VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger

Bert Beander

Digital Equipment Corporation
110 Spit Brook Road,
Nashua, NH 03062

ABSTRACT

Digital Equipment Corporation's VAX-11 Debugger, usually called VAX DEBUG or simply DEBUG, is an interactive, symbolic, and multilingual debugger which runs on the VAX-11 series of computers under the VMS operating system. The following gives an overview of VAX DEBUG and examines how it solves some of the problems inherent in the design of any such debugger. Particular attention is paid to how its command language is designed, how it distinguishes between addresses and values in command input, how it solves the problem of accessing and organizing symbol table information, and how it exercises control over the user program.

1. GENERAL OVERVIEW

VAX DEBUG is interactive, symbolic, and multilingual. These attributes are central from the user's point of view and deserve a brief description. DEBUG is interactive in the sense that the user program to be debugged is run interactively at a terminal under DEBUG control. When the program starts up, DEBUG gets control first and prompts for DEBUG commands. These commands may set up breakpoints, for example, and are normally followed by the GO command, which causes the user program to start executing.

The user program then executes until a breakpoint of some sort is encountered or an exception condition arises, at which point DEBUG again gets control and prompts for more command input. The user can then examine the state of the computation, alter it if so desired, and either continue program execution or simply exit the debugging session. In short, the user interactively follows the execution of the program and can examine or alter its state anywhere along the way.

VAX DEBUG is symbolic in the sense that program locations can be referred to by their symbolic names. The contents of a variable called X is thus examined by the command EXAMINE X; the actual address of X need not be specified. Similarly, a breakpoint is set on a routine called FOO by the command SET BREAK FOO; DEBUG itself determines what absolute address goes with the symbol FOO. Output is also symbolic wherever possible. Pascal enumeration-type values are displayed as enumeration-literal names, and program addresses are displayed as routine names and listing line numbers. How symbol information is accessed is described in Section 4 below.

VAX DEBUG is strictly an object program debugger which debugs programs at run time after they have been compiled and linked. It can thus only be used to debug compiled languages, not interpreted ones. It is a multilanguage debugger, however, supporting seven languages at present: assembly language, Fortran, Bliss, Basic, Cobol, Pascal, and PL/I. Being multilingual means that it understands the following for each supported language:

- How symbol names are composed in the language. It knows how identifiers are formed and how compound names are constructed. For example, it accepts A(2)->B as valid PL/I syntax and A[2]^.B as valid Pascal syntax.
- How language expressions are interpreted. It knows what operators are allowed and what their syntax and semantics are.
- How and when type conversions are done in the language. This is part of understanding how to interpret expressions and is needed to do assignments properly.
- How values in the language are displayed. For example, how enumeration type values are displayed in Pascal and how numeric values are displayed in Cobol.
- How the language scope rules work. It knows how to look up a symbol name in a specified scope according to language rules.

Basically, symbol names and expressions entered as part of DEBUG commands are parsed and understood in source language terms, and data values are displayed in source language terms. All other capabilities, including the rest of the DEBUG command language, are language independent; they do not vary with the source language of the program being debugged.

DEBUG understands multiple languages but operates according to the rules of only one language at a time. This language is called the current language. The current language is initially set to be the source language of the main program but can be changed at any time during the debugging session with the SET LANGUAGE command. For example, the command SET LANGUAGE PASCAL causes DEBUG to interpret subsequent command input according to PASCAL rules. This allows a user to debug a program written in multiple source languages using a single debugger, namely VAX DEBUG.

There are two main reasons for having a single multilingual debugger on the VAX instead of a separate debugger for each language. One is to support one of the key goals of the VAX system architecture: to allow each user to choose the language best suited to his application and to freely use packages written in other languages. Programs and applications can be written in multiple languages because they can call each other through a common calling standard, they can read and write each other's files through a common Record Management System, and they can be debugged together using a common debugger.

The other reason is that it is a lot cheaper to write one debugger that handles seven languages than it is to write seven debuggers. Most of what a debugger has to do is in fact language independent; the code that performs these tasks therefore needs to be written (and maintained) only once, not seven times. Finding the proper partitioning between the language-dependent and -independent parts has been a difficult problem and has required several major design iterations (which will not be described further here). But once a proper partitioning has been found, one can indeed make the language-specific parts quite localized. This makes it relatively easy to add support for additional languages and it makes the multilanguage debugger a very economical choice in the multilanguage environment.

2. FEATURES AND COMMAND LANGUAGE

The VAX DEBUG command language defines the capabilities of this debugger and constitutes the user interface. It therefore merits a short exposition. The intent here is not to give a complete description of the DEBUG command language (the reference manual [2] does that), but to give a brief synopsis outlining the main features of DEBUG and illustrating the flavor of the command language. The following are some of the more important commands accepted by DEBUG:

```
EXAMINE addr-expr
DEPOSIT addr-expr = lang-expr
EVALUATE lang-expr
SET SCOPE scope-spec
SET BREAK addr-expr
SET TRACE addr-expr
SET WATCH addr-expr
STEP
SHOW CALLS
GO
```

Here command language keywords are in upper case. The exact meanings of the constructs addr-expr (address expression) and lang-expr (language expression) are discussed in Section 3 below.

The EXAMINE command retrieves and displays the contents of a specified program location. The location is typically a variable and the display is formatted according to the variable's type. This is probably the most frequently used of all commands. The DEPOSIT command stores a new value into a specified location, again usually a variable. The EVALUATE command permits expressions in the current language to be evaluated and the results displayed. Together, these three commands are the primary means whereby a user examines or alters the state of his data areas.

Since variable names are not necessarily unique across a program with many compilation units (called "modules" in DEBUG terminology) or multiple routines, the SET SCOPE command is provided. This command specifies the scope in which subsequent symbol references are to be looked up. It can also specify a sequence of scopes to be searched. A scope specification is usually a module or routine name or the name of some other lexical entity (such as a Cobol section). The scope can also be defined to be whatever lexical entity contains the next instruction to be executed; this is the default scope. Given a scope, DEBUG looks up variable names from commands in that scope using the scope rules of the current language.

For a specific symbol, the user can override the current scope setting by prefixing the symbol name with a "pathname". MOD\ROUT\X, for example, means variable X in routine ROUT in module MOD; here MOD\ROUT\ is the pathname that defines the desired scope of this specific reference to X.

VAX DEBUG offers several ways to stop program execution on specified events. The SET BREAK command causes the program to stop when a specified instruction location is reached. The SET WATCH command causes execution to stop when a specified data location is written to. The STEP command stops the program when the next instruction or the next source line is reached; it is used to single-step the program. All these events cause DEBUG to gain control and to announce the event to the user. It then solicits DEBUG commands. The SET TRACE command is a variant of SET BREAK which announces events but does not stop to solicit commands. It thus "traces" these events.

The SHOW CALLS command displays the current state of the VAX call stack. First it shows at what routine and line number the program is currently stopped. Then it shows where that routine was

called in terms of a second routine name and line number, and then where that second routine was called, and so on. In short, SHOW CALLS shows the execution state (as opposed to the data state) of the process in symbolic terms. As such, it is a frequently used command.

The remaining control function is provided by the GO command, which starts or continues program execution. Control can optionally be transferred to a specified address, thus redirecting the execution stream, but the normal case is of course to start the program where it last stopped.

One remaining feature also deserves mention: source line display. DEBUG can display the source text associated with the current program location when single-stepping the program or when stopping at some event. There are also a TYPE command that types out (displays) a specified segment of the source text and a SEARCH command that searches the source text for a specified string. The source line display capability has proved very useful (and very attractive); it frequently eliminates the need for a source listing and thus takes the user one step closer to "paperless programming".

The above synopsis omits many significant features and of course nearly all detail, but hopefully gives an idea of what VAX DEBUG offers its users. The philosophy behind the DEBUG command language differs markedly from that of the DISPEL language described by Johnson [3]. DISPEL provides a number of low-level primitives from which a user can compose whatever high-level constructs he desires to do his debugging. DEBUG's design, on the other hand, attempts to predict what high-level debugging commands a user will typically want and provides those.

Of these, the DEBUG approach is probably the more practical one for most users. Just as most users are best served by high-level programming languages, they are best served by a high-level debugging language where the most frequently needed operations are provided ready-made. Still, the DEBUG approach has its drawbacks: It leads to frequent requests for new commands to handle specialized debugging situations. An ideal debugging language should probably combine both philosophies; it should provide both high-level debugging commands and extension mechanisms whereby the user can define his own specialized commands to handle specialized debugging situations.

3. ADDRESS EXPRESSIONS AND LANGUAGE EXPRESSIONS

In a debugger, there is a need to specify both addresses and values in the command language. A DEPOSIT command, for example, must specify both a value to deposit and an address to deposit into. In a symbolic object-time debugger like VAX DEBUG, it turns out to be particularly important to provide generalized ways to specify both addresses and values. Early in DEBUG's development, this point was not fully appreciated, but it eventually led to the development of two distinct kinds of expressions: address expressions and language expressions. These notions require some explanation as they are quite important in the command language.

VAX DEBUG accepts language expressions in the EVALUATE and DEPOSIT commands and in array subscripts. Such expressions are expressions in the current source language. They are scanned and parsed according to source language rules and the operators are interpreted according to source language semantics. They are somewhat restricted (no function calls, for example) but otherwise mimic the computations the language itself would perform.

More technically, language expressions are considered to consist of three kinds of entities: primary symbols, constants, and language operators. Primary symbols are symbol names but include all record component selection, pointer dereferencing, and subscripting. (Pointer dereferencing refers to operators like -> in PL/I and ^ in Pascal.) ABC and A.B[2][3] are thus primary symbols in Pascal while A-B-C and B OF A(2,3) are primary symbols in Cobol. Primary symbols are allowed in address expressions as well and are therefore considered to be distinct entities even though the operators within them are language specific. The term "language operators" thus refers to all operators in the current language that are not parts of primary symbols. In language expressions, language operators and constants are of course parsed and interpreted according to language rules. Constants can be numeric or string constants.

Language operators operate on the current values of their operands. The expression X+Y, where X and Y are variables in the user program, thus adds the current value of X to the current value of Y.

Language expressions serve two primary purposes. Most frequently, they compute array subscripts. In a simple command such as EXAMINE X(I), the I is a language expression that defines the subscript value. This expression may of course be arbitrarily complex. Language expressions also allow DEBUG to be used as a calculator during the debugging session. The command EVALUATE X+25 computes the value of X+25 and displays it. Similarly, DEPOSIT Y = X+25 computes a value to be deposited into Y.

Being able to evaluate language expressions while debugging is obviously useful, but the user frequently needs to compute the address, as opposed to the value, of something. For this reason DEBUG accepts "address expressions" in many commands. In an address expression, the operands are primary symbols or integer constants. Primary symbols are language dependent and may include component selection, pointer dereferencing, and subscripting as indicated above. However, the expression operators are language independent and perform address, not value, computations. The following operators are accepted: prefix "@" or "." (indirection through an address), "+" (addition), "-" (negation or subtraction), "*" (multiplication), and "/" (division). Of these, multiplication and division are seldom used but are sometimes applied to constants to compute offsets from addresses.

The operators in address expressions always operate on the addresses of the symbols and the values of the constants. Thus the address expression X+2 takes the byte address of X and adds 2 to it, yielding another address. Had this been a language

expression, 2 would have been added to the value (not the address) of X. In the address expression X(I-2)+4, the I-2 is a language expression yielding the subscript value. However, the "+" is an address operator which adds 4 bytes to the byte address of array element X(I-2).

Address expressions are used in DEBUG commands that do something to locations in the user program. They are thus accepted on the EXAMINE command (which displays the contents of a specified location) and the SET BREAK command (which sets a breakpoint on a specified code location), among others. They also specify the target locations of DEPOSIT commands.

The problem being addressed here is that addresses and values must both be specified to an object-time debugger and that generalized ways of specifying each are often needed. VAX DEBUG solves this problem by accepting two distinct kinds of expressions, address expressions and language expressions, each of which is parsed and evaluated by its own distinct rules to yield its own distinct kinds of results, namely addresses or values.

4. SYMBOL TABLE ACCESS

Because DEBUG is a symbolic debugger, it must have access to the symbol tables of the user program's compilation units. This information is passed from the compiler through the linker to DEBUG as follows:

- The compiler generates Debug Symbol Table (DST) records and inserts them into the relocatable object file it produces. These records describe the program's symbol table in a language-independent way.
- The linker passes the DST records from the object file into the executable image file. The linker does relocation and global symbol resolution on the DST text but does not otherwise interpret or understand any of the information in the DST.
- DEBUG picks up the Debug Symbol Table from the executable image file at run time. Through a pointer left in the image header by the linker, the DST is mapped into DEBUG's virtual address space.

Since the Debug Symbol Table can increase the size of an object file or executable image file many-fold, its emission is optional at both compile time and link time. (The image file for a large BLISS program can easily become six to eight times larger when the DST is included.)

One of the appealing properties of this scheme is that the propagation of the symbol table information is transparent to the user; it happens automatically as part of the compilation and linking process. There are no extra symbol table files to be handled by the user (when copying object files, for example), and debugging is a well integrated part of the operating system.

In the DST, each symbol is described by a DST record. In general, each such record contains three things: the symbol name, the symbol type, and the symbol address or value. (Variables have addresses while named constants have values.) The type of the symbol can be described at various levels of complexity. A single byte will do for an atomic data type such as integer, while much more elaborate type descriptions are used for complex types such as record, array, or enumeration types. Similarly, the symbol address or value can be described by as little as five bytes (one byte of control information and 32 bits of address or value) or by arbitrarily complex specifications of how the address or value is to be computed. In addition, each DST record contains its own length so that the location of the next record can be found.

Nesting in the symbol table is indicated by begin and end records bracketing the nested symbols. Thus a Routine Begin DST record and a Routine End DST record bracket the DST records for all symbols declared within that routine. They also give the start address and length of the routine. Similarly, Block Begin and Block End records bracket the symbols declared within a lexical block and Record Begin and Record End records bracket the DST records for record (structure) components. In addition, the DST for each independently compiled and linked unit (a "module" in DEBUG terminology) must begin with a Module Begin record and end with a Module End record.

The mapping between program counter values and listing line numbers is given by DST records which specify this information in a very compact encoding. This allows DEBUG to convert line numbers to program addresses and vice versa. Other DST records specify the mapping between listing line numbers and source file records. These DST records are able to describe a source stream as coming from multiple source files, including INCLUDE files. They allow DEBUG to display the source text for a specified range of line numbers or to go directly from a program address to the corresponding source text.

The Debug Symbol Table representation was designed to be very compact and relatively easy for compilers to generate. In this it succeeds reasonably well. However, the price paid for these advantages is that symbol information in the DST cannot be easily accessed--the only way is to make a linear scan of the DST from beginning to end. If done frequently, this is unacceptably slow.

Consequently, DEBUG builds a Run-Time Symbol Table (RST) which it then uses to do symbol lookups. The RST is hashed for fast lookups and has all the links needed to show the nesting structure of the symbol table. Each RST entry also has a pointer to the corresponding DST record. All symbol accesses are made through the RST even though specific attributes of a symbol (such as its name) remain resident only in the DST.

The RST is initialized when DEBUG starts up but is actually built one module (compilation unit) at a time in response to the SET MODULE command. This command thus defines the extent of the active sym-

bol table. Ideally the user should automatically have access to all symbols in his program, but in practice it is much too time consuming to construct an RST for the entire DST if the program is very large. Hence DEBUG lets the user control how much of the symbol table to make accessible and builds an RST for only those parts.

5. CONTROLLING THE USER PROGRAM

To fulfill its purpose, an interactive debugger must have a way of controlling the user program being debugged. This means stopping execution at specified breakpoints, single-stepping over instructions or source lines, tracing the program's execution, monitoring data locations, and taking control when error conditions occur in the user program. To provide the needed control, VAX DEBUG relies primarily on the the VAX/VMS exception-handling mechanism.

The VAX-11 architecture [1] provides an exception-handling mechanism whereby exceptions detected by the hardware or signalled by the user program itself are handled by user-specified exception handlers. (A handler is simply a routine which is called when an exception has occurred. It receives the exception information through a well-defined calling sequence.) A routine can declare an exception handler dynamically by putting the handler's entry address into a reserved location in the routine's call frame on the VAX call stack. Such a handler is called a stack handler because its address is found in the call stack. This handler then receives control when an exception occurs within that routine or within any routine called by that routine. When the routine returns, its call frame is popped and the handler is no longer declared.

When there are multiple call frames on the execution stack and an exception occurs, VMS searches the call stack for a call frame with a declared exception handler. The search starts at the top of the stack (i.e., at the call frame for the most recently called routine) and proceeds toward the bottom (i.e., toward the call frame for the main program) until a handler address is found. That handler is then called. If that handler decides it cannot handle this exception, it "resignals" the exception and VMS continues the search until it finds another handler to call. When a handler is found which does want to handle the exception, it may continue program execution after taking the appropriate action. No other handler gets to see the exception in this case. A handler may also "unwind" the call stack (force the return of some routine on the stack) or terminate program execution.

When activating a user program, VMS always creates a call frame at the bottom of the stack, that is, below the call frame for the main program. In this frame it stores a pointer to a handler called the final handler. When DEBUG is not used, this handler handles every exception by printing the corresponding error message and giving the error location. After that it terminates the program. This handler is thus what guarantees that the search for a handler in the call stack will terminate. The final handler will of course never see exceptions handled by the user's handlers since they are all declared higher in the call stack.

In addition, three special handlers can be declared through system service calls: the primary, secondary, and last chance handlers. If declared, the primary handler is always called first when an exception occurs. Then the secondary handler is called and after that the search for stack handlers begins. The last chance handler is called only if the call stack itself is corrupted or destroyed.

So, how does DEBUG control the user program's execution? It does so by declaring the primary handler and the last chance handler and by changing the final handler address to point to its own final handler. This means that all exceptions will go through DEBUG's primary handler first. DEBUG can thus cause exceptions, such as breakpoint faults, which it then intercepts and never resignals to the user's handlers. However, exceptions not caused by DEBUG are resignalled by the primary handler. The user's handlers (if any) can thus handle these exception just as they would if DEBUG were not present. However, if no user handler handles a given exception, DEBUG still gets control through its final handler when the handler search reaches the very bottom of the call stack.

DEBUG thus ensures its control by surrounding the user program on both ends with its own handlers. In addition, DEBUG's last chance handler gets control if the call stack is corrupted, and an "exit handler" is declared which gets control when the user program exits (i.e., terminates). In effect, DEBUG is nothing more than a collection of exception handlers.

While the final and last chance handlers catch errors in the user program, most control functions are exercised through the primary handler. Single-stepping over instructions is accomplished by DEBUG setting the trace bit (T-bit) in the user program's Processor Status Word. When control is returned to the user program, the T-bit causes one instruction to be executed, after which a "T-bit fault" is generated. DEBUG's primary handler intercepts this exception, prints the appropriate trace information, and then solicits DEBUG commands. Stepping over source lines is handled similarly except that several instructions must typically be stepped over before stopping and announcing the trace information.

Breakpoints are implemented as follows. When the user enters the SET BREAK command to DEBUG, a BPT (breakpoint) instruction is deposited into the specified program location. A later GO command transfers control from DEBUG to the user program. When the specified location is reached, the planted BPT instruction is executed, causing a "breakpoint fault". Again VMS looks for an exception handler and DEBUG's primary handler is called first. This handler checks that DEBUG in fact planted this particular BPT instruction, and if so announces the breakpoint. It then solicits DEBUG commands from the user.

If it turns out that DEBUG had not planted this particular BPT instruction, the primary handler

resignals the breakpoint fault. This means that user handlers get a chance to handle the exception. If no user handler is declared or they all resignal too, then DEBUG's final handler eventually gets called. This handler handles all exceptions by printing the error message associated with the exception condition and then soliciting DEBUG commands from the user.

In short, DEBUG's primary handler only handles exceptions caused by DEBUG's intervention in the user process. (The user can explicitly request that it should catch all exceptions regardless of cause, but this is not the normal mode of operation.) Other exceptions are resignalled so that user handlers can handle them just as if DEBUG were not present. Only if they all fail to do so will DEBUG reclaim these exceptions through its final handler.

Another control function also illustrates this principle. The VAX architecture provides no explicit support for monitoring accesses to data locations, but it does provide a memory page protection mechanism. A user who wishes to catch all write operations to a specified data location enters a SET WATCH command, which in DEBUG terminology sets a watchpoint.

DEBUG implements the watchpoint by write-protecting the page containing the watched data location. The user program can then run until a write access to that page occurs. The write access causes a memory "access violation" fault, and DEBUG's primary handler gets control. DEBUG then determines if the faulting address points to one of the watched memory locations on the page. If so, DEBUG announces the write access and prints the current value of the location. It then removes the page write protection and single-steps over the writing instruction by setting the trace bit, giving control to the program, and catching the resulting T-bit fault, again through the primary handler. The page protection is again restored. If the data location was watched, DEBUG now prints its new value and stops to solicit DEBUG commands. If the location was not watched, the user program is simply given control again without anything being announced to the user.

If a memory access violation occurs that was not caused by DEBUG's setting a watchpoint, then it is simply resignalled by the primary handler. Such an access violation is presumably a program error. It can be handled by the user's own handlers or it can fall through to the final handler, which will announce it as a program error and then solicit DEBUG commands.

Finally, the mechanism whereby infinite loops are stopped also uses the primary handler. To stop such a loop (or execution generally), the user presses the Control/Y key, causing VMS to suspend the user process and to prompt for a VMS command. If the user then enters "DEBUG", VMS raises the "debug exception". As usual, DEBUG's primary handler is called first and handles this exception by simply soliciting DEBUG commands from the user. The user can then enter SHOW CALLS to find out where execution stopped or otherwise examine the state of the program.

To summarize, DEBUG is in effect no more than a collection of exception handlers attached to the user process. DEBUG uses the same exception-handling mechanisms available to any user program and a user could in fact write his own debugger using these mechanisms. Of course, parts of the VMS exception handling scheme were explicitly designed to accommodate the needs of the debugger. In particular, the provision for a primary handler and a final handler is essential to give DEBUG the means to completely control the user process. A way of stopping loops and transferring control to DEBUG is another special hook. However, it is interesting to note that these relatively minor enhancements to an otherwise user-oriented exception-handling mechanism are enough to meet the full control needs of an interactive symbolic debugger.

6. CONCLUSION

In conclusion, we have seen that VAX DEBUG is an interactive, symbolic, and multilingual debugger and we have discussed how VAX DEBUG solves certain problems such a debugger must solve. We noted what is meant by being "interactive" and "symbolic" (namely the obvious meanings) and also what is meant in this case by being "multilingual", namely that programs written in seven separate source languages can be debugged with the same debugger. For each source language, DEBUG interprets symbol references and language expressions in source language terms and displays values as appropriate for that language. Also, the current source language can be changed at any time during the debugging session.

It was noted that DEBUG's command language constitutes a high-level, run-time debugging language that aims to provide all the high-level debugging capabilities a user typically needs. The command language has commands to examine and alter the current state of the computation, to show the call stack, to display program sources, and to interrupt and control the execution stream. It does not necessarily cater to every specialized situation, however. It was also noted that a symbolic object-time debugger needs generalized ways of specifying both addresses and values; in the VAX DEBUG command language, this problem is solved by accepting two distinct kinds of expressions, namely address expressions and language expressions.

Any symbolic debugger must have access to the symbol tables of the program being debugged. VAX DEBUG solves this problem by having the compilers output a language-independent Debug Symbol Table (DST), which is passed to DEBUG via the object and executable image files. The DST representation was chosen for compactness (to conserve file space) and ease of generation (to simplify compilers); ease of access was sacrificed, however. As a result, DEBUG must build a Run-Time Symbol Table (RST) which has the proper structure to show nesting relationships between symbols and to allow efficient symbol table access.

Finally, we discussed how VAX DEBUG controls the user program being debugged. VAX DEBUG achieves this control through the normal VAX/VMS exception-handling mechanism. By design this mechanism has

certain features which allow DEBUG to exercise the necessary control. The most important of these features are the provisions for a primary and a final exception handler. These handlers allow DEBUG to intercept exceptions both before and after the user program gets to see them. This, plus some other facilities, is enough to allow DEBUG to completely control the execution of the user program being debugged.

REFERENCES

1. -, VAX Architecture Handbook, Digital Equipment Corporation, 1981.

2. -, VAX-11 Symbolic Debugger Reference Manual, AA-D026D-TE, Digital Equipment Corporation, 1982.

3. Johnson, Mark Scott, "DISPEL: A Run-Time Debugging Language", Computer Languages, Vol 6, pp 79-94, 1981.