eVAX 1.1

A Software Emulator
Of a VAX Computer

User's Guide

Tom Cole
tom.cole@forest-edge.net

Draft 1.6
Monday, December 6, 1999

Table of Contents

Introduction to eVAX

This document describes "eVAX", a software emulation of a VAX processor. The emulator supports a generic VAX processor, with extensions for loading VAX/VMS images and executing them directly on the emulator. Runtime support is provided by a mixture of native VAX routines and a runtime library "shim" model in the emulator itself.

The emulator is written entirely in C, and is portable across Macintosh, Windows, UNIX, and VMS platforms. The emulator has been built using Metrowerks Codewarrior on Mac OS and Windows 98, using Microsoft Visual C on Windows NT, and using the GNU compiler tools on Linux, MachTen, and HP-UX.

The emulator is "open source", which means that the source code may be freely distributed, copied, or modified. The only licensing limitations are that any product derived from eVAX must itself be an open source product, and the product must retain a publicly visible statement similar to "Portions of this product were developed by Forest Edge Software.

The source archive is updated frequently (at least once a month) and can be found at the web location http://www.forest-edge.net/evax.html.

Basic Features

The eVAX emulator does not currently emulate a specific model of VAX. Rather, it attempts to meet the requirements for a VAX implementation as set forth in the "VAX Architecture Reference Manual", published by Digital Equipment Corporation (now part of Compaq).

The emulator supports most of the VAX instruction set (currently, notable exceptions include packed decimal, D_FLOAT, G_FLOAT and H_FLOAT instructions). Support for additional instructions continues at a slow pace; the ultimate goal is to support all instructions except G and H floating point.

The emulator fully supports virtual memory, though of course no paging mechanism is built in, since that must be supplied by the operating system booted on a VAX system. Additionally, the emulator supports all required VAX privileged registers, console terminal I/O, and the complete exception and interrupt model.

A built-in "console" provides support for controlling the configuration of the virtual VAX and for functions normally handled by console ROM or microcode in a real VAX. This includes sizing the amount of physical memory for the virtual VAX, establishing processor IPL, mode, etc. as well as setting privileged register and general-purpose register values. Page table entries can also be viewed or set from the console when virtual memory is enabled.

Additionally, eVAX supports a subset of a VAX assembler built in to the "console." This allows the user to easily create native VAX programs, or modify programs in memory. The assembler supports all instruction addressing modes and includes a symbol table manager and a set of pseudo-opcodes for creating procedure entry points, interrupt handlers, etc. Matched with the assembler is a minimal set of debugging tools for setting break points, single stepping code, disassembling memory, etc.

The console supports booting a VMB.EXE file (located in the host file system on which the emulator runs) and starting it, which is a first step to being able to bootstrap VMS on eVAX. This currently does not work, however, because the I/O model is unimplemented (see below).

Finally, the console has the ability to decode and load a VAX/VMS image into the eVAX memory and execute it. This allows you to create a program on a VAX system (using Macro or C for example) and run it using eVAX. Currently, shared library references in the program you run can be resolved to native VAX code you supply, or in some cases, bound to a set of run time support modules contained in the emulator itself. Ultimately, the emulator will contain common parts of LIBRTL, as well as mapping commonly used DECC$SHR routines to the native C runtime library support. The eVAX loader decodes sharable image fixups, and provides a symbolic mapping mechanism for redirecting these to runtime functions (called software “shims.”)

Initialization

The eVAX emulator is run as a conventional application in whatever environment you are running. For example, on a Mac or Windows system, double click the application. A “console” window is created representing the virtual VAX console. On a Unix or VMS system, console I/O is done to the current processes’s controlling terminal.

The eVAX system executes the commands in the file “vax.init” which must be located in the same directory as the emulator itself. This file can be edited freely. As distributed, it creates a virtual memory map and loads the default microkernel, located in the file “kernel.asm”.

The microkernel is initialized by executing starting at the label EXE$INITIALIZE. This prepares the VAX for user-mode execution, establishes required privileged registers, initializes the system control block, etc.

After the kernel initializes, it prints a message to the console. This indicates that the kernel is functioning correctly, including interrupts, processor mode, IPL, etc. The console then presents it’s default prompt “VAX>” and awaits a command.

You can enter commands to assemble code to execute, to query or modify the state of the VAX, to begin execution at arbitrary addresses, or perform debugging and diagnostic functions. You can also load a boot image file (VMB.EXE in VAX/VMS) or run a user-mode executable program.

The remainder of this document will describe the commands supported by the console and their function.

Console Commands

The following section describes each of the console commands. The console commands may be stored in text files and executed (as in the case with the initialization file “vax.init”), or may be typed in at the command prompt.

When a command completes, if there was an error, the symbol $STATUS is changed to contain the numeric value of the error. You can display this value and the error message associated with it by using the SHOW ERROR command. Note that if the command executes successfully, the $STATUS variable is not updated. However, the CLEAR ERROR command can be used to re-initialize the error value to 0, which means success.

If an error message has a severity of INFO or WARNING then commands continue to be read from command files if a command file is being used. Otherwise, execution of commands stops with the first ERROR or FATAL error severity.

You can get information about the syntax of most commands by using the HELP command, which is documented in this section. The HELP command can be executed with no parameters for a generalized information display, or can be followed by a specific command for detailed information on that command. Note that as of version 1.1, not all commands are documented.

Also, note that while the ASM command is part of the command set, it is documented separately in a different chapter on the micro-assembler.

In the following examples, commands are nearly always spelled out completely. However, please note that commands and keywords are only evaluated up to the first four characters. Therefore, the command CLEAR can be abbreviated CLEA without error.

Also, note that in the following examples, except where specifically noted, all numeric constants are hexadecimal numbers. You can change the default base of constants parsed by commands using the SET RADIX command. You can also specify an explicit radix by using radix prefix notation:

```
^X0200
^D512
0x03FE
```

These are examples of explicit radix prefix. The ^X prefix indicates a hexadecimal number, and the ^D prefix indicates a decimal number. In these examples, both constants refer to the same numerical value. Please also note that hexadecimal constants must start with a digit, so the value 0x0FF must be expressed with at least one leading zero, as in 0FF, even if the default radix is hexadecimal. Constants that start with a letter, “_”, or “$” characters are assumed to by symbols, and are resolved against the user or system symbol tables. The third example shown is present for compatibility with GNU assembler syntax for hexadecimal constants.

In most places where a numeric value can be specified, you can specify an expression. Expressions can consist of numeric constants (with an explicit radix or in the current default radix), register contents, symbolic values, built-in functions, and integer arithmetic operators. The operators supported are shown below in their order of precedence, with the first ones being evaluated first.

Operators	Description
`( )`	Controls the order of evaluation, expressions in parenthesis are evaluated regardless of operator precedence. 3+4*5 results in the decimal value 23, but (3+4)*5 results in the decimal value 35.
`<, <=, >, >=, =, <>`	Integer relational comparisons. All values are promoted to signed longword values before the comparison is done. The result is an integer value of zero if the comparison is false, or 1 if the comparison is true. Note that <> means “not equal”.
`*, /`	Multiplication and division. Divisions are done as integers and the fractional result, if any, is discarded. Division by zero results in a zero, and overflow is ignored.
`+, -`	Addition and subtraction. Overflow and underflow are ignored.
`DEFINED( “name”)`	This function determines if a symbol is defined or not. If the symbol is defined, the expression returns a 1, if it is not defined it returns a zero. The actual value of the symbol is not relevant.
`BYTE( addr ),` `UBYTE( addr )`	Returns the 8-bit byte (signed or unsigned) stored in virtual memory at the location defined by the expression “addr”.
`LONG( addr ),` `ULONG( addr )`	Returns the 32-bit longword (signed or unsigned) stored in virtual memory at the location defined by the expression “addr”.
`WORD( addr ),` `UWORD( addr )`	Returns the 16-bit word (signed or unsigned) stored in virtual memory at the location defined by the expression “addr”.
`PMEMSIZE()`	Returns the number of bytes in physical memory of the current virtual VAX machine.
`FILE_EXISTS( “name”)`	Returns a 1 if the file specified in the quoted string exists on the native file system, or zero if it does not.
`MKVALID()`	Returns a 1 if the microkernel is present, or zero if it is not present. Essentially this indicates if the default microkernel as been loaded without a subsequent boot of a “real” operating system.
`VMVALID()`	Returns a 1 if the console virtual memory map is present, or 0 if it is not. Essentially this reports if a VMINIT command has been executed without subsequently booting a “real” operating system.

You can include the name of any general-purpose or privileged register in an expression. The contents of the register are used in the expression. For example, if R0 contains the value 0200, then the expression (R0+5) evaluates to 0205. Note that you may only use privileged registers that can be read; if a register is marked as write-only then it cannot be used in an expression without an error. You do not have to be in kernel mode, however, to use the value of processor registers in expressions. You can use the contents of a register as an address in the EXAMINE command (documented in a later section) by prefixing the register name with an "@" character. For example,

```
EXAM R0        ; Display contents of register 0
EXAM @R0       ; Display memory pointed to by contents of register 0
```

CALL

The CALL command is used to start execution of a standard VAX procedure stored in the VAX memory. This is distinct from the GO command which simply begins execution at a given memory location.

The CALL command constructs a special call frame on the stack, nearly identical to the call frames generated by the CALLS VAX instruction. The important difference is that the saved PC and FP contain the reserved value 0xFFFFDEAF. This address (not used by any current version of VMS) is special in that both the FP and PC contain the same value (impossible in a normal runtime environment). This special value is known to the RET instruction, and allows the system to know when to stop execution of VAX instructions and return control to the console command handler.

The CALL instruction allows you to specify up to 15 arguments. These are all expressed as 32-bit integer values. They are pushed on the stack as part of the construction of the root call frame.

When the procedure being executed returns, the value of R0 (the normal return value from a procedure) is stored in the system variable $RC.

```
CALL LIB$PUT_OUTPUT( 7 )
```

In this example, a routine LIB$PUT_OUTPUT is called with a single parameter, the integer value of 7. The address of the routine here is expressed as a symbolic value, which requires that a routine of that name have been defined or assembled. You can also simply express the address you wish to call:

```
CALL 200( “Test” )
```

This example shows using a numeric address (200 in this case) to define the address of the entry mask for the procedure to be called. This example also shows a special capability of the CALL statement to pass a string descriptor. In this case, the contents of the descriptor are the byte sequence 0x54, 0x65, 0x73, 0x74. A descriptor is created in a special “scratch pad” area of memory, and the parameter is actually the address of the descriptor. See the SHOW STRING command for more information on where string descriptors are stored.

```
CALL/STEP MAIN( 200 )
```

In this example, the CALL command has the qualifier /STEP which indicates that the user wishes to single step into the called program. In this case, the first instruction of the procedure is executed, and then the program halts awaiting further console interaction, just like the STEP command. You can STEP to advance additional instruction(s) or GO to execute the code normally. The main difference between CALL/STEP and the STEP command is that the CALL/STEP command will construct the call frame to the routine and establish an execution context for the procedure, while STEP simply begins executing the next instruction.

CLEAR

The CLEAR command is used to reset the state of the virtual VAX or the console environment. The CLEAR command takes at least one argument, which is the sub-function that defines what is to be cleared. See the additional information for the following commands:

- CLEAR BREAKPOINT
- CLEAR INTERRUPTS
- CLEAR MEMORY
- CLEAR PROFILE
- CLEAR STRINGS
- CLEAR SYMBOLS
- CLEAR TB

CLEAR BREAKPOINT

The CLEAR BREAKPOINT command is used to clear breakpoints. A breakpoint is information stored in the console defining locations or events that will cause the emulation of VAX instructions to stop. A breakpoint can be a particular memory address or a fault or interrupt condition. See the SET BREAKPOINT command for information on establishing a breakpoint, and SHOW BREAKPOINT for information on displaying stored breakpoints.

A breakpoint on a memory location means that when the VAX instruction at that location is to be executed, the processor will first halt and display a message on the console. At that point, the user can examine the state of the machine, memory, etc. before resuming. A STEP or GO command is used to resume execution at a breakpoint. Note that a breakpoint is *not* implemented using the BPT instruction, but is a function of the emulation itself.

```
CLEAR BREAKPOINT 400
```

This command clears the stored breakpoint for location 400. If there is no stored breakpoint for memory address 400 then an error is displayed.

```
CLEAR BREAKPOINT/FAULT 20
```

This clears the breakpoint for fault code 20 (access violation). The /FAULT qualifier is used to distinguish between breakpoints for memory locations versus breakpoints for faults. A breakpoint on a fault will cause the console to regain control as soon as the fault is incurred, before the fault handler from the SCB is invoked.

```
CLEAR BREAKPOINTS/ALL
```

This command clears all the breakpoints stored in the console (both execution and fault breakpoints). If there are no breakpoints, no error is reported.

CLEAR INTERRUPT

The CLEAR INTERRUPT command is used to clear pending interrupt(s) from the VAX processor. When interrupts are generated by the virtual VAX, they are put in a queue. This queue can be viewed using the SHOW INTERRUPTS command. Because interrupts are usually generated synchronously by the serial nature of the emulation, interrupts are “aged” so that some number of instructions are executed before the interrupts are delivered. The CLEAR INTERRUPT command is used to delete entries from this queue.

```
CLEAR INTERRUPT 0FC
```

This instruction searches the interrupt queue for interrupts with code 0xFC, which is the Console Terminal Transmit interrupt. Any entries in the queue with this code are removed. The command reports the number of interrupt entries removed, which may be zero if there were no pending interrupts. Note that interrupts are identified by their code number, which is 4 times their offset in the SCB array of longwords.

```
CLEAR INTERRUPT/ALL
```

This clears all interrupts from the queue, regardless of IPL or age. The number of interrupts removed from the queue is reported. This number will be zero if there are no pending interrupts, but this is not an error condition.

CLEAR MEMORY

The CLEAR MEMORY command (or ZERO command, which is a synonym) is used to zero out the virtual VAX's memory. All physical memory is cleared, which implicitly zeros out virtual memory. When a CLEAR MEMORY command is given, it also turns virtual memory off by setting the privileged register MAPEN to zero, since the page tables will have been zeroed.

Like many commands that can destroy the context of the currently running VAX, this command can only be executed when the processor is in KERNEL mode. See the SET MODE command for information on changing modes.

```
CLEAR MEMORY
```

Because this command clears memory, it also clears all symbols in the user symbol table, since they will reference memory and data that no longer exists. Note that symbols marked as permanent are *not* cleared by a CLEAR MEMORY command. Additionally, many of the processor-privileged registers are reset since they depend on the contents of memory. Below is a table of the registers that are reset by the CLEAR MEMORY command.

Register	**Description**
AP	Set the size of VAX physical memory – 4 bytes (same as SP)
ESP	Set the size of VAX physical memory – 4 bytes (same as SP)
FP	Set the size of VAX physical memory – 4 bytes (same as SP)
ISP	Set the size of VAX physical memory – 4 bytes (same as SP)
KSP	Set the size of VAX physical memory – 4 bytes (same as SP)
MAPEN	Zeroed, indicating that virtual memory translation is disabled.
P0BR	Zeroed, indicating there is no P0 page table base
P0LR	Zeroed, indicating there is no P0 page table length
P1BR	Zeroed, indicating there is no P0 page table base
P1LR	Zeroed, indicating there is no P0 page table length
PC	Zeroed
SBR	Zeroed, indicating there is no S0 page table base
SLR	Zeroed, indicating there is no S0 page table length
SP, USP	Set the size of VAX physical memory – 4 bytes
SSP	Set the size of VAX physical memory – 4 bytes (same as SP)

CLEAR PROFILE

The CLEAR PROFILE command clears profiling statistics on the usage of each instruction. During execution of the virtual VAX, a counter is incremented for each instruction when it is used. This is used internally in the development of the emulator to identify instructions needing performance improvements. It can also be used to diagnose some things about a running user program. Note that currently the profile counts are kept are only event-based, not time based.

```
CLEAR PROFILE
```

This clears all the instruction counters, resetting them to ZERO. Instruction profiling resumes with the next instruction executed; you cannot turn this feature off. See the information on SHOW INSTRUCTION PROFILE for a sample of the information kept by instruction profiling.

CLEAR STRINGS

The CLEAR strings command is used to re-initialize the string constant pool used by the micro-assembler. These pages are reserved by the VMINIT command for use in constructing string descriptors and their contents. Strings are created when a quoted string is used as an argument to a CALL command or in assembling an instruction where an address would normally be expected.

The CLEAR STRINGS command will result in an error if the string pool as not been created by the VMINIT command or if it is no longer valid. Note that the string pool is used when running eVAX as a standalone system; if you were to boot a real operating system then the string pool would not be available.

```
CLEAR STRINGS
```

The CLEAR STRINGS command re-initializes the string pool memory. This means that any existing references to descriptors created in the string pool are no longer valid; the descriptors they point to will have been zeroed. In general, string pool strings should only be used for CALL arguments or statements in a DO command where the string is not expected to be valid after the statement completes.

See the documentation on the VMINIT command for more documentation on string pools. You can display the contents of the string pool with the SHOW STRING command, which also lists the console symbols that describe the start, size, and current allocation point in the pool.

CLEAR SYMBOL

The CLEAR SYMBOL command is used to delete a symbol stored in the console symbol table. Note that this symbol table is distinct from symbol tables maintained by operating systems or tools that are built using eVAX. The console symbol table is used to define aspects of the running system, and to support the micro-assembler.

```
CLEAR SYMBOL FOO
```

This command removes symbol FOO from the symbol table. The symbol will no longer be recognized and its value is no longer available. The symbol will be removed even if it is marked as a "permanent symbol"; explicit deletions of symbols are always permitted. The symbol may be a user or system symbol.

```
CLEAR SYMBOL/ALL
```

This command clears all user symbols, and all non-permanent system symbols. Permanent system symbols are not deleted by this command, and must be deleted explicitly by name using the CLEAR SYMBOL command described above.

See the documentation on SHOW SYMBOLS for information on displaying the current system and user symbol tables. See the documentation on SET SYMBOL for information on creating symbols, and on how to determine if a symbol is a system or user symbol.

```
CLEAR SYMBOL/TEMPORARY
```

This command clears all non-permanent symbols from the user symbol table. When assembling large programs, it may be convenient to free space in the symbol table by deleting symbols that are no longer required. If you mark symbols as permanent in the symbol table then they will not be deleted by this command.

CLEAR TB

The CLEAR TB command is used to clear the translation buffer managed by the virtual memory management hardware. When a virtual memory translation is made, the results are stored in a special cache called the translation buffer (TB). Future translations for the same page are first checked against the TB. This dramatically speeds up translations of pages since the actual page table entries (PTE's) do not have to be read from memory (a slower process than accessing the TB).

The CLEAR TB command is a debugging and diagnostic tools for correcting problems in the memory management hardware emulation or a page fault handler. The CLEAR TB command can be used to flush the translation buffer and force the system to re-acquire virtual page translation data on subsequent page access operations.

```
CLEAR TB
```

Note that translation buffer usage is controlled independently of the state of the TB itself. The privileged processor register TBDR controls whether translation buffer caching is performed. Setting the TBDR to 1 disables the use of the TB. Setting TBDR to zero re-enables it. For more information, see the documentation on privileged registers, including TBDR (TB Disable Register), TBIA (TB Invalidate All, similar to the CLEAR TB command), and TBIS(TB Invalidate Single, which invalidates a single TB entry). The SHOW TB command can be used to display the state of the TB and any valid cached translations.

DISASSEMBLE

The DISASSEMBLE command is used to display the contents of memory formatted as VAX instructions. The DISASSEMBLE command may also be entered as EXAMINE/INSTRUCTION for compatability with the VMS debugger.

```
DISASSEMBLE 0200
```

This disassembles a single instruction at location 0200 in memory. You can also disassemble a range of memory by giving a starting and ending address.

```
EXAM/INSTR MAIN MAIN+030
```

In this example, 30 bytes of memory are to be disassembled, starting at location described by the symbol MAIN. Note that if a range is given, and the ending byte is in the middle of an instruction, the entire instruction will be displayed.

When disassembling instructions, the system uses the user and system symbol tables to attempt to "symbolize" address values. For example, if the disassembly includes a location marked as an entry point, it will be displayed as an .ENTRY value.

Additionally, relative branch instructions (BRB, SOBGTR, etc.) will have the destination displayed as the actual address rather than the displacement stored in the instruction. This makes reading the destination of the branch easier.

Instructions with displacement addressing will show the displacement, but a comment will be placed at the end of the line showing what the effective address is of the displacement. This is helpful particularly in disassembling PIC code.

There are a number of ASSEMBLER flags that control the nature of the disassembly; see the documentation on the SET ASSEMBLER command for more information.

Below is a sample disassembly of a section of code that was generated by the DEC C compiler and loaded into eVAX for execution. In this case, the only symbolic name known is the entry point (defined by the image loader).

```
VAX> exam/instr main main+030
00000218:               .ENTRY MAIN, ^M<R2>
0000021A:               SUBL2    S^#30,SP
0000021D:               JSB      @L^229                         ; 0000044C
00000223:               MOVL     S^#01,R2
00000226:               BRB      0000022F
00000228:               RET
00000229:               HALT
0000022A:               SUBL2    S^#30,SP
0000022D:               CLRL     R2
0000022F:               MOVQ     B^DE,B^F0(FP)                  ; 00000210
00000234:               MOVAB    B^D2(FP),B^F4(FP)
00000239:               PUSHAB   B^C4                           ; 00000200
0000023C:               PUSHAB   B^D2(FP)
0000023F:               CALLS    S^#02,@L^20A                   ; 00000450
00000246:               PUSHAB   B^D2(FP)
00000249:               CALLS    S^#01,@L^204                   ; 00000454
```

DO

The DO command (which can also be specified as a single period "." character) is used to execute a single VAX instruction immediately. The instruction text follows the command verb. The instruction is assembled, and stored in the memory location defined by the symbol CONSOLE$SCRATCH. This symbol is created by the VMINIT command. If an operating system is booted on eVAX, then the DO command will no work unless the CONSOLE$SCRATCH symbol is directed to a memory locate that is writable by the current mode, and available for use. If the CONSOLE$SCRATCH symbol is not defined an error results from using the DO command.

```
DO MOVL R0, R1
```

This is an example of a DO command. It assembles a MOVL (move longword) instruction, whose function is to move the contents of general register 0 to general register 1. Any instruction can be assembled as a DO statement. However, statements that cause a flow of control change will not actually change the PC value. The instruction specified by the DO command is executed with the trace pending bit set, so that the console regains control after the instruction executes, and restores the PC to it's original value. However, instructions that modify the stack frame pointer, or other registers will work.

```
DO PUSHL @#MYDATA
```

This instruction pushes the longword stored at the memory location identified by the symbol MYDATA onto the stack. The stack pointer and stack contents are affected by this instruction, although the PC is not changed.

EXAMINE

The EXAMINE command is used to examine the contents of memory. The command takes an address to display. Optionally a second address is given and the instruction will display all the data between the range of addresses, inclusive. A variety of modifiers can be used to describe the data format being displayed. This type value must immediately follow the EXAMINE verb before the address (range) to be displayed.

Keyword	Description
/ASCII	Character data in ASCII encoding.
/AD	This is a synonym for /DESCRIPTOR, for compatibility with the VMS debugger.
/AZ	This is a synonym for /ZERO, for compatibility with the VMS debugger.
/COUNTED	The data is a counted ASCII string, where the first word (two bytes) describes the length of the data. A range specification for EXAM is not permitted in this case.
/DESCRIPTOR	The data is a string descriptor, containing a length and address value. The string described by the descriptor is displayed. A range specification for EXAM is not permitted in this case.
/ZERO	The data is a null (zero) terminated string. The character data up to the first zero byte is displayed. A range specification for EXAM is not permitted in this case.
/BYTE	Data is displayed as hexadecimal bytes
/D_FLOAT	The data is D_FLOAT data in 8-byte values.
/F_FLOAT	The data is F_FLOAT data in 4-byte values.
/LONG	The data is longword integers in 4-byte values.
/WORD	The data is word integers in 2-byte values.
/INSTRUCTION	The EXAM/INSTRUCTION command is a synonym for the DISASSEMBLE command. See the documentation on DISASSEMBLE for more information.

The address or range of addresses can be expressed as a numeric value, a symbol, or an expression consisting of values, symbols, and registers contents. Address ranges must specify the smaller address value first. Memory access to display the contents is done at the current access mode and uses the virtual memory mapping if enabled. Therefore, the EXAM command may generate an access violation error if the page is not readable at the current access mode of the processor.

```
EXAMINE 0200 0210
00000200: 73696854 20736920 65742061 00007473    This is a test..
00000210: 010E001D                               ....
```

This example displays the contents of memory locations from 00000200 to 00000210. The default data type is a longword. For integer data types, the memory is also displayed as text on the right side of the output.

```
EXAM /DESC TSTMSG
Length of descriptor string is 16 (0010) bytes
80005604: Here is a string.
```

This example shows the use of a complex data type (an ASCII DESCRIPTOR) and the use of a symbol (“TSTMSG”) to define a memory address. Note that for complex types (descriptors, counted strings, and null-terminated strings) you can only specify a single address to display, not a range.

```
EXAM /D_FLOAT 200 208
00000200: 1234.567000000
00000208:  8.000000000
```

This example shows displaying a range of addresses as floating point values. Note that the second value starts at memory address 208, which is included in the range. The value displayed actually resides at locations 208-20F, and the entire data item is shown even if parts of it are outside the given range.

```
EXAM @SP-8 @SP
7FFFFDF0: 00000001 80005614 20746865    .....V..eht
```

This example shows using a register value in an expression. This displays the contents of memory starting eight bytes before the current value of the stack pointer SP register. Note that you must specify the range in ascending order; i.e. the first value must be smaller than the second value or an error occurs.

```
EXAM/F_FLOAT R0
```

You can also examine the contents of registers using the data type modifiers such as /F_FLOAT in this case. The contents of the register itself (rather than the data it points to as an address) are formatted. You cannot use a complex data type such as /DESCRIPTOR in this mode.

GO

The GO command (or it's synonym EXECUTE) is used to start execution of the VAX virtual CPU, at the current value of the PC register. Optionally, a starting address can be given that is first loaded into the CPU.

Note that unlike many VAX computers, the virtual VAX CPU is not running when the console is active. The virtual VAX is a loop that reads instructions and executes them, as well as responding to events like interrupts. The virtual CPU is started by executing certain console commands such as GO or CALL.

```
GO 0200

GO MAIN+2
```

The first example stores the value 00000200 in the PC and starts execution of the virtual VAX.

The second example shows using an expression as the starting address. In this case, it is assumed that the entry point MAIN is probably a procedure. As such, the first two bytes are not instructions at all but rather an entry mask, which is not executable. This starts execution at the first instruction of the procedure. Note that this will still probably result in a fault when the RET instruction is executed since there is no valid call frame created by the GO command. If you wish to invoke a procedure from the console, you should use the CALL command instead.

Execution will continue until interrupted by the console, a HALT instruction occurs, an unimplemented exception occurs, or a breakpoint event occurs. You can interrupt from the console by using the attention-handling key for your system. For example, on a Unix implementation of eVAX, control-c is probably the attention handling key that generates a SIGINT interrupt. On a Mac, pressing command-period will interrupt execution.

If a HALT instruction is executed, then the processor stops. Note that if a HALT is executed in user mode, then a EXC_PRIV exception is signaled, and normal exception handling code is invoked, if installed in the SCB vector on the running VAX. When writing programs directly using eVAX it is often convenient to allow a HALT to simply halt the processor. In this case, use the SET DEBUG USERHALT command to allow a HALT in user mode to stop the emulation.

HELP

The HELP command is used to display information about the syntax and usage of commands in eVAX. The text of the help information is read from the file “vax.help” which must be located in the current default directory along with the eVAX application. The HELP command accepts as it’s argument a command, which it uses as a lookup key to find the associated help text. If no command is given, then HELP is assumed.

```
HELP

HELP SET

HELP SET DEBUG
```

Each of these examples displays information. The first command displays the generic introductory help, which explains how to use the command, and lists the basic keywords that you can use to get started.

The second example displays help about the SET command. The SET command accepts a second keyword after SET that determines what is to be set, usually followed by one or more additional arguments. The HELP SET command will show the SET command keywords, and the user must then issue a more detailed HELP command to get additional information.

The third example shows the more detailed case. In this example, the information displayed is specifically for the SET DEBUG command.

Note that the command you type is parsed and used as a search key in the file. Because of that, if you type a command that does not exist, you will just get a message saying there is no HELP for the topic given. If you issued a command like HELP SET FOO you would get this message, not a list of topics for HELP SET that are valid. In this way, the help facility is much less sophisticated than the superficially similar facility in VMS, for example.

The help data file “vax.help” is a simple text file. In the distributed form, it has a comment section at the beginning that describes the syntax for describing help text and keywords. Note that like the command parser itself, help keys must be unique in the first four characters of each command verb. This means that you cannot have a help entry for both PROCESSOR and PROCESS since they would be identical in the first four characters.

IF

The IF statement is used to conditionally execute a single command. The IF statement is followed by an expression. If the expression results in a non-zero result, then the remainder of the command is treated as a new command to be executed. If the results are zero, the rest of the command is ignored. The IF statement is mostly used in INCLUDE files (see below).

```
IF PC = 000000 SET PC = 0200
```

This statement evaluates the current contents of the PC in the emulated VAX. If it is zero, then the PC is reset to the value 00000200. If it is not zero, then it’s value is unchanged.

```
IF DEFINED( “SS$_NORMAL” ) = 0 THEN ASSEMBLE SSDEF.ASM
```

This IF statement uses a built-in function. In this case, if the symbolic value “SS$_NORMAL” is not defined (result of function is zero) then the file “SSDEF.ASM” will be assembled.

INCLUDE

The INCLUDE command is used to read console commands from an external text file. This command may be abbreviated as “@” for compatibility with VMS. The text file is read one line at a time and each command is executed as it is read. Processing the include file terminates when an error occurs or the end of file is reached.

Note that each command executes to completion before the next command is run. So if the include file contains commands that execute VAX programs, boot an operating system, etc., these will halt execution before the next command in the file is read.

You can use the IF statement to conditionally execute statements in the file, but unlike DCL or other shell scripting languages there is not flow-of-control operation, etc.

```
INCLUDE BOOT.COM

INCLUDE /LIST “myboot.com”
```

Note that the command name may be enclosed in quotation marks. All unquoted text on a console command line is converted to uppercase as part of the parsing operation. Some file systems (notably Unix and its derivatives) have case-sensitive file names. To ensure that the file name is used as a lower- or mixed-case name, enclose it in quotation marks. Additionally, if the file name is in an alternate directory, you should consider using quotation marks so that the path name syntax is not confused for qualifiers or additional parameters.

The /LIST qualifier can be added to specify that each line of the file is to be displayed as it is executed. If /LIST is not specifically given on the command line then the default VERIFY mode is used (see the documentation on SET VERIFY). Each line of the file is displayed prefixed by the name of the file it comes from. Note that included files may include files themselves, so the name helps to indicate which nesting level the command is from.

INITIALIZE

The INITIALIZE command is used to recreate an initialized VAX in a consistent state. This is similar to the state that a VAX is in immediately after power-up. The INITIALIZE command requires a single parameter, which is the size of physical memory the virtual VAX will use. This memory is allocated from the host operating system. If the memory specified is not available, then the INITIALIZE command will fail; however, the current virtual VAX is none-the-less destroyed.

```
INITIALIZE 2000
```

This command specifies a virtual VAX with 2000 (hexadecimal) pages of memory. Since each page is 512 bytes, this specifies 8192*512 bytes of memory, or a four-megabyte physical memory layout. The memory must be available as a single contiguous memory allocation.

Note that the INITIALIZE command will completely destroy the contents of the current virtual VAX state, so you must first have saved any memory or processor contents. See the SAVE command for more information. Because of this, the INITIALIZE command can only be issued when the processor is in kernel mode, to reduce the chance of accidentally destroying the machine state.

When the INITIALIZE command succeeds, the physical memory of the machine is set to all zeroes. Most general-purpose registers are set to zero. The stack pointer is set to the size of physical memory minus four. Virtual memory is disabled, and most privileged processor registers are reset to zero. IPL is set to 31 and the processor is in kernel mode. You must boot an operating system or load a microkernel before much of the processor will be useful, so that interrupts, I/O, etc. can be configured.

LOAD

The LOAD command is used to load a memory image from a file on the host system. Memory images are created by the SAVE command. A memory image contains the Processor State and the contents of physical memory, stored in a binary file format. When a memory image is loaded, the processor registers and state are restored, as is physical memory. Programs, etc. stored in this memory can be executed as they were when it was saved, with the exception that symbols present in the console are **not** saved in the memory image.

```
LOAD “the.vax”
```

This loads the memory image from a file called “the.vax”. Note that some operating systems (notably variants of UNIX) have case-sensitive file names. Because eVAX converts all command text to upper case as part of parsing the command, if the filename must contain lower-case letters, then enclose it in quotation marks.

```
LOAD
```

In this case, the LOAD command is issued without a file name. The default file name is “Saved.VAX”. This default name is used with the SAVE command when no file name is given.

One important note is that the LOAD command can only load binary memory images. If the state of the machine is saved with the SAVE TEXT command (instead of the SAVE command) then the resulting file is a text file that must be read with the INCLUDE command. If you attempt to load a text file into memory with the LOAD command, an error message is displayed.

Also, note that since the LOAD command *completely replaces* the state of the current virtual VAX, it can only be executed when in kernel mode to reduce the chance of accidentally destroying the state of the virtual VAX.

```
LOAD/ROM/BASE=20040000/SIZE=^d256 “ka360.txt”
```

This example shows loading a ROM dump file. A dump file is a text file that contains the output of the EXAM command on a real VAX system. The console file must be captured via a PC emulator or similar program as a text file on your host system. This file is read with the LOAD/ROM command.

In the above example, an explicit base address and size of the console ROM is given. Note that the size is expressed in units of kilobytes; i.e. the above example specifies a 256K ROM.

```
LOAD/ROM/NOERROR “default.rom”
```

This command does not specify a /BASE or /SIZE. These qualifiers cannot be used when the file being read is a ROM image instead of a ROM dump. A ROM image is created with the SAVE/ROM command, described later in this document. If a name is not given, the default name is “default.rom” in the directory containing the emulator.

The /NOERROR flag is used to suppress an error if the file name given cannot be found. This allows it to be used in the initialization file without causing an error if the given installation does not have a ROM file. Note that at the time of this writing, the ROM files are considered copyrighted material by Digital/Compaq. You must dump the ROM file of a machine you already legally own to satisfy this requirement, and you may not distribute the ROM file.

```
LOAD/NVRAM/NOERROR “default.nvram”
```

This command loads a non-volatile RAM image (NVRAM). This image reflects the non-volatile memory storage area found on many smaller VAX systems to contain configuration info, VAX console environment variables, etc. In this command, the NVRAM file must have been created by a SAVE/NVRAM command previously executed. If a name is not given, the default name is “default.nvram” in the directory containing the emulator. The /NOERROR qualifier is optional and has the same function as in the LOAD/ROM command to suppress error signaling if the NVRAM image file is not found.

```
LOAD/IMAGE “sieve.exe”
```

This command loads a VMS image into memory. The image file name must be specified in quotation marks if the host system has case-sensitive file names (such as Unix). The VMS image is decoded and loaded into memory. It is not executed. The command will print the starting address and defines a symbol MAIN that points to the transfer address.

PRINT

The PRINT command is used to display information on the console. This can be used from within an INCLUDE file, for example, to display information about the file being loaded. The command can be issued with no parameters, in which case it prints a blank line. If parameters are included, it prints each one in order, followed by a newline character. Each element of the PRINT command can be a quoted literal string or a numeric expression. Numeric expressions are formatted using the current default radix (see the SET RADIX command for more information).

```
PRINT “The result is “, R0
```

This example shows a simple display of a text label and a simple expression consisting of a general-purpose register name.

```
PRINT “Next address is “, R4+200
```

This is a similar example, but shows the use of an expression. When a general-purpose register such as AP or R0 is given in an expression, the value of that part of the expression is the contents of the register. In this case, the contents of general-purpose register four are added to the value 00000200 and the result is displayed.

QUIT

The QUIT command terminates the emulator. In non-windowing systems such as Unix or Alpha VMS, the application returns to the shell. On the MacOS implementation, it terminates the console window without saving the console text; you must do that manually before quitting. On the Windows implementation, you are prompted to press return before quitting. This is an artifact of the console window implemented by the Metrowerks runtime, and does not give the user additional ability to save their work.

```
QUIT
```

Note that the QUIT command does not save the window contents or the state of the machine; you must do that manually. See the documentation for the SAVE command for information on saving the state of the virtual machine. Use the windowing system to save the console window on systems that support it.

RUN

The RUN command allows you to load and run OpenVMS format images into the emulator's P0 space. Note that the emulator does not support a full compliment of runtime libraries, so the nature of the programs you can write and run are somewhat limited without extended support from the microkernel.

```
RUN "sieve.exe"
```

This locates and loads the file "sieve.exe" from the host environment's file system into the virtual memory space. The image is decoded using standard OpenVMS semantics for image sections, though some image section types (notably global sections and stacks) are not supported. The image is run using the default user mode stack in the virtual machine.

When loaded, the entry point to the image is defined using the user symbol MAIN so you can re-execute the program using the CALL MAIN command. If there are errors in loading (for example, references to runtime libraries that do not exist, etc.) then the code is not run.

Typical examples of code you might use this with are programs written using VAX MACRO assembler or the DEC C compiler that do not require runtime library support not available in the microkernel. This can however be used to test code by building it on a real VAX using OpenVMS and then running it on the emulator by FTP'ing or otherwise moving the file (in binary mode) to the system where the emulator is running.

If the program being loaded has references to runtime libraries, the loader will attempt to resolve those using "shim" references in the microkernel. Each entry point in each runtime library required by the program is converted to a system symbol. Supported entry points are defined in the microkernel using the .SHIM pseudo-opcode. See the documentation on the SHIM pseudo-opcode for more information.

Each G^ and .ADDRESS reference in the loaded image to a runtime library mapped to a symbol. The symbol name in SHIM$runtime_offset, where "runtime" is the name of the runtime library like VAXCRTL, and offset is a 4-digit hexadecimal number that is the offset into the shareable image transfer vector.

For example, the strcat() runtime routine is at offset 06B0 in the runtime library. References to the runtime library routine in the executable image are remapped through a G^ fixup by the VAX C compiler and VMS linker. When this image is run by eVAX, the fixup entries are created as if they were symbolic references in the micro-assembler. If the microkernel has already defined the runtime entry (SHIM$VAXCRTL_06B0, in this case) then the fixup vector is resolved to point to the microkernel runtime. If it does not exist, then it becomes an unresolved forward reference, and the program cannot run until it is resolved. However, it does allow the user to load appropriate runtime routines after loading an image (but before being able to execute it.)

The .SHIM pseudo-opcode in the micro-assembler allows the user to either map entry points in the micro-assembler to the associated runtime library entry point, or to map them to internal runtime routines in the emulator itself. The first case allows the user to write supporting runtime routines for conventional VMS runtime libraries without having to implement the entire runtime library. The second mechanism allows the authors of eVAX to extend the functionality of the runtime environment to connect to native runtime support. For example, the VAX C runtime routine open () could be mapped to the actual open () function call in the host operating system using a .SHIM appropriately coded to invoke a native runtime. This is documented more clearly in the section on the .SHIM pseudo-opcode in the section on the micro-assembler.

Note that the RUN command is not valid when an operating system is booted on the machine; the RUN command requires minimal support from the microkernel and the virtual memory layout supported by the VMINIT command. See the documentation on the VMINIT command for more information about this.

SAVE

The SAVE command is used to save the contents virtual VAX to a file in the host file system. The file can a binary image of memory and the virtual machine that is read in with the LOAD command. With the addition of the TEXT modifier, it can be a text file that is re-read using the INCLUDE command and is human-readable micro-assembler statements describing the state of the machine.

```
SAVE “saved.vax”
```

This saves a binary file that contains the state of the machine. The saved file contains the state of the virtual VAX processor (general-purpose registers, privileged processor registers, and console state) as well as the contents of physical memory. Note that the quotation marks around the name are used to preserve the case of the file name; this may or may not matter depending on the host system involved. VMS stores all files in upper-case only, while Windows and MacOS will preserve the case name in the file but do not consider it significant when specifying the file for reading later. Unix is completely case sensitive and requires the quotation marks to ensure that the file name will match when re-read later with the LOAD command.

```
SAVE /TEXT “myvax.com”
```

This saves the state of the machine as a text file that is readable using a standard text editor. The included file invokes the micro-assembler and uses pseudo-opcode directives to set the state of the processor and virtual memory, and re-establishes symbols, etc.

Here is an example of some of the contents of a saved TEXT file state:

```
;TXT;
;       Saved VAX state as of Sun,  Aug 29, 1999  12:39
;

INIT ^D2048
SET NODISASM
SET RADIX HEX

ASM

;       Non-zero internal registers

        .KSP    80002BFC
        .ESP    800033FC
        .SSP    80003BFC
        .USP    7FFFFDA0
        .ISP    000045FC
        .P0BR   80000A00
        .P0LR   00000400
        .P1BR   7F802400
        .P1LR   001FFE00
        .SLR    00000200
        .SCBB   00004A00
        .ASTLVL 00000004
        .ICCS   00000041
        .NICR   FFFFFE0B
        .ICR    FFFFFE0C
        .TXCS   000000C0
        .MAPEN  00000001
        .SID    03001200
        .PSL    03000001
;       Non-zero processor registers
        .SET  R0 00000001
        .SET  R14 7FFFFDFC
        .SET  R15 00000200
;
;       Non-zero storage
;
        .MAPEN 0              ;       Temporarily disable virtual memory
        .BASE  00000003
        .BYTE  0F0,001
        .BASE  00000007
        .BYTE  0F0,002
        .BASE  0000000B
… the remainder of physical memory is described in this way …
```

You can also save a ROM image if you have a ROM constructed in memory via a LOAD/ROM command. The saved ROM image is a binary file containing information for a future LOAD/ROM command to run faster than the dump format of LOAD/ROM.

```
SAVE/ROM “default.rom”
```

SET

The SET command is used to set the characteristics of the virtual VAX, the console, or the micro assembler. The following commands are described in detail on the following pages:

- SET ASSEMBLER
- SET BASE
- SET BREAK
- SET DEBUG
- SET MODE
- SET PSL
- SET PTE
- SET QUANTUM
- SET RADIX
- SET REGISTER
- SET SYMBOL
- SET TRACE
- SET VERIFY
- SET VM

SET ASSEMBLER

The SET ASSEMBLER command is used to set the state of various control flags in the operation of the micro-assembler. These settings are stored in the virtual VAX console state data, and can be saved with a SAVE command. They are not retained across executions of eVAX. In general, each command is of the form SET ASSEMBLER *option* where *option* is a keyword that may be prefaced with “NO” to negate it.

```
SET ASSEMBLER ADDRESS_PROMPT
SET ASSEMBLER NOADDRESS_PROMPT
```

When set, this flag indicates that the micro-assembler should display the address in which instructions and data will be written as part of the prompt for input. For example,

```
VAX> SET ASSEMBLER ADDRESS_PROMPT
VAX> ASSEMBLE
[00000200] ASM> MOVL R0, R1
[00000203] ASM> END
```

In this example, the address in which the instruction(s) will be stored is printed in the prompt within the brackets. If the option is set to NOADDRESS_PROMPT, then the prompt is just **ASM>** without the address in brackets.

```
SET ASSEMBLER BRANCH_DEST
SET ASSEMBLER NOBRANCH_DEST
```

When set, the BRANCH_DEST flag tells the disassembler component of the micro-assembler to display branch destinations as comments after symbolic branch labels. When an instruction is disassembled, if an operand is a branch destination, the disassembler will attempt to determine if a symbol exists that represents that branch destination. If so, then the symbol is used in the disassembly. If no symbol exists, the disassembly is shown as a hexadecimal displacement argument. However, when the symbol does exist it is still often convenient to know what numeric value it has; when this option is set the value appears as a comment. For example,

```
VAX> set assembler branch_dest
VAX> disassemble lib$put_output lib$put_output+30
80004E00:               .ENTRY LIB$PUT_OUTPUT, ^M<R3,R4>
80004E02:               MOVL     (AP),R3
80004E05:               TSTL     R3
80004E07:               BEQL     LIB$PUT_OUTPUT_EXIT        ;80004E2F
80004E09:               ADDL3    S^#04,AP,R4
```

In this example, the symbolic value of LIB$PUT_OUTPUT_EXIT is shown in the branching instruction BEQL, but the actual numeric value 80004E2F is shown as a comment on the same line of text. When the NOBRANCH_DEST setting is in effect, this comment is not displayed.

```
SET ASSEMBLER FORWARD_WARNINGS
SET ASSEMBLER NOFORWARD_WARNINGS
```

When set, the FORWARD_WARNINGS flag causes the micro-assembler to issue a warning whenever it completes assembling instructions that have unresolved forward references. This warning is issued when the source file is complete or an END statement is given interactively. When forward references are unresolved, the console will not allow the program to be executed since branch or addressing errors will result. See the documentation for SHOW UNRESOLVED SYMBOLS for information in displaying which symbols are unresolved.

```
SET ASSEMBLER SCOPE_NAMES
SET ASSEMBLER NOSCOPE_NAMES
```

The micro-assembler allows the user to specify locally scoped names that contain a leading underscore "_" character. These symbols are guaranteed to be made unique between entry points so individual procedures can be written to use common labels like "_EXIT" or "_ERR" as labels without collisions. When SCOPE_NAMES is set, the names are implicitly qualified by the name of the last .ENTRY specification, which effectively modifies them by concatenating the name in front of the temporary name; i.e. the symbol "_ERR" becomes "FOO_ERR" when referenced within the entry point FOO. When the SCOPE_NAMES attribute is disabled, temporary label names are just qualified with a unique number, so "_ERR" might become "__1_ERR" to make it unique. Unless there is an unexpected problem with symbolic name space, the SCOPE_NAMES flag should be left set to make symbol names more easily readable and related to the procedure that contains them.

```
SET ASSEMBLER SYMBOLS
SET ASSEMBLER NOSYMBOLS
```

When the SYMBOLS attribute is set, the disassembly portion of the micro-assembler will attempt to resolve addresses into symbolic values. This applies to branch destinations, absolute memory references, entry points, etc. When not set, the values are simply displayed as hexadecimal constants instead.

SET BASE

The SET BASE command establishes the new “next memory location” in which data is stored, in the current region. There is a separate base stored for each of P0, P1, and S0 regions. See the .REGION pseudo-opcode for information on changing default regions while assembling programs.

```
SET BASE 0200
```

This SET BASE statement means that the next location that an assembly statement will be stored in will be at address 0200. When a statement is assembled, the default base is incremented past where the instruction is stored so the next instruction or data will follow it. This can also be done within the assembly source using the .BASE pseudo-opcode.

You can reset the BASE at any time, or use the SHOW BASE command to display it’s current value. If you reset the base to a memory location that has already been used, there is no warning or error. You can use this to overwrite instructions or data that you wish to modify from the console, for example.

SET BREAK

The SET BREAK statement is used to create a breakpoint. Breakpoints are lists of memory locations that cause the console to stop execution of the VAX when they are processed as a PC value. A breakpoint can be set for a specific memory location, such that when the instruction at that location is about to be executed the console stops, allowing the user to examine registers or memory, resume execution, etc. Breakpoints can also be for specific machine exception or fault values, such that when that fault is about to be taken (i.e. a value is about to be removed from the System Control Block (SCB) to handle the fault) the console first halts the machine.

```
SET BREAK 0200
```

This sets an execution breakpoint at memory location 00000200. If the program counter (PC) register is set to this value as part of program execution then the console will halt the VAX. Note that the breakpoint is *not* taken if an instruction simply references location 00000200; the VAX must actually attempt to decode and execute the instruction at that location.

```
VAX> set break main
%VAX-I-BRKENTRY, breakpoint is an entry mask,
        changed to 00000202
```

In this example, the break point is specified symbolically instead of as a numeric value. Additionally, since the assembler knows that this address is the address of an entry point, it has marked the symbol MAIN as an entry point. The SET BREAK command knows that the program counter is never actually set to the entry point itself (since that contains the two-byte mask word) so it modifies the break point from the value of MAIN (00000200) to the first instruction following the entry mask. This only occurs when the address given is the address of a .ENTRY pseudo-opcode in the micro-assembler; the console cannot know if an arbitrary memory location is an entry point or not.

When a breakpoint is taken in this way, you can examine the state of the machine or memory, and modify them if necessary. When the breakpoint is taken, the instruction at the specification location has *not* been executed yet. After examining or modifying the environment as necessary, execution is usually resumed with a GO command or a STEP command if only a single instruction is to be executed.

```
SET BREAK/FAULT 0FC
```

This shows creating a breakpoint for a particular exception code. In this case, the exception code is 0FC, which is the console terminal transmit interrupt. This means that if the fault handler in the VAX must reference the offset 0FC in the SCB, it will first take the breakpoint and allow the user to examine the state of the machine, etc. When a FAULT breakpoint is taken, the processor has already changed mode to KERNEL and pushed the fault arguments on the correct stack (KSP or ISP). However, execution has not begun at the memory location defined in the SCB. At this point, you can examine or modify the fault parameters, etc. before resuming execution with a GO command.

SET DEBUG

The SET DEBUG command is used to set miscellaneous flags in the console to control operation of the virtual machine. Some are used to assist in debugging the emulator itself, and are useful in debugging user programs written for eVAX. The command has the syntax SET DEBUG *option* where *option* is a keyword that may be prefaced by "NO" to indicate that it is negated.

```
SET DEBUG EXCEPTIONS
```

When SET DEBUG EXCEPTIONS is in effect, debugging messages are displayed on the console when exceptions occur. This includes the implicit parameters that the exception pushes on the stack, the current PC and PSL, and the name of the exception. You can also learn about the most recent exception using the SHOW EXCEPTION command.

```
SET DEBUG INTERRUPTS
```

When SET DEBUG INTERRUPTS is enabled, the console displays information about interrupt handling. Unlike exceptions which are dispatched at the time they occur, interrupts are scheduled and the processed at a later time, either after a delay or when the processor IPL is lowered to allow the interrupt. Therefore, SET DEBUG INTERRUPT messages are printed when the interrupt is scheduled (i.e. a part of the emulator requests that an interrupt be delivered at some time in the future). Messages are also printed when interrupts are "aged" on a *quantum* boundary. The quantum in this case refers to an interval of instructions that are executed before interrupts are evaluated. The SET QUANTUM command is used to specify this interval. Every quantum, the queue of pending interrupts is reviewed, and each interrupt is "aged". Interrupts that are now dispatchable because their quantum has expired are evaluated for handling. Any interrupt that can be dispatched to a higher processor IPL is taken. When an interrupt is taken, then the SET DEBUG INTERRUPT code displays a message indicating which handler address is being used from the SCB to execute the code.

```
SET DEBUG REGISTERS
```

When set, this option causes extra code to be run when a STEP statement is executed to track register value changes. When a STEP statement executes an instruction, all general-purpose registers (including the PSL) that have changed during execution of the instruction are printed to the console.

```
SET DEBUG SYMBOLS
```

This is a tool primarily to help developers of the emulator. When set, this causes the symbol table manager to print information about when symbols are defined, evaluated, or reset.

```
SET DEBUG TB
```

This is a tool primarily to help developers of the emulator, though it can also be used to help determine when code paths can be made more efficient. When set, this causes debugging messages to be issued when the virtual memory translation buffer (TB) is used. When a memory reference is made that is not in the translation buffer, the new information is printed to the console.

```
SET DEBUG VM
```

When set, the VM debug option causes each virtual memory translation operation to be displayed on the console. During normal execution, this creates far too much information to be useful, but can be useful in debugging specific problems with the emulator or a microkernel.

```
SET DEBUG USERHALT
```

The USERHALT flag is used to indicate if the HALT instruction is considered a privileged instruction or not. By default, a HALT is a privileged instruction, and executing it when not in KERNEL mode causes a RESOP fault. When the USERHALT flag is set, the HALT instruction is not privileged and returns control to the console.

SET MODE

The SET MODE command is used to set the mode of the virtual VAX. The VAX architecture supports four basic modes: USER, SUPERVISOR, EXECUTIVE, and KERNEL. An extension of KERNEL is INTERRUPT, which indicates that the interrupt mode stack is in use. Each mode has it's own stack pointer and stack space, and the mode is used to control access to privileged instructions, privileged registers, and virtual memory protection modes. By default, the virtual VAX starts up in kernel mode, but the default microkernel and most operating systems run most functions in user mode to protect memory and data security.

```
SET MODE USER
```

This command sets the processor mode to USER. This affects the current mode and previous mode settings in the PSL, and sets the current stack pointer (to USP, in this case). This means that the SP register is stored away in the current stack pointer storage area in the processor for the previous mode and the SP is reloaded from the stored USP. If the current mode is already USER then no action takes place. Similarly, you can use the keywords SUPERVISOR, EXECUTIVE, KERNEL, and INTERRUPT in place of the keyword USER to specify a specific mode.

Note that many commands that would significantly modify the state of the virtual machine (such as BOOT, LOAD, INIT, or CLEAR MEMORY) require that the processor be in KERNEL mode before they can be executed. This is largely as a safety feature to prevent accidental destruction of the current machine state. Use the SET MODE KERNEL command to set the processor into kernel mode before executing these "privileged" commands.

SET PSL

The SET PSL command is used to set specific fields in the processor status longword (PSL) register. The PSL includes the processor status word (PSW) bits such as the condition bits, etc. which are accessible to user-mode programs. It also contains privileged state information such as the current and previous processor mode and the current interrupt priority level (IPL) of the machine.

The SET PSL command accepts one or more bit field names and values separated by an equals sign to specify the values to be put in each field of the PSL. All field values are numeric. In addition, the CUR_MOD and PRV_MOD fields accept the values USER, SUPERVISOR, EXECUTIVE and KERNEL as values. Below is a table of each field of the PSL and what it is used for. Use the field name in the command to set the value.

Field Name	Meaning
C	The carry bit
V	The integer overflow bit
Z	The zero bit
N	The negative bit
T	The trace bit
IV	Integer overflow enable
FU	Floating underflow enable
DV	Divide-by-zero enable
IPL	Interrupt priority level
CUR_MOD	Current processor mode
PRV_MOD	Last processor mode
FPD	First part done (not used on eVAX)
TP	Trace pending
CM	Compatibility mode (not supported on eVAX)

An example statement sets the integer overflow indicator and the negative indicator is shown below:

```
SET PSL C=1, V=1
```

Note that more than one field can be set at a time. When more than one field is given, use a comma “,” to separate the fields. Use the equal sign “=” to separate the field name from it’s value. A field value can be any valid expression.

SET PTE

The SET PTE command (which can also be entered as SET PAGE_TABLE_ENTRY) sets the value of a particular page table entry in the current virtual memory map, as defined by the various processor-privileged registers. Each valid virtual memory address can be translated using a page table entry, and this command allows you to modify the values of the fields in the PTE. You specify the PTE that you wish to modify by specifying the address of any byte in the page. Below is a table of the fields on the PTE that you can modify. The numeric values can be constants or expressions. For example, the PROT field is a 4-bit field, which can be expressed as a numeric constant or by the symbolic names used by VMS. An example of a symbolic name is PTE$K_KWUR, which means that the page can be written in kernel mode and read in any mode between kernel and user mode.

Field Name	Meaning
VALID	Is this PTE page valid (mapped)?
PROT	The protection code, such as PTE$K_KWUR
M	The modified bit, showing the page has been written to.
OWN	The owner field (used by VMS)
S	The software bit field (used by VMS)
PFN	The page frame number, maps to a physical page

Below is an example command that sets page 00000200 to be kernel write and user read, and sets the modified bit to zero.

```
SET PTE 200 PROT=PTE$K_KWUR  M=0
```

SET QUANTUM

The SET QUANTUM command is used to specify the quantum intervals for interrupt handling. This refers to the interval of instructions that are executed without interruption before interrupts are re-evaluated.

Currently, the quantum interval is expressed in terms of number of instructions. A quantum of 20 would mean that at least twenty instructions would execute before an interrupt is processed. When the emulator schedules an interrupt, it indicates the minimum number of instructions that must be processed before the interrupt can be handled. This is divided into quantums and each time a quantum interval occurs, interrupts are all "aged" to see which ones have become eligible for handling.

```
SET QUANTUM 20
```

This indicates that the minimum number of instructions that will execute before interrupt servicing is 20 instructions. This is the default value when the emulator initializes. Setting this number to a larger value will improve emulator performance at the expense of interrupt delivery responsiveness. In the future, the emulator will be extended to express interrupt quantums in terms of time rather than instruction counts.

SET RADIX

The SET RADIX command is used to set the default radix of constant values parsed in command and assembler expressions, and controls the radix of the output of the PRINT command and instruction disassembly.

```
SET RADIX HEX
```

This sets the radix to hexadecimal. This can also be specified as SET RADIX 16. In this case, integer values are all assumed to be 32-bit integer values in hexadecimal notation. In this case, numbers must always have a digit 0-9 as the first character of the constant, so if the value begins with A-F, use a leading zero for the constant.

```
SET RADIX DEC
```

This sets the default radix to decimal numbers. This can also be expressed as SET RADIX 10. Leading zeros are not required for any decimal number.

You can always specify a constant in a specific radix without worrying about the default by using a radix notation, like x^0FF for hexadecimal or d^10000 for decimal. In these cases, the radix notation overrides the default.

SET Register

The SET Register command is used to set the value of a general-purpose or processor-privileged register. Note that this command does not have a “REGISTER” keyword. Instead, the SET verb is followed by the name of the register, an equals sign (“=”) and the numeric value to be placed in the register. The value can be a constant or expression, including the name of another register.

```
SET R5 = R6 + X^100
```

This example sets the general-purpose register R5 to be the contents of register R6 plus 256. You can name a general-purpose register by the R*x* notation, such as R12, or by its alias name if it has one, such as AP, FP, SP, or PC.

```
SET NICR = 0FFFFFFE0
```

This is an example of setting a processor-privileged register (the next interval count register, NICR) to a value. You cannot set a privileged register unless the processor is in KERNEL mode. See the documentation on SET MODE for more information.

SET Symbol

The SET Symbol command is used to set the value of a symbol. The symbol is created if it does not exist. Note that the SET Symbol command does not have a "SYMBOL" keyword. Instead the SET command is followed optionally by a qualifier, then the name of the symbol, an equals sign ("=") and the value to set, expressed as an integer constant or expression. Note that a symbol name cannot also be the name of a general-purpose or processor-privileged register.

```
SET MAX_COUNT = d^100
```

This sets the symbol MAX_COUNT to contain the decimal value 100. If the symbol does not already exist, it is created as a user-mode symbol. The command SHOW SYMBOL can be used to display its value. After the SET statement, the symbol can be used in any expression in a command or micro-assembler statement.

```
SET /ENTRY MAIN = 200
SET /LABEL TEST = 220
```

The first statement shows using a qualifier to modify the symbol. In this case, the symbol is marked as being an entry point. This information is used when disassembling instructions or determining breakpoints in the code. The corresponding SHOW SYMBOL output will indicate that the flag is marked as an entry point. The second example uses a similar qualifier called /LABEL can be used to indicate that the symbol is an address in memory that can be used as the destination of a branch. This will cause the disassembler to display the label address when the instruction at location 220 is disassembled. Also, instructions that reference location 220 as an address (such as a branch instruction or absolute addressing mode) will substitute the symbol TEST in the disassembly text (assuming SET ASSEMBLER SYMBOLS is enabled).

```
SET /PERMANENT EXE_COUNTER = 0
SET /SYSTEM    EXE_INIT = D^1000
```

The first example here uses the /PERMANENT qualifier to mark the symbol as a permanent symbol. A permanent symbol in the user mode table will not be removed even when the CLEAR ALL SYMBOLS command is given. Symbols that are in the system table are always permanent by definition. The second example shows using the /SYSTEM qualifier to mark the symbol as being stored in the system symbol table rather than the user table. By default, symbols are stored in the user table unless they contain a “$” character in the name. In this case, they are stored in the system table. The /SYSTEM qualifier is used to force a symbol to be placed in the system table even if its name does not contain a “$” character.

SET TRACE

The SET TRACE command is used to control whether the console displays each instruction as it is executed.

```
SET TRACE
```

This enables the disassembly function. When enabled, the current stack and stack pointer, the current PC, and the instruction at that location are disassembled as each instruction is executed. The display is sent to the console before it is executed, so messages or results of a fault, exception, etc. are reported immediately after the instruction that causes the event to occur. The disassembly honors the various debugging flags such as the SET ASSEMBLER SYMBOLS, which determines if symbolic names are used for address operands. Below is a simple example:

```
VAX> set trace
VAX> call test
[USP 7FFFFD8C] 00000202:               MOVL     S^#02,R0
[USP 7FFFFD8C] 00000205:               ADDL2    S^#05,R0
[USP 7FFFFD8C] 00000208:               RET
```

To disable the disassembly output of each instruction, use the SET NOTRACE command. Note that generating the disassembly output produces a lot of output and slows the execution speed of the emulator by a factor of more than a hundred. It should only be used when it is required as a debugging or evaluation tool.

SET VERIFY

The SET VERIFY command is used to cause the console to display each command read from an INCLUDE file at the time that it is read, before it is executed. The display includes the name of the file currently being read, and the text of the command.

```
SET VERIFY
```

Following this command, each subsequently included file is displayed as it is read. This includes files read with the ASSEMBLE command as well as those read with the INCLUDE command.

```
SET NOVERIFY
```

This disables the VERIFY mode again, so that subsequent INCLUDE or ASSEMBLE commands are not displayed to the console.

SET VM

The SET VM command is used to enable or disable virtual memory. This is equivalent to writing a 1 or 0 to the MAPEN privileged register.

```
SET VM
```

When VM is set, then all memory references are first translated via the page table entries specified in the P0BR, P1BR, and SBR registers, as specified in the VAX architectural manual. Memory operations can fail due to a virtual address not being valid (mapped) or because of a page protection fault where the current processor mode is not allowed to access a memory page.

```
SET NOVM
```

When VM is disabled, each memory address is made without translation into the physical memory array of the virtual VAX. Note that some operations in the execution of the processor may temporarily turn virtual memory off, such as addressing the actual page table entries themselves.

SHOW

The SHOW command is used to display information about the state of the virtual VAX or the console and micro-assembler. The following pages describe the various SHOW commands in detail.

- SHOW ASSEMBLER
- SHOW BASE
- SHOW BREAKPOINTS
- SHOW CALLS
- SHOW CPU
- SHOW DEBUG
- SHOW ERROR
- SHOW EXCEPTION
- SHOW INSTRUCTIONS
- SHOW MEMORY
- SHOW MODE
- SHOW PSL
- SHOW PTE
- SHOW QUANTUM
- SHOW RADIX
- SHOW REGISTERS
- SHOW SCB
- SHOW STACK
- SHOW STRINGS
- SHOW SYMBOL
- SHOW TB

SHOW ASSEMBLER

The SHOW ASSEMBLER command displays the state of the various assembler flags. See the documentation on the SET ASSEMBLER command for more information. Here is a sample output of the command:

```
VAX> show assembler
ASSEMBLER SETTINGS:
    NOADDRESS_PROMPT        Include address in ASM prompt?
    FORWARD_WARNINGS        Warn if unresolved references after ASM END
    BRANCH_DESTINATION      Display branch destination addresses
    SYMBOLS                 Display addresses symbolically
    RESOLVE_TEMP            Require temporary sym resolution in scope
    SCOPENAMES              Use scope names in temp symbols
```

This shows the default settings when eVAX is initialized. The settings indicate that the address will not be included in any assembler prompts; unresolved forward references are printed when assembly complete; branch destinations are displayed as comments, symbols are used in disassembly, symbols are resolved in scope, and temp symbols are implicitly qualified with entry point names.

SHOW BASE

The SHOW BASE command displays the current base address in which the next instruction will be written by the micro-assembler. This is the base address for the current region (P0, P1, or S0) which is set with the .REGION pseudo-opcode in the micro-assembler.

```
VAX> show base

        Next storage address is 00000200
```

This shows that the next storage location in which instructions or data will be written is location 00000200.

SHOW BREAKPOINTS

The SHOW BREAKPOINTS command displays the currently active breakpoints. This includes instruction breakpoints and fault breakpoints. See the documentation on the SET BREAKPOINT command for more information.

```
VAX> show break
    Breakpoints:
    80004C08       EXE$TX
    80004E02       LIB$PUT_OUTPUT
  F 000000FC       Console Terminal Transmit
```

In this example, there are three breakpoints. The first two indicate breakpoints that are on memory locations. The SHOW BREAKPOINT command will symbolize the addresses if there are known labels or entry points. Note that the breakpoint at 80004E02 is really at the first instruction following the entry point mask for LIB$PUT_OUTPUT, but the symbolization shows it as the entry point since that will be the first instruction of the routine.

The last line shows a breakpoint on a fault or exception. The “F” in the first column notes this. The address is really the offset on the SCB for the code, and the text shows the name of the associated fault rather than a symbolic address value.

SHOW CALLS

The SHOW CALLS command is used to display the currently active call frame. Decoding the stack frame chain that is based on the current value of the frame pointer (FP) register does this.

```
VAX> show calls
    FRAME: 7FFFFDB0, SPA: 0, CALLS, MASK: 0018, PSW: 0000
        Saved AP : 7FFFFDF8
        Saved FP : 7FFFFDE4
        Saved PC : 00000239
        Saved R3 : 00000000
        Saved R4 : 00000000
        There is 1 argument:
            [ 1]: 7FFFFDD4

    FRAME: 7FFFFDE4, SPA: 0, CALLS, MASK: 0000, PSW: 0000
        Saved AP : 00000000
        Saved FP : <console>
        Saved PC : <console>
```

This example shows a nesting level of two calls above the current procedure frame. The frame base address (pointed to by the current or saved FP) is shown. The SPA value is a integer that is the stack pointer alignment value. When a CALLS or CALLG instruction is executed, the stack is aligned on a four-byte boundary when the new frame is created. The SPA value is the amount that must be added back to the SP to return it to it's saved value.

The CALLS or CALLG indicator shows what kind of instruction was used to create this frame, which determines if the argument list must be automatically popped off the stack or not by the RET instruction. The procedure mask used to save registers, and the PSW of the caller are also shown.

This is then followed by each of the saved registers (as defined by the register MASK) value that is on the stack. In the above example, the saved FP and PC for the final call frame specify "<console>" rather than the saved PC address. This is because the frame was created by the CALL console command and does not really mean a location that can be returned to. Finally, any arguments to the call are shown by referencing the current AP register.

By default the SHOW CALLS command will display all the call frames to the top of the current mode's stack. You can limit the number of call frames by specifying an integer value on the command, as shown below.

```
SHOW CALLS 3
```

This limits the number of call frames to three in the example shown. If there are fewer than three call frames on the stack then no error is displayed.

SHOW CPU

The SHOW CPU command shows a summary of the general-purpose and processor-privileged registers, as well as the current PSL values. Here is an example output:

```
VAX> show cpu


    Registers:

    R0:  00000001      R4:  00000000     R8:  00000000     AP:  7FFFFDCC
    R1:  00000000      R5:  00000000     R9:  00000000     FP:  7FFFFDB0
    R2:  00000000      R6:  00000000     R10: 00000000     SP:  7FFFFDB0
    R3:  00000000      R7:  00000000     R11: 00000000     PC:  80004E02

    PSL: 03000001
         PSW:  C=1, V=0, Z=0, N=0, T=0, IV=0, FU=0, DV=0
         PRIV: IPL=0, CUR_MOD=USER, PRV_MOD=KERNEL, FPD=0, TP=0, CM=0

    Stack pointers:

    USP: 7FFFFDFC   SSP: 80003BFC   ESP: 800033FC   KSP: 80002BFC
    ISP: 000045FC

    Privileged registers:

        P0BR  :  80000A00    P0LR  :  00000400    P1BR  :  7F802400
        P1LR  :  001FFE00    SBR   :  00000000    SLR   :  00000200
        PCBB  :  00000000    SCBB  :  00004A00    IPL   :  00000000
        ASTLVL:  00000004    SIRR  :  00000000    SISR  :  00000000
        ICCS  :  00000041    NICR  :  FFFFFE0B    ICR   :  FFFFFE43
        TODR  :  00000000    RXCS  :  00000000    RXDB  :  00000000
        TXCS  :  000000C0    TXDB  :  00000000    TBDR  :  00000000
        MAPEN :  00000001    TBIA  :  00000000    TBIS  :  00000000
        PMR   :  00000000    SID   :  03001200

        Next storage address is 00000448
```

SHOW DEBUG

The SHOW DEBUG command displays the state of the various internal debugging flags. Most of these are used to debug the emulator itself though some may be used to debug user programs as well. See the documentation on SET DEBUG for more information on each of the flag values. Below is a sample of the output:

```
VAX> show debug
DEBUG SETTINGS:
    NODEBUG                Invoke native debugger?
    NOVM                   Debug virtual memory translations?
    NOTB                   Debug translation buffer caching?
    NOSYMBOLS              Debug symbol table handling?
    NOEXCEPTIONS           Debug exception handling?
    NOINTERRUPTS           Debug interrupt handling?
    REGISTERS              Display changed registers on STEP?
    NOIMAGES               Display image info on RUN command?
    NOUSERHALT             HALT in user mode halts CPU?
```

This shows the default state of the debug flags when the emulator is first run. Note that SET DEBUG IMAGES and SET DEBUG DEBUG are reserved for internal use and are not documented here.

SHOW ERROR

The SHOW ERROR command can be used to display the last non-success error message displayed, or display the text of a specific error code. When a command is executed, it stores the result of the execution in the system symbol $STATUS if it was an error message.

```
VAX> show error
Last error code was 00080062, "%CPU-E-BREAK, Processor halted at break point
location"
```

In this example, the last error message encountered is displayed. In this case, the value of $STATUS was 00080062, and the associated error message text is displayed.

```
VAX> show error 0400EA
Error code 000400EA, "%EMUL-E-NOTKERNEL, Command not valid when not in KERNEL
mode"
```

In this case a specific error code is to be displayed. This might be saved away in an INCLUDE command file when a command executes by storing $STATUS in another variable, for example. If you specify a message code that does not exist, you get output like this:

```
VAX> show error 0400ff
Error code 000400FF, "%EVAX-?-UNRECOGNIZED, Unrecognized return code 000400FF"
```

SHOW EXCEPTION

The SHOW EXCEPTION command displays information about the last exceptions or interrupts that have occurred, and also will display the queue of pending interrupts that have been scheduled by the emulator but not yet handled.

```
VAX> show except
    Last exception/interrupt code was 000000FC, Console Transmit
        0 arguments,    PC=80004D27   PSL=00C00004
Pending interrupts:
    IPL  Quantum  Age Code
    14     30      1  FC Console Transmit
```

In this example, the last exception or interrupt that was handled by the processor was a 0FC interrupt, caused by a console terminal transmit completing and signaling that the hardware was ready to send another byte. The argument list to the exception was empty; if there had been arguments (such as there are to an access violation) they would be shown here.

Additionally, the output shows the queue of pending interrupts. In this case there is still another 0FC interrupt that hasn't been delivered yet. The code within the emulator requested that at least 30 instructions be executed before the interrupt could fire off. This was converted to a quantum interval age. In this case, one more quantum must pass before the interrupt can be taken. When it is ready to be taken, the processor will be raised to IPL of 14 to handle the interrupt.

See the documentation on SET QUANTUM for more information on how quantum intervals are used to schedule interrupts.

SHOW INSTRUCTIONS

The SHOW INSTRUCTIONS command is a tool to describe the current version of the emulator support for instructions, and as a reference guide for instruction formats. It has several variations.

```
SHOW INSTRUCTIONS
```

This format displays a table of all supported instructions. That is, all the instructions that the emulator can execute without getting an “unimplemented instruction” error. The display shows the instruction mnemonic and the opcode byte value for each instruction.

```
SHOW INSTRUCTIONS/UNIMPLEMENTED
```

This command displays the unimplemented or unsupported instructions. For example, eVAX does not currently support G or H floating data types, so all instructions that manipulate these items are considered unsupported. An attempt to execute one of these instructions results in a fault.

```
VAX> SHOW INSTRUCTION OPC$_ADDL3
       C1 ADDL3   src.l, src.l, dst.l
```

This format shows the display of a single instruction. The emulator predefines a symbol for each opcode in the VAX instruction set of the form OPC$_*name* as illustrated above. The opcode byte for an ADDL3 instruction is 0C1. This shows the operand format of the instruction (two source longwords and a destination longword). The operands will show immediate, branching, modify mode, etc. attributes of each operand.

```
VAX> SHOW INSTRUCTIONS/ALL
…output deleted…
       32 CVTWL   src.w, mod.l
       33 CVTWB   src.w, mod.b
       39 MATCHC  src.w, addr.b, src.w, addr.b
       3A LOCC    src.b, src.w, addr.b
       3B SKPC    src.b, src.w, addr.b
       3C MOVZWL  src.w, dst.l
       3D ACBW    src.w, src.w, mod.w, br.w
       3E MOVAW   addr.w, dst.l
       3F PUSHAW  addr.w
…output deleted…
```

You can display the format of all instructions supported in the emulator with the SHOW ALL INSTRUCTIONS command. Above is an edited example of the format of this output.

```
SHOW INSTRUCTION /PROFILE
SHOW INSTRUCTION /MODES
```

These are debugging tools that can be used to tune the performance of VAX assembler code, and are used to determine which instructions need tuning the most. The first command SHOW INSTRUCTION PROFILE dumps a count of the number of times each instruction is executed. The second command, SHOW INSTRUCTION MODES, displays the number of times each individual addressing mode is used in the executed instructions. You can use the CLEAR PROFILE command to zero these totals. They reflect all instructions executed, in any mode.

SHOW MEMORY

The SHOW MEMORY command displays the characteristics of the physical and virtual memory. If the memory was initialized using the VMINIT command, then the display shows the size and nature of the virtual memory layout of each of the regions P0, P1, and S0.

```
SHOW MEMORY
```

If the virtual memory layout is created by VAX software such as the VMS operating system, then the SHOW MEMORY display will not break down the memory regions. This is because the operating system rather than the microkernel owns the page tables. If virtual memory is disabled (the MAPEN register is zero) then the virtual memory layout will still be displayed if it is “owned” by the microkernel.

If virtual memory has never been initialized then the command only reports the size of physical memory, specified by the INITIALIZE command.

Below is a sample output of a memory layout created by the VMINIT console command.

```
Physical Memory
        00000800 (2048 decimal) pages
        Addresses  00000000 - 000FFFFF

    Virtual Memory (currently ENABLED)
        P0 Region
            Region size = 00000400 ( 1024 decimal) pages
            PFN database = 9 pages  P0BR = 80000A00    P0LR = 00000400
            Physical addresses = 00040000-000BFFFF
            Virtual addresses  = 00000000-0007FFFF

        P1 Region
            Region size = 00000200 (  512 decimal) pages
            PFN database = 5 pages  P1BR = 7F802400    P1LR = 001FFE00
            Physical addresses = 000C0000-000FFFFF
            Virtual addresses  = 7FFC0000-7FFFFFFF

        S0 Region
            Region size = 00000200 (  512 decimal) pages
            PFN database = 5 pages  S0BR = 00000000    S0LR = 00000200
            Physical addresses = 00000000-0003FFFF
            Virtual addresses  = 80000000-8003FFFF
```

In this example, the virtual memory layout is described for each region. This includes the number of pages for each region, the size and location of the page frame number PFN database, and the mapped virtual and physical addresses.

SHOW MODE

The SHOW MODE command displays the current mode setting of the virtual VAX. This defines the privilege of the currently executing code for instructions, processor registers, and memory. Additionally some console commands that could destroy the VAX context require that the processor be in KERNEL mode.

```
SHOW MODE
```

SHOW PSL

The SHOW PSL command displays the contents of the current processor status longword (PSL) which contains the condition bits, mode and IPL information, and status information about the processor.

```
VAX> show psl
    PSL: 03000001
         PSW:  C=1, V=0, Z=0, N=0, T=0, IV=0, FU=0, DV=0
         PRIV: IPL=0, CUR_MOD=USER, PRV_MOD=KERNEL, FPD=0, TP=0, CM=0
```

The first line describes the data stored in the processor status word, which can generally be manipulated by the user mode program directly. The second line defines the privileged parts of the PSL, such as the current IPL or the current processor mode. See the documentation on SET PSL for information about each field in the PSL.

SHOW PTE

The SHOW PTE command (which can also be entered as SHOW PAGE_TABLE_ENTRY) uses the current page table information to display the contents of a page table entry (PTE).

```
VAX> show pte 200
    Virtual address:   00000200
    Region:            00
    PTE Address:       80000A04
    PTE Longword:      A0000201
        VALID: 1
        PROT:  04 (PTE$K_ALL)
        M:     0
        OWNER: 0
        S:     0
        PFN:   00000201
    Physical address:  00040200
    Read  mode:  01
    Write mode:  03
    PSL cur_mod: 03
    Access:      VM_READ
```

This example shows displaying the PTE data for the address 00000200. The SHOW PTE command accesses the page frame number PFN database defined by the region, base and length registers, etc. In the display, the region code is shown (0=P0, 1=P1, 2=S0). The address of the page table entry is shown, as is the unformatted PTE longword. This is followed by a decoding of the PTE longword, showing the various fields of the PTE. Finally the physical address and protection data is displayed for the page and current process.

```
VAX> show pte/write 800040ef
    Virtual address:   800040EF
    Region:            02
    PTE Address:       00000080
    PTE Longword:      F0000020
        VALID: 1
        PROT:  0E (PTE$K_URKW)
        M:     0
        OWNER: 0
        S:     0
        PFN:   00000020
    Physical address:  000040EF
    Read  mode:  03
    Write mode:  01
    PSL cur_mod: 03
    Access:      VM_WRITE
ACCVIO, protection violation
%CPU-E-FAULT, A processor exception/fault occurred
```

By default the access evaluation of the page table entry is based on READ access to the page. You can specify the mode explicitly by putting the qualifier READ or WRITE on the command. In this example, WRITE mode is tested. Since the current processor is in USER mode (3) but the page write mode requires access of KERNEL mode. Therefore, the access would incur an access violation to write to this address in the current mode, and a message is printed.

SHOW QUANTUM

The SHOW QUANTUM command displays the processor quantum value used to schedule asynchronous events. A quantum is a unit of work that the virtual VAX processor is allowed to do before an interrupt may be serviced. Interrupts may wait for one or more quantums to simulate interrupt latency delays.

```
VAX> show quantum
QUANTUM
    INTERRUPTS  Initial=20  Current=3
    USER INTF   Initial=10000  Current=1554
```

This example shows that the interrupt quantum (controlled by the SET QUANTUM command) is twenty instructions. The “current” value shows how many instructions of the quantum interval are left. In this example, after executing three more instructions, the processor will be able to take an interrupt if one is queued up.

Additionally the user interface quantum is displayed. On systems like MacOS that require cooperative tasking for user interface events to be processed, this defines the number of instructions that can execute before the user interface is allowed to take control temporarily to handle messages. In this example, the processor stops to evaluate user interface events every ten thousand instructions. Events will be handled after 1554 more instructions are executed. Since the emulator should be able to run more than a half-a-million instructions per second, this number should be kept high to keep the emulation speed as high as possible.

Note that the user interface quantum has no effect on systems that do not require any action on thepart of the application to allow user events to occur, such as Unix or VMS.

SHOW RADIX

The SHOW RADIX command displays the current default radix, or base, in which numeric constants are expected to be represented. See the documentation on the SET RADIX command for more information.

```
SHOW RADIX
```

The output of the SHOW RADIX command is a message indicating that the radix is either 16 (hexadecimal) or 10 (decimal).

SHOW REGISTERS

The SHOW REGISTERS command displays the contents of all the registers. The SHOW command can also be used to display a single register value.

```
VAX> show registers


    Registers:

    R0:  00000001      R4:  00000000     R8:  00000000     AP:  00000000
    R1:  00000000      R5:  00000000     R9:  00000000     FP:  00000000
    R2:  00000000      R6:  00000000     R10: 00000000     SP:  7FFFFDFC
    R3:  00000000      R7:  00000000     R11: 00000000     PC:  00000200

    PSL: 03000001
         PSW:  C=1, V=0, Z=0, N=0, T=0, IV=0, FU=0, DV=0
         PRIV: IPL=0, CUR_MOD=USER, PRV_MOD=KERNEL, FPD=0, TP=0, CM=0
```

This example shows the display of the general-purpose registers and the processor status longword (PSL) value. The display of the PSL is identical to the format of the SHOW PSL command output. Note that processor-privileged registers are not displayed; use the SHOW CPU command to display both general-purpose and processor-privileged registers.

```
VAX> show r5
   REG      R5 = 00004EF0 (hex)         20208 (dec)
```

This is an example of using the SHOW command to display the contents of a single register, in this case general-purpose register R5. The register contents are displayed in both hexadecimal and decimal formats. You can use this form of the SHOW command to display either a general-purpose or a processor-privileged register such as ICR, P0LR, etc. The processor must be in KERNEL mode to display a register whose contents are privileged or an error occurs.

SHOW SCB

The SHOW SCB command displays the contents of the system control block base (SCBB) register and formats the contents of the system control block. The SCB contains an array of addresses that correspond to the handlers for each interrupt or fault class. When a fault or interrupt occurs, the processor uses the address in the SCB to branch to code that handles the associated condition.

```
VAX> show scb
SYSTEM CONTROL BLOCK
    SCBB register = 00004A00
    Slot  Name            Address  ISP?  Label
     06   EXC$RESOP       80004C38       EXE$RESOP
     07   EXC$RESADDR     80004C88       EXE$RESADDR
     08   EXC$ACCVIO      80004C20       EXE$ACCVIO
     10   EXC$CHMK        80004C10       EXE$CHMK
     22   EXC$SOFTWARE2   80004C58       EXE$DELIVER_AST
     30   EXC$RESERVEDC0  80004CA8       EXE$INTERVAL
     3F   EXC$CONWRITE    80004C08       EXE$TX
```

This is an example output of the SHOW SCB command. The SCBB register is always a pointer to physical memory since virtual memory cannot be used for some kinds of interrupt handling. Each non-zero (implemented) slot in the SCB is displayed. The address of the handler (a virtual or physical address) is shown. Additionally, the “ISP?” column indicates if the handler runs on the interrupt stack or the kernel stack. If the handler uses the interrupt stack (indicated by a 01 in the last two bits of the handler address) then the address is assumed to be a physical address and the handler runs with virtual memory disabled. If the handler has a 00 as the last two bits of the handler address, it runs in virtual memory mode on the kernel stack.

Finally the address of the handler is symbolized if it is a known label, typically in the default microkernel. When a real operating system (such as OpenVMS) is running, the SCB addresses will not have symbolic values.

SHOW STACK

The SHOW STACK command is used to display the contents of the default stack, or a specific stack.

```
VAX> show stack 5
    USER MODE SP = 7FFFFDF4:
      SP+0000 [7FFFFDF4]:  00000004
      SP+0004 [7FFFFDF8]:  000040EF
      SP+0008 [7FFFFDFC]:  00000000
      <end of stack>
```

The SHOW STACK command here displays the default stack, based on the current stack pointer (SP) register. The value of the SP register itself is displayed, along with one or more values from the stack. If a count is not given, then the top stack entry is displayed. If a count is given, as in the above example, then it determines how many stack items are displayed. In the above example, there are not five entries on the stack, so the “end of stack” indicator is shown. The command determines when the end of the stack is found by detecting a protected page, or the end of the region (P1 space). The default microkernel creates a read- and write-protected page at the top (highest address) end of each of the stacks.

```
VAX> show ksp 5
    KERNEL MODE SP = 80002BFC:
      SP+0000 [80002BFC]:  00000000
      <end of stack>
```

In addition to displaying the default mode stack, you can examine any of the stacks by explicitly specifying the stack to examine. In this example, the kernel mode stack is being displayed. You can use the keywords USP, SSP, ESP, KSP, or ISP to specify the user, supervisor, executive, kernel, and interrupt stacks respectively.

SHOW STRINGS

The SHOW STRINGS command displays the “string pool” that is maintained by the console in cooperation with the default microkernel. The string pool is an area of memory reserved by the microkernel initialization where string constants can be stored. String constants are created when a command or assembly statement that requires an address is given a quoted string instead. In this case, the string is stored in the string pool and a descriptor for it is created (also in the string pool area). The address of this descriptor is used in place of the quoted string in the command.

```
CALL LIB$PUT_OUTPUT( “The result is”, mydata, “ kilograms” )
```

This console command invokes the LIB$PUT_OUTPUT service. This service expects a list of string descriptors, which are all printed on the console output device. However, in this example several of the descriptors are specified as quoted strings rather than addresses. This creates strings in the string pool as shown below by the output of the SHOW STRINGS command.

```
VAX> show string
STRING POOL
    CONSOLE$STRINGPOOL_BASE       80005600
    CONSOLE$STRINGPOOL_SIZE       00002000
    CONSOLE$STRINGPOOL (Current)  80005634

    Descriptor 80005614 [ 13 bytes]  "The result is"
    Descriptor 8000562C [ 10 bytes]  " kilograms"
```

The location and size of the string pool are defined by the system symbols shown in the output. These symbols can be used in other expressions or assembly operations. They are defined by the default microkernel when it is built.

The string pool is actually a linked list of data in the VAX memory area. Each element in the list includes a pointer to the next element, followed by the 8-byte descriptor, followed by the text of the string constant. The address of the descriptor is used for the string data in the commands that create them. When the string pool is exhausted (because more data than CONSOLE$STRINGPOOL_SIZE bytes has been created) then an error is generated the next time a string constant is specified. Use the CLEAR STRINGS command to clear out the string pool.

If eVAX is running a real operating system rather than the microkernel, then the SHOW STRINGS command will simply indicate that there is no string pool. This is because operating systems like VMS do not support the string pool mechanism with the console.

The console determines that there is no string pool because the system symbol named CONSOLE$STRINGPOOL_BASE does not exist. The BOOT command, which loads and runs the operating system boot loader, deletes this symbol as part of initialization of the bootable machine.

You should not modify the values of these symbols. They correspond to storage that was set up in the microkernel based on the size of the virtual memory configuration in the VMINIT command, and the code contained in the microkernel itself. To adjust the size of the string pool, modify the source of the microkernel (in “kernel.asm”) and reload it.

SHOW SYMBOL

The SHOW SYMBOL command is used to display one or all of the symbols in the console internal symbol table. This symbol table is shared by the console commands and the micro-assembler. These symbols are not available to the running operating system. The SHOW SYMBOL command has several forms depending on what you want to view.

```
VAX> show sym main
    MAIN = 00000200 (hex)            512 (dec) entry
```

In it's simplest form you can display any symbol in the user or system symbol table. See the documentation on SET Symbol for more information on how symbols are created and which table they reside in. In this case, the symbol MAIN is displayed with both it's hexadecimal and decimal values shown. Additionally the flag "entry" indicates that this symbol is an entry point mask address.

Additional flag values include "label" which means it is a symbolic label, "perm" which means it is a permanent symbol not deleted on a CLEAR SYMBOL / ALL command, and "local" which means it was created with local scope to an entry point.

```
SHOW SYMBOL/ALL
```

This version of the command shows all symbols in the *user* symbol table. Just like the display of the individual symbols, the display includes the symbol name, hexadecimal and decimal versions of the value, and the symbol flag if any.

```
VAX> show symbol/unresolved
Unresolved symbols:
    MYDATA                              2 unresolved forward references
         abs.l   at 0000020E
         abs.l   at 00000204
```

This version of the SHOW SYMBOLS command displays only those symbols (in either the user or system table) that have unresolved forward references. After each symbol, the list of locations in memory that reference it are shown. For each reference it indicates if it is an absolute or displacement address, and the size of the field that must contain the resolved value when it is assigned (longword, word, or byte).

```
SHOW SYMBOL/SYSTEM
```

This command displays all system symbols. Most system symbols are created during initialization of the console code in the emulator. This includes symbolic values for opcode bytes, page table protections, XFC opcodes, etc. Additionally, the microkernel creates additional system symbols for each entry pointer or service routine in the microkernel, and pointers to data storage areas such as the string pool.

SHOW TB

The SHOW TB command displays the contents of the translation buffer used by the emulated virtual memory hardware to accelerate page translation and protection checks. This translation buffer contains recently translated addresses and protection information. When a memory reference can be satisfied from the translation buffer, it is much faster than reading and decoding the PTE from memory.

```
VAX> show tb
Translation buffer caching is enabled
    Tries=9075     Hits=9062    Ratio=99%
    Flushes=1   PFlushes=811
    TB(01)  VA=00000200  PA=00040200  PROT=ALL   MODE=USER WRITE
    TB(3E)  VA=7FFFFC00  PA=000FFC00  PROT=ALL   MODE=USER READ
    TB(45)  VA=80000A00  PA=00000A00  PROT=URKW  MODE=USER READ
    TB(46)  VA=80004C00  PA=00004C00  PROT=UR    MODE=-NONE-
    TB(47)  VA=80004E00  PA=00004E00  PROT=UR    MODE=USER READ
    TB(48)  VA=80005000  PA=00005000  PROT=URKW  MODE=-NONE-
    TB(4B)  VA=80005600  PA=00005600  PROT=ALL   MODE=USER READ
    TB(51)  VA=80002200  PA=00002200  PROT=URKW  MODE=-NONE-
    TB(55)  VA=80002A00  PA=00002A00  PROT=URKW  MODE=-NONE-
```

This output shows if the TB is enabled or not (when the processor-privileged register TBDR is non-zero, then translation is disabled). It also shows how effective the buffer is currently by recording the number of attempts to translate an address in the TB and the number of times it was successful. This is translated into a ratio indicating how effective the TB is being. Numbers close to 100% indicate that most translations are done using the translation buffer rather than main memory.

Additionally the display shows how many times the entire buffer was flushed and how many times the protection check result was flushed. The translation buffer is flushed when the virtual memory registers are rewritten (which occurs during VMINIT, operating system boot, or a process context switch). Additionally, the protection results are flushed whenever the processor mode changes.

Finally this display shows each entry in the translation buffer. It shows the index into the buffer, the virtual and physical address mapping, the page protection information, and the state of the last comparison. If the state is NONE then the comparison must be re-calculated. If the state is non-NONE then it shows the nature of the last successful page translation. In this case, most pages were read in USER mode but one page was written to in USER mode.

STEP

The STEP command is used to execute a single VAX instruction in the normal instruction stream. It can also be used to step over a flow-of-control change, or to execute until the current procedure returns. Unlike the DO command which assembles an instruction and executes it, the STEP command executes the next instruction in sequence pointed to by the program counter PC register.

```
VAX> step
[USP 7FFFFDFC] 00000202:     PUSHL    @#80005614
VAX> step
[USP 7FFFFDF8] 00000208:     CALLS    S^#01,@#LIB$PUT_OUTPUT ; 80004E00
                     AP:  7FFFFDF4
                     FP:  7FFFFDD8
VAX> step
[USP 7FFFFDD8] 80004E02:     MOVL     (AP),R3
                     R3:  00000001  1
```

In this example, the STEP command is used to step through a sequence of instructions. Each instruction is disassembled and executed. If SET DEBUG REGISTERS is enabled (the default) then any register values that change during execution of the instruction are printed, except for the SP and PC registers. In this example, then CALL instruction modifies the AP and FP registers, so they are reported. Also note that between the PUSHL and CALLS instructions, the stack pointer shown in the disassembly line decrements by four bytes indicating that the longword was pushed on the stack.

```
STEP 202
```

You can specify a starting instruction to begin the STEP operation. This simply loads the PC register with the given value before executing the instruction. If you specify a memory location that is known to be an entry point then the address is offset by two bytes to start after the entry mask word.

```
STEP/OVER
```

This instruction steps "over" the following instruction. If the next instruction is not a flow-of-control statement, then STEP / OVER is the same as STEP. However, if the STEP / OVER instruction transfers control to another location (via a CALLx, BRx, JSB, or JMP instruction) then the STEP / OVER instruction will set a temporary break point at the instruction following the next instruction, and let execution continue to that temporary breakpoint. If the next instruction is a CALLx instruction, for example, this has the effect of executing the procedure in it's entirety and the returning control to the user after the RET instruction from the called procedure. This is useful if you do not want to trace the execution of a routine but instead simply continue after the routine is executed.

```
STEP/RETURN
```

This instruction is used to exit from the current routine. The saved PC in the current call frame is evaluated and a temporary break point is set to that location. Then execution continues normally. When the current procedure executes the RET instruction, the breakpoint is taken and the program returns control to the console.

VMINIT

The VMINIT command creates a simplified virtual memory model for testing VAX programs. The command maps each page of physical memory to pages of memory in the P0, P1, and S0 address spaces. By default the physical memory is divided as evenly as possible into three regions.

The command accepts parameters that let you specify the size of one or more of the regions. If not all regions are specified, all remaining memory is divided among the unspecified region(s).

```
VAX> vminit p0 = ^d1024
VM initializing
        2048 pages of physical memory
        VM region sizes(decimal pages):  P0=1024  P1=512  S0=512
```

In this example, the VMINIT command is issued with a setting for the P0 region of 1024 (decimal) pages of memory. Regions are specified by their names, P0, P1, and S0. If more than one region is to be explicitly specified then each "region=size" expression should be separated by commas.

In the above example, the first 1024 pages of memory (one-half of a megabyte) is allocated to the P0 space. The remaining space is evenly divided between the P1 and S0 spaces. In this case, the physical memory size is 2048 pages, so the P1 and S0 spaces each receive 512 pages of memory.

Note that the virtual memory model created by the VMINIT command does not support paging of any kind. There is a direct one-to-one mapping of each physical page to a specific virtual address. See the output of the SHOW MEMORY command for information on how this virtual memory layout is created. In general, the P0 and S0 pages are allocated from the base of the region (00000000 and 80000000 respectively) and the P1 region is allocated such that the last (highest numbered) page is at 7FFFFFE0, since the P1 region usually contains the user mode stack and grows downward.

The VMINIT command creates all necessary page tables in the system space to map the three regions. In addition it creates the user mode stack in P1 space, and allocates room for each of the other mode stacks in the S0 space. Each stack has a read- and write- protected page at the high end to detect stack underflows. All stacks except the USP also have a guard page at the lowest address to detect stack overflow. The appropriate processor-privileged registers are set to identify the location of the page tables, stacks, etc.

VMINIT also allocates room for a system control block (SCB) and sets the SCBB register to point to it. The SCB is initialized to all zeroes. Finally, the VMINIT command allocates the string storage area and the console scratch area. See the documentation on the SHOW STRING command for information on the string storage area. The console scratch area is a page of memory identified by the symbol CONSOLE$SCRATCH that is user writable and located in S0 space. It is where instructions for the DO command are assembled, and it is also used to read the header block into VAX memory when the RUN command is issued.

The Micro-assembler

Assembler Overview

This section describes the micro-assembler that is built in to the console. The micro-assembler is used to build the microkernel each time the system is started from a source file. It can also be used to write and test user programs. Once an operating system is booted such as VMS, the micro assembler is not very useful any more since it does not interact with the running operating system. However, the disassembler component of the micro-assembler can be used to debug and diagnose problems with the running program or the operating system.

The assembler is invoked with the ASM command. This command either accepts console input of assembler source or will read a text file from the native file system of the host computer.

```
ASM/LIST “:examples:sieve.asm”
```

This example shows assembling a source file located on the host computer (in this case, a MacOS system). The name is in quotes to preserve case; this is important on Unix systems where file system case is significant.

The /LIST qualifier is optional; it essentially turns on the VERIFY mode during the assembly of the source. Each line of text is displayed on the console as it is read, along with the file name containing it. This is useful when the source uses the .INCLUDE directive to include nested source files.

```
VAX>    asm
ASM>    .entry main
ASM>    movl #1, R0
ASM>    ret
ASM>    end
VAX>
```

If the file name is omitted, then the console prompt becomes “ASM>” and source lines are read form the console, as shown in the above example. Input lines are read from the console until the END command is given, which returns control to normal console mode.

Assembler Source Format

Here is an example line of source for the assembler:

```
RETRY:  MOVL #1, @#RESET_FLAG  ; Reinitialize the flag
```

This example of source text shows the basic parts of each line of source. In any given line of source, any of the parts may or may not be present depending on the nature of the program. The parts of the line are the label, opcode, operands, and comment. Source lines may be entirely blank. Additionally, in place of actual instruction opcode and operands, you may specify pseudo-opcodes that perform specific functions in the assembler or generate code other than an instruction.

Labels

Source lines may have zero or one label on them. A label is a symbol (up to 31 characters, starting with a letter, "_", or "$" character) followed by a colon. Labels are not case sensitive and are stored in the symbol table in uppercase. The colon ":" character is what indicates that this field is a label. The symbol given is stored in the symbol table and marked as a label. It's value is the current program counter address, before the rest of the statement is processed.

Labels normally should not contain a "$" character. Labels that have a "$" are considered system labels, and are stored in the system symbol table rather than the user table. These symbols are not deleted under normal usage. If you use a "$" in a label for use mode code, it may be replaced or changed by the user and the label could be made to be invalid. In general, all user code should be written without the "$" symbol. Note that the microkernel, if used, is written to specify the "$" character in labels, because once loaded the microkernel is not typically changed or deleted.

Labels may begin with a "_" if they are to be considered local to the current entry point. See the documentation on the .ENTRY pseudo-opcode for information on defining an entry point. Labels that start with a "_" are unique to the current entry, and usually are modified to include the current entry name. However, this allows you to use common labels like "_DONE" or "_ERROR" in all your procedures without having to make them unique. This is similar to the use of the trailing "$" in the VAX MACRO-32 assembler.

Opcodes

Opcodes are the actual instructions to be stored in memory, or pseudo-opcodes (which are directives to the assembler itself). Pseudo-opcodes are identified by a leading "." character and are described later. Opcodes follow the standard mnemonic set defined in the VAX Architecture Reference Manual. Opcode names are not case sensitive.

Opcodes may be followed by one or more operands depending on the specific instruction. You can get a display of the instruction format by using the SHOW INSTRUCTION command. You can specify the specific instruction to display using the OPC$_xxx symbol notation. For example, SHOW INSTRUCTION OPC$_MOVL will display the format of the MOVL instruction.

Operands

Instructions that require operands must have them specified after the opcode, separated by one or more blanks or tabs. The operands must appear on the same line as the opcode itself. If you specify too many or too few operands, or using invalid formats for the instruction, then an error occurs. Note that if an error occurs in the formulation of either the opcode or operand, then a partial instruction may have been written to memory.

If there is more than one operand, the operands are separated from each other by a comma "," character.

The address of the destination or the label of the destination should identify operands that are branch addresses such as for the BRB or JSB instructions. The actual instruction encodes these as relative offsets from the instruction operand. The assembler handles this encoding for you; you do not have to calculate the offset yourself. Similarly when a branch address is disassembled, the destination is described by its symbolic address if it has one.

Operands use the standard addressing modes documented in the VAX Architecture Reference Manual. These are summarized in the table below with examples.

Example	Description
`#100` *or* `I^#100`	Immediate mode. The size of the operand is determined by the size of the data type for the instruction; i.e. a MOVL instruction will use a longword constant, while a MOVB will use a byte constant.
`#1` *or* `S^#1`	Short immediate mode. If the constant is less than 64 (decimal) it will be represented in a single byte using short immediate notation.
`R5`	Register notation. The register itself is used as the source or destination of data. You can specify general purpose register notation Rx or use the following extended mnemonics: AP, FP, SP, or PC. Note that PC is not valid for some instructions.
`@#LOC`	Absolute mode. The operand is located at address LOC in this example.
`@PTR`	Indirect mode. The contents of location PTR is used as an address which identifies the actual operand or destination.
`-(R5)`	Autodecrement mode. The register is decremented by the size of the operand and then used as the address of the actual operand.
`(R6)+`	Autoincrement mode. The register content is used as the address of the operand. After this address is determined, the register is incremented by the size of the operand.
`B^05(R1)`	Byte Displacement Mode. The signed displacement value is added to the contents of the register to determine the actual address of the operand.
`W^1503(R2)`	Word Displacement Mode. The signed displacement value is added to the contents of the register to determine the actual address of the operand.
`L^FFFFE3F0(R8)`	Longword Displacement Mode. The signed displacement value is added to the contents of the register to determine the actual address of the operand.
`@B^05(R1)`	Deferred Byte Displacement Mode. The signed displacement value is added to the contents of the register to determine an address. The content of this address is the actual address of the operand.
`@W^1503(R2)`	Deferred Word Displacement Mode. The signed displacement value is added to the contents of the register to determine an address. The content of this address is the actual address of the operand.
`@L^FFFFE3F0(R8)`	Deferred Longword Displacement Mode. The signed displacement value is added to the contents of the register to determine an address. The content of this address is the actual address of the operand.
`(R0)[R1]`	Indexed mode. The value in the base address (R0) is added to the value in the index register [R1] multiplied by the size of the operand to determine the actual address of the operand. The base address can be any valid addressing mode, not just the register mode shown here.

Comments

Comments follow the operands and are identified by a semicolon character. This is also the comment character for the console command language itself. Any text after the semicolon ";" character is discarded regardless of contents up to the end of the line.

Memory, and The Current Position

The assembler writes it's instructions to memory using the same functions as an executing instruction. While the assembler is not actually written in VAX code, it must follow the same rules as executing code. This means that the assembler can fail due to an access violation by addressing non-existent or protected memory. The assembler addresses memory according to the settings of the MAPEN register, using either physical or virtual memory as appropriate.

The console user must set the state of virtual memory and the current processor mode (USER, KERNEL, etc.) appropriately before assembling instructions. There are pseudo-opcodes that can assist in setting these modes in the assembler source.

The assembler operates with the concept of the "current position." This is an address in memory where the *next* byte of instruction code or data will be written by the assembler. Normally this is advanced automatically, such that each instruction is stored in the next available byte immediately following the previous storage.

Some pseudo-opcodes (described later) can be used to modify the current position. There is a current position maintained for system (S0) memory and for user (P0) memory. This allows you to assembler user and kernel code at the same time into two different areas of memory.

Pseudo-Opcodes

The assembler allows you to give directives to the assembler via the use of "pseudo-opcodes" or "pseudo-ops". These appear in the source of an assembly unit in the place of an opcode, and may have operands that go with them. They can be used to store data, create entry points, modify the behavior of the assembler, or assist in assembling complex instruction such as the CASEx instructions.

The following pages describe the pseudo-opcodes and their use.

.ALIGN

The .ALIGN directive is used to adjust the "next memory location" to be aligned on an address boundary that is an even multiple of a specific size. For example, to align the next storage location on a longword boundary, the size given is 4.

```
        .ALIGN 0200
```

This example gives the size of the alignment as 0200, or 512 (decimal). This directive adjusts the next memory location to be an even page boundary; i.e. the lower 9 bits of the address are guaranteed to be zero.

Alignment can be used to specify alignment requirements to match high-level programming languages, or to ensure that data starts on a fresh page of memory, etc. This is used in the micro-kernel, for example, to separate read-only code pages from read-write data pages.

Note that for simple user-mode programming, the .ALIGN directive is rarely useful. Unlike a real VAX computer, the emulator gets no advantage in reading data on aligned boundaries, except for data that crosses page boundaries.

.ASCII
.ASCIC
.ASCID
.ASCIZ

The .ASCIx set of directives are used to store character data in the ASCII encoding set into memory. The four directives determine how the text data is encoded, if at all. Each one accepts a quoted string as it's operand, and stores the string at the current memory location as determined by the directive format.

```
        .ASCII  “Hello, World”
```

It its simples form, this directive stores the individual bytes of the string in memory starting at the current location. There is no additional transformation or additional information stored with the data.

```
        .ASCIC  “This is a test”
```

This stores a counted ASCII string in memory. This string has a 16-bit integer length field followed by the bytes of data. The length field is stored starting at the current location. A length field of zero indicates that there are no additional bytes of data. The length does not include the two bytes to store the length data itself. In the above example, the length integer is 14 (decimal), followed by the individual bytes of the string. Because the length is a 16-bit integer, the longest string that can be encoded this way is 65535 bytes long.

```
        .ASCID  “OpenVMS”
```

This example constructs a string descriptor in memory. A string descriptor is an eight-byte structure that contains type, length, and address information. String descriptors are used heavily in VMS, and by the microkernel that comes with eVAX. The string descriptor created by ASCID contains no type information (indicating a static string by default). The length field is the length of the text, and the address field points to the text data. The text data is stored in memory directly after the string descriptor structure itself. The above directive would create the following memory layout, if the current location were address 00000200.

0200:	07	00	00	00
0204:	08	02	00	00
0208:	“O”	“p”	“e”	“n”
020C:	“V”	“M”	“S”	

The length word is first, followed by the type word of zero. This is followed by the longword address (00000208 in this case). This is followed by the data at the address given.

```
        .ASCIZ  “C string”
```

The .ASCIZ type creates a null-terminate (zero-byte) string in memory, suitable for use by C runtime library routines, etc. The data is stored starting at the current location, and an extra zero byte is written to the end of the storage. The above example stores 9 bytes of data (the string plus the extra zero byte).

.BASE

The .BASE operation resets the assembler's "next location" to an explicit address. This lets you store data or instructions in the exact memory locations desired. The "next location" is kept based on what region you are assembling into. By default, the region is always P0, but can be adjusted by the .REGION pseudo-opcode. There is a separate base address for each region.

```
        .BASE   0200
```

The operand of the .BASE operation is the address (physical or virtual, depending on whether virtual memory is currently enabled) to set the next address to. In the above example, the next instruction or data will be written to address 00000200, which is the start of user memory in the P0 region.

The address may be a constant as shown above, or may be an expression whose result is interpreted as a 32-bit address.

.BYTE
.WORD
.LONG

.BLKB
.BLKW
.BLKL

These pseudo-opcodes are used to create integer storage areas. The first three, .BYTE, .WORD, and .LONG are used to store specific integer values of the given size. The second .BLKx group, creates a storage area filled with zeros that will hold a given number of the integer type.

```
        .BYTE   0F0, 05, 0, 081
```

This is an example of a specific storage operation. The data byte is byte, and the operand list is a set of byte values. If more than one is given, they must be separated by a comma. The bytes are stored in memory at the current location in the order given. The byte values may be integer constants or expressions, but the result of the expression must fit into an 8-bit value.

```
        .BLKL   ^d100
```

This is an example of a block storage operation. The data type is a longword (the “L” in the BLKx opcode). A storage area is allocated at the current location that is large enough to hold 100 (decimal) longword values. This case, that would be a block of storage 400 bytes long.

.CASE

The .CASE directive is used to assist in constructing CASEx instructions. This VAX instruction accepts a number of operands defining the value to be tested, it's base value, and the number of values that are valid. This must be followed by a table of branch displacement words in memory. The .CASE directive is used to support creating this offset table.

```
          movl b^4(ap), r0          ; can be 5, 6, or 7
        caseb r0, #5, #2           ; Note limit is z-count of selectors
        .case _f5                  ; These could all be in a single
        .case _f6                  ;     .CASE with comma-separated
        .case _f7                  ;     values.
        brb _exit                  ; land here if outside range

_f5:    movl #000f5, r0
        ret

_f6:    movl #00f6, r0
        ret

_f7:    movl #00f7, r0
        ret

_exit:  movl #0ffff, r0                 ; land here if outside range
        ret
```

This (somewhat artificial) example shows using a value that must be in the range of 5, 6, or 7 to branch to the labels _F5, _F6, or _F7 based on the the value. The CASEB instruction identifies the value to test, and lowest value, and the zero-based number of selectors. This is followed by three .CASE directives. As the commands indicate, this could be a single .CASE directive with three values separated by commas.

.CLEAR

The .CLEAR directive is used to clear a specific symbol from the symbol table, or clear the entire symbol table.

```
.CLEAR TBLSIZE
```

This will clear the symbol named TBLSIZE from the symbol table. If the symbol contains unresolved forward references, then an error occurs since the fixup information for the symbol cannot be resolved if the symbol is deleted.

```
.CLEAR
```

The .CLEAR directive given without any symbol name will delete all symbols from the user symbol table that are not marked permanent. This is equivalent to the CLEAR ALL SYMBOLS console command.

.CONSOLE

The .CONSOLE directive allows the code being assembled to make use of console commands during the assembly process.

```
        .console        set page exe$ubase to exe$uend prot=pte$k_ur
```

This example shows a SET PTE command being used to change page protections for a range of pages. The SET PTE command is a console command, but will be executed by the assembler when the .CONSOLE directive is processed. This is used in the microkernel source, for example, to set protections on read-only pages of memory after they have been assembled. The above example shows modifying a range of addresses (EXE$UBASE to EXE$UEND) so that it can be read in USER mode but only written in KERNEL mode.

Any valid console command can be used, though some will have undesirable consequences (such as a QUIT command, for example).

.END

The .END directive ends an assembly of source. If this is omitted from a source file being assembled, the end-of-file is assumed to mean the .END directive. If the directive is read from the source file, the remainder of the source file is not read at all.

```
        .END    MAIN
```

You can optionally include the name of an entry point or it’s address as the operand to the directive. If this is given, then the assembler starts execution of the program at that address as soon as assembly completes. Note that if there are unresolved symbols then the program is not executed. If a starting address is not given, then the program is not run.

.ENTRY

The .ENTRY directive creates a procedure entry point that conforms to the VAX Calling Standard. The directive accepts an entry point name and optionally a saved-register mask.

```
      .ENTRY  MAIN, ^M<R2,R3,R4>
```

This example defines an entry point symbol MAIN. This can be referenced in a console CALL command, or by a CALLS or CALLG VAX instruction. The symbol is marked as an ENTRY symbol, which means that attempts to STEP to this memory location or set a breakpoin at that location will actually use the value of MAIN+2 following the save mask area.

A VAX entry point starts with a 16-bit integer register mask. This mask defines the registers that are to be saved on the procedure frame constructed on the stack. There is one bit for each register that is to be saved. Registers are included in this list if they are used in the procedure itself, indicating that the value should be restored from the stack when the procedure completes execution.

.F_FLOAT
.D_FLOAT
.BLKD
.BLKF

These directives store floating-point numbers in memory, using either the 32-bit F_FLOAT or 64-bit D_FLOAT values. Note that as of version 1.1 of eVAX, D_FLOAT numbers are *not* fully supported by the assembler, console, or instruction set.

The F_FLOAT and D_FLOAT instruction store specific floating-point numbers into memory. The BLKD and BLKF instructions create storage areas for the given data type filled with zeroes.

```
        .F_FLOAT       1.0, -2.3
```

This directive stores two F_FLOAT floating-point numbers in memory starting at the current location. The operands must be valid floating-point constants, which are converted to the VAX 32-bit representation and stored in memory. Note that the F_FLOAT and D_FLOAT directives only accept constants; you may not use symbols or expressions. If more than one constant is given as an operand, they must be separated by commas.

```
        .BLKF          d^20
```

This creates storage for 20 (decimal) F_FLOAT values. Since each value is 32-bits long, this allocates 80 bytes of storage. The storage area is initialized with zeroes.

.IF

The .IF directive functions like the IF console command. It evaluates an expression, and if the expression results in a non-zero result, the command following the expression is processed by the assembler.

```
        .IF DEFINED( “SS$_NORMAL” ) = 0 .INCLUDE “ssdef.asm”
```

This example shows conditionally including a source file. If the symbol SS$_NORMAL is *not* defined, then the assembler is directed to execute the .INCLUDE statement, presumably to define the SS$_x symbols.

If the expression (in this case, testing the result of the DEFINED function with zero) yields a zero result, then the rest of the line is not processed.

.INCLUDE

The .INCLUDE directive causes the assembler to pause in reading from the current input source (a text file or the console) and process the contents of a specified text file. When that file executes a .END directive or the end-of-file is encountered, then processing resumes with the current file.

```
; Read in file handling routine definitions
       .INCLUDE “filedef.asm”
; Now call the open routine
       CALLS          #1, @#FILE_OPEN
```

This example shows that the code calls for the processing of an “include” file called filedef.asm which contains additional text to assemble. After reading the contents of the file, assembly resumes with the next statement following the .INCLUDE operation.

There is little limitation on how many levels deep you can nest include files; it is limited by available memory or host file open quotas. However, if you recursively include a file (i.e. file A.ASM includes B.ASM which includes A.ASM) then the emulator will fail with an out-of-memory error.

Note that the file name is shown in quotation marks. If the quotation marks are not used, the file name is converted to upper-case letters. This may be undesirable on Unix system or other systems with case-sensitive file names. Use the quotation marks to preserve the case of the filename.

.MASK

The .MASK directive is used to store an entry mask in memory. Normally this is not necessary since the .ENTRY directive will do that as well as define an entry point. However, if you need to explicitly create an entry mask in memory, use this directive.

```
MAIN:   .MASK   <R3,R4,R5>
```

This creates a 16-bit integer entry mask at the current location (which happens to be labeled MAIN). The mask has a value of 0038, with a bit set for each register number identified in the mask.

Note that this is **not** the preferred way of creating an entry point in the assembler. In this case, the disassembler and other elements of the console cannot know that this is intended to be used as a procedure entry point. As such, the disassembler will not format the memory location as a mask, and commands like STEP and SET BREAK may not function as intended when given the address of this entry mask. This is because the label for the mask location is not marked as an ENTRY symbol, but rather a LABEL symbol.

.MICROKERNEL

This pseudo-directive sets flags in the emulator to indicate that there is a microkernel present. In this case, a microkernel refers to a minimal set of VAX-native support routines that allows execution of programs on the emulator. If a “real” operating system is booted, such as NetBSD or OpenVMS, the microkernel flag is cleared again.

```
.MICROKERNEL   ; Indicate that we are building a microkernel
```

The microkernel flag controls the availability of some of the console functions. For example, the RUN command that loads and runs a VMS image file depends on the presence of the microkernel. Also, certain assembler directives such as .SCB require that the microkernel flag be set, since the .SCB directive establishes system control block values on behalf of the microkernel, and is not valid when a true VAX operating system is run.

On the other hand, using BOOT to load and run an operating system does not require a microkernel to be present.

The .MICROKERNEL directive is only valid when in KERNEL mode, and requires that a successful VMINIT command be executed (the microkernel depends on an initialized virtual memory map).

The .MICROKERNEL directive is contained in the “kernel.asm” file loaded by default when eVAX is run.

.MODE

The MODE directive is used to set the processor mode *while the code is being assembled.* Because the micro-assembler uses the standard memory read and write routines, the assembler cannot write into KW protected memory unless the processor is in kernel mode. Note that changing modes will change the contents of the PSL and the stack pointer. It is suggested that when source code requires setting the .MODE to something other than USER mode, the mode be set back to USER at the end of the assembly.

```
        .MODE KERNEL
```

This is an example of setting the mode, in this case to KERNEL mode. The parameter may be one of KERNEL, EXECUTIVE, SUPERVISOR, or USER. This controls how memory can be accessed by the assembler. The processor starts in KERNEL mode when the microkernel is built by the assembler as part of the default initialization, and the microkernel source contains a .MODE directive to set mode back to user mode at the end of the assembly.

.PRINT

The .PRINT directive is used to display output on the console during the assembly of the source. For example, this could print descriptive information about the program being assembled. The output is displayed at the time the .PRINT directive is read, not when the program is run.

```
        .PRINT  “The size of the table is “, TBLSIZE
```

This example prints the contents of a text string and the symbol TBLSIZE. It shows that at the time the .PRINT directive was encountered, the TBLSIZE variable held a particular value. Note that the symbol must be a resolved symbol since the message is printed at the time the .PRINT is processed, not when the symbol value is defined later.

If more than one operand is given to the directive, they must be separated by commas. A .PRINT directive with no arguments just prints a blank line.

.REGION

This directive selections which region (P0 or S0) that code and data is being stored into. The assembler maintains two “next memory location” pointers, one for each of the P0 and S0 regions.

```
        .REGION P0
```

This selects the P0 region. The current memory pointer (whatever it’s value) is stored in the S0 region area, and the P0 region pointer is stored in the current memory location pointer. Note that using the .BASE directive carelessly with the .REGION directive can have very undesirable consequences. There is no check to determine that the current location is in fact in a specific region before it is saved away by this directive, and restored from the other saved region. If you use .REGION, only use .BASE to reset the pointer within the current region!

.SCB

The .SCB directive is used to set values in the system control block (SCB) vector for exception and interrupt handling. This assumes that the SCBB (SCB base register) is already set up correctly. The VMINIT command will allocate space for the SCB as part of its work, so the microkernel can use the .SCB directive to specify handlers in the code.

```
        .SCB    EXC$_CONREAD, KNL$RX, KSP
```

This defines a slot in the SCB. The first parameter is the slot identifier. This can be an integer value indicating the offset in the SCB, or a symbolic designation for that offset. In this example, the offset is EXC$_CONREAD, which defines the slot for the console read interrupt, at offset X^0F8.

The second parameter is the address of the handler for this routine. This address is expressed as the virtual address, but the .SCB directive will convert this to the required physical address.

Finally, the stack that the handler runs on is specified. Handlers either run on the kernel (KSP) stack, or the interrupt (ISP) stack.

Note that the .SCB directive is only valid when building a microkernel after the VMINIT command. This directive cannot be used when running under control of a real VAX operating system.

.SET

The .SET operation allows you to set a symbol's value in the source of the assembler code

```
        .SET    TBLSIZE = . – TBLBASE
```

This is an example of creating a symbol value from an expression. The "." in the expression always means the next memory location (the one defined by the .BASE opcode, for example). This example calculates the difference in address between the next available byte and the value of TBLBASE, presumably the address of the base of a table. The resulting symbol TBLSIZE is set to the given value.

Note that TBLSIZE may have already been used in assembler source and could be a forward reference value. For example,

```
        MOVL    I^#TBLSIZE, R0
```

This instruction uses the symbol TBLSIZE to put an immediate mode value in an instruction. This instruction might occur before TBLSIZE is defined; in which case when it is defined the value of the symbol is set in the instruction(s) that used it. See the SHOW UNRESOLVED SYMBOLS command for information on determining if a symbol is defined after it is first used.

The .SET directive accepts a number of optional qualifiers after the .SET keyword that defines the flags stored with the symbol.

Flag	**Description**
/ENTRY	The symbol is an entry point, and defines the location of a 16-bit integer register mask followed by the body of the procedure code.
/LABEL	The symbol is a label, and can be used in disassembling instructions to symbolically identify a branch destination.
/PERMANENT	The symbol is marked as a permanent symbol and will not be deleted by a CLEAR ALL SYMBOLS command.

.VECTOR

The .VECTOR directive is used to define the start of code in the microkernel that contains an exception or interrupt handler. Similar to the SCB directive, this can only be used in creating a microkernel that uses the memory layout created by the VMINIT console command. You cannot use the .VECTOR directive in writing code that runs under a real native VAX operating system.

```
        .VECTOR EXC$ARITH, KSP
        PUSHL           R1
        PUSHL           #SS$_ARITH
        CALLS           @#4, EXE$HANDLER
        …
```

This example shows a sample implementation of the handler for general arithmetic traps. The .VECTOR directive takes the name of an exception code, and the stack that the handler runs on. This information is stored in the system control block SCB vector entry. The directive is immediately followed by the actual code of the handler. The SCB entry contains the address of the first byte of code following the .VECTOR directive. This allows the SCB entry and handler code to be written in a self-documenting fashion.

Architectural Overview

Emulation Internals Introduction

This section describes the internal architecture of the emulator. It **is not** a general discussion of how a VAX works. For a good start on VAX architecture issues, consider "Computer Programming and Architecture – The VAX-11" by Henry M. Levy and Richard H. Eckhouse, Jr. The definitive technical reference on the operation of the central processor in a VAX is the VAX Architecture Reference Manual, published by Digital Equipment Corporation (now part of Compaq).

This section rather outlines the structure of the emulator code, the relationship between the "native console" and the "virtual VAX" and has hints on how to add new functions to the emulator. Finally, a "Futures" section attempts to address possible directions for as-yet-unimplemented aspects of the emulator.

Before going too much further, here are some general terms that are used in this discussion.

Host	The eVAX emulator can be run on a variety of computer systems, including Macs, Windows, Linux and Unix, and Alpha/VMS. The computer where eVAX runs is considered the host computer. The host computer provides resources such as memory that can be allocated by eVAX to support the virtual VAX.
Native Console	Unlike most VAX computers, the console for the virtual VAX is not implemented as VAX instructions, etc. Instead the console exists outside the virtual VAX, though it is still part of the emulator proper. The native console has access to the registers, memory, etc. of the virtual VAX. In this way, it is most similar to the way the original VAX 11/780 was implemented, with a PDP-11 functioning as the autonomous console computer.
Physical memory	The emulator is responsible for managing the memory of the virtual VAX. Most of the physical memory of the virtual VAX is represented by a single contiguous array of bytes that the emulator allocates from the host operating system. The amount of memory allocated is controlled by the INITIALIZE command that initializes the virtual VAX. The amount of physical memory in the virtual VAX is implicitly limited by the amount of host memory that the emulator is allowed to allocate.
Virtual VAX	The VAX being emulated. Registers, memory, etc. of the virtual VAX are manifested as data structures in the emulator.

Emulator Data Structures

The original purpose of eVAX was not to emulate an entire VAX but rather to provide a powerful pseudo-code (pcode) engine that was re-usable and modular. As such, the original design built most of the execution context around a single master data structure that descibed an instance of a virtual VAX. Originally it was expected that one or more virtual VAX systems might exist simultaneously at any given time.

This however proved to be a significant performance penalty since the context of the current VAX had to be passed to virtually every routine in the emulator as a parameter. When the virtual vax data structure was made a global value that all the routines could address directly, performance increased by almost 15% on several systems. This increase was significant enough to remove the ability to run simultaneous VAX systems in the same process.

Since almost all data structures for the emulator are anchored in on master structure (struct VAX *), this section will describe in detail what each field of the structure is used for. This will be used as background information in describing the function of each of the parts of the emulator, described elsewhere in this section.

Architectural Overview

In general, the emulator can be divided into several major segments, though they all interact with each other quite a bit. Currently these are not strongly partitioned; which is to say, any part of the emulator is allowed to call any other part of the emulator. The assembler and the console share symbol table management functions. The execution module interacts with the console. And so on. A future revision may be to abstract the various elements of the emulator more clearly, such that they can be distributed, abstracted, or replaced more easily.

The following sections describe each segment of the emulator in more detail.