process is waiting for this resource, its PCB$L_EFWM has been corrupted.

RWSWP Resource Wait

When a process is first created, a minimal swap file slot is allocated for it. The system has a table of installed page and swap files. It looks first in the swap files to allocate swap slots; if there is not enough room, the system allocates a swap slot from a page file with room. As the process pagefaults and its working set grows, a larger swap slot is allocated. The process's maximum swap slot is limited by its working set quota. If a larger slot is not available when the process's working set size is being increased, the process is waited on RSN$_SWPFILE in the access mode that incurred the pagefault. The RSN$_SWPFILE wait state can mean that the system is running out of swap space or that there is too much fragmentation to allocate a swap slot.

If you see processes in this wait state, issue the DCL command SHOW MEMORY/FILES/FULL to determine the state of your page and swap files. The SHOW MEMORY/FILES/FULL output identifies swap files as "used exclusively for swapping". You may be able to create and install an additional swap file to remove the process(es) from this wait.

The SYSBOOT parameter SWPFILCNT is the maximum number of swap files you can install; the parameter PAGFILCNT, the maximum number of page files. If you have fewer swap files than the value of SWPFILCNT, you may install a new one. (Enlarging an existing one would solve your problem but not without a reboot of the system.) If you can't install more swap files, then you may be able to install an additional page file as a temporary measure rather than rebooting the system immediately. Installation of either a page or swap file causes the resource RSN$_SWPFILE to be declared available. From the SYSTEM account or one set to a SYSTEM UIC, type the following DCL commands.

```
$ SHOW MEMORY/FILES/FULL
$ SET PROT=(SY:RWED,OW:RWED)/DEFAULT
$ MC SYSGEN
SYSGEN> USE ACTIVE
SYSGEN> SHOW SWPFILCNT
SYSGEN> SHOW PAGFILCNT
! specify a unique swap or page file name
SYSGEN> CREATE <file_spec>/SIZE=<size>
SYSGEN> INSTALL <file_spec>/SWAPFILE
!issue command below instead for page file
SYSGEN> INSTALL <file_spec>/PAGEFILE
SYSGEN> EXIT
```

A process is placed into this wait state in the access mode at which it pagefaulted. In theory, therefore, a user could type CTRL/C or CTRL/Y, followed by STOP, to have his process become computable again. However, any subsequent pagefault (except for pages paged through the system working set list) causes the process to wait again, from a different thread of execution. You should be able to delete or suspend the process through DCL command. Suspending the process may be a good alternative if there is only one very large process that is in this state because the swap file is fragmented

rather than almost full. When the system runs low on memory, the process's working set may be shrunk enough that it will fit its allocated swap slot. If this happens, you can resume it and have the user decrease his working set list size.

If you're looking at a crash dump, type the following SDA commands to see the state of the swap files.

```
SDA> READ SYS$SYSTEM:SYSDEF.STB
SDA> DEF ARRAY = @MMG$GL_PAGSWPVC
SDA> EXAM SGN$GW_SWPFILCT  !max # swap files
SDA> EXAM SGN$GW_PAGFILCT  !max # page files
SDA> ! repeat commands for sum of swap & page files
SDA> FORMAT @ARRAY         !format 1 page/swap file block
SDA> DEF ARRAY=ARRAY+4
SDA> ! ignore block if its address is MMG$GL_NULLPFL
SDA> ! PFL$L_FREPAGCNT is # of free pages
```

RWMPE Resource Wait

When a process faults a page, the pagefault service routine, MMG$PAGEFAULT, calls MMG$FREWSLE to find a working set list entry to describe the page to be added to the process's working set list. One possible working set list entry is a process page table page that is now inactive; that is, the page table page maps no valid pages. Such a working list entry can be re-used. If, however, the page table page still describes a page on the modified list, the modified page must be written to its backing store before the working set list entry used by the page table page can be released.

In such a case, the modified list high limit is temporarily set to zero to force a flush, and the process is placed into resource wait on RSN$_MPLEMPTY until its modified page has been written to its backing store. The modified page writer (part of the Swapper) declares RSN$_MPLEMPTY available when the modified page list is emptied.

The modified page writer can fail to write the entire list if there is insufficient page file or if a disk goes off line which contains a file (page, swap, or section file) to which modified pages are being written. If the modified page list grows above the SYSBOOT parameter MPW_WAITLIMIT, other processes may go into RSN$_MPWBUSY waits.

A process is placed into this wait state in the access mode at which it pagefaulted. In theory, therefore, a user could type CTRL/C or CTRL/Y, followed by STOP, to have his process become computable again. If the process pagefaults again, the process may be waited again, from a different thread of execution. However, if the process's modified page has already been written to its backing store or if the process faults a page which is paged through the system working set list, the process won't go into this wait again. You should be able to delete the process through DCL command.

RWMPB Resource Wait

A process which faults a modified page out of its working set is placed into this wait if the modified page list contains more pages than the SYSBOOT parameter MPW_WAITLIMIT. Typically, this resource wait is noticeable only on systems with insufficient page file space or incorrect parameter settings. That is, if the Swapper process cannot write the modified page list because there is insufficient space in the page file(s), the list continues to grow. When it reaches MPW_WAITLIMIT, processes are placed into RSN$_MPWBUSY resource wait. Modified page writing is triggered when the modified page list reaches the size of MPW_HILIMIT. Therefore, MPW_WAITLIMIT should never be less than the parameter MPW_HILIMIT. If it is, the system is likely to deadlock. This resource wait may also be noticeable on systems with compute bound realtime processes which block the Swapper process.

A process is placed into this wait state in the access mode at which it pagefaulted. In theory, therefore, a user could type CTRL/C or CTRL/Y, followed by STOP, to have his process become computable again. However, any subsequent pagefault (except for pages paged through the system working set list) causes the process to wait again, from a different thread of execution.

You are very unlikely to see processes in this wait state on a running system; that is, your own process is likely to be in the same state. However, if you do, see section HANGS, subsection System Hangs, the paragraph following the console message SYSTEM-W-PAGEFRAG, for directions on creating and installing a new page file and altering parameter MPW_WAITLIMIT.

If you are looking at a crash dump, type the following SDA commands to determine why processes were placed into RSN$_MPWBUSY wait.

```
SDA> EXAM SCH$GL_MFYCNT      !# pages on mfy. list
SDA> EXAM MPW$GL_WAITLIM     !MPW_WAITLIMIT
SDA> READ SYS$SYSTEM:SYSDEF.STB
SDA> DEF ARRAY=@MMG$GL_PAGSWPVC+(4*(@SGN$GW_SWPFILCT@10)@-10)
SDA> EXAM SGN$GW_PAGFILCT    !max # page files
SDA> ! repeat commands below for number of page files
SDA> FORMAT @ARRAY           !format 1 page file block
SDA> DEF ARRAY=ARRAY+4
SDA> ! ignore block if its address is MMG$GL_NULLPFL
SDA> ! PFL$L_FREPAGCNT is # of free pages
```

RWSCS Resource Wait

The V4 lock manager places a process into this wait when the lock manager must communicate with its counterparts on other VAXcluster nodes to obtain information about a particular lock resource. Typically, the process has requested the $ENQ[W] system service to enqueue a new lock or convert an existing lock on a resource

"mastered" on another node. A resource that is mastered on a remote node can be identified by the non-zero cluster system ID (CSID) in its resource block. A process requesting the $GETLKI system service to obtain lock information about a remote resource is also placed into this wait.

The first node in a cluster to take out a lock on a resource and create the resource is always master of the resource. For example, the first node to open a file becomes the master of the resource that represents that file. All other nodes in the cluster that want to open the file or lock records in the file must communicate with the master node.

When a process queues a lock at the root level of a resource, the local lock manager must first determine if a master already exists for the resource by sending a message to the resource "directory" node, which is determined by a hash algorithm. While the process waits for a reply from the directory node, it waits on resource RSN$_SCS. If that resource has been mastered on another node, the local lock manager must communicate with the master node and places the process into a resource wait on RSN$_SCS until the master node replies.

This wait happens frequently during normal system operation for relatively brief intervals, most often as the result of a $ENQ[W] request for a lock conversion and occasionally as the result of a $GETLKI request.

If you are looking at a dump and happen to see one or more processes in this state, most likely the state is merely an indication that there is remote lock activity at the time the system crashed.

If a process has been in this wait state for more than a few milliseconds, it may be an indiation of CI problems, an unstable cluster, or loss of quorum. Check the consoles for messages that might indicate cluster status and check the error logs and consoles for information about possible CI problems.

A process waits on this resource in kernel mode at IPL 2. This means that the process cannot be deleted, suspended, or easily examined with SDA and that the user cannot CTRL/Y out of the wait.

RWCLU Resource Wait

A V4 process which issues any lock requests on any node of a cluster in transition (that is, while a node is being added or removed) is placed into this wait state while the cluster membership stabilizes. This can be a relatively lengthy wait if node(s) are being removed and locks must be remastered.

A process waits on this resource in kernel mode at IPL 2. This means that the process cannot be deleted, suspended, or easily examined

with SDA and that the user cannot CTRL/Y out of the wait.

[more information TBS]

Hints And Kinks

1. Whenever you modify SYSBOOT parameters, remember to make AUTOGEN aware of your changes so that they propagate across AUTOGENs. Include any parameter changes you make in V3 SYS$SYSTEM:PARAMS.DAT or in V4 SYS$SYSTEM:MODPARAMS.DAT. See Chapter 11 in the Guide to VAX/VMS System Management and Daily Operations for further information on AUTOGEN.

Additional References

V3 VAX/VMS Internals and Data Structures Manual, Chapter 3, Dynamic Memory Allocation; Chapter 10, Scheduling; Section 14.5, Data Structures that Describe the Page and Swap Files; Section 15.5.2, Modified Page Writing; Section 19.5.4, Mailbox Driver

Guide to VAX/VMS Performance

VAX/VMS System Generation Utility Reference Manual

RESTART BUGCHECKS

VAX cpus halt in response to various severe error conditions, halt instructions, and console halt commands. The console (whether it is implemented as a separate processor or as cpu microcode) prints an error message and/or halt code on the console terminal. In the case of a powerfail recovery, the message and/or halt code is printed after power is restored.

The console's actions following a halt generally depend on cpu type and front panel switch settings. In no case will there be much, if any, information about the error causing the halt in the errorlog written by VAX/VMS. The console tests the auto restart switch. See the section HALTS <cpu_type> for cpu-specific details on the restart mechanism.

If auto restart is not enabled, the console prompts or reboots, depending on the switch setting. If auto restart is enabled, the console tests whether memory contents are valid. If they are, the console attempts to locate the Restart Parameter Block (RPB). If the console locates a valid RPB, it passes control to EXE$RESTART (whose physical address is contained in RPB$L_RESTART) with information in several general registers

- R10 - PC at the time of the halt
- R11 - PSL at the time of the halt
- AP - code indicating reason for halt
- SP - address of the end of the RPB page.

EXE$RESTART is entered in kernel mode, at IPL 31, with the PSL<IS> bit set, and with memory management disabled. It uses a temporary stack at the end of the RPB. EXE$RESTART turns on memory management, using information saved in the RPB.

EXE$RESTART's subsequent actions depend upon what kind of halt occurred. If a powerfail recovery occurred and if system state was saved completely prior to the powerfail, EXE$RESTART resumes system operations using information saved in the RPB.

If a powerfail recovery has occurred and system state was not saved completely, EXE$RESTART signals the fatal bugcheck STATENTSVD. This bugcheck generally means that there was not enough time between the powerfail interrupt grant and the total loss of power to the CPU for the volatile processor and general registers to be saved. It can also mean that through some hardware error, the system entered powerfail recovery without having taken a powerfail interrupt.

There is another possible cause of the STATENTSVD bugcheck on a VAX-11/780 or VAX-11/785. These systems have UNIBUS adapters which can interrupt at decimal IPL 20, as the result of failing power on the UNIBUS. VMS's response to this interrupt is to remap the SPTEs that mapped UNIBUS address space to prevent UNIBUS device drivers from getting machine checks when they access UNIBUS address space. If the UNIBUS adapter detects a pending powerfail before the cpu does

AND if a UNIBUS device driver is currently running at or above IPL 20, then the UNIBUS adapter interrupt is not granted until the driver lowers IPL or is done. If the driver remains at high IPL for too long, there may be no power on the UNIBUS when the driver tries to reference an address on the UNIBUS. This will result in a machine check exception. The machine check exception service routine runs at IPL 31 and thus blocks the cpu powerfail interrupt. This can result in a failure to enter the cpu powerfail interrupt service routine with sufficient time to save volatile system state. The subsequent restart results in the bugcheck STATENTSVD.

After any kind of halt other than powerfail recovery, EXE$RESTART crashes the system to provide information that might be useful in troubleshooting the halt and to preserve pending error log messages. (When the system is rebooted, SYSINIT will cause them to be written to the error log.) EXE$RESTART signals a bugcheck specific to the type of halt. If the code in AP is unknown, EXE$RESTART signals the fatal bugcheck UNKRSTRT.

Under V3, the VAX-11/785 halt ?CLOCK PHASE ERROR results in a UNKRSTRT crash if auto-restart is enabled. If the AP contains hex F (decimal 15) and this crash is from a VAX-11/785, see subsection ?CLOCK PHASE ERROR in the section HALTS - VAX-11/780 AND VAX-11/785.

The table below lists the various bugcheck names, the corresponding decimal halt codes, and to which VAX cpus they apply. A "Y" in the column under a cpu type means that halt and bugcheck type are possible on that cpu type; a blank means that the halt and bugcheck type are not applicable. Note that a VAX-11/725 is really a VAX-11/730 and that a VAX-11/782 is two VAX-11/780s connected through shared MA780 memory. Cpu types UV1 and UV2 in the table refer to MicroVAX I and MicroVAX II.

The subsections below describe each halt further and suggest approaches to analyzing the crashdump, if applicable. Read the section HALTS <cpu_type> for information on related hardware problems.

BUGCHECK NAME	HEX AP	DESCRIPTION	UV1	UV2	730	750	780	785	8600
UNKRSTRT	0,1, 2,>11	Unknown restart code	Y	Y	Y	Y	Y	Y	Y
IVLISTK	4	Invalid interrupt stack	Y	Y	Y	Y	Y	Y	Y
DBLERR	5	Double error halt	Y	Y	Y	Y	Y	Y	Y
HALT	6	Halt instruction	Y	Y	Y	Y	Y	Y	Y
ILLVEC	7	Illegal Vector code		Y	Y	Y	Y	Y	Y
NOUSRWCS	8	No user WCS for vector				Y	Y	Y	Y
ERRHALT	9	Error pending on halt					Y	Y	Y
CHMONIS	A	CHM on interrupt stack	Y	Y	Y	Y	Y	Y	Y
CHMVEC	B	CHM vector <1:0> .NE. 0	Y	Y	Y	Y			Y
SCBRDERR	C	SCB physical read error	Y	Y	Y				
WCSCORR	D	WCS error correction failed							Y
CPUCEASED	E	CPU ceased execution							Y
OUTOFSYNC	F	Processor clocks out of synch						Y	
ACCVIOMCHK	10	Machine check accvio		Y					
ACCVIOKSTK	11	KSP accvio		Y					

IVLISTK Bugcheck

This halt means that an attempted cpu write reference to the interrupt stack during interrupt or exception processing would have resulted in a translation not valid or access violation exception. There is no way to determine the SP value at the time of halt from looking at the crashdump. The PC at the time of the halt is in R10; the PSL at the time of the halt is in R11.

An IVLISTK bugcheck can result from software problems that corrupt the stack pointer, overflow or underflow the interrupt stack, or corrupt the System Page Table Entries (SPTEs) that map the interrupt stack. This bugcheck can also occur simply because the interrupt stack is too small.

1. Examine the pages allocated to the interrupt stack through the SDA command

   ```
   SDA> SHOW STACK @EXE$GL_INTSTKLM : @EXE$GL_INTSTK
   ```

   If the low address end of the interrupt stack contains several or more longwords of zero, there is a high probability that the stack did not overflow.

2. Examine the SPTEs that map the interrupt stack. Type the SDA command

   ```
   SDA> SHOW PAGE/SYSTEM @EXE$GL_INTSTKLIM : @EXE$GL_INTSTK
   ```

   SDA should display the page type as VALID and the protection as

ERKW. If the page has been modified, there will be an M by the protection. If the display of the lower SPTEs does not include the M bit, those pages have not been modified, and, therefore, the stack did not become invalid as the result of an overflow.

3. If you suspect an overflow, look for any repeated patterns that might be footprints of nested or recurring exceptions, particularly machine checks. Such repeating patterns generally indicate hardware problems.

4. Try to decipher the interrupt stack using section STACK PATTERNS - INTERRUPT STACK.

5. Determine the IPL at time of halt by examining R11. Decode it as a PSL using the layout in the section RELATED REFERENCE MATERIAL or the V4 SDA command EXAMINE/PSL.

6. Locate the PC at the time of halt (R10 contents) using section VIRTUAL ADDRESSES. Try to determine what code was running, using the PC and IPL. Check whether some thread might have been running at too low an IPL to block interrupts it was causing if you suspect an overflow. If you can identify the most recent thread that ran on the interrupt stack, read its code carefully looking for errors that may have resulted in stack pointer corruption or stack underflow.

7. If you suspect that the interrupt stack did overflow, a [temporary] workaround may be to reboot VMS with an increased SYSBOOT parameter INTSTKPAGES. You may wish to check the SPTE of the lowest interrupt stack page during system operation, to see whether the page has been modified, as in item 2 above. If the lowest stack page remains unmodified during heavy operations, it is likely that INTSTKPAGES is large enough

8. Read the section HALTS - <cpu_type> for information on disabling auto-restart and/or examining the SP and any other destroyed registers from the time of the halt in case this halt occurs again and for information on possible hardware problems.

DBLERR Bugcheck

This halt means that while the cpu was trying to write the microcode machine check logout onto the stack, another machine check occurred. Generally, this halt is an indication of hardware problems.

Analyzing a crashdump from a double error halt is unlikely to be useful. The needed information is in processor registers which have been overwritten. See the section HALTS - <cpu_type> for information on disabling auto-restart and/or examining the processor registers at the time of halt in case this halt occurs again and for information on possible hardware problems.

Look carefully at any error log entries from before and at the time of halt to see whether there are any related unexpected errors, for example, machine checks, bus errors, memory errors.

HALT Bugcheck

This halt may mean that some kernel mode code halted. (The HALT instruction can only be executed from kernel mode.) The various HALTs throughout VMS code are executed under extreme circumstances where no recovery is possible. Also, sometimes erroneous transfers of control or overwriting of code can cause the cpu to execute a byte of zero as a HALT instruction.

If this halt is the result of a software error, the crashdump contains useful information.

1. Determine the address of the HALT instruction and examine that code with the following SDA commands

   ```
   SDA> EVAL @R10-1
   SDA> EVAL BUG$FATAL
   SDA> EVAL BUG$FATAL+2D50
   SDA> EXAM/INST (@R10-1)
   ```

2. If the PC falls within the fatal bugcheck overlay (if it is within the approximate range BUG$FATAL - hex 200 : BUG$FATAL + hex 2D50), then the previous contents of that location have been overwritten. Locate the PC in source code using section VIRTUAL ADDRESSES to see whether there is a halt instruction in the sources. If there is a halt in the source code, read it to determine what anomaly caused the halt. If there is not a halt in the source code, there may have been a hardware error or there may have been a software problem that corrupted the code over which the fatal bugcheck overlay was written.

3. If the PC does not fall within the fatal bugcheck overlay, and the instruction you examined is not a halt, then possibly a hardware error (cache or instruction decode, for example) caused this problem. See the section HALTS - <cpu_type> for information on disabling auto-restart and/or examining the SP and any other destroyed registers and for information on possible hardware errors.

4. If the PC does not fall within the fatal bugcheck overlay, and the instruction you examined is a halt, locate the PC using section VIRTUAL ADDRESSES and try to determine whether code deliberately halted or whether a previous error (such as software corruption at that virtual address) caused this halt. Look at the contents around the halt and try to determine whether they make sense.

5. Decode the PSL at the time of the halt in R11 to determine whether the system was running on a process kernel stack or the interrupt stack. There is no way to determine from the crashdump what the SP contained at the time of halt. Use the PSL layout in the section RELATED REFERENCE MATERIAL or the V4 SDA command EXAMINE/PSL.

6. If the system was running on the interrupt stack, examine the stack with the SDA command

```
SDA> SHOW STACK @EXE$GL_INTSTK : @EXE$GL_INTSTKLIM
```

   Try to determine what was happening based on the PC at time of halt and stack footprints. See section STACK PATTERNS - INTERRUPT STACK.

7. If the system was running on a process's kernel stack, use the section KERNEL STACK LOCATIONS to determine the limits of the stack. Then display it with SDA.

ILLVEC Bugcheck

This halt means that an interrupt or exception dispatch was attempted through a System Control Block vector whose low two bits contain an illegal value, for example, binary 11. On some cpus, this halt can occur if the vector's low two bits are binary 10.

VMS software never deliberately sets the low two bits to binary 11 or 10. This halt might occur as the result of a previous software error's overwriting the System Control Block or memory errors. These bits may have been set deliberately by a user through the console or XDELTA or kernel mode code, in an attempt to determine through which vector an unexpected interrupt or exception is occurring.

Examine the SCB in the crashdump to see if there is a vector with the low two bits set. In an ILLVEC crashdump from a VAX-11/780 or VAX-11/785, R10 should contain the offset into the SCB of the vector that caused the problem. You can examine only this vector through the SDA command

```
SDA> EXAMINE (@EXE$GL_SCB) + @R10
```

If the crashdump is not from a VAX-11/780 or VAX-11/785, type one of the following SDA commands

```
SDA> EXAMINE @EXE$GL_SCB : @SWP$GL_BALBASE
SDA> SHOW STACK @EXE$GL_SCB : @SWP$GL_BALBASE
```

If there is no vector with the low two bits containing binary 11 or 10, there may be a hardware problem. See the section HALTS - <cpu_type> for information on disabling auto-restart and/or examining

the SP and any other destroyed registers from the time of the halt and for information on possible hardware problems.

NOUSRWCS Bugcheck

This halt means that an interrupt or exception dispatch was attempted through a System Control Block vector whose low two bits were binary 10 and that no user writable control store (WCS) exists on the cpu. Note that user WCS is only supported on the VAX-11/780, VAX-11/750, and VAX-11/785.

VMS software never deliberately encodes the low two bits as binary 10. This halt might occur as the result of a previous software error's overwriting the System Control Block or memory errors.

Examine the SCB in the crashdump to see if you can locate a vector with the low two bits equal to binary 10. Type one of the following SDA commands.

```
SDA> EXAMINE @EXE$GL_SCB : @SWP$GL_BALBASE
SDA> SHOW STACK @EXE$GL_SCB : @SWP$GL_BALBASE
```

If there is no such vector, hardware problems may be responsible. See the section HALTS - <cpu_type> for information on possible hardware problems. One very unlikely way that software error could cause this problem is corruption of the register PR$_SCBB. To confirm that PR$_SCBB contains its original value, examine the SPTE mapping the virtual address stored in EXE$GL_SCB, and compare its PFN to the contents of PR$_SCBB as displayed in the Process Registers screen from SHOW CRASH. Type the following SDA commands

```
SDA> SHOW CRASH             !to see processor register display
SDA> EVAL <scbb>/200   !convert contents of SCBB to PFN
SDA> SHOW PAGE/SYSTEM @EXE$GL_SCB;200 !to see PTE contents
```

ERRHALT Bugcheck

[TBS]

CHMONIS Bugcheck

This halt means that while the system was running on the interrupt

stack, an attempt was made to execute one of the change mode instructions (CHMU, CHMS, CHME, or CHMK).

This halt might occur as the result of software error; for example, some process context code's executing in system context, a user-written driver's erroneously requesting system services while executing on the interrupt stack, erroneous transfer of control to data or the middle of an instruction, etc.

The PC of the CHMx instruction is in the crashdump R10; the PSL at the time of halt is in the crashdump R11. There is no way to determine the SP value at the time of the crash from the dump. See the section HALTS - <cpu_type> for information on disabling auto-restart and/or examining the SP and any other destroyed registers from the time of the halt and for information on possible hardware problems.

1. Determine the address of the CHMx instruction and examine that code with the following SDA commands

   ```
   SDA> EVAL @R10
   SDA> EVAL BUG$FATAL - 200
   SDA> EVAL BUG$FATAL+2D50
   SDA> EXAM/INST @R10-20;30
   ```

   Most CHMK and CHME instructions are in the system service vectors. See section SYSTEM SERVICE VECTORS for more information on these vectors and their addresses.

2. If the PC falls within the fatal bugcheck overlay (if it is within the approximate range BUG$FATAL - hex 200 : BUG$FATAL + hex 2D50), then the previous contents of that location have been overwritten, and the SDA commands above will not display the contents at the time of the halt. Locate the PC in source code using section VIRTUAL ADDRESSES to see whether there is a CHMx instruction in the sources.

3. If there is not a CHMx in the source code, there may have been a hardware error or there may have been a software problem that corrupted the code over which the fatal bugcheck overlay was written.

4. If the PC does not fall within the fatal bugcheck overlay, and the instruction you examined is not a CHMx then possibly a hardware error (cache or instruction decode, for example) caused this problem. See the section HALTS - <cpu_type> for information on disabling auto-restart and/or examining the SP and any other destroyed registers and for information on possible hardware errors.

5. If there is a CHMx in the source code, try to determine whether this code was indeed intended to run on the interrupt stack. If it is intended to run on the interrupt stack in system context,

the CHMx instruction is definitely in error.

6. If the code appears to be process context code, then try to figure out why it is running on the interrupt stack. Examine the pages allocated to the interrupt stack through the SDA command

```
SDA> SHOW STACK @EXE$GL_INTSTKLM : @EXE$GL_INTSTK
```

Try to decipher the interrupt stack using section STACK PATTERNS - INTERRUPT STACK.

CHMVEC Bugcheck

This halt means that one of the four CHMx vectors in the System Control Block (SCB) has the low order two bits set to something other than binary 00.

VMS always encodes these bits as 00. This halt might occur as the result of a previous software error's overwriting the SCB or memory errors.

The PC at the time of halt is in the crashdump R10; the PSL at the time of the halt is in the crashdump R11. There is no way to determine the SP value at the time of the halt from the crashdump. See the section HALTS - <cpu_type> for information on disabling auto-restart and/or examining the SP and other registers at the time of the halt and for information on possible hardware problems.

Examine the SCB in the crashdump to see which, if any, CHMx vector has non-zero low bits. Type the following SDA commands

```
SDA> SHOW STACK @EXE$GL_SCB + 40 ;10
```

Under normal circumstances, if no vector has been altered, SDA's symbolic display for those vectors should be EXE$CMODKRNL, EXE$CMODEXEC, EXE$CMODSUPR, and EXE$CMODUSER. (It is remotely possible that on some systems the exception service routines for CHMK and CHME are, respectively, EXE$CMODKRNLX and EXE$CMODEXECX.) Any other values, including values such as EXE$CMODKRNL+1, indicate some kind of corruption.

One very unlikely way that software error could cause this problem is corruption of the register PR$_SCBB. To confirm that PR$_SCBB contains its original value, examine the SPTE mapping the virtual address stored in EXE$GL_SCB, and compare its PFN to the contents of PR$SCBB as displayed in the Process Registers screen from SHOW CRASH. Type the following SDA commands

```
SDA> SHOW CRASH            !to see processor register display
SDA> EVAL <scbb>/200       !convert contents of SCBB to PFN
SDA> SHOW PAGE/SYSTEM @EXE$GL_SCB;200 !to see PTE contents
```

SCBRDERR Bugcheck

This halt means that the cpu got an uncorrectable memory error trying to read an SCB vector. Generally this halt is an indication of a memory error or other hardware problem.

One very unlikely way that software error could cause this problem is corruption of the register PR$_SCBB. To confirm that PR$_SCBB contains its original value, examine the SPTE mapping the virtual address stored in EXE$GL_SCB, and compare its PFN to the contents of PR$SCBB as displayed in the Process Registers screen from SHOW CRASH. Type the following SDA commands

```
SDA> SHOW CRASH             !to see processor register display
SDA> EVAL <scbb>/200        !convert contents of SCBB to PFN
SDA> SHOW PAGE @EXE$GL_SCB;200 !to see PTE contents
```

Look carefully at any error log entries from before and at the time of halt to see whether there are any related unexpected errors, for example, machine checks, bus errors, memory errors.

WCSCORR Bugcheck

[TBS]

CPUCEASED Bugcheck

[TBS]

OUTOFSYNC Bugcheck

This halt means that the VAX-11/785 cpu and SBI clocks are out of phase. This error cannot be caused by software. For further information, see subsection ?CLOCK PHASE ERROR in section HALTS - VAX-11/780 AND VAX-11/785.

ACCVIOMCHK Bugcheck

[TBS]

ACCVIOKSTK Bugcheck

[TBS]

Hints And Kinks

1. For halts other than power failures, EXE$RESTART uses a temporary stack at the end of the page containing the Restart Parameter Block when it bugchecks. As a result, SDA's display in response to SHOW STACK is not very informative about the state of the stack at the time of the halt.

   If the halt occurred on the kernel stack, see the section LOCATING THE KERNEL STACK to determine the possibilities for its low and high limits.

   The interrupt stack high end is contained in EXE$GL_INTSTK; its low end in EXE$GL_INTSTKLM.

2. When SDA processes the SHOW CRASH command, SDA outputs the PC and PSL and the message "Remaining registers not available — wiped out by console." Despite this, it is possible and useful to examine the general registers. You can do this with the SDA command

   SDA> EXAM R<number>

   See section HALTS - <cpu_type>, subsection Restart Mechanism, for information on which general register contents are lost through the restart mechanism.

3. Note that for each V3 SDA COPY command used to copy the dump, the SP will be 8 bytes greater than its actual value; that is, SDA will show the SP pointing to a stack address 8 bytes higher than it should. This V3 bug has been corrected in V4.

Additional References

VAX Architecture Standard (DEC Standard 032), Section 12.7 Halts, Chapter 11 System Bootstrapping and Console

MicroVAX I CPU Technical Description, Chapter 2, Programming Interface.

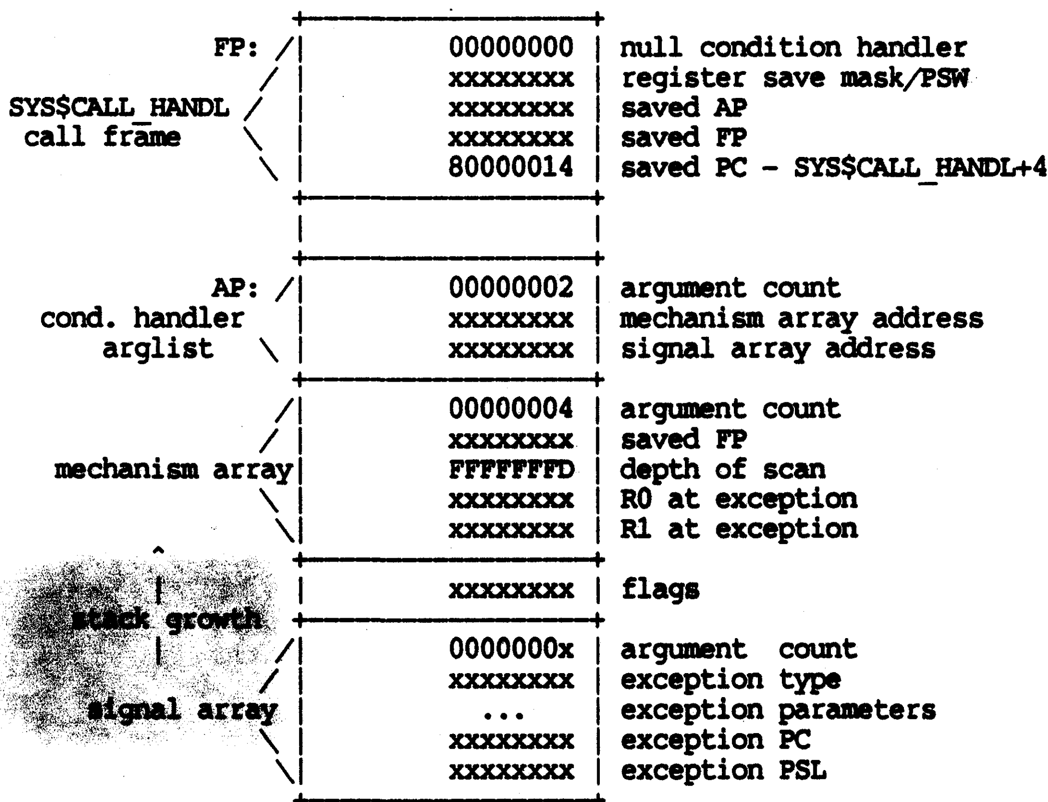
[MicroVAX II] KA630-A CPU Module User's Guide, Chapter 3, Booting and Console Program Interface

SSRVEXCEPT BUGCHECK

The SSRVEXCEPT bugcheck is signaled by the default last chance condition handlers for kernel mode and exec mode. That is, if process-context code has declared no other primary, secondary, or call frame condition handler to deal with a particular exception type, the last chance handler is invoked to bugcheck. In kernel mode, this bugcheck is always fatal. In exec mode, this bugcheck is fatal only if the SYSBOOT parameter BUGCHECKFATAL is 1; by default, BUGCHECKFATAL is 0.

The PC displayed by the SDA SHOW CRASH command reflects the last chance handler rather than the location of the exception. R0, R1, AP, and FP in the SHOW CRASH display have been altered by the exception dispatching code. The PC, R0, R1, AP, and FP at the time of the exception can be obtained as described below.

When this bugcheck is signaled, signal and mechanism arrays have already been built on the current stack and are pointed to by the condition handler argument list, also on the stack. The condition handler argument list is pointed to by AP. The newest information on the stack is a frame generated by the call to the last chance handler.



1. The FP register displayed by SHOW CRASH points to the frame generated by SYS$CALL_HANDL's call to the last chance handler. The saved AP and saved FP in this call frame are the AP and FP at the time the exception occurred. Typically, the saved AP contains the address of the argument list with which the most recent procedure was called. This saved FP usually points to a frame which contains the address of the previous saved FP. If the stack is intact, these saved FPs can be used to trace back the sequence of calls that occurred in this process.

2. Use the AP displayed by SHOW CRASH to obtain the addresses of the signal and mechanism arrays.

3. Locate the mechanism array. Saved R0 and saved R1 are the registers' values at the time the exception occurred.

4. Skip 1 longword, the flags longword.

5. The next longword, the beginning of the signal array, contains an argument count, the number of longwords that follow. Use the count to identify all entries in the signal array. The number of exception parameters present is a function of exception type and can be 0, 1, or 2 longwords.

6. The exception type is a status value, e.g., C (hex) or SS$_ACCVIO. The DCL command

   ```
   $ EXIT %X<exception_type>
   ```

   writes the message text associated with the EXCEPTION TYPE status value. The V4 SDA command

   ```
   SDA> EVAL/CONDITION <exception_type>
   ```

   writes the message text associated with the exception type status value.

   Typically, the exception is one generated by "hardware" (or microcode), for example, access violation. "Hardware" generated exceptions are listed with a description of their associated exception parameters in Section 10.1 of the VAX/VMS System Services Reference Manual. See section EXCEPTIONS for information about the more common hardware exceptions.

7. The exception PC in the signal array is the instruction whose [attempted] execution resulted in the unexpected exec or kernel mode exception. Whether the PC points to the beginning of the instruction or the end depends on whether the exception was a trap (end), fault (beginning), or abort (beginning). The reference above specifies whether each exception is a trap, fault, or abort. Identify in what source module the PC is. See section VIRTUAL ADDRESSES. Often examining instructions around the PC is helpful enough to eliminate a microfiche search. Try the SDA command

SDA> EXAMINE/INSTRUCTION <exception_pc>-20;30

Figure out why the instruction generated an exception. For example, if an access violation occurred, look at the operands to see which access was in error.

8. Decipher the current stack to trace control flow. See section STACK PATTERNS.

Hints And Kinks

1. Not all access violations are signaled by microcode. The pagefault exception service routine, MMG$PAGEFAULT, may signal an access violation if a process incurs a pagefault for a page in another process's process header.

2. Note that for each V3 SDA COPY command used to copy the dump, the SP will be 8 bytes greater than its actual value; that is, SDA will show the SP pointing to a stack address 8 bytes higher than it should. This V3 bug has been corrected in V4.

3. The VAX instruction set is sufficiently rich that most random data can be interpreted as instructions. Most system code deals with binary integer and character data. This means that if an EXAMINE/INSTRUCTION display includes many packed decimal and/or floating point instructions, you are probably examining a data area or using a start address which is not an instruction boundary.

   One common error that results in a nonsensical display is to examine instructions in the bugcheck overlay area. During a crash, fatal bugcheck code and message text overlay resident system image code, beginning one page before label BUG$FATAL, for a length of about 12000 decimal or 3000 hex bytes.

Additional References

V3 VAX/VMS Internals and Data Structure Manual, Chapter 4, for general exception dispatching and details of exceptions signaled by VMS system software

VAX Archititure Standard (DEC Standard 032) or VAX-11 Architecture Reference Manual, Chapter 6, Exceptions and Interrupts

VAX/VMS System Services Reference Manual, Chapter 10, Condition-Handling Services

STACK PATTERNS

Tracing flow of control is often necessary to determine the history of unexpected or erroneous system behavior. Whether the problem is a crash or a hung process, you need to determine the sequence of events that led to the current state. Normally, the best way to attempt that is to examine the contents of the appropriate stack and identify the "footprints" left on it by the thread(s) of execution which used that stack. You identify the footprints to trace what code routine(s) ran in that access mode.

Some examples of footprints are the return PC following a JSB or BSBB/W instruction, stack frames built by a CALLS/G instruction, and information pushed on the stack for temporary storage. Some footprints are easily identifiable patterns unique to particular access modes; these are described in the following sections on deciphering particular stacks. Other possible footprints are simple patterns common to all access modes, such as the return PC resulting from execution of a JSB or BSBB/W instruction. Patterns that simple are not unique enough to identify easily other than through reading the code which made them.

In general, you should start at the highest addresses, or oldest information on the stack. This is not the only approach; sometimes it is more expeditious to work backwards, from newer information to older, particularly when there are nested call frames on the stack. However, it is usually more reliable to trace a thread of execution from its start than to infer earlier events.

A useful approximation is that once VMS is running, the processor will be running in process context in user mode until some interrupt occurs or until execution of an instruction results in an exception. So, the question "how did the process or system change to this stack?" is a good place to start when you're examining inner access mode stacks. You should usually be able to answer this question, perhaps drawing inferences from the older stack contents and the PSL and perhaps the PC at the time of crash. The next question is "what happened in this access mode?". You should usually be able to answer this by drawing inferences from the footprints on the stack.

On a well-behaved system, that the current mode in a process is exec implies that the kernel stack is empty; that the system is running in process context implies that the interrupt stack is empty. There are two exceptions to this you may encounter: when you examine the process current at the time of a fatal bugcheck from exec mode, its kernel stack contains footprints left by execution of the fatal bugcheck code; a process running in an outer mode may have Files-11 XQP context saved on the XQP's private kernel stack.

1. If you have already identified in which stack you are interested, go on to item 4.

2. If you are looking at a crashdump, determine what stack was current at the time of the crash. The SDA command SHOW CRASH displays the contents of the PSL from the crash. This indicates the system-wide interrupt stack or the kernel or exec stack of the current process. (The system cannot be crashed from a process running in user or supervisor mode.) Decode the PSL using the layout in the section RELATED REFERENCE MATERIAL or with the V4 SDA command EXAMINE/PSL. If the system crashes on the interrupt stack, the current process is frequently (but not always) irrelevant to the crash.

3. If you are looking at a hung process on the current system, the appropriate stack is usually the one for the process's current access mode, as determined from the saved PSL displayed by the SDA command

   SDA> SHOW PROCESS/INDEX=<x>/REGISTER

   Decode the PSL using the layout in the section RELATED REFERENCE MATERIAL or with the V4 SDA command EXAMINE/PSL.

4. See the section corresponding to the stack of interest:
   STACK PATTERNS - EXEC MODE STACK
   STACK PATTERNS - INTERRUPT STACK
   STACK PATTERNS - KERNEL MODE STACK

Hints And Kinks

1. Although "deciphering stacks" and "identifying virtual addresses" are listed as single and separate steps, in practice, they are usually repetitive and intertwined. For example, that a particular longword can be interpreted as a particular address should be confirmed in the context of what code was executing and manipulating that longword. Usually this requires that some piece of the stack be deciphered. Another example is that identifying a particular footprint on the stack may require or result in the identification of addresses within that footprint.

2. Note that for each V3 SDA COPY command used to copy the dump, the SP will be 8 bytes greater than its actual value; that is, SDA will show the SP pointing to a stack address 8 bytes higher than it should. This V3 bug has been corrected in V4.

3. Occasionally you may find a stack whose contents make little or no sense. Although VMS keeps the stacks longword aligned almost all the time, you may be trying to examine an unaligned stack. Try one or more of the following SDA commands to see if any recognizable footprints or patterns emerge.

   SDA> SHOW STACK <low_address-1>:<high_address-1>

```
SDA> SHOW STACK <low_address-2>:<high_address-2>
SDA> SHOW STACK <low_address-3>:<high_address-3>
```

Additional References

VAX Architecture Standard (DEC Standard 032) or VAX-11 Architecture Reference Manual, Chapter 6, Exceptions and Interrupts; Chapter 7, Process Structure

V3 VAX/VMS Internals and Data Structures Manual, Section 1.3, Hardware Implementation of Operating System Kernel

STACK PATTERNS - EXEC MODE STACK

The system runs on the exec mode stack of the current process to service CHME exceptions and to deliver exec mode ASTs. In practice, most of the processes you see in exec mode are executing exec-mode system services or RMS services.

Possible patterns that you may see on a exec stack are described below. You may see these patterns more than once on the same stack, and you may see more than one of them. Some patterns should not be followed by other described patterns. Each pattern description includes any such restrictions.

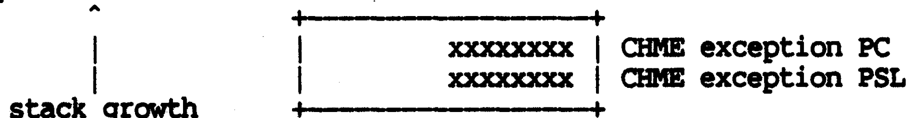
1. Identify the initial reason for the exec mode switch using the patterns below.

2. Account for as much of the stack as possible, using the patterns below.

3. Read the relevant code and try to determine what happened based on stack footprints, register contents, and data structure alterations made by the code. Use the section VIRTUAL ADDRESSES wherever appropriate.

Exec Mode Stack Patterns

1. One common pattern, most likely to be the highest (oldest) two longwords on the stack, is an exception PSL and PC from the system service vector area.



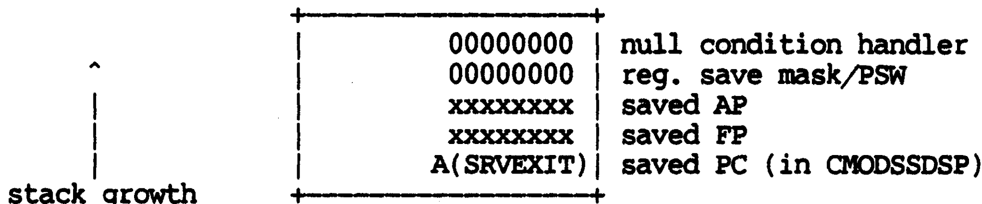
If SDA's symbolic interpretation of the hypothetical exception PC is of the form SYS$<service name> + 6, then these two longwords are a CHME exception PC and PSL, and the symbolic name shown is accurate. If SDA's symbolic interpretation is an offset other than 6 from a system service vector name, subtract 6 from the address, and determine to which system service vector, if any, the address corresponds, following the steps in the section SYSTEM SERVICE VECTORS.

When the process is already executing in exec mode and an exec mode system service is requested, you should see the frame from the CALL to the system service vector at stack addresses higher (older) than the CHME exception PC and PSL.

When you see a CHME exception PC and PSL on the exec stack, there

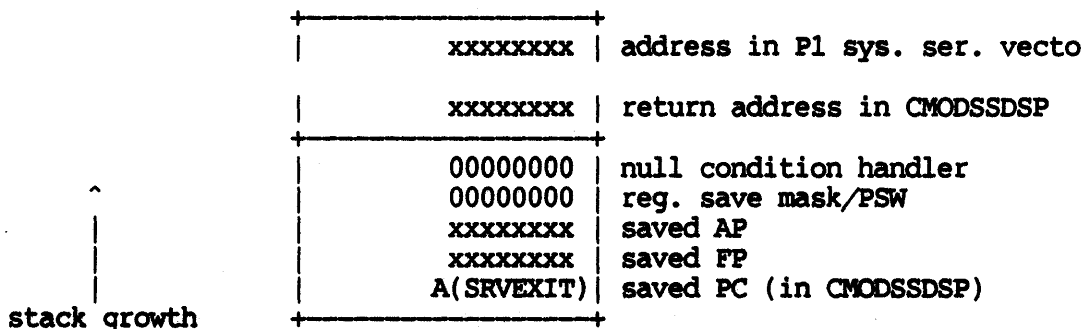
will usually be a change mode dispatcher call frame (described below) at the next lower (newer) stack addresses. The change mode dispatcher simulates a CALL to the real system service procedure, which is usually a global EXE$<service_name>, where <service_name> is from the SYS$<service_name> global.

2. The change mode dispatcher builds "by hand" a 5 longword call frame prior to entering a system service procedure. This call frame should never be the oldest information on the exec stack.



The V3 address of SRVEXIT is 8000CFE6. The V4 address of SRVEXIT is 8000FDCE. The saved AP is the address of the argument list with which the system service was called. The saved FP, typically a P1 address, is the address of the previous call frame.

3. In the case of change mode dispatching to a loadable exec mode system service, whether user added or VMS supplied, there are two extra longwords on the stack at addresses lower (newer) than the change mode dispatcher call frame.

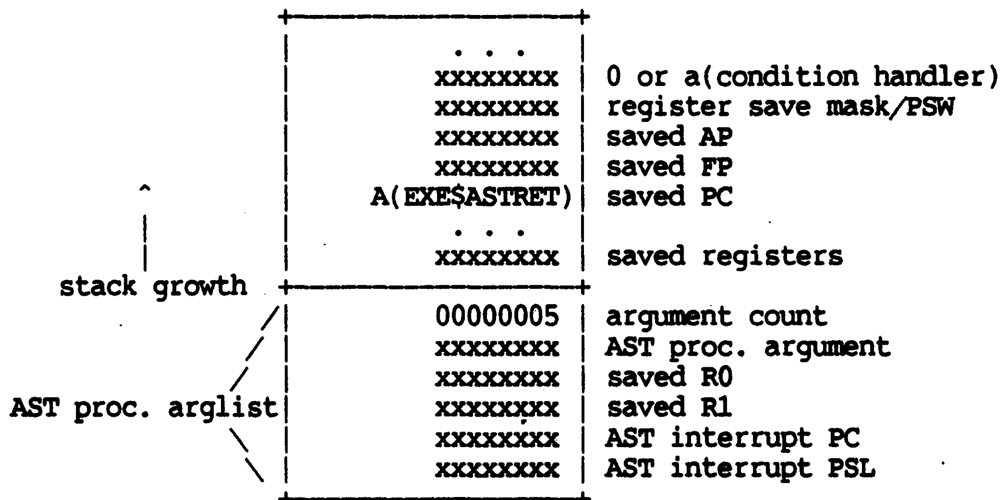


The V3 address of SRVEXIT is 8000CFE6. The V3 return address in CMODSSDSP is 8000CEBA. The V3 address in P1 sys. ser. page should fall within the range CTL$A_DISPVEC + 100 to CTL$A_DISPVEC + 1FF or 7FFEA100 to 7FFEA1FF.

The V4 address of SRVEXIT is 8000FDCE. The V4 return address in CMODSSDSP is 8000FCC7. The V4 address in P1 sys. ser. page should fall within the range CTL$A_DISPVEC + 100 to CTL$A_DISPVEC + 1FF or 7FFE6100 to 7FFE61FF.

4. One possible pattern results from an IPL 2 AST delivery interrupt and dispatch to an exec mode AST procedure. The IPL 2 AST delivery interrupt service routine REI's to exec mode to deliver the AST. This pattern is possible as the oldest exec stack

contents and as intermediate exec stack contents. That is, delivery of an exec mode AST is a reason for an access mode switch to exec, and delivery of an exec mode AST is possible to a process running in exec mode. This pattern includes the argument list with which the AST is entered and the frame built by CALLing the AST procedure.



The V3 address of EXE$ASTRET is 80008AFA. The V4 address of EXE$ASTRET is 80009E5E.

An exec mode AST should not be interrupted for delivery of another exec mode AST. That is, you should not see this pattern on an exec mode stack more than once.

Hints And Kinks

1. Occasionally you may find a stack whose contents make little or no sense. Although VMS keeps the stacks longword aligned almost all the time, you may be trying to examine an unaligned stack. Try one or more of the following SDA commands to see if any recognizable footprints or patterns emerge.

```
SDA> SHOW STACK <low_address-1>:<high_address-1>
SDA> SHOW STACK <low_address-2>:<high_address-2>
SDA> SHOW STACK <low_address-3>:<high_address-3>
```

Additional References

V3 VAX/VMS Internals and Data Structures Manual, Chapter 9, System Service Dispatching; Chapter 7, AST Delivery

STACK PATTERNS - INTERRUPT STACK

The system switches to the interrupt stack to service all hardware interrupts, to service all software interrupts above IPL 3, and to service some serious exceptions such as machine check. In addition, the system runs on the interrupt stack in the IPL 3 interrupt service routine after it has taken the current process out of execution and before it has placed a new process into execution.

Note that one interrupt stack thread of execution may be interrupted by another higher priority interrupt. Because of this, you usually should begin with the newer stack contents rather than older.

Because interrupts are asynchronous, because one interrupt may interrupt another interrupt service routine, and because many System Control Block vectors point to instructions that jump elsewhere in the system, deciphering the interrupt stack is more difficult than the kernel or exec stack.

1. If the stack contains more than just a PC and PSL, try to associate system space addresses it contains with a particular exception or interrupt service routine (E/ISR). Look particularly at contents newer than the interrupt PC and PSL, and keep in mind that most E/ISRs begin by saving registers (commonly R0 through R5) on the stack prior to using them. You might try to find the newest PC-PSL pair on the stack, then skip four to six longwords (saved registers), and then look for footprints.

   The general idea is to try to find a footprint (for example, a return PC following a JSB instruction) from the E/ISR on the stack, rather than beginning with the bugcheck PC, which may be in a routine called by another routine .... called by the E/ISR. See the IPL Usage table below and the notes following it for hints on determining E/ISR addresses. Use the section VIRTUAL ADDRESSES to translate any addresses of interest to source module names and offsets.

2. If you were unable to find a footprint from an E/ISR, try to associate the bugcheck PC with a particular E/ISR. Use the section VIRTUAL ADDRESSES to translate any addresses of interest to source module names and offsets.

3. If you cannot associate an address with a particular interrupt or exception service routine, the PSL<IPL> may be helpful. Try to associate the IPL with a particular E/ISR using the IPL USAGE table and notes below. This may be difficult since many service routines raise IPL. Some service routines save the previous IPL on the stack prior to raising it. If all else fails, look for a saved IPL (a hex number between 4 and 1D) on the stack as a clue to what interrupt may have occurred.

4. If you have identified the E/ISR, or if you have a candidate E/ISR, read the E/ISR code and that of any routines it calls, checking for footprints on the stack, data structure changes,

register contents, etc. to corroborate or disprove your hypothesis. Use the section VIRTUAL ADDRESSES wherever appropriate.

Interrupt Stack Priority Level Usage Table

HEX IPL	TYPE	E/ISR	SOURCE MODULE	NOTE(S)
0	illegal			1
1	unexp. interrupt			2
3	rescheduling	SCH$RESCHED	[SYS]SCHED	
	782 rescheduling	MPS$RESCHED	[MP]MPSCHED	
4	IOPOST	IOC$IOPOST	[SYS]IOCIPOST	
5	XDELTA request	INI$MASTERWAKE	[SYS]INIT	3
	782 rescheduling	MPS$RESCHED5	[MP]MPSCHED	
6	fork dispatch	EXE$FRKIPL6DSP	[SYS]FORKCNTRL	4,10
7	software timer	EXE$SWTIMINT	[SYS]TIMESCHDL	5
8	fork dispatch	EXE$FRKIPL8DSP	[SYS]FORKCNTRL	5,6,10
9	fork dispatch	EXE$FRKIPL9DSP	[SYS]FORKCNTRL	7,10
A	fork dispatch	EXE$FRKIPL10DSP	[SYS]FORKCNTRL	8,10
B	fork dispatch	EXE$FRKIPL11DSP	[SYS]FORKCNTRL	9,10
C	console request	EXE$IPCONTROL	[SYS]IPCONTROL	3
D	unused			
E	unused			
F	782 XDELTA request	INI$MASTERWAKE	[SYS]INIT	3
10	unused			
11	unused			
12	unused			
13	unused			
14	device interrupts (BR4/SBIREQ4)			11
	780, 785, 730 console devices			
	750, 8600, MicroVAX I, MicroVAX II console terminals			
15	device interrupts (BR5/SBIREQ5)			11
16	device interrupts (BR6/SBIREQ6)			11
	uVAX interval timer	EXE$HWCKLINT	[SYS]TIMESCHDL	
17	device interrupts (BR7/SBIREQ7)			11
	750, 8600 console block storage			
18	interval timer	EXE$HWCKLINT	[SYS]TIMESCHDL	
19	cpu-specific interrupt			12
1A	cpu-specific interrupt			12
1B	cpu-specific interrupt			12
1C	cpu-specific interrupt			12
1D	cpu-specific interrupt			12
1E	powerfail	EXE$POWERFAIL	[SYS]POWERFAIL	
1F	machine check exc.	EXE$MCHK	[SYSLOA]MCHECKxxx	12,13
	invalid ksp exc.	EXE$KERSTKNV	[SYS]EXCEPTION	

NOTES on INTERRUPT PRIORITY LEVEL USAGE TABLE

1. Being on the interrupt stack at IPL 0 is an inconsistent state that should be very short-lived. That is, if any interrupt or exception occurs while the processor is in this state, the REI from its service routine should result in a reserved operand exception (on the interrupt stack) to which VMS's normal reaction is a fatal bugcheck.

2. The IPL 1 interrupt is currently unused. Any IPL 1 interrupt is due to an error, most likely a hardware error. IPL can also be raised to 1 as the result of executing a SVPCTX instruction from IPL 0, as could happen through hardware or software error; the microcode raises IPL to 1 because being on the interrupt stack at IPL 0 is an inconsistent and illegal state.

3. These interrupt are requested only by a human at the console terminal depositing into the software interrupt request register. The interrupts at IPL 5 and IPL hex F are used to awaken XDELTA, if present, through a BPT instruction at a location known to XDELTA. The interrupt at IPL hex C is used primarily to cancel mount verification or force recomputation of cluster quorum instead of crashing the system and may also be used to awaken XDELTA. If someone was using XDELTA prior to a crash, you should consider the possibility that user corruption of data structures or interference with normal system operation contributed to the crash.

4. IPL 6 fork dispatching is used primarily by V3 drivers which run at higher fork IPLs and which need to create a thread to execute some code which would affect a system wide data base synchronized at IPL$_SYNCH. Many fork IPL 8 drivers which created IPL 6 forks for this reason still do under V4, although IPL$_SYNCH is now 8. IPL 6 is also the fork IPL of the connect to interrupt driver.

5. Under V3, IPL 7 is the value of IPL$_SYNCH, the IPL used to serialize access to system databases such as the scheduler and memory management databases. It is also the IPL of the software timer interrupt. Under V4, the software timer interrupt is requested at IPL 7, but EXE$SWTIMINT runs primarily at IPL 8, the V4 value of IPL$_SYNCH.

6. Fork IPL 8 is used by most device drivers. It is the IPL associated with System Communication Services. It is also the IPL associated with distributed lock management and cluster management code.

7. Fork IPL 9 is unused by any known VMS drivers.

8. Fork IPL A hex is unused by any known VMS drivers.

9. Fork IPL B hex is used by the mailbox driver ([SYS]MBDRIVER) and shared memory mailbox driver ([DRIVER]MBXDRIVER). This IPL is also used for synchronizing nonpaged pool variable list

allocation and deallocation.

10. A fork process is entered with R5 pointing to the fork block, which is frequently part of some other data structure, such as a UCB or CDRP. If the system crashes at a fork IPL, formatting the fork block may be helpful; in particular, the offset FKB$L_FPC usually contains the address to which the fork dispatching code passed control. Type the following SDA commands to format the data structure.

```
SDA> READ SYS$SYSTEM:SYSDEF.STB !read symbol definitions
SDA> FORMAT @R5/TYP=FKB         !format fork block portion
SDA> FORMAT @R5                 !format anything else
```

11. Device ISRs are entered at hex IPLs 14 through 17. Also driver fork processes sometimes raise IPL to that associated with their devices to block interrupts during a critical section of code.

    To find out what devices are present on a system and their associated device IPLs, display the I/O database with the SDA command SHOW DEVICE. The device IPL is displayed as part of the UCB information.

    To get a list of many ISR addresses and global names, type the following SDA command.

```
SDA> SHOW STACK (@EXE$GL_SCB+F0):(@SWP$GL_BALBAS -4)
```

    The low order two bits of the vector do not contain address information. In practice, this means that you must subtract one from each of the contents displayed. This display includes the addresses of the console ISRs, unexpected interrupt service routines, nexus ISRs, and any directly vectored UNIBUS ISRs.

    In most cases, there is an extra level of indirection in that the SCB contents are the addresses of dispatch instructions. Under V4, the real console interrupt routines are within SYSLOAxxx.EXE images. For many nexus ISRs and any directly vectored UNIBUS ISRs, the addresses point to a Controller Request Block (CRB) JMP to the appropriate ISR (generally within a driver image).

    If you suspect an address to be within a device driver ISR, first determine which driver using the section VIRTUAL ADDRESSES - SYSTEM SPACE. The driver name should be of the form <device_name>DRIVER. Then examine any Controller Request Block (CRB) associated with that driver to see the addresses of ISRs within that driver. Type the following SDA command.

```
SDA> SHOW DEVICE <device_name>.
```

12. The cpu-specific ISRs and the machine check ESR are part of code loaded during system initialization. The SCB vectors point to instructions which dispatch into the loaded code. The sources are in [SYSLOA]MCHECKxxx, where xxx designates a cpu. "xxx" in

the MCHECKxxx names is the same as in the SYSLOAxxx.EXE names.

For further information on the SYSLOAxxx names and on the mechanisms for dispatching into SYSLOA, see subsection SYSLOAXXX.EXE in the section VIRTUAL ADDRESSES. For information on cpu-specific interrupts, see section CPU-SPECIFIC INTERRUPTS. For information on machine checks, see section MACHINE CHECKS. Although these interrupts occur at IPLs in the hex range 19 to 1D, their ISRs immediately raise IPL to 1F.

13. Microcode initiation of a machine check exception or invalid kernel stack exception causes an IPL raise to hex 1F. Drivers and other system code occasionally raise IPL to 1F to block all interrupts. Also, the system runs at this IPL during system initialization and restart following a halt.

Hints And Kinks

1. Occasionally you may find a stack whose contents make little or no sense. Although VMS keeps the stacks longword aligned almost all the time, you may be trying to examine an unaligned stack. Try one or more of the following SDA commands to see if any recognizable footprints or patterns emerge.

```
SDA> SHOW STACK <low_address-1>:<high_address-1>
SDA> SHOW STACK <low_address-2>:<high_address-2>
SDA> SHOW STACK <low_address-3>:<high_address-3>
```

Additional References

V3 VAX/VMS Internals and Data Structures Manual, Chapter 5, Hardware Interrupts; Chapter 6, Software Interrupts; Section 8.3, Machine Check Mechanism

VAX Architecture Standard (DEC Standard 032) or VAX-11 Architecture Reference Manual, Section 6.6, System Control Block

VAX Architecture Standard (DEC Standard 032) Section 12.4, Format and [Cpu-specific] Contents of the System Control Block

STACK PATTERNS - KERNEL MODE STACK

The system runs on the kernel mode stack of the current process to service IPL 2 AST delivery interrupts, to deliver kernel mode ASTs, to service most exceptions, and to execute kernel mode system services (a special case of exception servicing). In practice, many exceptions are ultimately serviced by process-declared condition handlers in the access mode that incurred the exception, and most of the processes you will see in kernel mode are executing kernel mode system services.

In addition, the SWAPPER process and NULL process execute only in kernel mode, and newly created processes execute EXE$PROCSTRT and any code it invokes in kernel mode.

Possible patterns that you may see on a kernel stack are described below. You may see these patterns more than once on the same stack, and you may see more than one of them. Some patterns should not be followed by other described patterns. Each pattern description includes any such restrictions.

1. First, see the section KERNEL STACK LOCATIONS to identify its high and low limits and to determine whether the process of interest is running on its usual kernel stack.

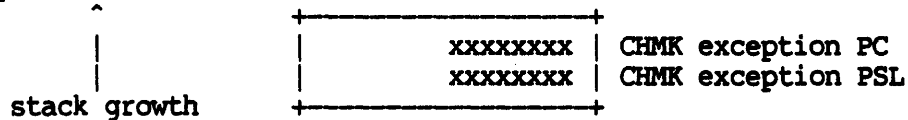
2. If the process of interest is NOT the SWAPPER or NULL job, identify the initial reason for the kernel mode switch, using the patterns below. Account for as much of the stack as possible, using the patterns below. Read the relevant code and try to determine what happened based on stack footprints, register contents, and data structure alterations made by the code. Use the section VIRTUAL ADDRESSES wherever appropriate.

3. If the process of interest is the SWAPPER, read its code ([SYS]SWAPPER) following the path from label LOOP. (Whenever the SWAPPER is awakened, it resumes at a location near label LOOP, a local symbol.) Try to determine what it did based on stack footprints, register contents, and data structure alterations made by the code.

4. The NULL job consists of 1 instruction at location EXE$NULLPROC
```
          10$:  BRB  10$
```
   Under normal circumstances the NULL job should incur no exceptions and should not receive ASTs. It may, however, be interrupted by a spurious AST interrupt intended for the previous process.

Kernel Mode Stack Patterns

1. One common pattern, most likely to be the highest (oldest) two longwords on the stack, is an exception PSL and PC from the system service vector area.

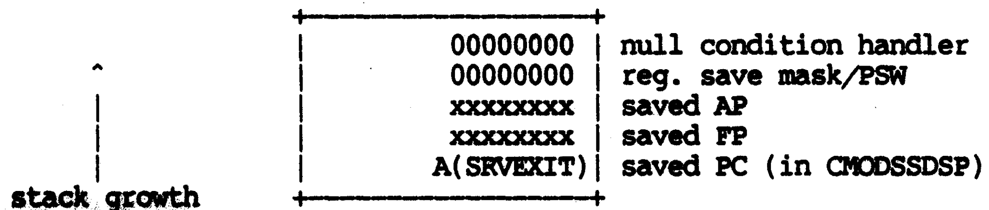


   If SDA's symbolic interpretation of the hypothetical exception PC is of the form SYS$<service_name> + 6, then these two longwords are a CHMK exception PC and PSL, and the symbolic name shown is accurate. If SDA's symbolic interpretation is an offset other than 6 from a system service vector name, subtract 6 from the address, and determine to which system service vector, if any, the address corresponds, following the steps in the section SYSTEM SERVICE VECTORS.

   When the process is already executing in kernel mode and a kernel mode system service is requested, you should see the frame from the CALL to the system service vector at stack addresses higher (older) than the CHME exception PC and PSL.

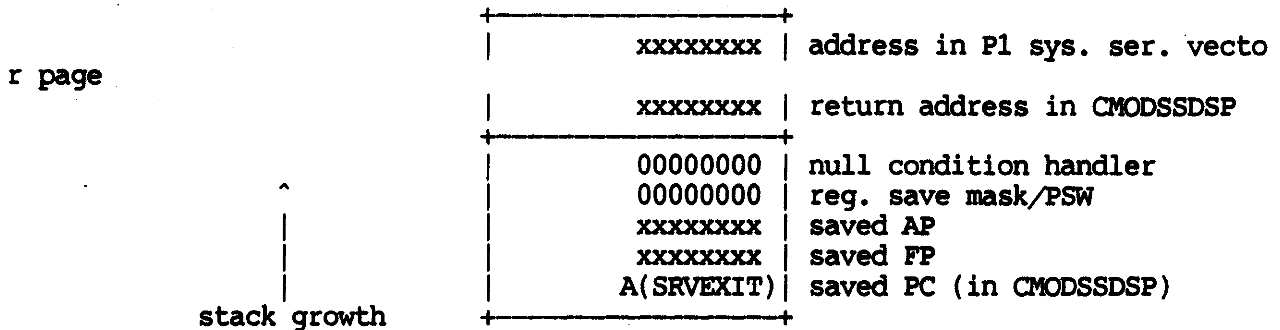
   When you see a CHMK exception PC and PSL on the exec stack, there will usually be a change mode dispatcher call frame (described below) at the next lower (newer) stack addresses. The change mode dispatcher simulates a CALL to the real system service procedure, which is usually a global EXE$<service_name>, where <service_name> is from the SYS$<service_name> global.

2. The change mode dispatcher builds "by hand" a 5 longword call frame prior to entering a system service procedure. This call frame should never be the oldest information on the kernel stack.



   The V3 address of SRVEXIT is 8000CFE6. The V4 address of SRVEXIT is 8000FDCE. The saved AP is the address of the argument list with which the system service was called. The saved FP, typically a P1 address, is the address of the previous call frame.

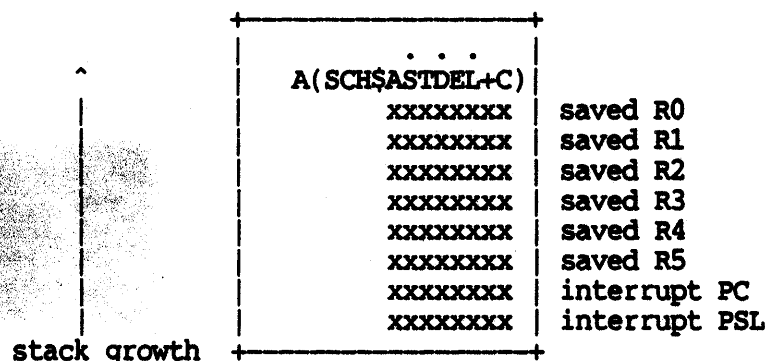
3. In the case of change mode dispatching to a loadable kernel mode system service, whether user added or VMS supplied there are two extra longwords on the stack at addresses lower (newer) than the change mode dispatcher call frame.



The V3 address of SRVEXIT is 8000CFE6. The V3 return address in CMODSSDSP is 8000D11E. The V3 address in P1 sys. ser. page should fall within the range CTL$A_DISPVEC to CTL$A_DISPVEC + FF or 7FFEA000 to 7FFEA0FF.

The V4 address of SRVEXIT is 8000FDCE. The V4 return address in CMODSSDSP is 8000FF0D. The V4 address in P1 sys. ser. page should fall within the range CTL$A_DISPVEC to CTL$A_DISPVEC + FF or 7FFE6000 to 7FFE60FF.

4. One possible pattern results from an IPL 2 AST delivery interrupt and dispatch to a special kernel mode AST. This pattern is possible as the oldest kernel stack contents and as intermediate kernel stack contents. That is, delivery of an AST is a reason for an access mode switch to kernel, and delivery of a kernel mode AST is possible to a process running in kernel mode at an IPL below 2. During execution of the special kernel AST, the process's current IPL should be no lower than 2, and the stack should contain the following pattern.

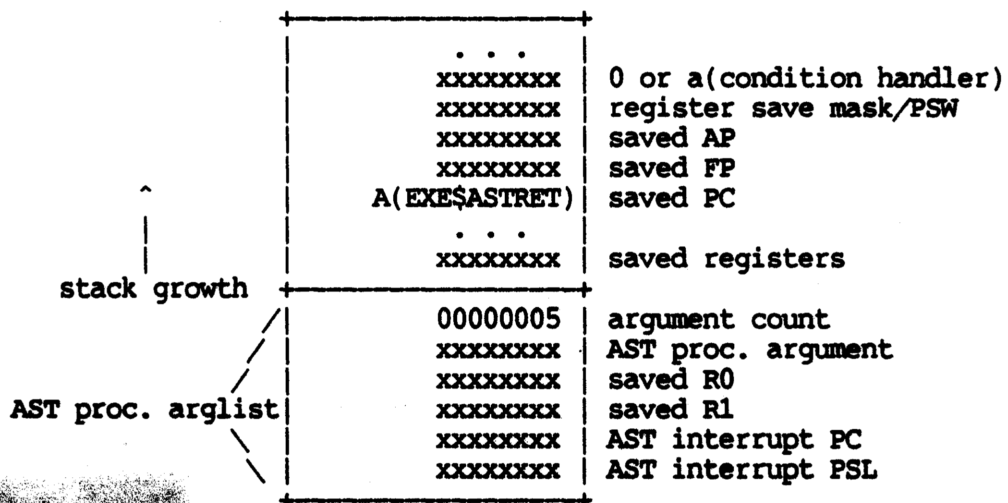


The V3 address of SCH$ASTDEL + C is 80008A38. The V4 address of SCH$ASTDEL + C is 80009D9C. SCH$ASTDEL is within the module [SYS]ASTDEL.

During the execution of most special kernel ASTs, R5 contains the address of the AST Control Block (ACB). Well behaved special kernel AST routines do not invoke system services; therefore, you should not expect to see the pattern above followed (i.e., at lower addresses) by a system service vector CHMK exception PC and PSL and change mode dispatcher call frame. While IPL is at 2 or higher, any other AST delivery interrupts are blocked, so you should not expect to see this pattern followed by the pattern for delivery of another special kernel AST or the pattern for delivery of a normal kernel AST.

The exception to both those restrictions occurs in V3 process deletion. The special kernel AST routine DELETE lowers IPL to 0 to allow delivery of other kernel ASTs, from subprocesses of the process being deleted. In addition, DELETE requests various system services.

5. One possible pattern results from an IPL 2 AST delivery interrupt and dispatch to a normal kernel mode AST. This pattern is possible as the oldest kernel stack contents and as intermediate kernel stack contents. That is, delivery of an AST is a reason for an access mode switch to kernel, and delivery of a kernel mode AST is possible to a process running in kernel mode at an IPL below 2. This pattern includes the argument list with which the AST is entered and the frame built by CALLing the AST procedure.



The V3 address of EXE$ASTRET is 80008AFA. The V4 address of EXE$ASTRET is 80009E5E.

A normal kernel AST procedure may be interrupted for delivery of a special kernel AST but not for delivery of another kernel mode AST.

The exceptions to that restriction occur in V4 process deletion and suspension. The kernel AST procedure DELETE clears the

PCB$B_ASTACT bit to enable further normal kernel mode ASTs to be delivered to the process. The normal kernel AST procedure SUSPEND also clears the PCB$B_ASTACT bit to enable further normal kernel mode ASTs to be delivered to a process with outstanding Files-11 XQP activity.

6. If the process of interest is running on the Files-11 XQP stack, the oldest (highest) information on the stack will be a call frame from routine DISPATCH in [F11X]DISPATCH (entered on the previous kernel stack via normal kernel AST) to routine DISPATCHER in [F11X]DISPAT (which runs on the XQP's private kernel stack).

7. Another event that can leave kernel stack footprints is an exception. For most exceptions, VMS dispatches to a condition handler declared by the process; that is, VMS locates the condition handler, cleans up the kernel stack, REIs to the access mode that incurred the exception, and CALLs the condition handler. Conceivably, a bugcheck could occur somewhere in this sequence prior to the kernel stack cleanup and REI. In this case, the older stack contents should contain (partial) signal and mechanism arrays and resemble the stacks pictured in sections INVEXCEPTN BUGCHECK and SSRVEXCEPT BUGCHECK.

8. Another possibility is that an exception occurred which VMS handles itself in kernel mode, for example, translation not valid. In this case, there may be a footprint on the stack which is an address within an exception service routine.

Hints And Kinks

1. Occasionally you may find a stack whose contents make little or no sense. Although VMS keeps the stacks longword aligned almost all the time, you may be trying to examine an unaligned stack. Try one or more of the following SDA commands to see if any recognizable footprints or patterns emerge.

```
SDA> SHOW STACK <low_address-1>:<high_address-1>
SDA> SHOW STACK <low_address-2>:<high_address-2>
SDA> SHOW STACK <low_address-3>:<high_address-3>
```

Additional References

V3 VAX/VMS Internals and Data Structures Manual, Chapter 4, Condition Handling; Chapter 9, System Service Dispatching; Chapter 7, AST Delivery

SYSTEM SERVICE VECTORS

System service vectors are system global procedure names CALL'ed to invoke a particular service. They contain small procedures which execute in the mode of the caller and which serve as a bridge between the caller and the actual procedure(s) which implement the service request. The actual procedures may be part of SYS.EXE or some other loaded image such as RMS.EXE and may execute in an inner access mode.

System Service Vector Addresses

The values of system service vectors are fixed across all VMS releases, so that user programs need not be relinked for a new VMS version. System service vectors are located in the lowest pages of system space. They are also located (doubly-mapped) in P1 space. The P1 definitions were added in V3 to allow per-process redirection of selected system services. Per-process redirection of selected system services is currently unused.

As of V3, the Linker uses by default the module SYS$P1_VECTOR in SYS$LIBRARY:STARLET.OLB to resolve system service vector globals to P1 space addresses. Earlier Linkers resolved system service vector globals to system space addresses. These system space addresses are defined by the module SYS$VECTOR in SYS$LIBRARY:STARLET.OLB.

The only system service global names known to SDA by default are those referenced within SYS.EXE, a small subset of the total system service and RMS service globals. This means that SDA can make valid symbolic interpretations of only those system service vector addresses.

System Service Vector Contents

System service vectors begin with a register save mask. What follows the save mask varies, in part as a function of the access mode in which the actual procedure executes. For the very few system services that execute in the access mode of the caller, the save mask is followed only by a JMP to the actual procedure, which is most often part of SYS.EXE. For services that execute in an inner mode, the save mask is followed by a CHME or CHMK instruction with an operand identifying the service request and, usually, by a RET to return to the invoker. However, some system service vectors, commonly called "composite vectors", contain lengthier procedures; the CHME/K instruction is followed by something other than a RET.

One example of a composite system service vector is SYS$QIOW, which includes a CHMK #QIO and a CHMK #WAITFR request. The RMS system service vectors are composite vectors that branch to RMS synchronization code which conditionally stalls the process until all I/O associated with its request is complete. The RMS synchronization

code uses the status in the FAB or RAB associated with the request to determine whether the I/O is complete and executes a CHMK #WAITFR if it is not.

V4 contains a number of additional composite system service vectors; all the services which guarantee not to return to the invoker until all I/O associated with the request is complete have composite vectors. These V4 system services, called "synchronous services", use a combination of status block (for example, IOSB or lock status block) contents and event flag to test for I/O completion. Examples of synchronous services are $QIOW, $UPDSECW, and $GETJPIW.

The V4 composite system service vectors for these synchronous services contain a CHMx instruction that dispatches in the usual way to the actual procedure, whose responsibilities include clearing the associated event flag and zeroing the contents of the status block if one was specified. The CHMx instruction is followed by a branch to common synchronization code.

The synchronization code first tests the status block, if one was specified. If its status word is zero, the synchronization code waits on the event flag. Whenever the flag is set and the process placed into execution, the synchronization code tests the status word and, if it is zero, clears the flag and waits for it again. If the user specifies a status block with the system service request, this mechanism eliminates the traditional problem of returning before I/O is complete as a result of concurrent multiple uses of the same flag.

System Service Vector Stack Footprints

Executing the CALLS/G to a system service vector always generates a call frame on the current access mode stack.

For system services that execute in the mode of the caller (for example, $FAO), the actual procedure executes on this stack and RETurns to the instruction following the call to the system service vector.

For inner access mode system services, executing the CHME/K instruction causes an exception. When such a system service request is made from an outer mode, the CHMx exception results in a stack switch and an access mode change to the mode in which the system service will actually be performed. The address following the CHMx instruction is the exception PC saved on the target access mode stack, whether the access mode in which the CHMx is executed is outer to or the same as the target access mode.

These exception PCs are commonly found as the oldest contents on a kernel or exec mode stack. Resolving such an address to its system service name is an important step in tracing what happened in exec or kernel mode in a process which is hung or a process in whose context the system crashed. Note that these addresses within system service

vectors may also appear in the middle of an exec or kernel mode stack, as the result of one system service's requesting another or as the result of an AST procedure's requesting a system service.

The exception PC following a CHME/K instruction is frequently the address corresponding to SYS$<service_name> + 6 and, thus, relatively easy to resolve as a system service global name. (6 = 2 bytes of register save mask + 1 byte of CHMx opcode + 1 byte of operand specifier + 2 bytes of immediate operand) However, the composite system service vectors contain or branch to synchronization code which issues other CHME/K instructions. As a result, identifying the original system service global name is more difficult in these cases.

Resolving System Service Vector Addresses

To determine whether a particular address is within the system service vectors and which system service it is, follow the directions below.

1. The P1 vectors begin at the symbol P1SYSVECTORS. The V3 value of P1SYSVECTORS is 7FFEDE00. The end of the V3 P1 vectors is 7FFEE5FF. The V4 value of P1SYSVECTORS is 7FFEDE00. The end of the V4 P1 vectors is 7FFEE7FF.

   To create a list of P1 space system service vector addresses and their global names, use the following commands.

   ```
   $ LIBR/OUT=P1VECTOR/EXTRACT=SYS$P1_VECTOR -
   $       SYS$LIBRARY:STARLET.OLB
   $ LINK/NOEXE/MAP/FULL P1VECTOR
   ```

2. If the address is between 80000000 and the symbol MMG$A_ENDVEC, it is within the system space system service vectors. The V3 value of MMG$A_ENDVEC is 80000800. The V4 value of MMG$A_ENDVEC is 80000A00.

   To create a list of system space system service vector addresses and their global names, use the following commands.

   ```
   $ LIBR/OUT=SYSVECTOR/EXTRACT=SYS$VECTOR -
   $       SYS$LIBRARY:STARLET.OLB
   $ LINK/NOEXE/MAP/FULL SYSVECTOR
   ```

3. If the address is not within either of the ranges above, it is not a system service vector.

4. If the address is within one of the ranges above, search the relevant map by eye, with the SEARCH utility, or with your favorite editor to locate the address of interest and obtain its corresponding global name.

5. If your address is within the system service vector range but you cannot find a corresponding global name of the form SYS$<service_name>, then the address is likely to follow an aditional CHMx instruction within a composite system service vector. To identify the original system service, locate the original call instruction using the directions below.

   - If the stack contains a change mode dispatcher call frame, then use its SAVED FP value as the address of the frame built by the original call. (See the section STACK PATTERNS - KERNEL MODE for a layout of the change mode dispatcher call frame.)

   - If there is no change mode dispatcher call frame, then use the contents of the FP register.

   - In the frame pointed to by the SAVED FP value or FP register, locate the SAVED PC. (See the section RELATED REFERENCE MATERIAL for the layout of a call frame.) This should be the address of the instruction following the call to the system service vector.

   - Issue the following SDA command to see the original CALLS/G instruction

     ```
     SDA> EXAM/INST <saved_pc>-10;10
     ```

   - The target of the CALLS/G is the address of the original system service vector and should be the value of a SYS$<service_name>. Search the relevant map to locate the corresponding global name if SDA is unable to resolve it.

Additional References

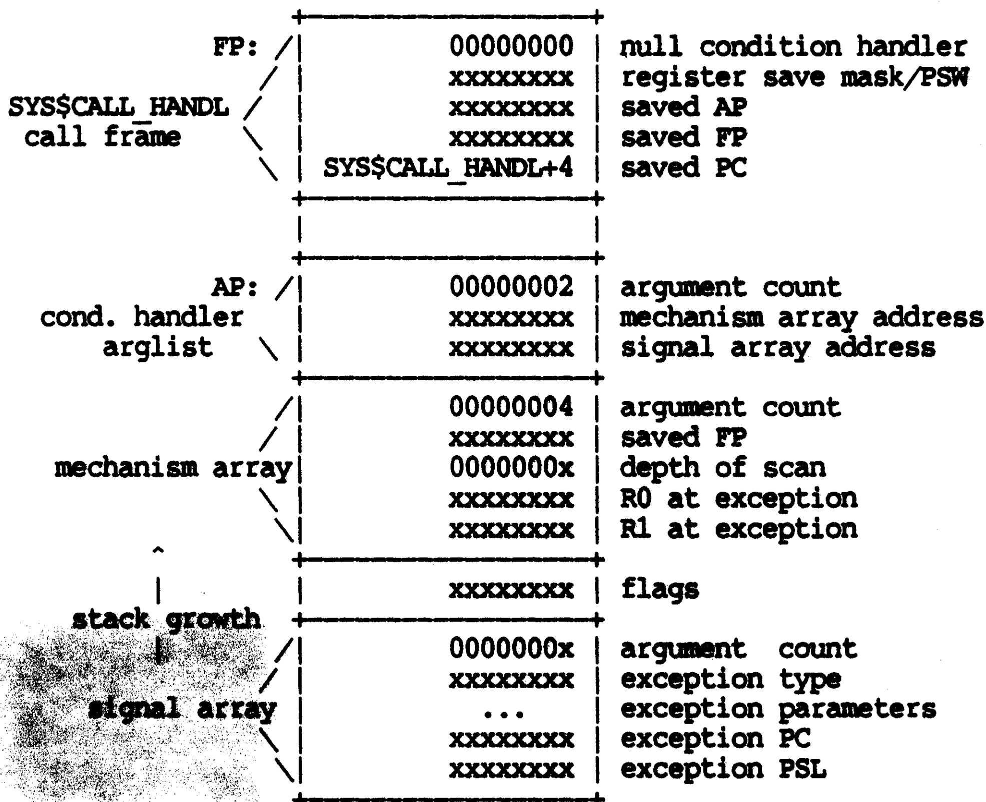
V3 VAX/VMS Internals and Data Structure Manual, Chapter 9, for more information on system service vector contents and dispatching to system service procedures

UNXSIGNAL BUGCHECK

The UNXSIGNAL bugcheck is signaled by a V4 Files-11 XQP (ODS-2), V3 Files-11 ACP (ODS-2), a magtape ACP, or Files-11 ODS-1 ACP condition handler. In all cases, the bugcheck indicates that some unexpected exception has occurred. In all cases, the stack patterns are similar, with condition handler argument list, signal and mechanism arrays, and a call frame to the condition handler as newest stack contents.

In the case of the Files-11 XQP, the bugcheck text "unexpected signal in ACP", is historically true but somewhat misleading in that V4 Files-11 ODS-2 support is procedure-based code that runs in the context of the process requesting the I/O rather than as an ACP.

The PC displayed by the SDA SHOW CRASH command reflects the condition handler signaling the bugcheck rather than the location of the exception. R0, R1, AP, and FP in the SHOW CRASH display have been altered by the exception dispatching code. The PC, R0, R1, AP, and FP at the time of the exception can be obtained as described below. The stack pattern follows.



1. The SHOW CRASH output includes the name of the current process and its image. This indicates whether the bugcheck was signaled from code within a Files-11 ODS-1 process, a magtape ACP, a V3

Files-11 ODS-2 process, Files-11 XQP procedures running in some process's context.

The process names of ACPs are constructed from the mounted device name, an ACP type letter, and "ACP". For Files-11 ODS-1, the type letter is A; for Files-11 ODS-2, the letter is B. For magtape ACPs, the type letter is A. For example, DUA0BACP is an V3 Files-11 ODS-2 ACP.

2. The FP register displayed by SHOW CRASH points to the frame generated by SYS$CALL_HANDL's call to the condition handler. The saved AP and saved FP in this call frame are the AP and FP at the time the exception occurred. Typically, the saved AP contains the address of the argument list with which the most recent procedure was called. This saved FP usually points to a frame which contains the address of the previous saved FP. If the stack is intact, these saved FPs can be used to trace back the sequence of calls that occurred in this thread of execution.

3. Use the AP displayed by SHOW CRASH to obtain the addresses of the signal and mechanism arrays.

4. Locate the mechanism array. Saved R0 and saved R1 are the registers' values at the time the exception occurred.

5. The depth value specifies how many nested procedures there are on the current stack between the exception and the procedure that declared the condition handler.

6. Skip 1 longword, the flags longword.

7. The next longword, the beginning of the signal array, contains an argument count, the number of longwords that follow. Use the count to identify all entries in the signal array. The number of exception parameters present is a function of exception type and can be 0, 1, or 2 longwords.

8. The exception type is a status value, e.g., C (hex) or SS$_ACCVIO. The DCL command

```
$ EXIT %X<exception_type>
```

writes the message text associated with the exception type status value. The V4 SDA command

```
SDA> EVAL/CONDITION <exception_type>
```

writes the message text associated with the exception type status value.

Typically, the exception is one generated by "hardware" (or microcode), for example, access violation. "Hardware" generated exceptions are listed with a description of their associated exception parameters in Section 10.1 of the VAX/VMS System

Services Reference Manual. See section EXCEPTIONS for information about the more common hardware exceptions.

9. The exception PC in the signal array is the instruction whose [attempted] execution resulted in the unexpected exec or kernel mode exception. Whether the PC points to the beginning of the instruction or the end depends on whether the exception was a trap (end), fault (beginning), or abort (beginning). The reference above specifies whether each exception is a trap, fault, or abort.

10. Identify in what module the exception PC is. Figure out why the instruction generated an exception. For example, if an access violation occurred, look at the operands to see which access was in error.

    If the current image is a Files-11 ODS-1 ACP, then the bugcheck has been signaled by its exec mode condition handler MAIN_HANDLER and is always fatal. The exception PC is most likely within F11AACP.EXE, described by [F11A]F11AACP.MAP.

    If the current process is a magtape ACP, then the bugcheck has been signaled by the exec mode condition handler EXCEPT_HANDLER and is fatal only if the SYSBOOT parameter BUGCHECKFATAL is 1. By default, BUGCHECKFATAL is 0. The exception PC is most likely within MTAAACP.EXE, described by [MTAACP]MTAAACP.MAP.

    If the current process is not an ACP, then the bugcheck has been signaled by one of several Files-11 XQP kernel mode condition handler. It is always fatal. The initial Files-11 XQP procedure declares MAIN_HANDER as a call frame condition handler. Other Files-11 XQP procedures declare call frame condition handlers that issue this bugcheck: ACL-related code declares the handler BUILD_HANDLER; the procedure READ_ATTRIB declares the handler READ_HANDLER. The exception PC is most likely within F11BXQP.EXE, which is loaded into the process's P1 space. For the location of the Files-11 XQP and its stack, see subsection Files-11 XQP Regions in section VIRTUAL ADDRESSES - P1 SPACE.

    If the PC is not within one of those images, see section VIRTUAL ADDRESSES for information on identifying its source.

11. Decipher the current stack to trace control flow. See section STACK PATTERNS.

Hints And Kinks

1. Not all access violations are signaled by microcode. The pagefault exception service routine, MMG$PAGEFAULT, may signal an access violation if a process incurs a pagefault for a page in

another process's process header.

2. Note that for each V3 SDA COPY command used to copy the dump, the SP will be 8 bytes greater than its actual value; that is, SDA will show the SP pointing to a stack address 8 bytes higher than it should. This V3 bug has been corrected in V4.

3. The VAX instruction set is sufficiently rich that most random data can be interpreted as instructions. Most system code deals with binary integer and character data. This means that if an EXAMINE/INSTRUCTION display includes many packed decimal and/or floating point instructions, you are probably examining a data area or using a start address which is not an instruction boundary.

   One common error that results in a nonsensical display is to examine instructions in the bugcheck overlay area. During a crash, fatal bugcheck code and message text overlay resident system image code, beginning one page before label BUG$FATAL, for a length of about 12000 decimal or 3000 hex bytes.

Additional References

V3 VAX/VMS Internals and Data Structure Manual, Chapter 4, for general exception dispatching and details of exceptions signaled by VMS system software

VAX Architure Standard (DEC Standard 032) or VAX-11 Architecture Reference Manual, Chapter 6, Exceptions and Interrupts

VAX/VMS System Services Reference Manual, Chapter 10, Condition-Handling Services

VIRTUAL ADDRESSES

When you are looking at a crashdump and trying to determine the sequence of events that led to a problem, it is necessary to identify what code executed and what data structures were referenced. This means that you must associate virtual addresses with the code or data structures they contain. You must associate code or data with source modules; you must identify type and format of dynamically created data structures.

All executable code, both in system space and process space, consists of source modules compiled and linked into images. Images that execute in P0 space are loaded by the image activator. P1 space images are limited to command language interpreters and, under V4, the Files-11 XQP. The Files-11 XQP is mapped as a global section. Command language interpreters are mapped into P1 space by merged image activation. Images that execute in system space are loaded by system code. Many system space images, including SYS.EXE, are loaded during system initialization. Other images are loaded later. For example, SYSGEN loads driver images in response to CONNECT and/or LOAD commands; 782 support (MP.EXE) is loaded in response to the DCL command START/CPU.

By default, SDA knows about symbols defined in SYS.STB, the SYS.EXE symbol table, and some other self-defined symbols. (See section 6.2.4 of the VAX/VMS System Dump Analyzer Reference Manual.) SDA will attempt symbolic interpretation of virtual addresses based on the symbols it knows. That is, SDA will interpret an address as a positive offset less than hex 1000 from the closest symbol with a smaller value. This means that spurious labels may be attached to displayed data and addresses; for example, SDA will interpret the hex value 00002336 as SS$_NOSHRIMG + 17A. (SS$_NOSHRIMG, a status value unrelated to any P0 address, coincidentally has the hex value 2336 - 17A.)

1. If your hypothetical address is between 0 and 3FFFFFFF, follow the directions in section VIRTUAL ADDRESSES - P0 SPACE.

2. If your hypothetical address is between 40000000 and 7FFFFFFF, follow the directions in section VIRTUAL ADDRESSES - P1 SPACE.

3. If your hypothetical address is between 80000000 and BFFFFFFF, follow the directions in section VIRTUAL ADDRESSES - SYSTEM SPACE.

4. If your hypothetical address is C0000000 or above, it is not a valid address. Go back to the path of investigation that led you here and develop another hypothesis.

Hints And Kinks

1. Although "deciphering stacks" and "identifying virtual addresses" are listed as single and separate steps, in practice, they are usually repetitive and intertwined. For example, that a particular longword can be interpreted as a particular address should be confirmed in the context of what code was executing and manipulating that longword. Usually this requires that some piece of the stack be deciphered. Another example is that identifying a particular footprint on the stack may require or result in the identification of addresses within that footprint.

VIRTUAL ADDRESSES - P0 SPACE

P0 space is defined dynamically at image activation by the image a user runs and its referenced shareable images and their referenced shareable images, and so on. Also, a running image may create P0 address space for its own use through system services such as $CRETVA, $CRMPSC, or $MGBLSC.

In addition, if the process runs out of storage in the P1 process allocation region, the process allocation region may be increased to include a segment of P0 space if the image has not explicitly prohibited this through the Linker option NOP0BUFS.

As the V4 image activator processes an image and its references to other images, the image activator builds a list in P1 space of work items for itself. Each work item is an image described by an Image Control Block (ICB). As an image is activated, the ICB describing it is moved to a list of completed ICBs. You may be able to examine this list of ICBs to determine what images are mapped into P0 space. (The Debugger traverses this list to define its SHARE$ symbols.)

The directions below will not work for V3. The V3 analogue is to examine the shareable image list in the fixup vector section. See Additional References below for a pointer to more information.

1. First, confirm that your hypothetical P0 space address falls within the range of P0 space for this process. Find the high end of its defined P0 space by typing the following SDA commands.

   ```
   SDA> SET PROCESS/INDEX=<n>
   SDA> EVAL 200 * @P0LR
   ```

   If your address is larger than that, it is not a legal P0 space address for this process. Go back to the path of investigation that led you here and develop another hypothesis.

2. If the address is legal, examine the image activator ICB list to find out whether the address is within an activated image. Try the following SDA commands.

   ```
   SDA> EVAL IAC$GL_IMAGE_LIST !get address listhead
   SDA> DEF ICB = IAC$GL_IMAGE_LIST
   SDA> ! beginning of repeat loop
   SDA> DEF ICB = @ICB !get address next ICB
   SDA> EVAL ICB
   SDA> ! if ICB address is not equal to listhead, continue
   SDA> EXAM ICB + 48  !start address of image
   SDA> EXAM ICB + 4C  !end address of image
   SDA> EXAM ICB + 15 ;((@(ICB+14)@18)@-18) !image name
   SDA> ! go to beginning of repeat loop
   ```

   You may not be successful with these, because the listhead and ICBs are pageable.

3. The low end of the process's defined P0 space varies with the image it is running. An image is linked to a default base of 200 hex, unless another base address is specified through the Linker BASE option.

   To determine whether location x is valid for this process, type the following SDA command to display the page table entry that maps that location.

   ```
   SDA> SHOW PROCESS/PAGE <x>;200
   ```

Hints And Kinks

1. Any SDA commands to examine P0 or P1 space assume that you have already established the process to be examined by issuing one of the following commands.

   ```
   SDA> SET PROCESS/INDEX=<n>
   SDA> SET PROCESS <processname>
   ```

2. SDA uses special kernel ASTs to access the P0 or P1 space of another process on the current system. The special kernel AST, running in the context of the target process, examines its address space and sends the information back to the process running SDA. Delivery of the special kernel AST cannot happen if the target process is being waited at IPL 2. This means that you cannot examine that process's context until the process lowers its IPL.

Additional References

V3 VAX/VMS Internals and Data Structures Manual, Section 21.1.2, The Address Relocation Fixup System Service

VIRTUAL ADDRESSES - P1 SPACE

P1 space is primarily defined by the prototype page table and other assembly-time information in [SYS]SHELL. Some pieces of P1 space are created dynamically at process creation, others at image activation, and others by images that run in P1 space.

P1 space is divided into a permanent portion and a nonpermanent portion. The global CTL$GL_CTLBASVA contains the current boundary between the two portion; the address range below its contents is deleted at image exit. This nonpermanent portion of P1 space includes the user stack and the extra pages of image I/O segment. If the user creates a per-process message section through the SET MESSAGE command, the boundary moves to include the newly mapped message section.

1. Determine whether your hypothetical P1 space address falls within the range of P1 space for the relevant process. Find the low end of its defined P1 space by typing the following SDA commands.

   ```
   SDA> SET PROCESS/INDEX=<n>
   SDA> EVAL 40000000+(200*@P1LR)
   ```

   If your address is smaller than that, it is not a legal P1 space address for this process. Go back to the path of investigation that led you here and develop another hypothesis.

2. Using the table below, determine into which region of P1 space the address falls. If the address does not fall within an image identified in the table below, go to item 7.

3. Subtract the starting address of the loaded image from your virtual address to determine the offset of the address into the loaded image.

4. Reading the subsections below the table, identify to what VMS facility the loaded image belongs and where in the linked image the loaded code begins.

5. Locate the facility in the source fiche. The last sheet of the source fiche contains an index to the rest of the fiche. Maintenance update additions to the source fiche contain an updated index as the last sheet. The facilities are ordered alphabetically in the fiche. Each facility includes link maps and source listings for the components of the facility.

6. Reading the map for that module, determine the relevant source module and offset within the source module. Then return to the path of investigation that led you here.

7. If the virtual address does not fall within the boundaries of a loaded image, but is in the process allocation region, then it may be the address of a data structure. The following SDA commands should identify a data structure with a standard dynamic

data structure header.

```
SDA> READ SYS$SYSTEM:SYSDEF.STB  !if you haven't already
SDA> FORMAT <address>
```

SDA may report that there are no symbols to format a block of type <xxx>. Most likely this means that the block has a standard dynamic data structure header, but that neither SYS.STB nor SYSDEF.STB contains symbolic definitions for its fields. If this happens, you might try to generate the symbols yourself by typing the following sequence.

```
SDA>CTRL/Y
$ SPAWN
$ MACRO/OBJ=SYS$LOGIN:<xxx>DEF SYS$INPUT: -
$_ + SYS$LIBRARY:LIB/LIB
     $<xxx>DEF    GLOBAL
     .END
CTRL/Z
$ LO
$ CONT
SDA>READ SYS$LOGIN:<xxx>DEF.OBJ
SDA>FORMAT <address>
```

8. If the address is not within the process allocation region or a loaded image, determine into which other regions it falls. Read the subsection under the table that discusses that region for further information, and return to the path of investigation that led you here.

The following table describes the various "regions" of P1 space. In this context, region means a distinct area, with defined boundaries and characteristics. One example region of P1 space is the pages that the CLI image occupies. The table is ordered by increasing virtual address. Each region is described briefly in each table by its contents and protection and the SDA commands to determine its boundaries. More detailed descriptions of each region follow the table. Use the first table for V3 systems and the second for V4 systems.

V3 P1 Space Organization

```
         REGION         SDA COMMANDS
          PROTECTION

User Stack    UW    READ SYS$SYSTEM:SYSDEF.STB        !if you haven't already
                    EXAM @CTL$GL_PCB + PCB$L_PHD      !address of PHD
                    EVAL @(@.+PHD$L_FREP1VA) + 200    !low address of stack
                    EVAL @(CTL$AL_STACK+C)-@SGN$GL_EXUSRSTK !high address

Extra User Stack    EVAL @(CTL$AL_STACK+C)-@SGN$GL_EXUSRSTK !low address
              UW    EXAM CTL$AL_STACK + C             !high address

Image I/O Segment   READ SYS$SYSTEM:RMSDEF.STB        !if you haven't already
            UREW    EXAM CTL$AL_STACK + C             !start address
                    EXAM PIO$GW_IIOIMPA + IMP$L_IOSEGADDR
                    EVAL @. + @(PIO$GW_IIOIMPA + IMP$L_IOSEGLEN) !end address

Per-Process   UR    EXAM CTL$GL_PPMSG                 !start address - 0=none
 Message Section    EXAM . + 4                        !end address

CLI Symbol Table    EXAM CTL$AG_CLIDATA+10            !start address
              SW    EVAL @.+ @(.-4)                   !end address

CLI Image URSW/UR   EXAM CTL$AG_CLIMAGE               !start address
                    EXAM . + 4                        !end address

Channel Table       EVAL @CTL$GL_CCBBASE-(10*((@SGN$GW_PCHANCNT@10)@-10))
            UREW                                      !low address
                    EXAM CTL$GL_CCBBASE               !high address

P1 Window to PHD    EXAM CTL$GL_PHD                   !start address
            URKW    EVAL PIO$GL_FMLH                  !end address

RMS Process URKW    EVAL PIO$GL_FMLH                  !start address
 Context Area       EVAL PIO$GL_FMLH + 200            !end address

Process I/O         EVAL PIO$GL_FMLH + 200            !start address
 Segment    UREW    EVAL PIO$GL_FMLH + PIO$C_SEGSIZ   !end address

Per-Process   UW    EVAL CTL$A_COMMON-@CTL$GQ_COMMON!start address
 Common for Users   EVAL CTL$A_COMMON                 !end address

Per-Process   UW    EVAL CTL$A_COMMON                 !start address
 Common Digital     EVAL CTL$A_COMMON+@CTL$GQ_COMMON!end address

Compatibility UW    EVAL CTL$AL_CMCNTX                !start address
 Mode Data Page     EVAL CTL$AL_CMCNTX+200            !end address

User Mode Data      EVAL CTL$GL_DCLPRSOWN             !start address
 Page         UW    EVAL CTL$GL_DCLPRSOWN+200         !end address
```

```
Unused Pages       2 pages

Image Activator    EVAL CTL$GL_IAFLINK              !start address
 Context UREW      EVAL CTL$GL_IAFLINK+200          !end address

Process Allocation EVAL CTL$A_PRCALLREG              !start address
 Region      UREW  EVAL CTL$A_PRCALLREG+(CTL$C_PRCALLSIZ*200) !end address

CLI Data Pages     EVAL CTL$AL_CLICALBK             !start address
             URSW  EVAL CTL$AG_CLIDATA+CTL$C_CLIDATASZ !end address

Image Activator    EVAL MMG$IMGACTBUF               !start address
 Scratch UREW      EVAL MMG$IMGACTBUF + 1000        !end address

Debugger Context   EVAL MMG$IMGACTBUF + 1000        !start address
 Pages        UR   EVAL CTL$A_DISPVEC               !end address

Dispatch Vectors   EVAL CTL$A_DISPVEC               !start address
             UREW  EVAL MMG$IMGHDRBUF               !end address

Image Header       EVAL MMG$IMGHDRBUF               !start address
 Buffer       UW   EVAL MMG$IMGHDRBUF+200           !end address

Guard Page         1 page

Kernel Stack SRKW EXAM CTL$AL_STACKLIM              !low address
                  EXAM CTL$AL_STACK                 !high address

Exec Stack   SREW EXAM CTL$AL_STACKLIM+4            !low address
                  EXAM CTL$AL_STACK+4               !high address

Sup. Stack   URSW EXAM CTL$AL_STACKLIM+8            !low address
                  EXAM CTL$AL_STACK+8               !high address

System Service     EVAL P1SYSVECTORS                !start address
Vector Pages URKW EVAL P1SYSVECTORS+(SGN$C_SYSVECPGS*200) !end address

Spare Pages for System   12 (decimal) pages
 Service Vectors

P1 Pointer Page    EVAL CTL$GL_VECTORS              !start address
             URKW  EVAL CTL$GL_VECTORS+200          !end address

Debugger Symbol    EXAM CTL$GQ_DBGAREA+4            !start address
 Table        UW   EVAL @. + @(.-4)                 !end address
```

V4 P1 Space Organization

```
        REGION       SDA COMMANDS
         PROTECTION
User Stack    UW  READ SYS$SYSTEM:SYSDEF.STB        !if you haven't already
                  EXAM @CTL$GL_PCB + PCB$L_PHD      !address of PHD
                  EVAL @(@.+PHD$L_FREP1VA) + 200    !low address of stack
                  EVAL @(CTL$AL_STACK+C)-@SGN$GL_EXUSRSTK !high address

Extra User Stack  EVAL @(CTL$AL_STACK+C)-@SGN$GL_EXUSRSTK !low address
              UW  EXAM CTL$AL_STACK+C                !high address

Extra Image I/O
 Segment    UREW  (see subsection below)

Per-Process   UR  EXAM CTL$GL_PPMSG                  !start address - 0=none
 Message Section  EXAM . + 4                         !end address

CLI Symbol Table  EXAM CTL$AG_CLIDATA+10             !start address
           UR/SW  EVAL @.+ @(.-4)                    !end address

CLI Command       EXAM CTL$AG_CLITABLE               !start address
 Tables       UR  EXAM . + 4                         !end address

CLI Image     UR  EXAM CTL$AG_CLIMAGE                !start address
                  EXAM . + 4                         !end address

Files-11 XQP      CTRL/Y                             !leave SDA to get symbols
 Data         KW  SPAWN                              !create subprocess
                  MACRO/OBJ=SYS$LOGIN:F11BDEF SYS$INPUT:+SYS$LIBRARY:LIB/LIB
                          $F11BDEF    GLOBAL
                          .END
                  CTRL/Z
                  LOGOUT                             !attach main process
                  CONTINUE                           !return to SDA
                  READ SYS$LOGIN:F11BDEF.OBJ         !read few XQP symbols
                  DEF XQP=@CTL$GL_F11BXQP
                  EXAM XQP+F11B$L_IMPBASE            !data start address
                  EVAL @.+@(XQP+F11B$L_IMPSIZE)      !data end address

Files-11 XQP  ER  EXAM XQP+F11B$L_CODEBASE           !code start address
 Image            EVAL @.+@(XQP+F11B$L_CODESIZE)     !code end address

Image I/O Segment EXAM PIO$GQ_IIODEFAULT+4           !start address
            UREW  EVAL @. + @(.-4)                   !end address

Process I/O       READ SYS$SYSTEM:RMSDEF.STB         !read symbols
 Segment    UREW  EXAM PIO$GW_PIOIMPA + IMP$L_IOSEGADDR
                  DEF PIOEND=@.+@(PIO$GW_PIOIMPA+IMP$L_IOSEGLEN)
                  EVAL PIOEND-(200*((@SGN$GW_PIOPAGES@10)@-10))
                                                     !start address
```

```
                    EVAL PIOEND                        !end address

Process      UREW   EVAL PIOEND                        !start address
 Allocation Region EVAL PIOEND + (200*((@SGN$GW_CTLPAGES@10)@-10))
                                                       !end address

Channel Table       EVAL @CTL$GL_CCBBASE-(10*((@SGN$GW_PCHANCNT@10)@-10))
             UREW                                      !low address
                    EVAL @CTL$GL_CCBBASE+10            !high address

P1 Window to PHD    EXAM CTL$GL_PHD                    !start address
             URKW   EVAL PIO$GL_FMLH                   !end address

RMS Process UREW    EVAL PIO$GL_FMLH                   !start address
 Context Area       EVAL PIO$A_TRACE                   !end address

RMS Tracepoint      EVAL PIO$A_TRACE                   !start address
 Page        UREW   EVAL PIO$A_DIRCACHE                !end address

RMS Directory       EVAL PIO$A_DIRCACHE                !start address
 Cache       UREW   EVAL PIO$A_DIRCACHE+400            !end address

RMS IFAB/IRAB       READ SYS$SYSTEM:RMSDEF.STB         !if you haven't already
 Table       UREW   EXAM PIO$GW_PIOIMPA+IMP$L_IFABTBL!start address
                    EVAL @. + 200                      !end address

Per-Process    UW   EVAL CTL$A_COMMON-@CTL$GQ_COMMON!start address
 Common for Users EVAL CTL$A_COMMON                    !end address

Per-Process    UW   EVAL CTL$A_COMMON                  !start address
 Common Digital     EVAL CTL$A_COMMON+@CTL$GQ_COMMON!end address

Compatibility Mode EVAL CTL$AL_CMCNTX                  !start address
 Data Pages    UW   EVAL CTL$AL_CMCNTX+400             !end address

User Mode Data      EVAL CTL$GL_DCLPRSOWN              !start address
 Page          UW   EVAL CTL$GL_DCLPRSOWN+200          !end address

Unused Pages   NA   2 pages

Security Audit      EVAL NSA$T_IDT                     !start address
 Data Pages    KW   EVAL NSA$T_IDT+600                 !end address

Image Activator     EVAL CTL$GL_IAFLINK                !start address
 Context     UREW   EVAL CTL$GL_IAFLINK+200            !end address

CLI Data Pages      EVAL CTL$AL_CLICALBK               !start address
             URSW   EVAL CTL$AG_CLIDATA+CTL$C_CLIDATASZ !end address

Image Act Scratch EVAL IAC$AL_IMGACTBUF                !start address
 Pages       UREW   EVAL IAC$AL_IMGACTBUF+1000         !end address

Debugger Context    EVAL IAC$AL_IMGACTBUF+1000         !start address
 Pages         UW   EVAL CTL$A_DISPVEC                 !end address
```

```
Dispatch Vectors  EVAL CTL$A_DISPVEC               !start address
            UREW  EVAL MMG$IMGHDRBUF               !end address

Image Header      EVAL MMG$IMGHDRBUF               !start address
 Buffer     URSW  EVAL MMG$IMGHDRBUF+200           !end address

KRP Lookaside     EVAL CTL$GL_KRP                  !start address
 List       URKW  EVAL CTL$GL_KRP+(CTL$C_KRP_COUNT*CTL$C_KRP_SIZE)
                                                   !end address

Guard Page    NA  1 page

Kernel Stack  NA  EVAL CTL$GL_KSTKBASEXP           !low address
 Expansion Pages  EVAL CTL$GL_KSTKBAS              !high address

Kernel Stack SRKW EVAL CTL$GL_KSTKBAS              !low address
                  EVAL CTL$GL_KSPINI               !high address

Exec Stack   SREW EXAM CTL$AL_STACKLIM+4           !low address
                  EXAM CTL$AL_STACK+4              !high address

Sup. Stack   URSW EXAM CTL$AL_STACKLIM+8           !low address
                  EXAM CTL$AL_STACK+8              !high address

System Service UR EVAL P1SYSVECTORS                !start address
 Vector Pages     EVAL P1SYSVECTORS+(SGN$C_SYSVECPGS*200) !end address

Spare Pages for System   11 (decimal) pages
 Service Vectors NA

P1 Pointer Page   EVAL CTL$GL_VECTORS              !start address
            URKW  EVAL CTL$GL_VECTORS+200          !end address

Debugger Symbol   EXAM CTL$GQ_DBGAREA+4            !start address
   Table      UW  EVAL @. + @(.-4)                 !end address
```

User Stack

The pageable user stack is normally allocated at the lowest defined end of P1 space to enable its automatic expansion on demand. To examine the current contents of this stack, type the following SDA command.

```
SDA> SHOW STACK @USP:(@(CTL$AL_STACK+C)-@SGN$GL_EXUSRSTK)
```

Extra User Stack Pages

Extra pages are allocated at the high end of the user stack for use by the system during exception processing if the user stack is corrupted. The size of this region is defined by SYSBOOT parameter EXUSRSTK, with a default value of 2 pages. Usually, these pages show up as extra zeros at the end of what SDA displays in response to the command SHOW STACK/USER.

Image I/O Segment

The pageable image I/O segment contains RMS data structures for files which can be open only during the life of an image. Under V4, EXE$PROCSTRT allocates a default image I/O segment that is SYSBOOT parameter PIOPAGES pages long. An image needing more space than this should be linked with the IOSEGMENT option. When an image which specified an IOSEGMENT bigger than the default is activated, the image activator allocates virtual address space equal in size to the difference between the default segment size and the size specified at link time. This additional portion of image I/O segment lies in virtual addresses just higher than the end of the user stack. RMS data structures allocated from the Image I/O Segment include Internal File Access Blocks (IFABs), Internal Record Access Blocks (IRABs), Buffer Descriptor Blocks (BDBs), I/O buffers, etc.

To display the data structures allocated from this region, type the SDA command SHOW PROCESS/RMS.

Pieces of the image I/O segment that have been used and then deallocated are linked together at the listhead PIO$GW_IIOIMPA + IMP$L_FREEPGLH. (The IMP$ symbols are defined by SYS$LIBRARY:LIB.MLB macro $IMPDEF and also in SYS$SYSTEM:RMSDEF.STB.) Any additional I/O segment allocated by the V4 image activator is inserted on this list as well. The portion of the image I/O segment that has never been allocated is defined in the image I/O segment context area. Its starting address is in PIO$GW_IIOIMPA + IMP$L_IOSEGADDR; its size is in PIO$GW_IIOIMPA + IMP$L_IOSEGLEN

Per-Process Message Section

A message section is mapped into a process's P1 space as the result of the DCL command $SET MESSAGE <filename>. The $GETMSG system service uses this message section in addition to the system message section to translate status codes into message text.

CLI Symbol Table

LOGINOUT.EXE creates this pageable virtual address space with a $EXPREG of SYSBOOT parameter CLISYMTBL pages. The CLI uses this to store the definitions of process global and local symbols, labels, and scratch storage. DCL manages this area using EXE$ALLOCATE and EXE$DEALLOCATE.

CLI Command Table

LOGINOUT.EXE maps the command table for the process's CLI. A user can override the default tables with the SPAWN and login qualifier /TABLES. The usual tables for DCL and MCR are SYS$SHARE:DCLTABLES.EXE and SYS$SHARE:MCRTABLES.EXE. These tables contain all the command definitions for the CLI. They can be replaced or altered with the SET COMMAND utility. Under V4, the filename of the mapped command tables is in CTL$GT_TABLENAME.

CLI Image

LOGINOUT.EXE, using the authorization file record and/or command used to create the process, determines which command language interpreter the process will use. It maps the CLI into pageable P1 space.

Under V4, to find out which CLI a process has mapped, type the following SDA command.

```
    SDA> EXAMINE CTL$GT_CLINAME;8
```

The CLI images are loaded starting at offset 0. That is, the contents of CTL$AG_CLIMAGE correspond to offset 0 in the linked image. The DCL CLI is in facility [DCL]. The V3 MCR CLI is in facility [MCR]. The V4 MCR CLI is a layered product, shipped separately from VMS.

Files-11 XQP Regions

Under V4 the Files-11 XQP runs in process context and is mapped into P1 space by EXE$PROCSTRT. The location in F11BXQP.EXE that

concurrently or if you have too many concurrent F$SEARCH contexts, you may run out of space in this region and receive the error RMS-F-DME, dynamic memory exhausted. In extreme cases, there may not be an error message, merely that status value set in the DCL symbol $STATUS.

Process Allocation Region

This region is a pageable pool for process-specific data structures. Mounted volume list entries for privately mounted volumes and the process logical name table are allocated from it. Under V4, this address space is sized by SYSBOOT parameter CTLPAGES and created by EXE$PROCSTRT and is also used for the allocation of process logical name directories and image control blocks, among other things. Under V4, the process allocation region can be increased to include a segment of P0 space, if it becomes full and if the image has not explicitly prohibited this through the Linker option NOP0BUFS.

Blocks not in use are linked in a singly linked list whose head is at CTL$GQ_ALLOCREG. Each block contains the pointer to the next free block at offset 0 and its own size at offset 4.

There is no useful way to display this region other than through repeated EXAMINE commands. The DCL command SHOW PROCESS/MEMORY displays information about this region, its size, number of bytes free, etc.

Insufficient process allocation region is one possible cause of the error SYSTEM-F-INSFMEM.

Channel Control Block Table

The pageable Channel Control Block (CCB) Table has room for SYSBOOT parameter CHANNELCNT CCBs. Whenever a process issues the $ASSIGN system service, a CCB is allocated to describe the state of the process's connection to the assigned device unit. The "channel number" returned from the system service is the positive byte index which is subtracted from the base of the table to form the address of the assigned CCB. Channel 0 is left unassigned for error detection. CTL$GW_CHINDX contains the positive byte index of the highest channel assigned at any time in the life of this process.

Unassigned CCBs have CCB$B_AMOD equal to 0. An assigned CCB has CCB$B_AMOD equal to 1 plus the access mode from which the channel was assigned.

The CCB table is the key to determining what I/O is or has been going on in the process. An assigned CCB contains the Unit Control Block (UCB) address of the assigned device unit and an indirect pointer to any file opened on that channel. CCB$W_IOC is the number of I/O

corresponds to the contents of CTL$GL_F11XQP is global symbol XQP_QUEUE. Files-11 XQP sources are in facility [F11X].

EXE$PROCSTRT calls the Files-11 XQP's initialization routine. This routine creates an impure area and private kernel stack on which the XQP keeps the context of an I/O request in progress, and locks the private kernel stack and some of the Files-11 image pages into the process working set, making it nonpageable. To determine the boundaries of the XQP stack, type the following SDA commands.

```
SDA> EXAM @CTL$GL_F11BXQP+2C  !low end
SDA> EXAM @CTL$GL_F11BXQP+28  !high end
```

The SYS$LIBRARY:LIB.MLB macro $F11BDEF defines symbols for several variables at the beginning of the data area. A layout of the entire impure area is in [F11X]FCPDEF, in the macro GLOBAL_STORAGE. The contents of CTL$GL_F11BXQP correspond to the location XQP_QUEUE in that layout.

Process I/O Segment

The pageable process I/O segment contains RMS data structures describing "process permanent" files, those which can, and usually do, remain open across image activations. SYS$INPUT, SYS$OUTPUT, and files opened through the DCL command OPEN are examples of process permanent files. These data structures include Internal File Access Blocks (IFABs), Internal Record Access Blocks (IRABs), Buffer Descriptor Blocks (BDBs), I/O buffers, etc.

To display the data structures allocated from this region, type the following SDA commands.

```
SDA> !   save value of image I/O segment context
SDA> DEF SAVE=PIO$GW_IIOIMPA
SDA> !   set symbol to process I/O segment context
SDA> DEF PIO$GW_IIOIMPA=PIO$GW_PIOIMPA
SDA> !   display process I/O data structures
SDA> SHOW PROCESS/RMS
SDA> !   restore value of image I/O segment context
SDA> DEF PIO$GW_IIOIMPA = SAVE
```

Pieces of the process I/O segment that have been used and then deallocated are linked together at the listhead PIO$GW_PIOIMPA + IMP$L_FREEPGLH. The portion of the process I/O segment that has never been allocated is defined in the process I/O segment context area. Its starting address is in PIO$GW_PIOIMPA + IMP$L_IOSEGADDR; its size is in PIO$GW_PIOIMPA + IMP$L_IOSEGLEN (The IMP$ symbols are defined by SYS$LIBRARY:LIB.MLB macro $IMPDEF and also in SYS$SYSTEM:RMSDEF.STB.)

This size of this region is determined by the SYSBOOT parameter PIOPAGES. If you have too many process permanent files open

requests outstanding on this channel. The outstanding I/O requests are described by I/O Request Packets (IRPs) that contain the process ID of this process in IRP$L_PID, the contents of CCB$L_UCB in IRP$L_UCB, and the negative "channel number" in IRP$W_CHAN.

If a file is open on the channel, CCB$L_WIND contains its Window Control Block (WCB) address. If a section has been mapped on the channel, CCB$L_WIND contains the section table index.

Under V4, a negative value in CCB$B_AMOD identifies the single channel control block reserved for the use of Files-11 XQP. (Files-11 XQP alters CCB$L_UCB dynamically in order to issue an I/O request to a particular disk.) Files-11 XQP zeroes CCB$B_AMOD to issue an I/O request through that CCB; when the request completes, Files-11 XQP stores a negative value in that field to prevent the channel from being deassigned by image rundown.

To display the region under V3, select one of the following sets of SDA commands.

```
SDA> !use this method to scan through CCBs quickly
SDA> !       to locate a particular CCB
SDA> READ SYS$SYSTEM:SYSDEF.STB !if you haven't already
SDA> EXAM @CTL$GL_CCBBASE
SDA> !repeat next command til CCB is all zeroes
SDA> FORMAT .-10/TYP=CCB
SDA> !or use this method to scan through CCBs determining
SDA> !           assigned device for each
SDA> READ SYS$SYSTEM:SYSDEF.STB !if you haven't already
SDA> DEF X = @CTL$GL_CCBBASE    !base address of CCB table
SDA> !next CCB
SDA> DEF X = X-10                       !address of next CCB
SDA>                 !if X < @CTL$GL_CCBBASE- (@CTL$GW_CHINDX*10)
SDA>                 !  then you're done
SDA> FORMAT X/TYP=CCB                   !display CCB
SDA>                 !if CCB$B_AMOD=0, go to next CCB
SDA> DEF UCB=@(X+CCB$L_UCB)     !address of UCB
SDA> EXAM @(UCB+UCB$L_DDB)+DDB$T_NAME;8 !device name
SDA> EXAM UCB + UCB$W_UNIT      !unit number in low word
SDA>                !go to next CCB
```

To display this region under V4, type the following SDA command.

```
SDA>SHOW PROCESS/CHANNEL
```

P1 Window To Process Header

All of the process's header, except for the page table pages, is double-mapped in P1 space so that kernel mode code can access it using P1 addresses which are invariant across process outswaps and inswaps (which can result in allocation of a different system space balance set slot for the process's header). This region is not

pageable.

To display these pages, type the following SDA commands.

```
SDA> FORMAT @CTL$GL_PHD/TYP=PHD!display fixed part of PHD
SDA> SHOW PROCESS/REGISTER       !display hardware PCB
SDA> SHOW PROCESS/WORK   !display working set list
SDA> SHOW PROCESS/PROC   !display section table
```

RMS Process Context Area

RMS uses this page to locate and control per-process RMS-32 resources and data structures. It is divided into several areas: an overall area that includes globals that define process RMS defaults and listheads for directory caches and free memory; a process I/O segment context area; and an image I/O segment context area. This page is pageable.

The best way to look at the overall area is by repeated SDA EXAMINE commands, reading [SYS]SHELL, since the globals are not all longwords. The globals are defined beginning in the source module at subtitle PROCESS I/O SEGMENT and continue up to PIO$GW_PIOIMPA.

The process and image I/O segment context areas contain pointers to some of the data structures in their respective segments that describe RMS operations.

To display the process and image I/O segment context areas, type the following SDA commands.

```
SDA> READ SYS$SYSTEM:RMSDEF.STB
SDA> FORMAT PIO$GW_PIOIMPA/TYPE=IMP !process I/O context
SDA> FORMAT PIO$GW_IIOIMPA/TYPE=IMP !image I/O context
```

RMS Tracepoint Page

This pageable page contains RMS per-process statistics, counters for the various RMS operations. The symbolic offsets in this page are globals of the form TPT$L_<name> defined in SYS$SYSTEM:RMS.STB. Look in the source fiche at module [RMS]RM0DUMMY for the values of these symbols and the comments that explain them.

Type the following SDA commands to display this page.

```
SDA> READ SYS$SYSTEM:RMS.STB
SDA> FORMAT PIO$A_TRACE/TYP=TPT
```

RMS Directory Cache

These pageable pages are used to cache the mappings between a directory name and the file id of the file that contains that directory. This caching minimizes Files-11 directory file lookups for heavily used directory paths.

The format of the cache elements is defined by the macro $DRCDEF, whose symbols are also in SYS$SYSTEM:RMSDEF.STB. The cache elements are linked together in a tree structured list whose head is at PIO$GL_DIRCACHE and the next longword. PIO$GL_DIRCACHE heads a queue of cache elements, each describing a volume, linked together through DRC$L_NXTFLNK and DRC$L_NXTBLNK. Each of these volume elements contains a list head (DRC$L_LVLFLNK and DRC$L_LVLBLNK) for top level directories contained on that volume. The cache elements for top level directories on a particular volume are themselves linked together through DRC$L_NXTFLNK and DRC$L_NXTBLNK, with the element at the end of the list pointing back to its upper level cache element. A cache element for a top level directory contains a list head for the next lower level subdirectory (DRC$L_LVLFLNK and DRC$L_LVLBLNK), and so on.

Unused cache elements are singly linked at PIO$GL_DIRCFRLH.

RMS IFAB/IRAB Tables

This page contains the process I/O Internal File Access Block (IFAB) and Internal Record Access Block (IRAB) tables. The IFAB/IRAB Table page is pageable. Whenever the process opens a process permanent file, an IFAB is allocated and an Internal File Identifier (IFI) is associated with that file. The address of the IFAB is stored in the IFAB table at the longword entry indexed by the IFI. Whenever a record stream is connected to such a file, an IRAB is allocated and an Internal Stream Identifier (ISI) is associated with the file. The address of the IRAB is stored in the IRAB table at the longword entry indexed by the ISI.

Under V4, these tables each have room for 64 (decimal) entries. Under V3, these tables each have room for 12 (decimal) entries. An entry containing a zero indicates that its corresponding identifier is not in use. This page is displayed as part of the process I/O segment with the SDA commands in the subsection above on Process I/O Segment. RMS finds these tables by following the pointers from the process I/O segment context area, PIO$GW_PIOIMPA + IMP$L_IFABTBL and IMP$L_IRABTBL.

Per-Process Common Regions

The four pages beginning at CTL$A_COMMON are for use by VAX-11 BASIC to provide "core common" required to pass data when one image chains

to another. An identical area, negatively displaced, is allocated for use by customers and CSS. These are pageable.

Compatibility Mode Data Pages

One use of these pageable pages is communication between the initial service routine for compatibility mode exceptions, EXE$COMPAT, and any compatibility mode handler declared through $DCLCMH. In most cases, the declared compatibility mode handler is part of the RSX-11 AME. EXE$COMPAT saves R0-R5, the compatibility mode exception parameter, and the faulting PC and PSL in this page.

User Mode Data Page

This page is writable from user mode and pageable. It is used by user-mode VMS components. It contains two globals used as pointers for CLI context and work area.

Security Audit Data Pages

These pages are used by routines in V4 module [SYS]SECAUDIT to construct security audit journal records and/or OPCOM messages. The SYS$LIBRARY:LIB.MLB macro $NSAIDTDEF defines the layout of this region.

Image Activator Context Page

This page contains image activator context that remains after an image is activated, for use by the $IMGFIX system service and image rundown. It is pageable.

CLI Data Page

This page contains CLI data common to both MCR and DCL. It contains, for example, the name of the CLI image used by the process and the address ranges into which the CLI and its command tables have been mapped.

Beginning at CTL$AG_CLIDATA is an area used for communication between LOGINOUT and the CLI it maps. It contains, for example, information about the size and location of the CLI symbol table; the file ids, IFIs, ISIs, and channel numbers of SYS$INPUT and SYS$OUTPUT. It also contains a pointer to the data structure DCL uses to describe its current state. This data structure includes the listheads for local

and global symbols, labels, and available space in the CLI symbol table pages.

To examine these areas, type the following SDA commands.

```
SDA> READ SYS$SYSTEM:DCLDEF.STB
SDA> !display CLI-LOGINOUT communication area
SDA> FORMAT CTL$AG_CLIDATA/TYPE=PPD
SDA> !get address of DCL per-process area
SDA> EXAM CTL$AG_CLIDATA+PPD$L_PRC
SDA> FORMAT @./TYPE=PRC    !display DCL per-process area
```

To see the definitions of these data structures, look at the source fiche

```
PPD$ symbols    [LOGIN]PPDDEF.MDL
PRC$ symbols    [DCL]DCLDEF.MDL
```

Image Activator Scratch Pages

This pageable region is used as local storage by the $IMGACT system service. It is used to store the image header of the image being activated and various RMS data structures for files opened by the Image Activator.

Debugger Context Pages

[TBS]

Dispatch Vectors For User-Written System Services And Messages

These pages contain the linkages for dispatching into privileged shareable libraries (user-written system services and rundown routines) and the linkages for per-process message sections. The pages are initialized by EXE$PROCSTRT. Actual linkages are established by the image activator and cleared at image rundown. The change mode dispatchers (EXE$CMODKRNL and EXE$CMODEXEC) use these pages to dispatch to user-written system services, and EXE$RUNDWN to dispatch to user-written inner access mode rundown routines.

Image Header Buffer

If an image is currently active in the process, CTL$GL_IMGHDRBUF contains the address of this pageable region. When CTL$GL_IMGHDRBUF is nonzero, this region's contents are valid. Image rundown clears CTL$GL_IMGHDRBUF. The page contains part of the image header from

the image and the image file descriptor block (IFD).

To display this region, type the following SDA commands.

```
SDA> READ SYS$SYSTEM:IMGDEF.STB
SDA> FORMAT @MMG$IMGHDRBUF/TYPE=IHD !display image header
SDA> DEFINE IFD = @(MMG$IMGHDRBUF+4)!address of IFD
SDA> FORMAT IFD/TYPE=IFD        !display image file desc.
SDA> !display filename of image being run
SDA> EXAMINE @(IFD+IFD$Q_CURPROG+4);@(IFD+IFD$Q_CURPROG)
```

KRP Lookaside List

Under V4, there is a lookaside list of buffers used by kernel-mode code. The most common use of these buffers is to hold equivalence names returned from logical name translations. Formerly, space was allocated on the kernel stack for this purpose. EXE$PROCSTRT initializes the list. Available blocks are queued to the listhead at CTL$GL_KRPFL and CTL$GL_KRPBL. Allocation and deallocation from the list is done via REMQUE and INSQUE.

Inner Access Mode Stacks

The supervisor, exec, and kernel mode stack sizes and locations are fixed for a given release of VMS. Their highest addresses are recorded in CTL$AL_STACK, a longword array indexed by access mode. Their lowest addresses are recorded in CTL$AL_STACKLIM, a similar array. The hardware PCB of a process whose context has been saved contains the values of the current pointers into each of these stacks (PHDL_USP, PHDL_SSP, PHD$L_ESP, and PHD$L_KSP) The kernel stack is nonpageable; the others are pageable.

Under V4, the kernel stack can be expanded into several pages of address space reserved for this purpose. When the kernel stack is expanded, the low limit stored in CTL$AL_STACKLIM is also altered.

Under V4, the Files-11 XQP runs on its own private kernel stack. See subsection FILES-11 XQP Regions.

To examine the current contents of these stacks in a dump, type the following SDA commands.

```
SDA> SHOW STACK/KERNEL
SDA> SHOW STACK/EXEC
SDA> SHOW STACK/SUPER
```

See the section STACK PATTERNS for hints on interpreting the contents of the stacks.

System Service Vector Pages

Several pages of P1 space double-map the system service vectors. See the section SYSTEM SERVICE VECTORS for a description of this region.

P1 Pointer Page

This page is defined in [SYS]SHELL and contains much of the process-specific data used by the exec. The page is a permanent member of the process working set and thus nonpageable. See section A.3.1 in the V3 VAX/VMS Internals and Data Structure Manual for a list of its global variables and their contents.

The best way to look at the P1 pointer page is by repeated SDA EXAMINE commands, reading [SYS]SHELL, since the globals are not all longwords. The globals are defined beginning in the source module at subtitle BODY OF SHELL PROCESS.

Debugger Symbol Table

[TBS]

Hints And Kinks

1. All of the above SDA commands to examine P1 space assume that you have already established the process to be examined by issuing one of the following commands.

   ```
   SDA> SET PROCESS/INDEX=<n>
   SDA> SET PROCESS <processname>
   ```

2. SDA uses special kernel ASTs to access the P1 space of another process on the current system. The special kernel AST, running in the context of the target process, examines its address space and sends the information back to the process running SDA. Delivery of the special kernel AST cannot happen if the target process is being waited at IPL 2. This means that you cannot examine that process's context until the process lowers its IPL.

Additional References

V3 VAX/VMS Internals and Data Structures Manual, Section 1.5.2, The Control Region (P1 Space); Section 26.4, Sizes of P1 Space; Section A.3, Process Specific Executive Data

VIRTUAL ADDRESSES - SYSTEM SPACE

Since much of system space is variable length, sized by SYSBOOT, and since many images other than SYS.EXE reside in system space, decoding a system space address is not trivial.

1. Determine whether your hypothetical system space address falls within the range of system space defined in the system under examination. Find the end of defined system space by typing the following SDA command.

   ```
   SDA>EXAM MMG$GL_MAXGPTE
   ```

   If your hypothetical address is larger than that, it is not a legal address for this system. Go back to the path of investigation that led you here and develop another hypothesis.

2. Using the tables below, determine into which [sub-]region of system space the address falls. The first table is for V3 systems; the second is for V4 systems. The most likely possibilities are SYS.EXE, RMS.EXE, nonpaged pool, and the lookaside lists. If the address does not fall within an image identified in the table below, go to item 7.

3. Subtract the starting address of the loaded image from your virtual address to determine the offset of the address into the loaded image.

4. Reading the subsections below the table, identify to what VMS facility the loaded image belongs and where in the linked image the loaded code begins.

5. Locate the facility in the source fiche. The last sheet of the source fiche contains an index to the rest of the fiche. Maintenance update additions to the source fiche contain an updated index as the last sheet. The facilities are ordered alphabetically in the fiche. The fiche for each facility includes link maps and source listings of its components.

6. Reading the map for that module, determine the relevant source module and offset within the source module. Then return to the path of investigation that led you here.

7. If the virtual address does not fall within the boundaries of a loaded image, but is in paged pool, nonpaged pool, or one of the lookaside lists, then most likely it is the address of a data structure. If the virtual address does not fall within pool, then go to item 9. The following SDA commands should identify a data structure with a standard dynamic data structure header.

   ```
   SDA> READ SYS$SYSTEM:SYSDEF.STB  !if you haven't already
   SDA> FORMAT <address>
   ```

   SDA may issue the error

SDA-E-NOSYMBOLS, no "<xxx>" symbols found to format this block

Most likely this means that the block has a standard dynamic data structure header, but that neither SYS.STB nor SYSDEF.STB contains symbolic definitions for its fields. If this happens, you might try to generate the symbols yourself by typing the following sequence.

```
SDA>CTRL/Y
$ SPAWN
$ MACRO/OBJ=SYS$LOGIN:<xxx>DEF SYS$INPUT: -
$_ + SYS$LIBRARY:LIB/LIB
     $<xxx>DEF    GLOBAL
     .END
CTRL/Z
$ LO
$ CONT
SDA>READ SYS$LOGIN:<xxx>DEF.OBJ
SDA>FORMAT <address>
```

8. If the pool virtual address is still not identified, try examining memory on either side of it. If it appears to be instructions in nonpaged pool, rather than data, possibily it is part of an extended AST Control Block allocated for use by $GETJPI system service, on-line SDA, or DELTA. Return to the path of investigation that led you here.

9. If the address is not within pool or a loaded image, determine in which other regions or sub-regions it falls. Read the subsection under the table that discusses that [sub-]region for further information, and return to the path of investigation that led you here.

The following table describes the various "regions" of system space. In this context, region means a distinct area, with defined boundaries and characteristics. One example region of system space is the system image itself. It has "sub-regions", that is, pieces with defined boundaries and possibly different characteristics, permanently resident and pageable, for example. The table is ordered by increasing virtual address of the regions; to the extent possible, the sub-regions are also ordered by increasing virtual address. Each [sub-]region is described briefly in the table by its contents and protection and the SDA commands to determine its boundaries. Use the first table for V3 systems and the second for V4 systems.

V3 System Space Organization

```
        REGION        SDA COMMANDS
         PROTECTION

SYS.EXE               EVAL G                              !start address
                      EVAL MMG$A_SYS_END                  !end of system image

  system service      EVAL G                              !start address
  vectors URKW        EVAL MMG$A_ENDVEC                   !end address

  nonpaged exec       EVAL MMG$A_ENDVEC                   !start address
  data   URKW         EVAL MMG$FRSTRONLY                  !end address

  nonpaged exec       EVAL MMG$FRSTRONLY                  !start address
  code   UR           EXAM MMG$GL_PGDCOD                  !end address

  pageable exec       EXAM MMG$GL_PGDCOD                  !start address
        UR            EVAL MMG$AL_PGDCODEN                !end address
--------------------------------------------------------------------------
allocatable system EVAL MMG$A_SYS_END                     !inital start address
    space             EVAL G + @BOO$GL_SPTFREL*200        !current low address
                      EVAL G200 + @BOO$GL_SPTFREH*200 !end address

    adapter           DEF IOSPACE =@(@MMG$GL_SBICONF) !define symbol
        I/O space EVAL IOSPACE                            !start address
           KW         EVAL @(@MMG$GL_SBICONF+((@EXE$GL_NUMNEXUS-1)*4))+200
                                                          !approx. end address

    coninterr sptes EXAM EXE$GL_RTIMESPT                  !if 0, none allocated
        KR or KW      DEF RTSPTE = G+@(@EXE$GL_RTBITMAP)*200 !define symbol
                      EVAL RTSPTE                         !start address
                      EVAL RTSPTE + 200*(@EXE$GL_RTIMESPT)
                                                          !end address

    system disk       READ SYS$SYSTEM:SYSDEF.STB          !define UCB symbols
         SVPN         EVAL G+(@(SYS$GL_BOOTUCB+UCB$L_SVPN)*200) !start address
           KW         EVAL G200+(@(SYS$GL_BOOTUCB+UCB$L_SVPN)*200) !end address

    black hole        EVAL G+((@EXE$GL_SVAPTE-@MMG$GL_SPTBASE)*80) !start address
       page KW        EVAL G200+((@EXE$GL_SVAPTE-@MMG$GL_SPTBASE)*80) !end addres
s

    RMS.EXE           EVAL RMS                            !start address
        UR            EXAM EXE$GL_SYSMSG                  !end address

   SYSMSG.EXE         DEF SYSMSG = @EXE$GL_SYSMSG         !define symbol
         UR           EVAL SYSMSG                         !start address
                      ![TBS]                              !end address

   device driver SVPNs                                    !see subsection below
        KW
```

```
RPB                 DEF RPB = @EXE$GL_RPB               !define symbol
          URKW      EVAL RPB                            !start address
                    EVAL RPB+200                        !end address

PFN database        DEF PFNDATA = @PFN$A_BASE           !define symbol
          URKW      EVAL PFNDATA                        !start address
                    EVAL @PFN$AB_TYPE+(@MMG$GL_MAXPFN-@MMG$GL_MINPFN)
                                                        !end address

paged pool          DEF PAGEDYN = @MMG$GL_PAGEDYN       !define symbol
          URKW      EVAL PAGEDYN                        !start address
                    EVAL PAGEDYN + @SGN$GL_PAGEDYN      !end address
--------------------------------------------------------------------------
nonpaged pool       DEF NPAGEDYN = @MMG$GL_NPAGEDYN     !define symbol
     variable       EVAL NPAGEDYN                       !start address
          ERKW      EXAM MMG$GL_NPAGNEXT                !actual end address
                    EVAL NPAGEDYN + @SGN$GL_NPAGEVIR    !"virtual" end address

    device driver   SHOW DEVICE                         !DPT is start address
     images                              !DPT plus DPT size is end address

    MP.EXE          EXAM EXE$GL_MP                      !if 0, then not loaded
                                                        !   else start address
                    DEF MP = @EXE$GL_MP                 !define symbol
                    EVAL MP + (@(MP+8)@10)@-10          !end address

    SCSLOA.EXE      EXAM/INST SCS$ACCEPT                !if JMP @#EXE$LOAD_ERROR,
                                                        !  then SCS is not loaded
                    DEF SCSLOA = @(SCS$ACCEPT+2)-C      !define symbol
                    EVAL SCSLOA                         !start address
                    EVAL SCSLOA + (@(SCSLOA+8)@10)@-10  !end address

    SYSLOAxxx.EXE
                    EXAM EXE$GB_CPUTYPE                 !read cpu type low byte
                    DEF SYSLOA = MCHK-2BC               !if cpu = 3 = 730/725
                    DEF SYSLOA = MCHK-2C8               !if cpu = 2 = 750
                    DEF SYSLOA = MCHK-D0                !if cpu = 1 = 780/785/782
                    EVAL SYSLOA                         !start address
                    EVAL SYSLOA + @SYSLOA               !end address

    TTDRIVER.EXE    EVAL TTDRIVER                       !start address
                    EVAL TTDRIVER+(@(TTDRIVER+8)@10)@-10 !end address

    system disk     READ SYS$SYSTEM:SYSDEF.STB          !get RPB symbols
    boot driver     DEF IOVEC = @(@EXE$GL_RPB +RPB$L_IOVEC) !define symbol
                    EVAL IOVEC                          !start address
                    EVAL IOVEC + @(@EXE$GL_RPB+RPB$L_IOVECSZ)
                                                        !end address
    CI microcode    EXAM SCS$GL_MCADR                   !start address
                    EVAL @. + (200*12)                  !end address
--------------------------------------------------------------------------

LRP list            DEF LRPLIST = @IOC$GL_LRPSPLIT      !define symbol
```

```
                ERKW   EVAL LRPLIST                        !start address
                       EXAM MMG$GL_LRPNEXT                 !approx. end address
                       EVAL LRPLIST + (@SGN$GL_LRPCNTV*@IOC$GL_LRPSIZE)
                                                           !"virtual" end address

IRP list               DEF IRPLIST = @EXE$GL_SPLITADR      !define symbol
                ERKW   EVAL IRPLIST                        !start address
                       EXAM MMG$GL_IRPNEXT                 !approx. end address
                       EVAL IRPLIST + (@SGN$GL_IRPCNTV*A0) !"virtual" end address

SRP list               DEF SRPLIST = @IOC$GL_SRPSPLIT      !define symbol
                ERKW   EVAL SRPLIST                        !start address
                       EXAM MMG$GL_SRPNEXT                 !approx. end address
                       EVAL SRPLIST + (@SGN$GL_SRPCNTV*@SGN$GL_SRPSIZE)
                                                           !"virtual" end address

null access page       EVAL @EXE$GL_INTSTKLM - 200         !start address
                  NA   EXAM EXE$GL_INTSTKLM                !end address

interrupt stack        EXAM EXE$GL_INTSTKLM                !low address (newest stack)
                ERKW   EXAM EXE$GL_INTSTK                  !high address (oldest stack
)

null access page       EXAM EXE$GL_INTSTK                  !start address
                       EVAL @EXE$GL_INTSTK+200             !end address

System Control         DEF SCB = @EXE$GL_SCB               !define symbol
      Block ERKW       EVAL SCB                            !start address
                       EXAM SWP$GL_BALBASE                 !end address

balance set slots      DEF BALBASE = @SWP$GL_BALBASE       !define symbol
                ERKW   EVAL BALBASE                        !start address
                       EVAL BALBASE + (@SGN$GL_BALSETCT*@SWP$GL_BSLOTSZ*200)
                                                           !end address

system header          DEF SYSPHD = @MMG$GL_SYSPHD         !define symbol
                ERKW   EVAL SYSPHD                         !start address
                       EVAL SYSPHD + @MMG$GL_SYSPHDLN      !end address

system page table      DEF SPT = @MMG$GL_SPTBASE           !define symbol
                ERKW   EVAL SPT                            !start address
                       EXAM MMG$GL_GPTE                    !end address

global page table      DEF GPT = @MMG$GL_GPTE              !define symbol
                URKW   EVAL GPT                            !start address
                       EXAM MMG$GL_MAXGPTE                 !end address
```

V4 System Space Organization

```
        REGION        SDA COMMANDS
           PROTECTION

SYS.EXE             EVAL G                              !start address
                    EVAL MMG$A_SYS_END                  !end of system image

   system service EVAL G                                !start address
   vectors UR     EVAL MMG$A_ENDVEC                     !end address

   nonpaged exec EVAL MMG$A_ENDVEC                      !start address
   data URKW/UREW   EVAL MMG$FRSTRONLY                  !end address

   nonpaged  UR EVAL MMG$FRSTRONLY                      !start address
   exec code    EXAM MMG$GL_PGDCOD                      !end address

   pageable     EXAM MMG$GL_PGDCOD                      !start address
   exec UR      EVAL MMG$AL_PGDCODEN                    !end address
------------------------------------------------------------------------------
allocatable         EVAL MMG$A_SYS_END                  !inital start address
    system space    EVAL G + @BOO$GL_SPTFREL*200        !current low address
                    EVAL G200 + @BOO$GL_SPTFREH*200     !end address

    adapter         DEF IOSPACE =@@MMG$GL_SBICONF       !define symbol
       I/O space    EVAL IOSPACE                        !start address
         KW         EVAL @(@MMG$GL_SBICONF+((@EXE$GL_NUMNEXUS-1)*4))+200
                                                        !approx. end address

    coninterr sptes EXAM EXE$GL_RTIMESPT                !if 0, none allocated
         KR or KW DEF RTSPTE = G+@(@EXE$GL_RTBITMAP)*200 !define symbol
                    EVAL RTSPTE                         !start address
                    EVAL RTSPTE + 200*(@EXE$GL_RTIMESPT)
                                                        !end address

    system disk     READ SYS$SYSTEM:SYSDEF.STB          !define UCB symbols
       SVPN         EVAL G+(@(SYS$GL_BOOTUCB+UCB$L_SVPN)*200) !start address
         KW         EVAL G200+(@(SYS$GL_BOOTUCB+UCB$L_SVPN)*200) !end address

    mount verify    EVAL G+((@EXE$GL_SVAPTE-@MMG$GL_SPTBASE)*80) !start address
       page KW      EVAL G200+((@EXE$GL_SVAPTE-@MMG$GL_SPTBASE)*80) !end addres
s

    erase pattern EXAM EXE$GL_ERASEPB                   !start address
     buffer KW      EVAL @EXE$GL_ERASEPB+200            !end address

    erase pattern EXAM EXE$GL_ERASEPPT                  !start address
     PPT KW         EVAL @EXE$GL_ERASEPPT+200           !end address

    RMS.EXE         EVAL RMS                            !start address
           UR       EXAM EXE$GL_SYSMSG                  !end address

    SYSMSG.EXE      DEF SYSMSG = @EXE$GL_SYSMSG         !define symbol
             UR     EVAL SYSMSG                         !start address
                    ![TBS]                              !end address
```

```
    MSCP.EXE       EVAL  MSCP                        !if 0, then not loaded
                                                     !  else, start address
                   EVAL MSCP + @MSCP                 !end address

    device driver  SVPNs                             !see subsection below
         KW
---------------------------------------------------------------------------
RPB                DEF RPB = @EXE$GL_RPB             !define symbol
         URKW      EVAL RPB                          !start address
                   EVAL RPB+200                      !end address

PFN database       DEF PFNDATA = @PFN$A_BASE         !define symbol
           ERKW    EVAL PFNDATA                      !start address
                   EVAL @PFN$AB_TYPE+(@MMG$GL_MAXPFN-@MMG$GL_MINPFN)
                                                     !end address

paged pool         DEF PAGEDYN = @MMG$GL_PAGEDYN     !define symbol
           ERKW    EVAL PAGEDYN                      !start address
                   EVAL PAGEDYN + @SGN$GL_PAGEDYN    !end address
---------------------------------------------------------------------------
nonpaged pool      DEF NPAGEDYN = @MMG$GL_NPAGEDYN !define symbol
     variable      EVAL NPAGEDYN                     !start address
           ERKW    EXAM MMG$GL_NPAGNEXT              !actual end address
                   EVAL NPAGEDYN + @SGN$GL_NPAGEVIR !"virtual" end address

    device driver  SHOW DEVICE                       !DPT is start address
      images                         !DPT plus DPT size is end address

    MP.EXE         EVAL MP                           !if 0, then not loaded
                                                     !  else, start address
                   EVAL MP + @(MP+4)                 !end address

    VAXEMUL.EXE    EXAM MMG$GL_VAXEMUL_BASE          !if 0, then not loaded
                                                     !  else, start address
                   EVAL @.+ @(@.)                    !end address

    FPEMUL.EXE     EXAM MMG$GL_FPEMUL_BASE           !if 0, then not loaded
                                                     !else, start address
                   EVAL @.+ @(@.)                    !end address


    CLUSTRLOA.EXE EVAL CLUSTRLOA                     !if 0, then not loaded
                                                     !  else, load address
                   EVAL CLUSTRLOA+380                !start address ; V4.0
                   EVAL CLUSTRLOA+380+@(CLUSTRLOA+380)!end address

    SYSLOAxxx.EXE EXAM EXE$GB_CPUTYPE                !read cpu type low byte
                   EXAM MMG$GL_SYSLOA_BASE           !load address
                   DEF SYSLOA = MCHK-18              !cpu=1=780/2/5 V4.0-V4.2
                   DEF SYSLOA = MCHK-CE0             !cpu=2=750 V4.0-V4.2
                   DEF SYSLOA = MCHK-A74             !cpu=3=730 V4.0-V4.2
 |                 DEF SYSLOA = MCHK-18              !cpu=4=VAX 8600 V4.1-V4.2
 |                 DEF SYSLOA = MCHK-A6C             !cpu=7=uVAX I V4.1-V4.2
```

```
                   DEF SYSLOA = MCHK-AEC             !cpu=7=WS I V4.1-V4.2
                   DEF SYSLOA = MCHK-C44             !cpu=8=uVAX II V4.2
                   DEF SYSLOA = MCHK-CC4             !cpu=8=WS II V4.2
                   EVAL SYSLOA                       !start address
                   EVAL SYSLOA + (@(SYSLOA+8)@10)@-10
                                                     !end address

  SCSLOA.EXE       EVAL SCSLOA                       !if 0, not loaded
                                                     !  else, load address
                   EVAL SCSLOA+2B0                   !start address; V4.0
                   EVAL SCSLOA+2B0+@(SCSLOA+2B0)     !end address; V4.0

  TTDRIVER.EXE     EVAL TTDRIVER                     !start address
                   EVAL TTDRIVER+(@(TTDRIVER+8)@10)@-10 !end address

  system disk      READ SYS$SYSTEM:SYSDEF.STB        !get RPB symbols
  boot driver      DEF IOVEC = @(@EXE$GL_RPB +RPB$L_IOVEC) !define symbol
                   EVAL IOVEC                        !start address
                   EVAL IOVEC+@(@EXE$GL_RPB+RPB$L_IOVECSZ)
                                                     !end address

  CI microcode     EXAM SCS$GL_MCADR                 !if 0, not loaded
                                                     !  else, load address
                   EVAL @. + (200*12)                !end address
```

```
LRP list           DEF LRPLIST = @IOC$GL_LRPSPLIT    !define symbol
          ERKW     EVAL LRPLIST                      !start address
                   EXAM MMG$GL_LRPNEXT               !approx. end address
                   EVAL LRPLIST + (@SGN$GL_LRPCNTV*@IOC$GL_LRPSIZE)
                                                     !"virtual" end address

IRP list           DEF IRPLIST = @EXE$GL_SPLITADR    !define symbol
          ERKW     EVAL IRPLIST                      !start address
                   EXAM MMG$GL_IRPNEXT               !approx. end address
                   EVAL IRPLIST + (@SGN$GL_IRPCNTV*D0) !"virtual" end address

SRP list           DEF SRPLIST = @IOC$GL_SRPSPLIT    !define symbol
          ERKW     EVAL SRPLIST                      !start address
                   EXAM MMG$GL_SRPNEXT               !approx. end address
                   EVAL SRPLIST + (@SGN$GL_SRPCNTV*@SGN$GL_SRPSIZE)
                                                     !"virtual" end address

null access page   EVAL @EXE$GL_INTSTKLM - 200       !start address
              NA   EXAM EXE$GL_INTSTKLM              !end address

interrupt stack    EXAM EXE$GL_INTSTKLM              !low address (newest stack)
          ERKW     EXAM EXE$GL_INTSTK                !high address (oldest stack

null access page   EXAM EXE$GL_INTSTK                !start address
              NA   EVAL @EXE$GL_INTSTK+200           !end address

System Control     EXAM EXE$GL_SCB                   !start address
     Block ERKW    EXAM SWP$GL_BALBASE               !end address
```

```
balance set slots DEF BALBASE = @SWP$GL_BALBASE   !define symbol
           ERKW   EVAL BALBASE                    !start address
                  EVAL BALBASE + (@SGN$GL_BALSETCT*@SWP$GL_BSLOTSZ*200)
                                                  !end address

system header     EXAM MMG$GL_SYSPHD              !start address
           ERKW   EVAL @MMG$GL_SYSPHD + @MMG$GL_SYSPHDLN  !end address

system page table EXAM MMG$GL_SPTBASE             !start address
           ERKW   EXAM MMG$GL_GPTE                !end address

global page table EXAM MMG$GL_GPTE                !start address
           URKW   EXAM MMG$GL_MAXGPTE             !end address
```

SYS.EXE

SYS.EXE is, of course, the system image. It contains most of the system services, the software interrupt service routines, scheduling and memory management code, etc. Its first several pages contain the system service vectors. For more information on them, see the section SYSTEM SERVICE VECTORS.

The nonpaged exec code sub-region has different protections at different times. Initially, its protection prohibits any write. However, when a fatal bugcheck occurs, nonpaged exec code is made writable so that the fatal bugcheck overlay can overwrite it. Thus, if you look at the System Page Table in a crash dump, the protection in the SPTEs that map the nonpageable exec code is URKW. Also, the protection of this sub-region is changed temporarily by XDELTA so that breakpoints can be set and other modifications made.

Between the end of the pageable exec and MMG$A_SYS_END is virtual address space used by the INIT module and XDELTA. INIT executes during system initialization. At its completion, it releases the physical pages it occupied (and those of XDELTA, if XDELTA is not to remain resident) to the free list. The SPTEs which mapped it have no further use and are left invalid.

SYS.EXE modules are in the facility [SYS]. The location in SYS.EXE that corresponds to the start of the loaded image is 80000000.

Allocatable System Space

During system initialization, a number of system page table entries (SPTEs) are reserved for variable allocation of system address space. These allocatable SPTEs begin at the end of the address space used to map SYS.EXE. RMS, connect to interrupt driver SPTEs, and I/O adapters' physical register spaces are among the things mapped in this region.

The start and end sytem virtual page numbers (SVPN) available for allocation are recorded in BOO$GL_SPTFREL and BOO$GL_SPTFREH. As SPTEs are allocated from the low end, BOO$GL_SPTFREL is updated to reflect the new low SVPN. Allocation is no longer possible when BOO$GL_SPTREL and BOO$GL_SPTFREH are equal. Allocation from these SPTEs is generally permanent and is usually done through invoking the system routine IOC$ALOSPT.

Adapter I/O Space

During system initialization VAX/VMS determines what kind of adapter or controller, if any, is present at each backplane interface, or nexus. A nexus is a physical connection to the backplane which transmits and responds to commands. Each nexus has a unique number

used for identification and for priority arbitration. Each nexus also is assigned a range of physical address space for device control registers. See the VAX Hardware Handbook for a brief description of the physical address space layouts for each processor. Additionally, VAX I/O space includes an assignment of 128K words for the entire UNIBUS address space of each UNIBUS the processor supports. After system initialization code turns on memory management, access to nexus registers and UNIBUS address space requires that page table entries contain page frame numbers (PFN) corresponding to these physical VAX I/O addresses.

VAX/VMS allocates system virtual address space for each nexus present in the hardware configuration to allow virtual access to the nexus's registers. The physical address range reserved for each nexus is considerably larger than current nexus use. VMS maps only the portion of the range that corresponds to actual adapter registers. VMS maps only the I/O portion of the UNIBUS address range to enable access to UNIBUS peripherals' registers. The table below lists how many pages are allocated for each adapter type.

ADAPTER TYPE	NUMBER OF PAGES (DECIMAL)
local memory controller	1
multiport memory controller	1
MASSBUS adapter	8
UNIBUS adapter	24
Adapter registers (8)	
UNIBUS I/O space (16)	
DR780 or DR750	4
CI780 or CI750	8

In addition, one page of system address space is allocated for each nexus without a controller or adapter to allow for an adapter's being brought on line subsequent to system initialization. Currently, this mechanism is used only for MA780s.

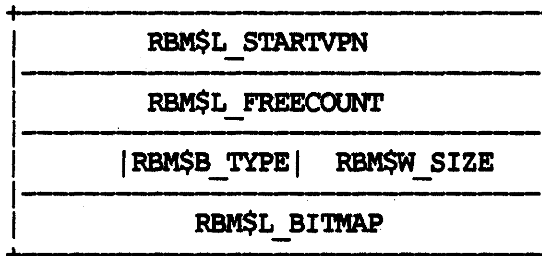
Two arrays are allocated from nonpaged pool and filled in to record the nexus present on the system and the starting system virtual address space for the nexus registers. The global symbol EXE$GL_NUMNEXUS contains the number of entries in each of the two arrays. The global symbol MMG$GL_SBICONF contains the address of a longword array of starting system virtual addresses.

Although only VAX-11/780s have SBIs, all VAXen have nexus and adapters whose registers must be accessed virtually during VMS's operations. The name is historical, reflecting hardware on the first VAX, the VAX-11/780.

The global symbol EXE$GL_CONFREG contains the address of a byte array of nexus device types. The values for the different nexus device types are defined by the macro $NDTDEF in SYS$LIBRARY:LIB.MLB. To obtain the adapter type and starting virtual address of a particular nexus, use the nexus number as a subscript or index into these two arrays.

CONINTERR SPTEs

If the SYSBOOT parameter REALTIME_SPTES is nonzero, SPTEs are allocated for the Connect to Interrupt driver's use in double-mapping user supplied code and buffers into system space. The protection on the pages is either KW or KR, depending on whether or not the user's QIO request is IO$_CONINTREAD. A data structure is allocated in nonpaged pool to describe which SPTEs are available and which are in use. The data structure offsets are defined with SYS$LIBRARY:LIB.MLB macro $RBMDEF. EXE$GL_RTBITMAP contains the starting address of the data structure.



RBM$L_STARTVPN contains the starting SPTE number. RBM$L_FREECOUNT contains the number of CONINTERR SPTEs still available for allocation. The bitmap has 1 bit for each SPTE, with the bit clear to indicate that the SPTE is in use. If you suspect a problem related to the Connect to Interrupt driver or user supplied code mapped into this virtual address range, read the driver module [DRIVER]CONINTERR, the user program that issues $QIOs to this driver, and the V3 VAX/VMS Real-Time User's Guide, Chapter 4.

Black Hole Page

The black hole page consists of a physical page of memory (whose PFN is recorded in EXE$GL_BLAKHOLE). Under V3, an SPTE is also allocated to map the black hole page. When a UNIBUS adapter powers down, all SPTEs that mapped adapter register and UNIBUS I/O space are modified to point to the black hole page in order to prevent drivers' causing machine checks by referencing the no-longer accessible adapter I/O space. The power failure of a MASSBUS adapter results in similar use. Under V3, the black hole page is also used as a one page buffer for mount verification's use in reading a disk's home block. With V4, a separate physical page is used for this purpose.

The sources for the black hole page are six thousand light years away, in the direction of Cygnus X-1.

Mount Verify Page

This page is a buffer for mount verification's use in reading a disk's home block.

Erase Pattern Buffer

During system initialization, the erase pattern buffer is allocated and initialized as a page of binary zeros. The contents of the buffer are the default source for overwriting the contents of files marked "erase on delete".

Erase Pattern Pseudo Page Table

During system initialization, the erase pattern pseudo page table is allocated and initialized as 128 page table entries, each of which maps to the physical page containing the erase pattern buffer. The page table is used to enable DMA transfer of potentially 128 copies of the erase pattern buffer to overwrite a large piece of a file in one I/O request.

RMS.EXE

The RMS image contains the procedures that make up the Record Management Services. During system initialization, system address space is allocated to map RMS.EXE as a pageable section. Its starting address is stored in MMG$GL_RMSBASE and in CTL$GL_RMSBASE. SDA automatically defines the symbol RMS to be the contents of MMG$GL_RMSBASE. Since the system message file is mapped immediately above RMS, the simplest way to determine RMS's end boundary is as the start of the message file.

Most RMS modules are in the facility [RMS]; the exceptions are [INSTAL]INSKFSCAN, [VMSLIB]MATCHNAME, and [SYS]RMSVECTOR. The location in RMS.EXE that corresponds to the start of the loaded image and to SDA symbol RMS is local symbol RMS$DISPATCH in module [SYS]RMSVECTOR. For debugging RMS problems there is a symbol table, SYS$SYSTEM:RMSDEF.STB, which contains symbol definitions for user RMS data structures (e.g., FAB) and RMS status codes.

If you are debugging a problem involving RMS, type the following SDA command to get addresses in the loaded image resolved to RMS.EXE symbols.

```
SDA> READ /RELOCATE = @MMG$GL_RMSBASE SYS$SYSTEM:RMS.STB
```

SYSMSG.EXE

During system initialization, system address space is allocated to map SYSMSG.EXE as a pageable section. Its starting address is stored in EXE$GL_SYSMSG. SYSMSG.EXE is a shareable image consisting entirely of data, the standard system message text. The $GETMSG

system service uses this section to translate message identification codes to message text.

Device Driver SVPNs

SPTEs needed by VMS direct I/O device drivers are also allocated from this region. Typically, a disk driver uses its SPTE to make a virtual mapping to a process buffer page to which ECC correction must be applied. Determining all the SPTEs allocated to DMA device drivers requires looking through the I/O data space for DMA device units and examining UCB$L_SVPN for each one. UCB$L_SVPN records the system virtual page number allocated to that device unit.

MSCP.EXE

MSCP.EXE, the MSCP server for local disks, is loaded into this region as a result of issuing the MSCP command to the SYSGEN image. It accepts as input MSCP command packets received from other cluster nodes and translates them into the corresponding QIO requests to local disks. MSCP.EXE returns MSCP status and data to the requesting node.

MSCP modules are in facility [MSCP]. The location in MSCP.EXE that corresponds to the start of the loaded image is offset 0 in the image. Its starting address is stored in SCS$GL_MSCP. V4 SDA automatically defines the symbol MSCP to be the contents of SCS$GL_MSCP.

Restart Parameter Block (RPB)

The RPB is a data structure that is physically and virtually based; it is not pageable. CPU console restart code locates it in order to restart a system following a powerfail or other halt. The RPB also communicates information between the various steps in system initialization. To display it, type the following SDA commands.

```
SDA> READ SYS$SYSTEM:SYSDEF.STB
SDA> FORMAT @EXE$GL_RPB/TYPE=RPB !necessary to specify type
```

Page Frame Number (PFN) Data Base

The PFN data base, which is nonpageable, is used to record the current state of any pages of physical memory whose virtual state can change. It is actually eight different arrays, each of which is indexed by page frame number. The highest valid subscript into those arrays is the contents of MMG$GL_MAXPFN. If MMG$GL_MINPFN is

nonzero, its contents are the lowest valid subscript; otherwise, the lowest valid subscript is 1.

The 8 longwords beginning at PFN$A_BASE contain the starting addresses of the various PFN arrays. You can use the first SDA command shown below to obtain the information for a specific physical page; the second command displays the entire data base; and the others display pages on the selected list

```
SDA> SHOW PFN_DATA <pfn>  !show data for page <pfn>
SDA> SHOW PFN_DATA/SYSTEM !show entire PFN data base
SDA> SHOW PFN_DATA/FREE   !show PFN data for free list pages
SDA> SHOW PFN_DATA/MODIFY !show PFN data for mod. list pages
SDA> SHOW PFN_DATA/BAD    !show PFN data for bad list pages
```

Paged Pool

Paged pool is pageable dynamic data storage. It is typically used for logical name blocks (LOGs), mount list entries (MTLs), global section descriptors (GSDs), known file headers (KFHs), known file entries (KFIs), and, under V4, process quota blocks (PQBs).

To display allocation from this region, type either of the following SDA commands.

```
SDA> SHOW POOL/PAGED         !display data structure contents
SDA> SHOW POOL/PAGED/HEADER  !display only structure headers
```

Blocks not in use are linked in a singly linked list whose head is at EXE$GL_PAGED. Each block contains the pointer to the next free block at offset 0 and its own size at offset 4.

Under V4, PQBs, initially allocated from paged pool, are deallocated to a lookaside list whose head is at EXE$GL_PQBFL. Process creation code attempts to allocate a PQB through removing an entry from this list, as a faster alternative to general pool allocation.

To see the state of paged pool on a running system, type the DCL command SHOW MEMORY.

Insufficient paged pool is one possible cause of the error SYSTEM-F-INSFMEM.

Nonpaged Pool Variable Length List

The nonpaged pool variable length list is used for allocating nonpaged pool that doesn't fit the allocation constraints of any of the lookaside lists. Typically, the larger unit control blocks (UCBs) and loaded images such as device drivers are allocated here.

Nonpaged pool is extended dynamically as needed, up to a total of SYSBOOT parameter SGN$GL_NPAGEVIR bytes. MMG$GL_NPAGNEXT contains the start address of the extensible area. That is, pool addresses lower than this are valid.

To display the allocation from this region, type either of the following SDA commands.

```
SDA> SHOW POOL/NONPAGED        !display data structure contents
SDA> SHOW POOL/NONPAGED/HEADER !display structure headers
```

Blocks not in use are linked in a singly linked list whose head is at EXE$GL_NONPAGED + 4. Each block contains the pointer to the next free block at offset 0 and its own size at offset 4.

To see the state of nonpaged pool on a running system, type the DCL command SHOW MEMORY.

Insufficient nonpaged pool is one possible cause of the error SYSTEM-F-INSFMEM. Dynamically loaded images are usually placed in the nonpaged pool variable list, although if they are smaller than an LRP, they may be loaded into an allocated LRP. The following subsections describe the various images that are loaded dynamically into nonpaged pool.

Device Driver Images

Most VMS device drivers are loaded dynamically into nonpaged pool. The initial output (titled DDB list) of the SDA SHOW DEVICE command shows the starting address (DPT) and size (DPT size) of most loaded drivers. SDA omits the terminal class driver, TTDRIVER, from its display, unless virtual terminals are enabled, but does define a symbol TTDRIVER as equal to the contents of TTY$GL_DPT. VMS driver images have the same filenames as those displayed in SDA's SHOW DEVICE output. Most driver modules are part of facility [DRIVER]. The exceptions are

[TTDRVR]DZDRIVER, TTDRIVER, YCDRIVER
[TMDRVR]TMDRIVER (V3 only)
[NETACP]NDDRIVER, NETDRIVER
[SYS]MBDRIVER, NLDRIVER
[SYSLOA]OPDRIVER

Driver images are linked to a base of 0 with $$$105_PROLOGUE as the first PSECT. This PSECT is defined through the invocation of the DPTAB macro which also names the driver and builds a header for the Driver Prologue Table (DPT) dynamic data structure. The loaded driver image begins with PSECT $$$105_PROLOGUE and its DPT header. All loaded drivers are queued together through the DPT header's first two longwords. The listhead for the queue of loaded drivers is at IOC$GL_DPTLIST and the following longword. SDA defines a symbol for each loaded driver it finds, using the name in the DPT. For example,

SDA's symbol LPDRIVER would correspond to offset 0 in PSECT $$$105_PROLOGUE in [DRIVER]LPDRIVER.

MP.EXE

VAX-11/782 support is loaded into nonpaged pool on a suitable configuration in response to the START/CPU DCL command. The nonpaged pool area includes data storage, the attached processor's interrupt stack, code executed by the attached processor, and code executed by the primary processor. If you are debugging a 782 crash, read the MP symbol table with the following SDA command to get addresses in the loaded image resolved to MP.EXE symbols.

```
SDA> READ /RELOCATE = @EXE$GL_MP SYS$SYSTEM:MP.STB
```

MP modules are in facility [MP]. The location in MP.EXE that corresponds to the start of the loaded image is global symbol MPS$BEGIN. Its starting address is stored in EXE$GL_MP. V4 SDA automatically defines the symbol MP to be the contents of EXE$GL_MP.

VAXEMUL.EXE

VAXEMUL.EXE is loaded into nonpaged pool during system initialization on MicroVAX CPUs to emulate VAX instructions not supported by MicroVAX microcode, such as the character istructions, decimal instructions, and EDITPC. Two vectors in the System Control Block, hex offsets C8 and CC, point to exception service routines within the loaded code. The protection on the pages containing this code is altered to be URKW, to allow execution of the code from all access modes.

The VAXEMUL.EXE image is built as part of the [EMULAT] facility. The location in VAXEMUL.EXE that corresponds to the start of the loaded image is global symbol VAX$BEGIN.

FPEMUL.EXE

FPEMUL.EXE is loaded into nonpaged pool during system initialization on any VAX CPU that doesn't have microcode support for any class of floating point instructions. The Opcode Reserved to Digital vector in the System Control Block, hex offset 10, points to an exception service routine within the loaded code. FPEMUL's service routine transfers back to EXE$OPCDEC any opcode reserved to Digital exceptions with opcodes other than the floating point ones it emulates. The protection on the pages containing this code is altered to be URKW, to allow execution of the code from all access modes.

The FPEMUL.EXE image is built as part of the [EMULAT] facility. The location in FPEMUL.EXE that corresponds to the start of the loaded image is global symbol FPS$BEGIN.

CLUSTRLOA.EXE

CLUSTRLOA.EXE, a module containing cluster connection management and distributed lock management code, is loaded into nonpaged pool on all the nodes of a VMS cluster. That CLUSTRLOA.EXE is loaded implies that SCSLOA.EXE is, as well. The array beginning at system global CLU$AL_LOAVEC defines CLUSTRLOA globals to the rest of SYS.EXE and to other modules linked against SYS.STB. Most of these CLUSTRLOA globals are the addresses of JMP instructions. Before CLUSTRLOA.EXE is loaded, the target of the JMP instructions is EXE$LOAD_ERROR. After CLUSTRLOA.EXE is loaded, the JMP destinations are altered to cause dispatch into the loaded code, and CLU$GL_LOA_ADDR contains the address at which CLUSTRLOA is loaded. After CLUSTRLOA's initialization is complete, CLUSTRLOA deallocates its initialization code. Thus, the contents of CLU$GL_LOA_ADDR no longer point to the beginning of the loaded code.

The CLUSTRLOA.EXE image is built as part of the [SYSLOA] facility. The location in CLUSTRLOA.EXE that corresponds to the contents of CLU$GL_LOA_ADDR is offset 0. V4 SDA automatically defines the symbol CLUSTRLOA to be the contents of CLU$GL_LOA_ADDR.

SCSLOA.EXE

SCSLOA.EXE, a module containing port-independent System Communication Services (SCS) routines, is loaded into nonpaged pool on systems which use the disk class, tape class, or DECnet class drivers. The array beginning at system global SCS$AL_LOAVEC defines SCS globals to the rest of SYS.EXE and to other modules linked against SYS.STB. Most of these SCS globals are the addresses of JMP instructions. Before SCSLOA.EXE is loaded, the target of the JMP instructions is EXE$LOAD_ERROR. After SCSLOA.EXE is loaded, the JMP destinations are altered to cause dispatch into the loaded code. SCS$GA_EXISTS contains the address at which SCSLOA is loaded. V4 SDA automatically defines the symbol SCSLOA to be the contents of SCS$GA_EXISTS. After SCSLOA's initialization is complete, SCSLOA deallocates its initialization code. Thus, SCS$GA_EXISTS no longer points to the beginning of the loaded code.

The SCSLOA.EXE image is built as part of the [SYSLOA] facility. The location in SCSLOA.EXE that corresponds to the contents of SCS$GA_EXISTS is offset 0. For debugging, there is a symbol table, SYS$SYSTEM:SCSDEF.STB, which contains symbol definitions for all the SCS data structures.

SYSLOAxxx.EXE

Cpu-specific support is loaded into nonpaged pool during system initialization. Cpu-specific support consists of routines such as the machine check exception service routine, the service routines for the cpu-specific interrupts, and a routine to purge UNIBUS adapter buffered datapaths. [SYSLOA], the facility for these modules, contains sources for one image for each cpu type. The image name is of the form SYSLOAxxx.EXE, where xxx designates the cpu type.

CPU	IMAGE NAME
MicroVAX I	SYSLOAUV1.EXE
MicroVAX II	SYSLOAUV2.EXE
VAX-11/730	SYSLOA730.EXE
VAX-11/750	SYSLOA750.EXE
VAX-11/780	SYSLOA780.EXE
VAX-11/785	SYSLOA780.EXE
VAX 8600	SYSLOA790.EXE
VAXstation I	SYSLOAWS1.EXE
VAXstation II	SYSLOAWS2.EXE

The array beginning at SYS.EXE global EXE$AL_LOAVEC defines SYSLOA globals to the rest of SYS.EXE and to other modules linked against SYS.STB. Most of these SYSLOA globals are the addresses of JMP instructions. Before SYSLOAxxx.EXE is loaded, the target of the JMP instructions is EXE$LOAD_ERROR. After SYSLOAxxx.EXE is loaded, the JMP destinations are altered to cause dispatch into the loaded code. Under V4 MMG$GL_SYSLOA_BASE contains the address at which the SYSLOA image is loaded. Offset 0 in the SYSLOAxxx.EXE images corresponds to the contents of MMG$GL_SYSLOA_BASE. After SYSLOA's initialization is complete, SYSLOA deallocates its initialization code. Thus, MMG$GL_SYSLOA_BASE no longer points to the beginning of the loaded code.

Global symbol EXE$LOAD_SIZE in the V3 SYSLOAxxx.EXE images corresponds to the start of the loaded images.

TTDRIVER.EXE

TTDRIVER.EXE, the terminal class driver, does device-independent processing of terminal I/O. SYSBOOT builds the name of the terminal class driver image using the parameter TTY_CLASSNAME, the default contents of which are "TT". SYSBOOT then locates the class driver and loads it into nonpaged pool. Under V4, the SYSINIT process loads OPDRIVER, the operator console port driver, and invokes routines which establish connections between OPDRIVER and the terminal class driver. SYSGEN builds the I/O data base for a terminal controller, loads its terminal port driver, and invokes driver initialization routines which establish connections between the terminal port and class drivers.

TTDRIVER.EXE modules are in facility [TTDRVR]. The location in TTDRIVER.EXE that corresponds to the start of the loaded image and to SDA symbol TTDRIVER is global symbol TT$DPT.

System Disk Boot Driver

The system disk boot driver is a minimal driver used during system initialization to read images off the system disk and during fatal bugcheck processing to write the contents of physical memory to the crashdump file. It consists of some device-independent code that interprets $QIO requests ([BOOTS]BOOTDRIVER) and some code which is specific to the boot device.

There are a number of different device-specific modules linked with VMB.EXE, the primary bootstrap. Using register arguments that describe the type and location of the system disk, VMB selects the appropriate device-specific module. During a later step in system initialization psect BOOTDRIVR_1 from the BOOTDRIVER module and psect BOOTDRIVR_4 from the appropriate device-specific module are moved to nonpaged pool for use during a system crash. Offset RPB$L_IOVEC in the RPB contains the starting address of the combined modules and offset RPB$L_IOVECSZ their size.

The simplest way to determine the boundary between the two pieces in nonpaged pool is to read the size of psect BOOTDRIVR_1 in the VMB.MAP corresponding to the VMB that booted the system.

The beginning of psect BOOTDRIVR_1 is a table, defined by macro $BQODEF in SYS$LIBRARY:LIB.MLB. BQO$L_SELECT contains the offset from the beginning of the boot driver to the device-specific qio code in the BOOTDRIVR_4 portion. In the device-specific boot driver module, this offset corresponds to the value of the ADDR argument in the $BOOT_DRIVER macro.

The device-specific modules are in facility [BOOTS]. Following is a list of these modules and their corresponding devices:

FILENAME	DEVICE
CVBTDRIVR	VAX 8600 console RL02
DDBTDRIVR	VAX-11/730, VAX-11/750 console TU58
DLBTDRIVR	RL01/RL02
DMBTDRIVR	RK06/RK07
DQBTDRIVR	VAX-11/730 RB730 RB02/RB80
DXBTDRIVR	VAX-11/780 console RX01
MBBTDRIVR	MASSBUS disks
PABTDRIVR	CI MSCP device
PUBTDRIVR	UDA

CI Microcode

During system initialization, CI microcode is loaded into nonpaged pool from a file called CI780.BIN on the console block storage medium. This filename is used regardless of processor type, since the microcode is the same for all current implementations of the CI. PADRIVER port initialization loads the microcode from pool into the CI at startup, following a powerfail, and after certain serious CI errors, such as CI local store parity error.

When you copy CI780.BIN, you must override the default copy modes that FLX and EXCHANGE use. See subsection Hints and Kinks for more information.

This area of pool has a secondary use as storage for contents of the CI port local store (device registers, virtual circuit descriptor table, translation cache, etc.) during crashes initiated by the PADRIVER. If the PADRIVER detects a serious inconsistency, it copies 1000 hex longwords from the CI port into this area so that the information is available in the crashdump and usable to someone familiar with the CI.

Large Request Packet (LRP) Lookaside List

This region is typically used for the allocation of DECnet receive buffers (NETs). Note that on systems with a large value for LRPSIZE parameter, many loaded images, such as device drivers or SCSLOA.EXE, may be allocated off the LRP lookaside list rather than the variable length list.

The LRP lookaside list is extended dynamically as needed, up to a total of SYSBOOT parameter SGN$GL_LRPCNTV packets. MMG$GL_LRPNEXT contains the start address of the extensible area. That is, LRP addresses lower than this are valid.

To display allocation from this region, type either of the following SDA commands.

```
SDA> SHOW POOL/LRP        !display data structure contents
SDA> SHOW POOL/LRP/HEADER !display only structure headers
```

Packets not in use are inserted onto the lookaside list through their first two longwords. The queue header is at IOC$GL_LRPFL and IOC$GL_LRPBL. No longer needed LRPs are deallocated by inserting them at the back of the queue, and LRPs are allocated by removing them from the front of the queue. Thus it is sometimes possible to find an intact LRP which has been deallocated but whose contents are of interest, by starting at IOC$GL_LRPBL and following the back links in the queued LRPs.

I/O Request Packet (IRP) Lookaside List

This region is typically used for the allocation of I/O request packets (IRPs), process control blocks (PCBs), job information blocks (JIBs), volume control blocks (VCBs), unit control blocks (UCBs), class driver request packets (CRDPs), and larger buffered I/O buffers.

The IRP lookaside list is extended dynamically as needed, up to a total of SYSBOOT parameter SGN$GL_IRPCNTV packets. MMG$GL_IRPNEXT contains the start address of the extensible area. That is, IRP addresses lower than this are valid.

To display allocation from this region, type either of the following SDA commands.

```
SDA> SHOW POOL/IRP        !display data structure contents
SDA> SHOW POOL/IRP/HEADER !display only structure headers
```

Packets not in use are inserted onto the lookaside list through their first two longwords. The queue header is at IOC$GL_IRPFL and IOC$GL_IRPBL. No longer needed IRPs are deallocated by inserting them at the back of the queue, and IRPs are allocated by removing them from the front of the queue. Thus it is sometimes possible to find an intact IRP which has been deallocated but whose contents are of interest, by starting at IOC$GL_IRPBL and following the back links in the queued IRPs.

Small Request Packet (SRP) Lookaside List

This region is typically used for the allocation of file control blocks (FCBs) and window control blocks (WCBs). Other data structures commonly found here are ACP queue blocks (AQBs), timer queue elements (TQEs), interrupt dispatch blocks (IDBs), channel (controller) request blocks (CRBs), typeahead buffers (TYPAHDs), device data blocks (DDBs), and smaller nonpaged pool buffers (BUFIOs).

The SRP lookaside list is extended dynamically as needed, up to a total of SYSBOOT parameter SGN$GL_SRPCNTV packets. MMG$GL_SRPNEXT contains the start address of the extensible area. That is, SRP addresses lower than this are valid.

To display allocation from this region, type either of the following SDA commands.

```
SDA> SHOW POOL/SRP        !display data structure contents
SDA> SHOW POOL/SRP/HEADER !display only structure headers
```

Packets not in use are inserted onto the lookaside list through their first two longwords. The queue header is at IOC$GL_SRPFL and IOC$GL_SRPBL. No longer needed SRPs are deallocated by inserting

them at the back of the queue, and SRPs are allocated by removing them from the front of the queue. Thus it is sometimes possible to find an intact SRP which has been deallocated but whose contents are of interest, by starting at IOC$GL_SRPBL and following the back links in the queued SRPs.

Interrupt Stack

The system switches to the interrupt stack to service all hardware interrupts, all software interrupts above IPL 3, and some serious exceptions such as machine check. In addition, the system runs on the interrupt stack in the IPL 3 interrupt service routine after it has taken the current process out of execution and before it has placed a new process into execution. It is not pageable. To display this region, type one of the following SDA commands.

```
SDA> SHOW STACK/INTERRUPT
SDA> SHOW STACK @EXE$GL_INTSTKLM:@EXE$GL_INTSTK
```

If the interrupt stack is empty, and its stack pointer points to the high end of the stack, the first command displays nothing. The second command displays the current contents of all pages allocated to the stack. The second display is difficult, perhaps impossible, to interpret since, as events occur and the stack expands and contracts, one usage [partially] overwrites previous usages.

An invalid page with no access to any access mode is allocated on both sides of the interrupt stack so that underflow and overflow of the interrupt stack cause a system halt.

System Control Block

The System Control Block (SCB) contains the interrupt and exception vectors for the system. Its size is a function of cpu type and number of UNIBUSes: a VAX 8600 SCB is four pages long; a VAX-11/780 SCB is one page long; the SCB for a VAX-11/750 with 1 UBI is two pages; the SCB for a VAX-11/750 with 2 UBIs is three pages; the SCB for a VAX-11/730 is two pages. A MicroVAX I SCB is two pages. A MicroVAX II SCB is two pages. A vector contains the system virtual address of the service routine for its interrupt or exception, plus information in the low order two bits specifying whether the service routine should run on the interrupt stack or the current process's kernel stack. The SCB is built by SYSBOOT and filled in by various system initialization steps. It is also modified as a result of SYSGEN AUTOCONFIGURE and CONNECT commands. The SCB is not pageable. To display it with the names of many interrupt and exception service routines, type the following SDA command.

```
SDA> SHOW STACK @EXE$GL_SCB : (@SWP$GL_BALBASE - 4)
```

Balance Set Slots

The balance set slot region isa table containing resident process headers. Each process header is SWP$GL_BSLOTSZ pages large, some of it pageable and some of it nonpageable. If one process faults a page in another process's header, as could happen with careless use of on-line SDA, MMG$PAGEFAULT signals an access violation.

PHV$GL_PIXBAS contains the address of a word array that identifies the owner process of each header slot. The array has SGN$GL_BALSETCT elements. An element in the array contains either a zero or the index part of the PID to which the corresponding header belongs. To examine this array, type the following SDA command.

```
SDA> EXAMINE @PHV$GL_PIXBAS ; (2 * @SGN$GL_BALSETCT)
```

To display an entire process header, type the following SDA commands.

```
SDA> SET PROCESS/INDEX=<n>      !replace <n> with pix
SDA> SHOW PROCESS/PHD           !display fixed portion of phd
SDA> SHOW PROCESS/REGISTER      !display hardware pcb
SDA> SHOW PROCESS/PAGE          !display P0 and P1 page tables
SDA> SHOW PROCESS/WORK          !display working set list
SDA> SHOW PROCESS/PROC          !display process section table
```

System Header

The system header is structured somewhat like a process header. It contains the system working set list and global section table and basically enables system code to be pageable. It is not pageable. To display it, type the following SDA commands.

```
SDA> SHOW PROCESS/SYSTEM/PHD
SDA> SHOW PROCESS/SYSTEM/WORK !display working set list
SDA> SHOW PROCESS/SYSTEM/PROC !display global section table
```

System Page Table

The system page table is used to translate system virtual addresses to physical addresses. It is not pageable, is typically placed in the highest physical memory, and must be physically contiguous. On a VAX-11/782 system, the primary processor and the attached processor share one system page table. To display the system page table, type the following SDA command.

```
SDA> SHOW PAGE_TABLE/SYSTEM
```

Global Page Table

The global page table is unlike other page tables in that it is not used by cpu memory management microcode for address translation. VMS memory management code uses the global page table to keep track of the state and location of global pages. The global page table is pageable. To display it, type the following SDA command.

```
SDA> SHOW PAGE_TABLE/GLOBAL
```

Hints And Kinks

1. The console block storage medium has an RT-11 file structure. The RT-11 file structure implements three different record formats: stream ASCII, formatted binary, and fixed-length record. Under VMS you use the V3 FLX utility or the V4 EXCHANGE utility to transfer files to and from the console.

   Both FLX and EXCHANGE select a default record transfer mode based on file extension type. For example, extensions of OBJ and BIN default as EXCHANGE /RECORD=BINARY and FLX /FB transfer modes.

   Occasionally the default based on file extension type is inconsistent with the file's record format. In particular, CI780.BIN, the CI microcode; WCSxxx.PAT, the VAX-11/780 microcode; and PCS750.BIN, the VAX-11/750 microcode, will not be copied correctly unless you override the default transfer mode.

   If you are not sure what the transfer mode should be, you can use the EXCHANGE qualifier /RECORD_FORMAT=STREAM or the FLX switch /FA for all text files (e.g. command files). Use the EXCHANGE qualifier /RECORD_FORMAT=FIXED (or /TRANSFER_MODE=BLOCK) or the FLX switch /IM for all other files (binary files such as images, microcode files, patch files). The VMS console contains no formatted binary files, so you will never want /RECORD_FORMAT=BINARY or FLX's /FB.

Additional References

V3 VAX/VMS Internals and Data Structures Manual, Section 31.2, Use of Map Files; Appendix A, Executive Data Areas; Chapter 26, Size of Virtual Address Space.

INDEX

-A-

Access violation exception
  software induced, 29
Access violation fault, 28
Adapter I/O space, 205
ASYNCWRTER bugcheck, 18, 19

-B-

Bad page list, 103
Balance set slot, 219
Black hole page, 207
Boot driver, 215
BRDMSGLOST bugcheck, 128
Bugcheck
  analysis, 11
  ASYNCWRTER, 18, 19
  BRDMSGLOST, 128
  BUG_CHECK macro, 12
  CHMONIS, 141
  CHMVEC, 143
  DBLERR, 138
  FATALEXCEPT, 33
  HALT, 139
  ILLVEC, 140
  INVEXCEPTN, 84
  IVLISTK, 137
  KRNLSTAKNV, 89
  MACHINECHK, 98
  mechanism, 22
  NOUSRWCS, 141
  OPERATOR, 13
  OUTOFSYNC, 51, 144
  overlay, 23
  PGFIPLHI, 113
  Restart, 135
  SCBRDERR, 144
  SSRVEXCPT, 146
  STATENTSVD, 135
  UNKRSTRT, 51, 59, 136
  UNXINTEXC, 17
  UNXSIGNAL, 170

-C-

Cache parity error
  VAX-11/780 and VAX-11/785, 105 to 106
Call frame
  change Mode dispatcher, 162
  layout, 116
Channel control block, 188
CHMONIS bugcheck, 141
CHMVEC bugcheck, 143
CI microcode, 216
CI780.BIN
  See CI microcode
CLI
  command table, 186
  data page, 192
  image, 186
  image name, 186
  symbol table, 186
Cluster Connection Management
  See CLUSTRLOA.EXE
CLUSTRLOA.EXE, 213
Common
  per-process, 191
Compatibility mode
  data page, 192
CONINTERR SPTEs, 207
Control store parity error
  VAX-11/780 and VAX-11/785, 106 to 107
Corrected read data
  See CRD error
Cpu timeout
  VAX-11/780 and VAX-11/785, 19
Cpu type
  identification, 98
Cpu-specific interrupt
  VAX-11/780 and VAX-11/785, 17 to 20
Cpu-specific support
  See SYSLOAxxx.EXE
  interrupt, 16
Crashdump file, 12
  backup of, 12
  deletion, 21, 22
  PAGEFILE.SYS, 21, 22, 24
  size, 21
  size affected by PHYSICALPAGES, 21
  size alteration, 22
Crashdump requirements, 21
CRD error
  VAX-11/780 and VAX-11/785, 17, 102

-D-

DBLERR bugcheck, 138
Debugger
  symbol table, 195
DELTA
  kernel mode deposit, 122
Device driver SVPNs, 209
Dispatch vector
  user-written system service,
      193
Dump file
  see Crashdump file

-E-

Emulation
  VAX character instructions
    See VAXEMUL.EXE
  VAX decimal instructions
    See VAXEMUL.EXE
  VAX floating point instructions
    See FPEMUL.EXE
Erase pattern
  buffer, 208
  pseudo page table, 208
Error confirm error
  VAX-11/780 and VAX-11/785, 108
      to 111
Error log entry
  AW
    VAX-11/780 and VAX-11/785, 19
  BE
    VAX-11/780 and VAX-11/785, 19
  bugcheck, 13, 23
  HE
    VAX-11/780 and VAX-11/785,
        103
  MC, 98
  pending, 22, 24
  SA
    VAX-11/780 and VAX-11/785, 18
  SE
    VAX-11/780 and VAX-11/785, 18
Exception, 25
  access violation
    see Access violation fault
  dispatching, 26
  hardware, 25
  reserved addressing mode
    see Reserved addressing mode
        fault
  reserved opcode
    see Reserved opcode fault
Exception (Cont.)
  reserved operand
    see Reserved operand
        exception
  software, 25
EXE$GL_MCHKERRS
  asynchronous write timeout
    VAX-11/780 and VAX-11/785, 19
  machine check, 98
  SBI fault
    VAX-11/780 and VAX-11/785, 19
EXE$GL_MEMERRS
  corrected read data
    VAX-11/780 and VAX-11/785, 18
  RDS error
    VAX-11/780 and VAX-11/785,
        103
  SBI alert
    VAX-11/780 and VAX-11/785, 18

-F-

FATALEXCEPT bugcheck, 33
Fiche organization
  See Microfiche organization
Files-11 XQP, 186
  current IRP, 93
  location, 186
  pending IRP, 93
  stack, 87
Forced crash, 43
FPEMUL.EXE, 212

-G-

Global page table, 220

-H-

Halt
  CHMx on interrupt stack
    VAX-11/780 and VAX-11/785, 50
  clock phase error
    VAX-11/785, 51
  double error
    VAX-11/780 and VAX-11/785, 51
  halt instruction
    VAX-11/780 and VAX-11/785, 48
  illegal vector
    VAX-11/780 and VAX-11/785, 52
  invalid interrupt stack
    VAX-11/780 and VAX-11/785, 53
  no user WCS
    VAX-11/780 and VAX-11/785, 54

Halt (Cont.)
  pathological
    VAX-11/780 and VAX-11/785, 54
  VAX-11/780, 46
  VAX-11/780 and VAX-11/785
      console message, 47
  VAX-11/785, 46
HALT bugcheck, 139
Hang, 65
  process, 75
  system, 65

-I-

I/O request
  location, 92
I/O request packet lookaside list
  See IRP lookaside list
ILVISTK bugcheck, 137
Image activator
  context page, 192
  scratch page, 193
Image header buffer, 193
Image I/O segment, 185
INVEXCEPTN bugcheck, 84
IPL usage, 157
IRP lookaside list, 217

-K-

Kernel request packet lookaside
    list
  See KRP lookaside list
Kernel stack
  see Stack, kernel
KRNLSTAKNV bugcheck, 89
KRP lookaside list, 194

-L-

Large request packet lookaside
    list
  See LRP lookaside list
Lookaside List
  LRP
    See LRP lookaside list
  SRP
    See SRP lookaside list
Lookaside list
  IRP
    See IRP lookaside list
  KRP
    See KRP lookaside list
LRP lookaside list, 216

-M-

Machine check
  VAX-11/780, 101
  VAX-11/785, 101
MACHINECHK bugcheck, 98
Message section
  per-process, 186
  system
    See SYSMSG.EXE
Microcode not supposed to be here
  VAX-11/780 and VAX-11/785, 107
      to 108
Microfiche organization, 196
MicroVAX I
  console single step, 71
MicroVAX II
  console single step, 72
MP.EXE, 212
MSCP server
  See MSCP.EXE
MSCP.EXE, 209
Mutex
  global name, 119
MWAIT state, 117

-N-

Nonpaged pool variable list, 210
NOUSRWCS bugcheck, 141
Null Job
  kernel stack, 87

-O-

OPERATOR bugcheck, 13
OUTOFSYNC bugcheck, 51, 144

-P-

P1 pointer page, 195
PADRIVER bugcheck
  CI port local store, 216
Paged pool, 210
PFN data base, 209
PGFIPLHI bugcheck, 113
Process
  MWAIT, 117
  resource wait, 117
Process allocation region, 188
Process header
  access with online SDA, 6
  location, 219
  P1 window, 189

Process I/O segment, 187
Processor status longword
  layout, 116
PSL
  See Processor status longword

-R-

RDS error
  VAX-11/780 and VAX-11/785, 17,
    102 to 104
Read data substitute
  See RDS error
Read timeout error
  VAX-11/780 and VAX-11/785, 108
    to 111
Reserved addressing mode fault,
  30
Reserved opcode fault, 30
Reserved operand exception, 31
Resource wait, 117
  RWAST, 120
  RWBRK, 126
  RWCLU, 133
  RWIMG, 129
  RWLCK, 129
  RWMBX, 123
  RWMPB, 132
  RWMPE, 131
  RWNPG, 124
  RWPAG, 126
  RWPFF, 126
  RWQUO, 129
  RWSCS, 132
  RWSWP, 130
RESTAR.CMD
  editing, 59
Restart parameter block
  contents, 209
  use during restart, 135
  use during VAX-11/780 restart,
    58
  use during VAX-11/785 restart,
    58
  virtual location, 209
RMS
  directory cache, 191
  image I/O segment, 185
  location, 208
  per-process statistics, 190
  process context area, 190
  process I/O segment, 187
  process permanent IFAB/IRAB
    table, 191
RMS (Cont.)
  tracepoint page, 190
RNS$_IACLOCK, 129
RPB
  See Restart parameter block
RSN$_ASTWAIT, 120
RSN$_BRKTHRU, 126
RSN$_CLUSTRAN, 133
RSN$_JQUOTA, 129
RSN$_LOCKID, 129
RSN$_MAILBOX, 123
RSN$_MPLEMPTY, 131
RSN$_MPWBUSY, 132
RSN$_NPDYNMEM, 124
RSN$_PGDYNMEM, 126
RSN$_PGFILE, 126
RSN$_SCS, 132
RSN$_SWPFILE, 130
RW*** code
  see Resource wait

-S-

SBI alert
  VAX-11/780 and VAX-11/785, 18
SBI fault
  VAX-11/780 and VAX-11/785, 19
SBI silo compare
  VAX-11/780 and VAX-11/785, 17
SCB
  See System control block
SCBRDERR bugcheck, 144
SCS
  See System communication
    services
SCSLOA.EXE, 213
Security audit data page, 192
SHOW ERROR output
  cpu, 98
    VAX-11/780 and VAX-11/785, 19
  memory
    VAX-11/780 and VAX-11/785, 18,
      103
Single step
  CI sanity timer, 82
  MicroVAX I console commands, 71
  MicroVAX II console commands,
    72
  VAX 8600 console commands, 70
  VAX-11/730 console commands, 71
  VAX-11/750 console commands, 70
  VAX-11/780 console commands, 70
  VAX-11/785 console commands, 70

Small request packet lookaside list
  See SRP lookaside list
SRP lookaside list, 217
SSRVEXCEPT bugcheck, 146
Stack
  exec, 152 to 155, 194
  Files-11 XQP, 87
  interrupt, 156 to 160, 218
  kernel, 87, 161 to 165, 194
    invalid, 89
    location, 87
  Null Job, 87
  supervisor, 194
  Swapper, 87
  user, 185
    extra pages, 185
STATENTSVD bugcheck, 135
Swapper
  kernel stack, 87
SYS.EXE, 205
SYSLOAxxx.EXE, 214
SYSMSG.EXE, 208
System communication services
  See SCSLOA.EXE
System control block, 218
System header, 219
System image
  See SYS.EXE
System message file
  See SYSMSG.EXE
System page table, 219
System service
  name, 166
  vector, 166 to 169

-T-

Terminal class driver
  See TTDRIVER.EXE
Translation buffer parity error
  VAX-11/780 and VAX-11/785, 104 to 105
TTDRIVER.EXE, 214

-U-

UNKRSTRT bugcheck, 51, 59, 136
UNXINTEXC bugcheck, 17
UNXSIGNAL bugcheck, 170
User mode data page, 192

-V-

VAX 8600
  console single step, 70
VAX-11/730
  console single step, 71
VAX-11/750
  %%, 66
  console single step, 70
VAX-11/780
  auto restart, 57
  console halt message, 47
  console single step, 70
  Cpu-specific interrupt, 17 to 20
  Halt, 46
  machine check, 101
VAX-11/782 support
  See MP.EXE
VAX-11/785
  auto restart, 57
  console halt message, 47
  console single step, 70
  Cpu-specific interrupt, 17 to 20
  Halt, 46
  machine check, 101
VAXEMUL.EXE, 212
Virtual address, 174
  P0 space, 176 to 177
  P1 Space, 178 to 196
  system space, 196 to 220

-X-

XQP
  see Files-11 XQP