OpenVMS Technical Journal V6

Fatal Bugchecks on OpenVMS Alpha and OpenVMS I64 Systems

Fatal Bugchecks on OpenVMS Alpha and OpenVMS I64 Systems

Ruth Goldenberg and Richard Bishop, OpenVMS Engineering

Overview

When OpenVMS code detects an internal inconsistency, such as a corrupted data structure or an unexpected exception, it generates a bugcheck. If the inconsistency is not severe enough to prevent continued system operation, the bugcheck generated is nonfatal and merely results in an error log entry.

If the error is serious enough to jeopardize system operation and data integrity, OpenVMS code generates a fatal bugcheck. This generally results in aborting normal system operation, recording the contents of memory to a dump file for later analysis, and rebooting the system.

This article describes how fatal bugchecks are handled on OpenVMS Alpha and OpenVMS I64, how the dump file is found, how memory is written to the dump file, and how the dump file is organized.

Initial Bugcheck Handling

On Alpha, a bugcheck is generated by placing a bugcheck value in R16 and executing a `CALL_PAL BUGCHK` instruction. On I64 a bugcheck is generated by placing a value in R17 and executing a `BREAK BREAK$C_SYS_BUGCHECK` instruction. This article refers to these simply as the `BUGCHECK` instruction. In both cases, the bugcheck value identifies the type and severity of the bugcheck and determines whether a crash should be followed by a cold or warm reboot. If the bugcheck is fatal, bit 2 of the value is 1; otherwise it is 0. If the bugcheck is fatal and a cold reboot is requested, bit 0 of the value is 1; otherwise, it is 0. Bit 1 is not used and is always 0.

Regardless of the type of bugcheck, executing the instruction causes an exception. The `BUGCHECK` instruction results in changing access mode to kernel mode and passing control to code that services the exception. In this article, this service routine and the routines it calls are referred to as BUGCHECK code.

OpenVMS BUGCHECK code is implemented in several parts. Initially, the exception is dispatched to platform-specific code. On I64 platforms, BUGCHECK code flushes all stacked registers to their memory backing store. On both platforms, BUGCHECK code ensures that interrupt priority level (IPL) is at least 3. This prevents rescheduling and resumption on a different CPU.

BUGCHECK code tests whether this is a recursive bugcheck, that is, whether it has already saved bugcheck context in a nonpaged pool per-CPU structure called the per-CPU database. If so, it refrains from overwriting the description of the original bugcheck. If this is not a recursive bugcheck, BUGCHECK code records information such as the following in the per-CPU database:

- Access mode stack pointers and, on I64 platforms, access mode register stack pointers
- Page table base registers
- Address space numbers
- AST summary and enable information
- Software interrupt summary information
- Process unique value, which identifies a user thread
- Processor status at bugcheck, including IPL and access mode
- Number identifying the specific bugcheck
- Process and system cycle counters

- Address of the stack structure describing the BUGCHECK exception: on Alpha, the exception stack frame, or, on I64, the INTSTK structure built by software interrupt services (SWIS) code

BUGCHECK code decides whether subsequent processing can occur in the current hardware context or whether it needs to switch contexts. If this is an outer mode bugcheck (user or supervisor), no switch is necessary. If this is an inner mode fatal bugcheck, BUGCHECK code switches to its own context, which has a larger kernel mode stack.

On Alpha platforms, BUGCHECK code saves on the current stack the integer registers not already saved in the exception frame. If the process is performing floating-point calculations, it saves all the floating-point registers on the stack. On certain Alpha platforms, it also triggers the on-chip logic analyzer to dump any program counter (PC) data and to stop logging more.

On I64 platforms, the general registers, branch registers, predicate registers, application registers, and minimal floating point registers (F6 through F11) have already been saved in the INTSTK structure. BUGCHECK code saves the rest of the low bank of floating point registers (F0 through F31) and, if the process is using the high bank, saves F32 through F127.

BUGCHECK code's subsequent actions vary, depending on the access mode in which the bugcheck occurred and the severity of the bugcheck. The following sections describe the BUGCHECK code actions.

Bugchecks from User and Supervisor Modes

The OpenVMS operating system itself generates few bugchecks from user or supervisor mode. It provides the mechanism for use by other software. When a bugcheck is generated from user or supervisor mode code running in a process with BUGCHK privilege, BUGCHECK code writes an error log message.

The SYSGEN parameter BUGCHECKFATAL has no effect on bugchecks generated from user or supervisor mode. Only bit 2 of the bugcheck value determines whether a given bugcheck is fatal. Fatal user and supervisor mode bugchecks affect only the current process.

If the bugcheck is fatal, BUGCHECK code returns to the access mode of the bugcheck and requests the Exit ($EXIT) system service. It specifies the value SS$_BUGCHECK as the final image status. What happens as a result of this service request depends on whether the process is executing a single image (without a command language interpreter, CLI, to establish a supervisor mode exit handler) or is an interactive or batch job.

- If the process is executing a single image, a fatal bugcheck from user or supervisor mode typically results in process deletion.

- If the process has a CLI, a fatal bugcheck generated from an interactive or batch job typically causes the currently executing image to exit and control to be passed to the CLI through its supervisor mode exit handler. The CLI prompts for the next command.

In either case, the only difference between fatal user and supervisor mode bugchecks is that user mode exit handlers are not called when a fatal bugcheck is generated from supervisor mode.

If the bugcheck is not fatal, BUGCHECK code restarts Alpha on-chip logic analyzer data collection (if it was active), restores saved registers, and dismisses the exception. Execution continues with the instruction following the `BUGCHECK` instruction.

Bugchecks from Executive and Kernel Modes

Various OpenVMS components generate bugchecks from executive and kernel modes. If an executive or kernel mode bugcheck value is not fatal and the SYSGEN parameters BUGCHECKFATAL and SYSTEM_CHECK are both zero, BUGCHECK code proceeds as it does for nonfatal bugchecks for the outer two access modes. It writes an error log message, restores the saved registers, and dismisses the exception, passing control back to the instruction following the `BUGCHECK` instruction.

Typically, execution continues with no further effects. However, the routine that detected the error and generated the bugcheck can take further action. One example of such a routine is the last chance handler for executive mode exceptions. It generates the nonfatal bugcheck SSRVEXCEPT (unexpected system service exception). On the presumption that process data structures are inconsistent, it then requests the $EXIT system service. Exiting from executive mode results in process deletion. Another example is the Record Management Services (RMS) routine that generates the nonfatal bugcheck RMSBUG. On the presumption that process RMS data structures are inconsistent, it deletes the process by requesting the Delete Process ($DELPRC) system service.

If BUGCHECKFATAL is 1 or SYSTEM_CHECK is nonzero, any executive or kernel mode bugcheck is treated as fatal, independent of bit 2 of the bugcheck value. By default, BUGCHECKFATAL and SYSTEM_CHECK are 0, which means that a nonfatal inner access mode bugcheck does not cause the system to crash. BUGCHECK performs fatal bugcheck processing under any of the following circumstances:

- BUGCHECKFATAL = 1
- SYSTEM_CHECK ≠ 0
- The bugcheck is fatal.

In the case of a fatal bugcheck, the most important function of BUGCHECK code is to record the contents of memory and of the error log buffers. BUGCHECK code does not use standard I/O mechanisms to write this data because they may be affected by the system inconsistency that triggered the fatal bugcheck. Instead, it performs I/O through the bootstrap system device driver, the one used during system initialization. After recording memory contents, BUGCHECK halts the system to prevent any further system operations in case they might lead to data corruption.

After the system reboots, during system initialization, the error log buffers are copied to nonpaged pool for processing by the ERRFMT process. The dump file can be examined with the System Dump Analyzer (SDA) to determine the cause of the crash.

Fatal Bugcheck Processing on a Uniprocessor System

In processing a fatal bugcheck, BUGCHECK code takes the following steps:

1. On an I64 system, BUGCHECK turns off SWIS tracing and disables virtual hash page table walking (a hardware mechanism to improve performance of handling translation buffer misses). On an Alpha system, BUGCHECK turns off performance monitoring. On both platform types, BUGCHECK raises IPL to 31 to disable all interrupts.

2. BUGCHECK tests whether there have been four or more recursive bugchecks. If so, it displays on the console an error message, information about the most recent exception, and the stack. It then

reboots or shuts the system down, depending on the value of SYSGEN parameter BUGREBOOT.

3. If all of the following conditions are met, BUGCHECK displays on the console the location of crash-related information and enters XDELTA:

   - There have been no recursive bugchecks.
   - BUGREBOOT is zero.
   - The bugcheck has not been triggered by shutting the system down or running OPCCRASH.
   - The system has been booted with XDELTA or the remote debugger.

   Entering XDELTA enables a human operator to examine the state of the operating system through XDELTA commands and request a crash dump at will.

4. Based on the value of the BUGREBOOT parameter and bit 0 of the bugcheck value, BUGCHECK selects one of the following halt action values: perform a cold bootstrap, perform a warm bootstrap, or remain halted. It stores the value in the hardware restart parameter block (HWRPB) per-CPU slot.

5. If this system is a Galaxy node, BUGCHECK notifies other sharing set members that this node is going down unless there have been three or more recursive bugchecks. This test allows for the possibility that attempting to notify the other members caused some of the bugchecks.

6. BUGCHECK shuts down any system communication services (SCS) circuits unless there have been three or more recursive bugchecks. This test allows for the possibility that attempting to shut down SCS circuits caused some of the bugchecks.

7. BUGCHECK shuts down all adapters and initializes the adapter that connects the system device circuits unless there have been three or more recursive bugchecks. This test allows for the possibility that attempting to shut down adapters caused some of the bugchecks.

8. BUGCHECK stores the value of the bugcheck code in the per-CPU database.

9. If there have been no recursive bugchecks, BUGCHECK begins to write information about the bugcheck to the console terminal: it announces that the system is crashing. If this is the second nested bugcheck, BUGCHECK writes a message saying that this is a recursive bugcheck and displays information about the most recent exception.

10. BUGCHECK validates the checksum of the boot control block, the data structure containing the locations of the error log and dump files. If the checksum is bad, no dump can be written.

11. If the checksum is good and the system disk is shadowed, BUGCHECK determines the unit number of the master disk in the shadow set. If bit 2 in DUMPSTYLE is set, BUGCHECK also selects the first valid dump device from the DUMP_DEV environment variable, as described later in this article.

12. BUGCHECK writes information about the bugcheck to the console terminal. This information can include the bugcheck message, addresses of loaded executive images, current process name, current image name, privileges of the current security persona, contents of registers, and contents of stacks relevant to the crash.

    How much information is written depends on the value of bit 1 in SYSGEN parameter DUMPSTYLE. The default value of 0 inhibits most output. The console output is written before the dump file and should not be interrupted by halting the processor from the console terminal. Such an interruption prevents the dump file from being written.

13. BUGCHECK writes the dump header, trap information blocks, the bugcheck error log entry, and the error log buffers to SYS$SPECIFIC:[SYSEXE]SYS$ERRLOG.DMP. For simplicity, the layout within SYS$ERRLOG.DMP matches the layout within the dump file. After writing the error log buffers, BUGCHECK rewrites the dump header to indicate that the error log buffers have been written.

    If the system disk is a member of a shadow set, BUGCHECK writes the same set of information to every member of the shadow set. This ensures that regardless of which member is master when the system reboots, system initialization code can process the error log buffers from the system disk before the complete shadow set is mounted and made consistent.

14. BUGCHECK determines whether a dump is to be written and, if so, what kind of dump, based on the following criteria:

    - If the SYSGEN parameter DUMPBUG is 0, no dump is written. Its default value is 1.
    - If neither SYS$SPECIFIC:[SYSEXE]SYSDUMP.DMP nor PAGEFILE.SYS exists on the system disk or a disk specified by the DUMP_DEV environmental parameter, no dump is written.
    - If, during system initialization, resources cannot be allocated for BUGCHECK's use, no dump can be written. BUGCHECK initialization code attempts to allocate system page table entries to map I/O requests, its memory stack, and, on an I64 platform, its register stack. It needs additional page table entries if the dump is to be compressed.
    - If this is an operator-requested shutdown generated through the system shutdown command procedure, no dump is written.
    - If bit 0 in DUMPSTYLE is set, memory is dumped selectively; otherwise, a physical memory dump (also known as a full dump) is written. The default value of this parameter specifies a selective dump.
    - If bit 3 in DUMPSTYLE is set, the memory contents in the dump will be compressed. BUGCHECK needs physical memory to serve as compression buffers. It borrows memory from the resident code granularity section after copying the pages' current contents to a temporary location in the dump file. If insufficient pages are available, an uncompressed dump is written. The default value of this parameter specifies a compressed dump.

15. If no dump is to be written, BUGCHECK continues with step 23.

16. If any type of dump is to be written, BUGCHECK switches, if necessary, to the disk containing the dump file. This could be needed if BUGCHECK had been writing to shadow set members' SYS$ERRLOG.DMP or if it is writing a "dump off system disk" (DOSD). It then writes the dump header, trap information blocks, bugcheck error log entry, and the error log buffers to the dump file. Then it rewrites the dump header with a status indicating that the dump contains the error log allocation buffers.

17. If the system disk is shadowed, BUGCHECK outputs a message to the console terminal indicating which member it is dumping to and whether that member is the master unit.

18. BUGCHECK outputs a message saying that it is starting to dump memory and specifying whether the dump is physical or selective and whether it is compressed.

19. If the dump is compressed, BUGCHECK zeroes the blocks in the dump that will contain the compression map.

20. If the dump is physical, BUGCHECK writes a map of memory to the dump file so that SDA can associate blocks of the file with physical pages. (Memory is likely to be discontiguous and may not start at page 0.) It then writes physical memory to the dump file, starting with the lowest page and, if requested, compressing as it goes. A later section in this article describes in detail the layout of a physical memory dump and the compression algorithm.

21. If the dump is selective, BUGCHECK writes selected virtual address spaces to the dump, compressing as it goes if compression was requested. Each selected virtual address space is represented by a logical memory block (LMB).

    Not all virtual addresses in the range spanned by an LMB are necessarily included in it. Because a virtual page not in memory cannot be written to the dump file, it represents a hole in the virtual address space. An LMB with holes in its address space contains a **hole table**, which lists the pages of address space not present in the dump.

    The general sequence in writing an LMB is as follows:

    a. BUGCHECK writes an LMB header in the next block of the dump.

    b. BUGCHECK scans the page tables that describe the address space to be dumped, looking for invalid pages that are not transition pages. It writes an entry in a hole table for each such sequence of pages found. It writes the hole table to the next block (or blocks) of the dump. It rewrites the LMB header and the dump header to reflect the presence of the hole table in the dump.

    c. BUGCHECK scans the hole table, filling in its allocated system page table entries with information from each valid or transition page table entry that it found. That is, it double-maps those pages so that it can write virtually noncontiguous pages in one I/O request.

    d. When BUGCHECK has written all the valid and transition pages in a particular LMB to the dump file, it rewrites the block containing the LMB header with correct information about the number of holes in the address space and the number of data blocks (valid and transition pages) in the LMB.

    e. It rewrites the dump header to increase the chances that the dump can be analyzed even if the dump is incomplete.

    If BUGCHECK reaches the end of a file sized for selective dumps before it reaches the end of the LMB list, it rewrites the descriptor of the current LMB with the hole count and actual number of data blocks written. It then rewrites the dump header, filling in status information such as the number of I/O errors encountered while writing the dump file, the number of process LMBs written, and so on.

    In writing a selective dump, BUGCHECK must defend against the possibility that whatever error led to the bugcheck might also have corrupted the data structures necessary to write virtual address space. BUGCHECK replaces the page fault and access violation exception service routines with its own routines to prevent recursive bugchecks if either of those errors occurs. It also performs consistency checks on certain key data structures. For example, it checks whether an address presumed to be that of a process header is syntactically correct; that is, the address must be within known address boundaries and at an integral number of process headers from the beginning of the address range.

22. If a dump was written, BUGCHECK rewrites the dump header so that information such as the error count and number of blocks of memory in the dump reflects what was actually written to the dump file.

23. If SYSGEN parameter BUGREBOOT is 0, BUGCHECK writes a message on the console terminal. As a last step, it halts the system. In response, the console subsystem gains control and acts on the halt action value stored in the per-CPU slot of the hardware restart parameter block (HWRPB) during step 4. Typically, BUGCHECK performs a warm bootstrap.

Fatal Bugcheck Processing on a Symmetric Multiprocessing (SMP) System

When one CPU member of an SMP system incurs a fatal bugcheck, all members crash; the executive takes the conservative approach that an inconsistency severe enough for operations on one CPU to cease is likely to be systemwide. All members of the active set participate in fatal bugcheck processing.

The CPU that first incurs a fatal bugcheck (known as the CRASH CPU) drives the crash. It informs the other active CPUs that a bugcheck sequence has been initiated and takes a number of steps to ensure that a consistent system state can be saved. In response, the other active CPUs crash with the fatal bugcheck CPUEXIT. The primary CPU performs most of the remaining fatal bugcheck processing.

The following figure shows the sequence of some of the steps in fatal bugcheck processing as they might occur concurrently on the CRASH CPU (which is not pictured as the primary processor), the primary processor, and the other active set members. Steps shown in different columns but on the same line do not necessarily execute at the same time on all CPUs. The numbers in the figure correspond to the steps described after the figure, not all of which are represented in the figure.

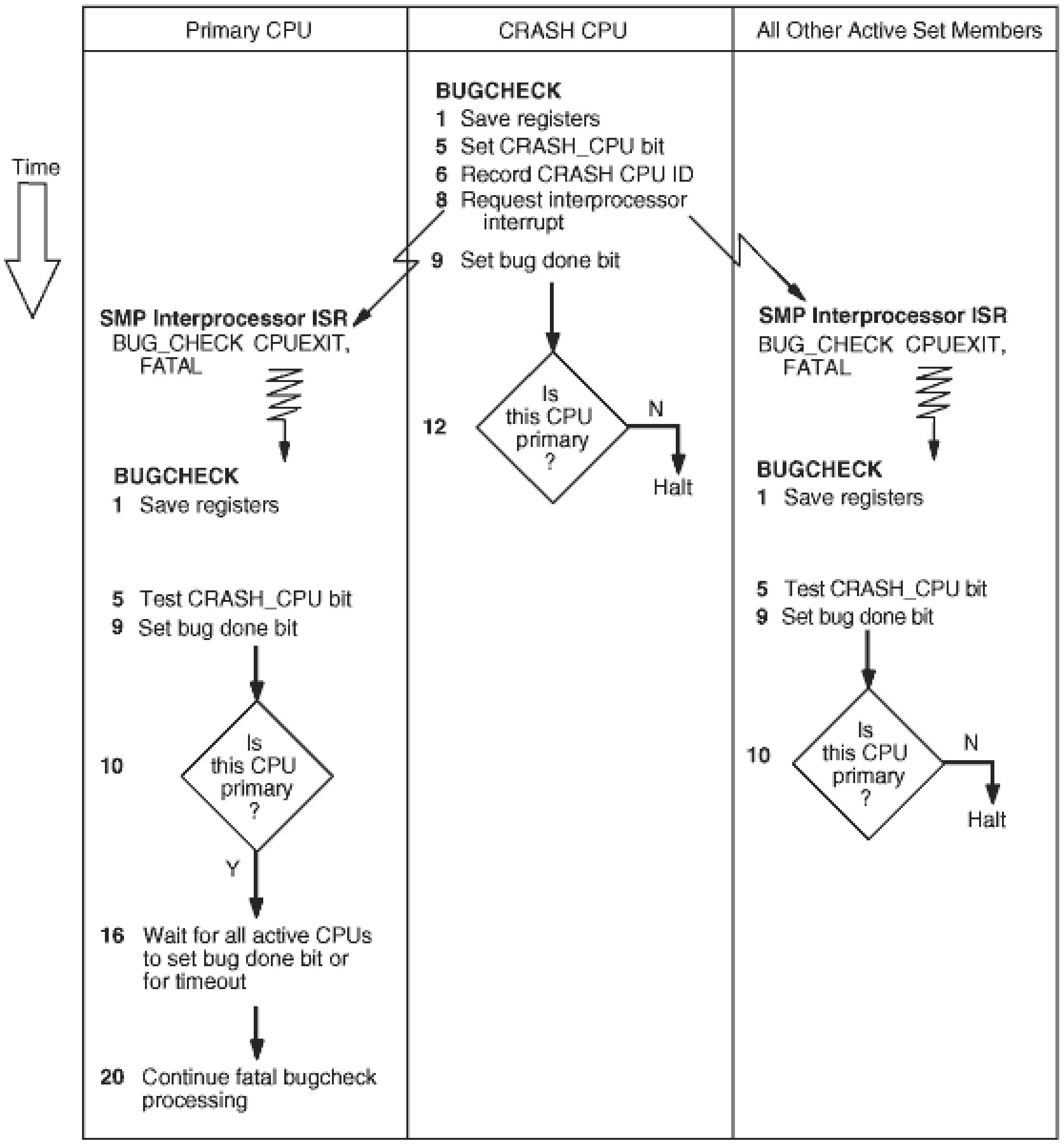

Figure 1 Steps in Fatal Bugcheck Processing on an SMP System

The following steps, which correlate to numbers in the figure, focus on SMP-specific processing and omit most steps common to uniprocessor processing.

1. BUGCHECK initially runs on the CRASH CPU and subsequently on the other SMP members. It records information in the per-CPU database.

2. The primary processor switches to BUGCHECK's hardware context; the secondary processors switch to the system hardware context.

3. BUGCHECK saves registers in the per-CPU database.

4. BUGCHECK tests whether it is running on a member of the active set. If not (a pathological and unlikely case), it proceeds to step 9 rather than taking any steps that might interfere with SMP operations.

5. If BUGCHECK is running on a member of the active set, it tests and sets the bit SMP$V_CRASH_CPU in SMP$GL_FLAGS. Only the first CPU to crash sets this bit and thus becomes the CRASH CPU.

   If the bit is already set, BUGCHECK continues with step 9. Use of the bit prevents confusion during concurrent independent crashes.

6. BUGCHECK records the ID of the CPU on which it is running in SMP$GL_BUGCHKCP as the CRASH CPU.

7. BUGCHECK acquires the CPU mutex to prevent any other processors from joining the active set.

8. BUGCHECK requests an interprocessor interrupt of every other member of the active set, specifying bugcheck as the work request type. The interprocessor interrupt service routine (ISR) generates the fatal bugcheck CPUEXIT.

9. BUGCHECK then sets its CPU ID bit in SMP$GL_BUG_DONE to indicate that it has saved its context.

10. BUGCHECK compares its CPU ID to that in SMP$GL_PRIMID to determine whether it is executing on the primary processor. If it is not, it halts. If it is in the failover set in a Galaxy system, it will be reassigned to another partition.

11. This and the following steps execute only on the primary processor because it is the only member guaranteed access to the console terminal.

    BUGCHECK sets its CPU ID in SMP$GL_OVERRIDE, adding itself to the override set. As a member of the override set, its spinlock acquisitions and releases are not subject to the usual checks.

12. BUGCHECK turns off the sanity timer to prevent sanity timer timeouts during the bugcheck.

13. On a Galaxy system, BUGCHECK reassigns any CPUs that were already stopped at the time of the bugcheck.

14. BUGCHECK waits up to a maximum of 10 seconds plus the value of SYSGEN parameter SMP_SPINWAIT (whose default value is 10 seconds) for all active members to save their context. Under normal circumstances, much of this wait does not occur. However, if one member restarts following a halt, it can take the member a significant time to complete that process and respond to the interprocessor interrupt requesting bugcheck processing. If 10 seconds passes before all members are done, BUGCHECK proceeds.

15. On a Galaxy system, BUGCHECK reassigns previously active secondary CPUs that have already saved their context.

16. BUGCHECK again waits, up to a maximum of 10 seconds plus SMP_SPINWAIT, for all active members to save their context. This wait ensures that the CRASH CPU has saved its context.

17. Still running on the primary CPU, BUGCHECK tests whether the CRASH CPU has saved its register context. If not, it uses its own per-CPU database in the following steps.

18. BUGCHECK uses the bugcheck code in the CRASH CPU's per-CPU database to select the bugcheck message text. This field is initialized to BUG$_CPUCEASED in case a problem on the CRASH CPU prevents it from recording the real bugcheck code.

19. BUGCHECK writes bugcheck information from the CRASH CPU's per-CPU database to the console terminal.

20. Running on the primary CPU, BUGCHECK then continues execution at step 13 in the earlier section titled “Fatal Bugcheck Processing on a Uniprocessor System”.

The System Dump File

System initialization code locates and opens the error log dump file SYS$SPECIFIC:[SYSEXE]SYS$ERRLOG.DMP and the system dump file SYS$SPECIFIC:[SYSEXE]SYSDUMP.DMP. If the error log dump file is not found, the error log buffers are not dumped when the system crashes or shuts down, and therefore cannot be recovered by system initialization code during the subsequent reboot.

Absence of a system dump file on the system disk does not necessarily mean a system dump cannot be written. If SYS$SPECIFIC:[SYSEXE]PAGEFILE.SYS exists, the executive writes the system dump there instead. Subsequent analysis of a dump written to the page file requires that the SYSGEN parameter SAVEDUMP be 1.

If the “dump off system disk” (DOSD) bit of the SYSGEN parameter DUMPSTYLE is set, BUGCHECK locates and opens SYSDUMP.DMP on an alternate disk. If such a dump file is found, it is used instead of a file on the system disk (SYSDUMP.DMP or PAGEFILE.SYS). If such a file is not found, the system disk dump file is used. If no dump file is found on either disk, then no dump is written. DOSD is discussed in more detail at the end of this article.

There are two types of dump:

- A **full** or **physical** dump is a dump of physical memory from the lowest numbered physical page to the highest.

- A **selective** dump is a dump of the virtual memory in use at the time of the system crash.

Both physical and selective dumps are divided into several sections, as shown in the following figure and descriptive text.

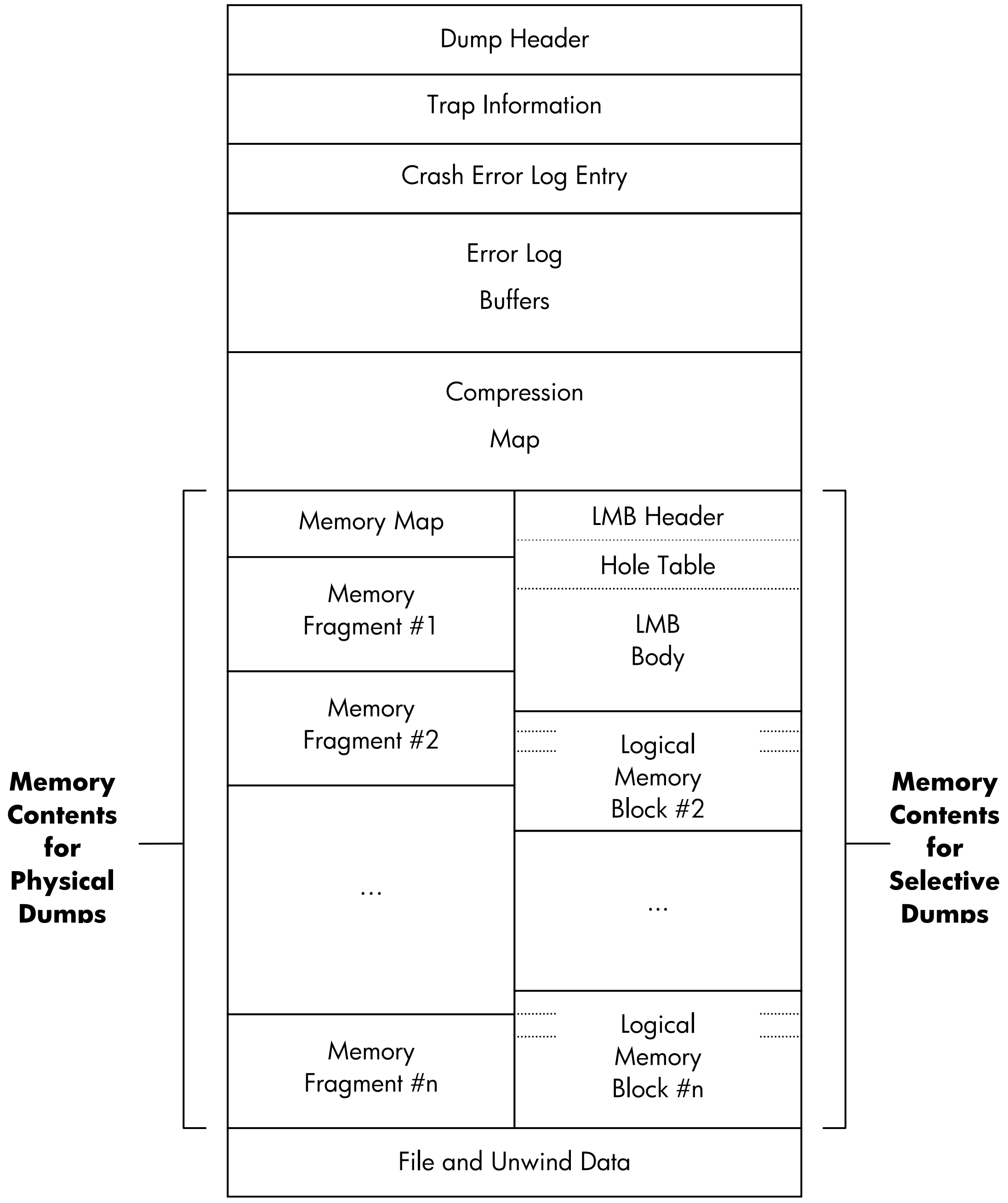

Figure 2 Layout of Physical and Selective Dumps

- Dump Header. This consists of two blocks of information describing the dump, including data such as the date and time of the system crash, the bugcheck type, the size of the dump, and so on.

- Trap Information. In case the BUGCHECK code itself incurs a trap such as an ACCVIO while it is writing the dump, blocks are set aside for the data from the trap to be saved. The saved data consists of the exception frame plus some additional fields. Currently, one block is required on Alpha, and two blocks on I64.

- Crash Error Log Entry. This is the error log entry that describes the system crash. Two blocks are used on both Alpha and I64.

- Error Log Buffers. These are the error log entries that were recorded during the period immediately preceding the system crash and have not yet been written to the error log file ERRLOG.SYS. The number of blocks required for these buffers is calculated by multiplying the SYSGEN parameters ERRORLOGBUFF_S2 and ERLBUFFERPAG_S2. (When the system reboots after a crash, SYSINIT copies the Error Log Buffers and the Crash Error Log Entry into nonpaged pool. Later, the ERRFMT process records them in ERRLOG.SYS).

- Compression Map. If a compressed dump is being written, space is allocated in the dump for the compression map. The format of this map is discussed later in this article. The BUGCHECK code calculates the worst-case size for the map – the number of blocks that would be needed if no compression were possible and if the dump file were completely filled.

- Memory Contents. The actual contents of memory is dumped next. The layout of this section depends on the type of dump (physical or selective) being written. The two layouts are described in more detail later in this article.

- File and Unwind Data. Starting in the OpenVMS release that follows Version 8.2, additional data is appended to the dump after the system reboots. This data is appended when the dump is copied by SDA during system startup. The appended data comprises:

  - Data to translate file identification numbers (FIDs) to file names
  - Unwind tables for images activated by any processes (OpenVMS I64 only)

  The file and unwind data section is described in more detail later in this article.

Physical Dumps

In a physical dump, all the memory in the system is written to the dump file, with two exceptions:

- If the SYSGEN parameter PHYSICAL_MEMORY has been set to less than the size of the system's memory, any ranges of pages marked "Available" are not dumped.

- In the case of a system crash following a MACHINECHECK due to a double-bit error in memory, the page with the error is not dumped.

In a physical dump, memory is handled in fragments. A **fragment** is a contiguous range of pages with common attributes (for example, console-owned, OpenVMS-owned, or Galaxy shared memory). In an OpenVMS Galaxy system, usually the only fragments dumped are those owned by the member that has crashed, plus shared memory. The one exception to this is the page range containing the Galaxy Configuration Tree (GCT). The GCT, which needs to be included in the dump, is normally located in the local memory of member zero. If a system crash occurs in another member, an additional fragment must be created by BUGCHECK code to describe the page range of the GCT.

The memory portion of a physical dump begins with the **memory map**. The memory map describes the memory fragments (physically contiguous ranges of pages) in the system. Up to 16 memory fragments are described by each block of the memory map. The remainder of the dump has the following contents, in this order:

- A memory fragment for the GCT (if BUGCHECK code created such a fragment)

- All console-owned pages
- All OpenVMS-owned pages
- Any Galaxy shared memory pages

Console-owned pages are written before OpenVMS-owned pages to ensure that the level 1 page table page created by the console is written early. On many Alpha platforms, this page table is in a high-numbered page, and if pages were written in ascending order, the page would often fail to be written if the dump file was even slightly too small.

An **uncompressed** or **raw** physical dump requires a dump file large enough for all of memory, plus space for the headers, error log buffers, memory map, and so on. If the dump file is too small, it is quite likely that no analysis will be possible. A **compressed** physical dump requires a dump file about half this size, but the exact size depends on the contents of memory. The memory map is not compressed. Dump compression is described later in this article.

Selective Dumps

In a selective dump, only memory in use at the time of the system crash is written to the dump file, and it is ordered by its virtual memory address. System memory is dumped first, followed by the memory of processes and global sections, in units called **Logical Memory Blocks** (LMBs). An LMB consists of a 1-block header, followed by a hole table (one or more blocks in length), followed by the actual memory contents. The hole table describes the ranges of virtual addresses not written for the section within the LMB.

The complete set of possible LMB types, and their order in a selective dump, is as follows:

1. PT Space (the page table for all of system space (S0/S1 and S2)).

2. S0/S1 space (32-bit system space)

3. S2 space (64-bit system space)

4. RAD-specific data (virtual pages in S0/S1 and S2 space that map to different physical pages in systems that support Resource Affinity Domains). There can be multiple RAD-specific LMBs in a dump.

5. Key Process page tables (one for each key process). An explanation of key processes follows later in this article.

6. Key Process memory (one for each key process). For each process, the page table LMB and memory LMB are written together as a pair.

7. Key Global pages (any global pages in the working set of at least one key process at the time of the system crash).

8. Non-key Process page tables (one for each non-key process)

9. Non-key Process memory (one for each non-key process)

10. Remaining Global pages (any global pages in the working set of a process at the time of the system crash that were not included in the “Key Global pages” LMB).

In contrast to physical dumps, it is impossible to determine exactly the size needed for an uncompressed or raw selective dump. For analysis to be possible, the dump file must be large enough to contain all system space plus the current process on the crash CPU. Best results are obtained if the dump file is large enough to accommodate all key processes and key global pages, as discussed in the next section. A compressed selective dump requires a dump file about half this size, but, as with physical dumps, the exact size depends on the contents of memory. The LMB headers and hole table blocks are not compressed. Dump compression is described later in this article.

Incidentally, **process dumps** use a format similar to selective dumps. Process dumps contain two LMBs – one for all process address space (both page tables and memory), and one for all system space memory that is pertinent to the process being dumped. Process dumps do not contain error logs or space reserved for trap information. Instead, process dumps include blocks following the dump header that contain an invocation context block and other data. This information provides a snapshot of the processor registers at the time of the failure.

Key Processes

When a dump is analyzed, the necessary information is usually found in the address space of the process that was current at the time of the system crash, or in the address space of a process that contains important system data structures. For this reason, certain processes, known as **key** processes, are dumped first in a dump. These processes, and the order in which they are dumped, are as follows:

- The current process on the CPU where the system crash was initiated
- The SWAPPER process
- Current processes on any CPUs that did not respond to the request from the crash CPU to perform a `CPUEXIT BUGCHECK` instruction
- Current processes on any other CPUs
- Processes registered as priority processes by the SYSMAN DUMP_PRIORITY ADD command
- Processes designated by Hewlett-Packard to be key processes. This list of processes is built into the BUGCHECK code and comprises the following:
  MSCPmount
  AUDIT_SERVER
  NETACP
  NET$ACP
  REMACP
  LES$ACP
  SHADOW_SERVER
- Processes in any miscellaneous wait state: for example, RWAST, MUTEX, WTBYT, and so on.

SYSMAN DUMP_PRIORITY ADD commands can be entered directly by the system manager or can be included in the installation procedures of layered products and third-party products. Careful use of SYSMAN DUMP_PRIORITY ADD commands to include all key processes ensures that a selective dump includes useful data first. This prevents the dump file being filled with the memory contents of processes that have no relevance to the system crash.

Dump Compression

Dump compression was introduced in OpenVMS Alpha Version 7.0 and has been enhanced since then, most recently in Version 7.3-1. It does not use an established compression algorithm, but one designed to be most effective with common OpenVMS memory patterns, and which can be used by the BUGCHECK code.

Following the dump header and error logs, a compressed dump contains a compression map. This is a sequence of quadwords, each describing up to 128 blocks of the compressed dump. (In a selective dump, additional quadwords describe the LMB headers and the hole tables for each LMB. These quadwords can easily be identified because bit 63, DMP$V_NOCOMP, is set).

The data in each quadword includes the final raw VBN represented by this compression map entry, the number of compressed blocks written for this entry, and flags such as DMP$V_NOCOMP. When SDA is reading the dump file, it scans the compression map looking for the entry containing the raw VBN it wants. Having counted the compressed blocks written for preceding entries, SDA can then read the exact set of blocks containing the raw VBN of interest and decompress just those blocks to get to the raw data.

The BUGCHECK code attempts to compress four blocks of raw data at a time using the algorithms described below. The compressed data is appended to a buffer until the maximum size of a compressed section (127 blocks) is reached.

The BUGCHECK code looks for the following byte patterns:

- Repeated sequences of a single value (DMP$K_REPEAT)
- An ascending sequence of values (DMP$K_INCREMENT)
- A descending sequence of values (DMP$K_DECREMENT)
- A sequence containing a limited set of values (up to 2, 4, 8, 16, 32, or 64) (DMP$K_REENCODE_n, n=1-6)
- A sequence where a small number of values (1-7) is dominant (DMP$K_BITMAP_n, n=1,3,7)
- A sequence where no compression is possible (DMP$K_NOCOMP)

For each byte pattern, the compressed data has a fixed header of three bytes. The first byte contains the pattern type (DMP$K_xxx) and some flag bits (to be described later). The second and third bytes contain a word count of the number of raw bytes represented in the compressed section.

For DMPK_REPEAT, DMPK_INCREMENT, and DMP$K_DECREMENT, a fourth byte contains the first character of the sequence. A minimum sequence of 16 bytes is required for the pattern to be recognized. If successive complete DMP$K_REPEAT sections are found for the same byte value, these are merged to a maximum raw size of 64 blocks.

For DMP$K_NOCOMP, the sequence of raw byte values follows the header. If successive complete DMP$K_NOCOMP sections are found, these are merged, also to a maximum raw size of 64 blocks. If two successive 64-block DMP$K_NOCOMP sections occur, the 3-byte headers are dropped, the entire 128 blocks are written to the dump file, and the DMP$V_UNCOMP bit is set in the compression map entry. Without this optimization, the compressed dump would end up being larger than the raw dump if memory patterns are such that little or no compression is possible.

DMP$K_REENCODE_n sections contain the same 3-byte header, followed by bytes containing the values that occur in the sequence (from 2**(n-1)+1 to 2**n such values). So that SDA can determine the end of the set of values, if the full set (2**n) is not used, the last value is always hex FF, even if it does not occur in the sequence. To understand how re-encode compression works, consider a sequence of bytes that contains only the hex values 17, 31, 65, A4, and E9. This can be compressed as a DMP$K_REENCODE_3 section, using binary 000 for every 17, binary 001 for every 31, 010 for 65, 011 for A4, and 100 for E9. The five raw values, followed by hex FF, are written to the dump, followed by as many 3-bit groups as there were bytes in the raw data. If a sequence of 100 bytes containing just these five values was found, its compressed size would be 3+6+(100*3+7)/8 =

47 bytes. (That's three bytes of fixed header, six bytes of raw values including the hex FF terminator, and 100 3-bit groups for the actual re-encoded values, rounded up to the next complete byte.)

DMP$K_BITMAP_n sections contain the 3-byte header, followed by bytes containing the dominant values that occur in the sequence (exactly n such values). There is no need to have a filler value, as do DMP$K_REENCODE_n sections, because any sequence of bytes that contains fewer than seven distinct values is more efficiently compressed using re-encoding. Following the set of dominant values is a set of 1-bit, 2-bit, or 3-bit groups, with one group for each byte in the raw data. The value in each group is the offset in the set of dominant bytes unless it is the maximum value for the bit group (1, 3, or 7). In that case, the value indicates that some other less common (latent) value occurs in the raw data. These latent values follow the bit groups in the same order they occur in the raw data. The number of bytes required to contain the compressed data using bitmap compression depends on the repeat counts for each dominant value.

The BUGCHECK code looks first for repeat, increment, or decrement sequences, temporarily ignoring intervening bytes. Having identified such a sequence, it then scans the ignored bytes, counting occurrences of all possible values. Using the collected statistics, it decides if a re-encoded compression is possible or if a bitmap compression would produce a better result. If no approach will compress the data, it becomes a DMP$K_NOCOMP section. The result is written to the dump, followed by any previously-identified repeat, increment, or decrement section.

But compression does not end here. In many situations, the compressed data is itself compressible, so the BUGCHECK code attempts further compression, up to four times. While it is rare to perform the maximum number of compression passes, two or three passes are common, especially with certain patterns that occur in page tables and the PFN database. Flag bits in the first byte of the 3-byte header indicate to SDA when it has completely decompressed a section.

File and Unwind Data

Beginning with the OpenVMS release that follows Version 8.2, additional data is appended to the dump after the system reboots. However, the actual dump file cannot be modified. For example, when a dump is written to a page file, any unused space in the dump file is made available for paging as soon as the system reboots. Therefore, the additional data is appended to the copy when the dump is copied by SDA. The appended data comprises the following:

1. File identification to file name translation data. Collection of this data allows SDA to display file names when analyzing dumps for the following commands:
   ```
   SHOW PROCESS /CHANNELS
   SHOW GLOBAL_SECTION_TABLE
   SHOW MEMORY /FILES
   ```

2. Unwind tables for images activated by any processes. OpenVMS I64 uses unwind tables to describe when and where registers have been saved during the execution of a procedure. The unwind tables for an activated image are in pageable sections. As a result, the tables and the data structures that describe them have often been unavailable in system dumps. Their absence affects the SHOW CALL command, preventing a complete analysis of all call frames if any frame is for a PC in an activated image. Collection of this data allows SDA to access the data it needs to display such call frames.

   The SHOW UNWIND *address* command also uses the collected data if the given address is for a PC in an activated image. However, the SHOW PROCESS /UNWIND [=ALL] command can still fail because it also relies on the data structures that describe the in-memory unwind tables; those cannot be reconstructed after the crash.

It is worth noting that because the unwind tables for loaded executive images are always resident, the SHOW UNWIND [/ALL] command is never affected. Likewise, SHOW CALL can always display call frames whose PC is in a loaded executive image.

By default, a copy of an original dump, as written at the time of the system crash, includes collection data. You can execute COPY /NOCOLLECT to override this. Conversely, a copy of a dump previously copied by SDA without collection (COPY /NOCOLLECT) does not include collection data, but COPY /COLLECT can be used to override this. Copying a dump that already contains an appended collection will always include that collection.

For all file and unwind data to be collected successfully, all disks that were mounted at the time of the system crash should be remounted and accessible to the process running SDA. SDA is usually invoked early during startup to save the contents of the dump, for example by defining the logical name CLUE$SITE_PROC to point to an SDA command procedure that includes a COPY command. If some disks are not mounted until a batch job is run, some file or unwind data might not be collected successfully. To avoid this, use the COPY/NOCOLLECT command in the CLUE$SITE_PROC command procedure; then, once all disks have been mounted, execute an additional COPY/COLLECT command to save file and/or unwind data.

Dump Off System Disk (DOSD)

On many systems, disk space can be at a premium, especially on a cluster-common system disk. For this reason, OpenVMS on both Alpha and I64 provides the ability to write system dumps to a dump file on a disk other than the system disk. To use this feature, the system manager must perform the following setup steps:

1. Set the "dump off system disk" (DOSD) bit in the SYSGEN parameter DUMPSTYLE. This is bit 2 (value 4).

2. Put the name of the disk to be used in the DUMP_DEV environment variable. On Alpha, this is done with a SET DUMP_DEV command at the console prompt. On I64, it is done using the command procedure SYS$MANAGER:BOOT_OPTIONS.COM. If there are multiple paths to the dump disk, all paths should be included in DUMP_DEV. The disk cannot be a member of a shadow set or part of a volume set.

3. On the desired dump disk, create the file [SYSn.SYSEXE]SYSDUMP.DMP using the SYSGEN CREATE /SIZE=size command. The file must be in the same system root that is used to boot the system. Using SYSGEN to create the dump file ensures that, even if the file is not contiguous, it will not be so badly fragmented that an extension header is required to record its location on the disk.

When the system crashes, if the BUGCHECK code finds the DOSD bit set in DUMPSTYLE, it attempts to access each entry in DUMP_DEV in turn and tries to locate the dump file, using the same primitive file system that is available to SYSBOOT when the system is booted. Once the BUGCHECK code finds the dump file, it writes the dump in exactly the same way as it does for a system disk dump file.

The error log dump file is always located on the system disk because SYSINIT must access it when the system reboots. The BUGCHECK code uses DUMP_DEV to locate alternate paths to the system disk if it failed over to a different path while OpenVMS was running. For this reason, if there is more than one path to the system disk, DUMP_DEV should include all paths. If the system disk is a shadow set, the BUGCHECK code attempts to write the contents of the error log buffers to all members of the shadow set. Therefore, DUMP_DEV should include all paths to all members of the system disk shadow set.

Unfortunately, on Alpha there is a limit to the number of paths that can be included in DUMP_DEV because the console provides only a single fixed-length buffer for the list. On some Alpha platforms, only a single device can be specified; on others, there is room for 4 to10 devices, depending on their type. As a result, the system manager must determine which devices to include. At a minimum, all paths to the dump device should be included if DOSD is being used. As many paths to the system disk as possible should then be added; in the case of a shadow set, start with the member that is usually the master – the member used in BOOTDEF_DEV.

On I64, the limit on the number of paths is 99, so there should be no difficulty in defining all paths to the dump disk and all paths to all members of the system disk shadow set.

For more information

For more information about the creation and analysis of system crash dumps, refer to the following OpenVMS manuals:

HP OpenVMS System Analysis Tools Manual
http://h71000.www7.hp.com/doc/82FINAL/6549/6549PRO.HTML

HP OpenVMS System Manager's Manual, Volume 2: Tuning, Monitoring, and Complex Systems
Refer to the chapter titled "Managing Page, Swap, and Dump Files."
http://h71000.www7.hp.com/doc/82FINAL/aa-pv5nj-tk/aa-pv5nj-tk.HTMl