subject:

STEP: A Stack-based Controller for HDM Tap Managers

date: **July 29, 1997**

from: **Dale E. Parson**
Org. 538862000
PAI-830 55E-218
610-712-3365
dparson@lucent.com

ESST (Embedded System Software Tools) License

1. Introduction: STEP and the need for a lightweight interpreter in HDM

This is an overview document for STEP (Stack-based Threaded Emulated Processor), a Forth-like virtual machine [1] written in C++. STEP design and implementation came about as a result of the Modeling / HDM engineers' experience with both networked HDM and the interpreted language Tcl [2]. STEP's run-time machinery is a natural extension of the run-time networked-function-call machinery of networked HDM. STEP is in alpha form in the TEEM build and test environment. It is undergoing regression test construction and initial application to HDM.

The STEP VM (virtual machine) is a small integer-code interpreter, similar in concept (but simpler in architecture and implementation) to earlier byte-code interpreters such as UCSD P-code for Pascal and Sun's Java VM. A run-time interpreter is appropriate for HDM because the addition of new processor types requires the addition of small amounts of processor-specific code to the target HDM environment. In cmd16xx's networked HDM implementation the addition of new processor core variations results at times in the addition of new networked HDM functions. Each new networked function requires deployment of a new HDM target server executable file. Any new cmd16xx host process is incompatible with any prior networked HDM target server because the target server lacks new functions. New target servers may be incompatible with prior cmd16xx host processes because of bug fixes or changes in networked function parameters. The result of these incompatibilities has been a proliferation of target server deployments, with some customers running multiple target servers to serve multiple DSP1600 variant processors.

The STEP VM solves the problem of target server extension because it is an interpreter that can receive its processor-specific code at run time. The target server process is generic and extensible. The HDM host process downloads processor-specific code for interpretation. The 1600/16k HDM engineering team discussed the use of Tcl as a run-time interpreter for the target HDM environment, but we did not use Tcl for two major reasons. First, Sun does not support Tcl on MSDOS or Win16 platforms. We do not wish to supported a local version of Tcl. Second, Tcl incurs run-time memory allocation and string handling overhead that is inappropriate for HDM. What we really need is a fast way to download not only sequences of instructions into HDM—networked HDM already does this for cmd16xx—but also a way to download conditional tests that can select sequences of instructions to run within the target environment. Viewed from this perspective, our need is a small addition of conditional logic to the basic command-sequencing capabilities of networked HDM.

Threaded intermediate code exactly satisfies this need [3]. The varieties of threaded code are designed to be interpreted at run time with a minimum of interpreter overhead. Typically a few arithmetic operations are all that is necessary to decode and execute a threaded operation.

Forth is a premier example of a threaded virtual machine that has been used within embedded applications for over twenty years [1]. The boot ROM within Sun workstations is written in Forth. We contacted FORTH, Inc. via the Internet and found that they are still providing tools for robust embedded system solutions a quarter century after their founding [4]. They have tools for developing extremely lean Forth-based control systems on MSDOS. TargetView HDM's target server will run on MSDOS.

Forth, Inc.'s tools have two limitations for networked HDM support. First, they do not provide for downloading of threaded code from a remote host. All text-to-threaded code translation occurs in the run-time environment. This interleaving of the compilation and run-time systems is typical of

Forth's approach to program interpretation, but run-time compilation is inappropriate for HDM. Forth, Inc. indicated that they support a custom set of tools for one customer that includes remote compilation of threaded code, but we were concerned with the costs in time and money of buying and integrating a non-standard product.

The second limitation consists of the fact that Forth, Inc.'s support products do not include a networked cross-debugger capable of debugging Forth programs running on a disk-less, terminal-less TargetView MSDOS box from a UNIX or Win32 debugging environment. Debugging embedded HDM Forth programs would be difficult.

Enter TEEM! Given some expertise in threaded code techniques [5], I estimated that it would take me about a week to get a Forth-like virtual machine up and running and another week to integrate it into TEEM's debugging environment. STEP has taken almost exactly two weeks (with some late nights) to bring to this state, including creation of a compiler, linker and loader for STEP's Forth-like language. This source code generation system (SGS) is written in Tcl and runs in the UNIX and Win32 host environments. It creates integer threaded code that a host process can download into the target STEP VM.

Currently STEP supports loading, execution and debugging in a stand-alone environment. When cmd16xx's networked HDM is enhanced and added to TEEM, STEP will support networked loading and debugging by using networked HDM machinery from TEEM. In fact the STEP VM appears to TEEM as a remote hardware processor. It has provided the first means for testing the upper levels of the TEEM-to-HDM interface by serving as a test HDM machine. Thanks to TEEM we have a command-line and graphical STEP debugger that supports conventional debugging and that will support networked cross-debugging.

Three sections follow. Section 2 outlines the operation of the STEP virtual machine. Section 3 describes STEP's Forth-like programming language. Section 4 describes STEP's debugging environment under TEEM.

2. STEP: A Forth-like virtual machine

2.1. STEP memory and registers

Figure 1 illustrates the STEP virtual processor. The **data stack**, **thread stack**, **dictionary**, and **commands buffer** are areas of memory. Registers include **ds**, **ts**, **dictused**, **cmdsused**, **ip**, **opcode** and **state**.

Forth-like machines use two stacks, the **data stack** and the **thread stack**. The **control stack** is another name for the thread stack. The data stack houses temporary data, while the thread stack houses return addresses for STEP subroutine calls. This two-stack arrangement differs from procedural languages that intermix subroutine arguments, local variables and return addresses on a single stack. The data stack can pass multiple values into a subroutine, provide temporary storage for that subroutine, and it can pass return values out of a subroutine. Returning from a subroutine pops an address off of the thread stack, but returning does not affect data on the data stack.

Early Forth implementations worked with a single, 16-bit integer data type. Later Forths added support for various integer sizes as well as floating-point data and operations. STEP uses a single, signed integer data type #defined as STEPWORD in awmpstep.h as its single native data type. STEPWORD is currently defined as a C long integer. A single type is sufficient for HDM control. A 32-bit long meets the needs of all anticipated processors accessed by HDM, and it provides

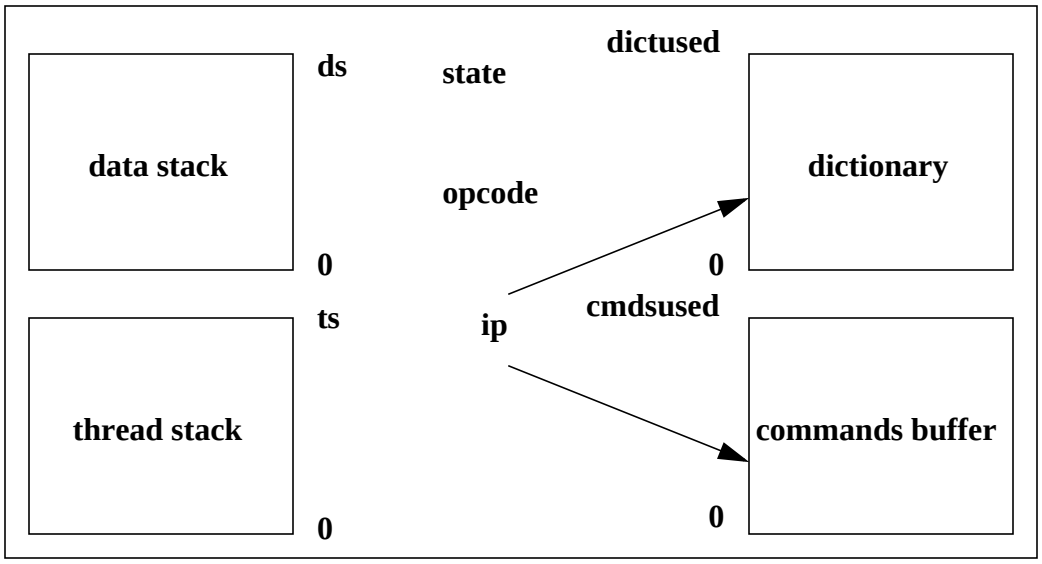

Figure 1: The Forth-like STEP virtual processor

portability between STEP processes compiled for 32-bit execution (UNIX and Win32) and 16-bit execution (Win16 and MSDOS HDM servers).

The **data stack** is an array of STEPWORD. The **ds** register is in fact a pointer to a STEPWORD in the STEP implementation. It points to the top of the data stack. Viewed from outside STEP ds appears as a STEPWORD offset into the data stack, but it has a pointer implementation for efficiency. A virtual value of -1 for ds signifies an empty stack, while values 0 through n-1 signify offsets into the data stack.

The **thread stack** houses return addresses for subroutine calls. A return address consists of a copy of a value for the **ip** (instruction pointer) register. This register can point into either the **dictionary** or the **commands buffer**. The dictionary is a standard Forth region that houses both user-defined subroutines as well as static variables and arrays. When ip points into the dictionary it serves as a program counter. When a STEP subroutine calls another STEP subroutine in the dictionary, the STEP virtual machine pushes ip to the thread stack. Executing a **return** operation pops ip's value. The **ts** register gives the top of the thread stack. It is a STEPWORD** that appears to a STEP user as a STEPWORD offset into the thread stack.

The **commands buffer** is not a conventional Forth region. Forth as an interpreter allows execution of "immediate mode" commands as a user enters them. Forth's interleaving of incremental compilation of text with threaded code execution gives support for immediate execution via the dictionary. STEP on the other hand separates compilation from execution. STEP supports compilation only on the Tcl-capable UNIX and Win32 platforms. In this compile-load-and-execute environment, the commands buffer provides support for an "immediate mode" of execution. Compilation directs subroutine bodies into the dictionary and immediate commands into the commands buffer. STEP's load function can load both the dictionary and commands buffer, and it can execute the latter immediately upon load. In this way STEP can appear as a conventional interpreter.

The dictionary and the commands buffer are arrays of STEPWORD. Registers **dictused** and

cmdsused give the number of words currently in use in these two regions. Both data and opcodes take the form of STEPWORD. Register ip, on the other hand, is a pointer to a STEPWORD in the dictionary or commands buffer. When it points within the dictionary it appears outside STEP as a STEPWORD offset into the dictionary. When it points within the commands buffer it appears as a value of -1 outside STEP. Typically the commands buffer will contain as little as a call to the "main" subroutine within the dictionary, where most of the execution occurs. Register ip starts out pointing within the commands buffer, but any subroutine call from the commands buffer pushes a commands buffer address to the thread stack and points ip within the dictionary. There is no way to call out of the dictionary.

The **opcode** register is a STEPWORD that houses the current opcode during STEP's execute phase. The **state** register records a STEP machine's state. STEP has the following states.

- **s_ready**: The STEP VM is freshly constructed (as a C++ object) and perhaps loaded with an object file, but it has not yet run. STEP's **awmpStep::reset()** function places a STEP object back into the s_ready state.
- **s_run**: STEP is executing code in the commands buffer or dictionary. A STEP client can initiate execution using STEP's binary **awmpStep::load()** function.
- **s_pause**: A running STEP program has executed the **pause** instruction, returning control to STEP's caller. STEP provides support for cooperative multitasking via this pause mechanism. Each STEP object houses its own memory regions and registers, so each STEP object can be treated as a coroutine. A client can call **awmpStep::resume()** to resume execution of a paused STEP program.
- **s_bpt**: A running STEP program has executed the **breakpoint** instruction, returning control to STEP's caller. STEP supports ip-based breakpoints via this breakpoint instruction. STEP's **awmpStep::setbp(addr)** function saves the opcode at addr in a look-aside table and inserts a breakpoint opcode at that location. STEP's **awmpStep::clrbp(addr)** function restores the opcode and deletes the table entry. Executing a breakpoint instruction causes breakpoint entry by setting state to s_bpt and returning to the client similar to s_pause. When **awmpStep::resume()** resumes execution, the re-executed breakpoint command retrieves the original opcode from the look-aside table and forces execution of that operation. From outside STEP s_bpt appears to work the same as s_pause. The breakpoint opcode gives STEP a "hardware breakpoint" capability from TEEM's perspective. STEP is not constructed from TEEM modeling objects because we do not support them for MSDOS. STEP appears as a hardware processor with ip-based hardware breakpoints to TEEM.
- **s_halt**: A running STEP program has executed the **halt** instruction, or an error occurred within STEP, or STEP reached the end of execution of the current program. Functions **awmpStep::load()** and **awmpStep::resume()** return 0 on error-free execution and a non-0 error flag on an error. The STEP VM halt instruction pops its return value from the data stack, allowing STEP programs to enter s_halt state for application-determined normal or abnormal termination. A client must call **awmpStep::reset()** or load a new program with **awmpStep::load()** before the STEP VM can become ready to execute again. The STEP VM can re-execute a program at any time via **awmpStep::reset()** and **awmpStep::resume()**.
- **s_block**: A running STEP program has called **awmpStep::block()**, typically from within an IO primitive operation. (There is no block opcode available inside the STEP VM.) STEP

will not execute from within state s_block. An external caller (for example an IO service provider such as the chain manager for HDM) can call **awmpStep::unblock()**. Unblock() moves a STEP object from state s_block to state s_pause, making it available for **awmpStep::resume()**.

2.2. STEP execution of threaded code

STEP opcodes take the form of STEPWORD values. Forth opcodes on the other hand take the form of direct pointers to machine code. A Forth opcode decoder executes an immediate jump to the code that implements a primitive Forth operation. The primitive operation jumps back to the opcode decoder. Constructing this run-time mechanism is possible because Forth is built incrementally from assembly language and Forth. On the other hand C does not support pointers to arbitrary code locations. Both because of this limitation of C and C++, and because we wish to support execution of STEP programs across heterogeneous platforms for networked HDM, STEP implements opcodes as simple integers (STEPWORD). This implementation is slightly less efficient than Forth, but it is about as efficient as a C-implemented run-time interpreter can be. STEP's compiler can create opcodes on one platform and a networked load function can download them to another platform for execution.

STEP assigns a number of consecutive integers, starting at 0, as opcodes for primitive operations. Files awmpstep.h and stepexe.h document these basic operations, known as **primitives** in Forth terminology. A system design can extend the set of primitives by deriving a STEP-child class from awmpStep that extends STEP's primitives with application-specific primitives. Files awmpstep.h, stepexe.h and stepsgs.tcl document primitive-supplied extension.

The second means for extending the STEP machine is via the definition of **STEP subroutines**. An opcode known as a **secondary** in Forth identifies a threaded subroutine. A STEP secondary opcode is nothing other than an offset into the dictionary that points to the start of a subroutine. STEP identifies a secondary opcode by setting the topmost bit to 1. There is no need for an explicit "call" to a subroutine in STEP; the topmost bit triggers the call.

STEP's run-time fetch-and-decode logic in file stepexe.h looks like this:

```
while (state == s_run) {
        opcode = *ip++ ;
        switch (opcode) {
                case STEP_ADD:          // add is a primitive
                        tmp1 = *ds-- ;  // pop augend into tmp1
                        *ds += tmp1 ;   // replace addend with sum
                        break ;
                // other primitive opcodes work similarly
                default:
                        if ((opcode & TOPBIT) != 0 &&
                                        (opcode ^ TOPBIT) < dictused) {
                                ++ts ;
                                *ts = ip ;              // push ip to thread stack
                                // point ip to subroutine
```

```
                    ip = &dict[opcode ^ TOPBIT] ;
                } else {
                    HALT with an error code
                }
                break ;
        }}
```

This C++ switch statement takes the place of pointer chasing in conventional Forth implementations. An opcode is a conceptual pointer, and the switch statement is the mechanism by which control follows this conceptual pointer to its machine code. Most primitives fetch and store data using the data stack. Primitives that include immediate data in the instruction stream access it using ip. Application-specific IO primitives can bring data into STEP's stack and emit data from STEP's stack.

A 1 value in the TOPBIT position identifies a secondary. A secondary is the compiled body of a STEP subroutine. The opcode interpreter calls this subroutine by pushing the current ip value to the thread stack and then pointing ip to the subroutine. Now interpreter control returns to the top of the while loop, and unless the primitive has taken the STEP VM out of the s_run state, execution continues with the next opcode. In the case of a subroutine call ip fetches the next opcode from the called subroutine.

Subroutine completion ends with the execution of the return primitive:

```
case STEP_RETURN:
    ip = *ts-- ;
    break ;
```

This interpretation mechanism is about the smallest possible for a C-implemented run-time virtual machine interpreter.

3. Forth and STEP programming languages

3.1. Forth and STEP are postfix machines

Because of the close association of data with the data stack, and because of the desire to minimize interpreter overhead, Forth uses postfix notation. This Forth expression:

```
3 4 +
```

generates the following three operations:

- Push a constant of 3 to the stack. The STEP compiler creates a push of an in-line constant using STEP's constant primitive:

```
case STEP_CONST:
    ++ds ;
    *ds = *ip++ ;          // push an in-line constant
    break ;
```

- Push a constant of 4 to the stack.
- Pop the top two elements and push their sum, as seen for STEP_ADD above.

Forth was designed as a tight, small text interpreter + run-time interpreter for embedded control applications on tiny computers, and now small microcontrollers. Postfix notation helps to minimize the machine space and time spent compiling at the expense of the Forth programmer. (Forth enthusiasts argue in favor of postfix.) STEP has stayed with postfix to minimize the time needed to create a compiler, and also to avoid the creation of a new language. STEP's language is basically HP calculator expressions + control constructs + static variable allocation constructs. Section 3.2 discusses STEP lexical issues. Section 3.3 looks at symbolic constants and variable allocation constructs, Section 3.4 at control constructs, Section 3.5 at public STEP primitives for postfix expressions, and Section 3.6 at private STEP primitives used only by the compiler. Section 3.7 discusses the STEP source interpreter and compiler.

3.2. STEP lexical issues

STEP lexemes are non-empty, white space-delimited strings within a STEP program.

Lexemes subdivide into reserved words and non-reserved words. Any non-reserved word is available for use as a constant, variable, array or procedure name.

Reserved words consist of compiler keywords, public primitives and private primitives. STEP programmers may use all reserved words except private primitives. Programmers may not redefine reserved words.

STEP compiles a quoted string surrounded in double quotes (e.g.,: "a quoted string") as a 0-terminated sequence of ASCII integers in the dictionary. Substrings "\n" and "\t" translate into **newline** and **tab** characters respectively. In addition to compilation of a quoted string into the dictionary, the appearance of a quoted string results in a push of the string's dictionary address to the data stack, similar to the appearance of an **array** name (see next section). STEP's **prints** primitive (below) can print such a string. Quoted strings and **prints** are useful mostly for printing debugging information.

The appearance of a decimal literal, a 0x-prefixed hexadecimal literal or a 0-prefixed octal literal into a STEP program results in a push of that value to the data stack. The STEP compiler compiles a literal character such as 'a' into a push of its ASCII integer code. Octal sequences such as '\007' or '\0' are also legal.

STEP treats a sequence of tokens preceded by "[" and followed by "]" as a call to a **Tcl** interpreter embedded within the STEP compiler. Tcl commands embedded within "[]" delimiters execute immediately, and the STEP compiler glues a string returned from Tcl into the STEP program for subsequent parsing. For example the following immediate commands to STEP (commands are underlined) return the accompanying output (not underlined):

step> [set a {"this is a Tcl escape"}] prints crlf ...
this is a Tcl escape
step> [set a] prints crlf ...
this is a Tcl escape
step> [set a 3] printi crlf ...
3
step> [set a] printi crlf ...
3

In the above example **prints** prints a compiled string whose dictionary address is on top of the stack, while **printi** prints an integer that is on top of the stack, and **crlf** prints a newline character. Tcl serves as a powerful macro language that can do everything from maintaining and calculating constant-valued expressions via **expr**, to embedding generators for STEP programs.

The sequence **...** (three dots, seen in the above example) is significant only when STEP runs as a text interpreter that reads **stdin** (standard input) from a terminal. In that environment **...** signals the text reader function to pass its input to the STEP compiler. Everything typed before **...** passes to the STEP compiler, and the STEP compiler compiles the program and causes immediate execution. The **...** string does not go to the STEP compiler. When compiling a program contained in a non-terminal file, the STEP reader simply reads the entire file and sends it to the compiler. The **...** delimiter should not appear within a program contained in a file.

The unquoted // sequence (pair of slashes) begins a comment as it does in C++. The comment runs to the end of its line.

Programmer-defined names for constants, variables and procs (below) may not contain the two-character "@@" substring (pair of *at* signs). The compiler uses "variable@@proc" to denote a variable within a proc in symbol table information passed to the TEEM debugger.

3.3. STEP constants, static variables and arrays

The **constant** command defines a numeric constant:

- constant *constant-name constant-value*

where *constant-name* is a programmer-defined name for the constant, and *constant-value* is a numeric constant, ASCII character literal or previously defined constant. Here are some examples:

```
step> constant three 3
step> constant c3 three
step> 3 three c3 printi printi printi crlf ...
333
```

A constant defined within a **proc** (procedure, see below) is local to that proc, while a constant defined outside any proc is a global constant. The constant definition does not allocate any code in the dictionary, but a constant invocation generates code to push that constant to the data stack at run time. Proc-local constant definitions can redefine non-reserved words.

The **variable** command allocates and names a STEPWORD storage location in the dictionary:

- variable *variable-name*

where *variable-name* is a programmer-defined name for that storage location. A variable defined within a **proc** (procedure, see below) is local to that proc, while a variable defined outside any proc is a global variable. The variable definition allocates one STEPWORD in the dictionary. A variable invocation generates code to push the dictionary address of that variable to the data stack at run time. Global and proc-local variable definitions can redefine non-reserved words. Global and proc-local variables are static variables in the sense that they are allocated in the persistent dictionary, not on the data stack.

The dictionary address pushed by a variable invocation is useful to the **@f** (fetch) and **@s** (store) commands. Fetch replaces a dictionary address on top of the data stack with the contents of that

dictionary location. Store expects a dictionary address on top of the stack, with a STEPWORD value beneath it; store stores the value at the address. Here are some examples that include some compiler information messages:

```
step> variable x ...
stepCompile stdin, 1 words to dictionary in this pass, 1 total
step> variable y ...
stepCompile stdin, 1 words to dictionary in this pass, 2 total
step> x printi crlf // prints address of x in dictionary ...
0
step> y printi crlf // prints address of y ...
1
step> 3 x @s // store 3 at x ...
step> 4 y @s // store 4 at y ...
step> x @f printi crlf // fetch and print x ...
3
step> y @f printi crlf // fetch and print y ...
4
```

An **array** is a multiple-element variable. An array declaration allocates storage for a sequence of STEPWORDS in the dictionary; invoking an array name pushes the dictionary address of the first element of the array to the data stack:

- array *array-name constant-value*

where *array-name* is a programmer-defined name for the array, and *constant-value* is the number of elements in the array. Here are some examples:

```
step> constant ele 3
step> array a3 ele ...
stepCompile stdin, 3 words to dictionary in this pass, 3 total
step> array b50 50 ...
stepCompile stdin, 50 words to dictionary in this pass, 53 total
step> -1 b50 @s // b50[0] = -1 ...
step> -100 b50 49 + @s // b50[49] = -100 ...
step> b50 @f printi crlf // fetch and print b50[0] ...
-1
step> b50 49 + @f printi crlf // fetch and print b50[49] ...
-100
```

The dictionary address pushed by invoking “b50” is the STEPWORD offset into the dictionary of the first element (i.e., index 0 out of 0 through 49) of b50. The address of the final element, element 49, is gotten by adding 49 to the base address of b50. Thus array indexing is synonymous with address arithmetic.

A programmer can construct an initialized array by using the **table - endtable** construct:

- table *array-name value-0 value-1 ... value-n* endtable

where each value value-0 through value-n is a STEPWORD value. For example:

```
step> constant two 2
step> table mytable -1 two -3 100 -100 endtable ...
stepCompile stdin, 5 words to dictionary in this pass, 5 total
step> mytable @f printi crlf // print mytable[0] ...
-1
step> mytable 4 + @f printi crlf // print mytable[4] ...
-100
step> mytable 1 + @f printi crlf // print two at mytable[1] ...
2
```

An array or table defined within a **proc** (procedure, see below) is local to that proc, while an array or table defined outside any proc is global.

Here is a way to determine the number of elements in a global table without hard-coding them:

```
step> table tentab 10 20 30 40 50 60 70 80 90 100 endtable
step> variable tabsize
step> tabsize tentab - // get number of elements in tentab
step> tabsize @s // store that count in tabsize
step> tabsize @f printi crlf // print tabsize ...
stepCompile stdin, 11 words to dictionary in this pass, 16 total
10
```

Variable tabsize is allocated immediately after the previous dictionary allocation, in this case that of tentab. Subtracting the address of tentab from that of tabsize gives the number of elements in tentab. **This trick works only for global arrays and tables.** The current STEP compiler generates **goto** instructions around variables and arrays defined within procs in order to simplify compilation; proc operations and proc variables and arrays are intermixed in the dictionary. These goto instructions would change the above calculations of dictionary addresses, yielding an incorrect result for size.

Constant, variable, array and table names must be defined before they are used.

3.4. STEP control constructs

STEP contains three private control primitives, **goto**, **goto0** and **gotox**, documented in the STEP private primitives section below. The compiler generates these primitive opcodes, but the STEP programmer cannot access them directly. The STEP programmer specifies control using the control constructs of this section. All control constructs except proc-endproc may be nested.

3.4.1. Keywords proc, endproc and return define secondaries

The outline of a **proc** definition takes this form:

```
proc PROCNAME     // bundles threaded calls as a new, secondary opcode
        // body of the proc -- these invoke primitives and secondaries
endproc
```

Procs may contain local constant, variable and array declarations. A proc may not contain nested proc definitions. **All other control constructs must appear within proc bodies.** A proc may call itself, but no indirect form of recursion is supported. A proc name must be defined before it is used.

The programmer may issue a **return** statement anywhere within a proc. The required **endproc** generates a default return instruction at the end of the proc body.

3.4.2. Keywords if-then-else-endif specify selection

The “if” construct is similar to its C counterpart:

```
if      // control keywords are syntactic sugar
        test-code // leave a 0 or non-0 on top of stack
then
        success // execute code here on non-0 test-code-result
else
        failure // execute code here on 0 test-code-result
endif
```

The section between “if” and “then” consists of code that leaves a 0 or non-0 value on the data stack. Keyword “then” pops this value, executing the “then” part if the test value is non-0, executing the “else” part if the test value is 0. The “else” part is optional.

3.4.3. Keywords while-do-endwhile, dountil-until-enduntil, break and continue: loops

The first looping construct executes its test at the top:

```
while
        test-code
do
        loop-code // execute loop while test-code-result is non-0
endwhile
```

The *test-code* leaves a 0 or non-0 value on the data stack. Keyword **do** pops and tests this value, executing the loop on a non-0 value.

The second looping construct executes its test at the bottom:

```
dountil
        loop-code // execute loop until test-code-result is non-0
until
        test-code // note that non-0 test result EXITS the loop
enduntil
```

This construct is more like the corresponding Pascal construct than the corresponding C construct. It exits the loop on a non-0 test result. Here *test-code* leaves a value on the data stack that **enduntil**

pops and tests.

Keyword **break** exits its inner loop immediately, while **continue** jumps to the **while** or **until** section of its inner loop, executing the test code. These words may appear in the *loop-code* section of a loop. A proc **return** may also exit a loop by exiting its proc.

3.4.4. Keywords switch-case-endcase support table-driven proc calls

STEP's limited switch construct selects from a table of proc addresses:

```
switch
        index-code // generate an index, value 0 through n-1, onto stack
case
        proc0 // runs if 0 found on stack
        proc1 // runs if 1 found on stack
        ...
        procn-1 // runs if n-1 OR anything else found on stack
endcase
```

The *index-code* pushes an index offset into the subsequent table of procs.

3.4.5. Primitives pause and halt affect control

Two public STEP primitive operations, **pause** and **halt**, affect program control. These first appeared in Section 2.1.

The **pause** instruction causes **awmpStep::load()** or **awmpStep::resume()** to return a value of 0 (no error), putting the STEP VM in the **s_pause** state. A STEP client can resume execution by calling **awmpStep::resume()**. **Pause** supports cooperative multitasking, where a scheduler calls **awmpStep::resume()** for multiple STEP objects, one at a time.

The **halt** instruction pops the top of the data stack, returning that value from **awmpStep::load()** or **awmpStep::resume()**, and putting the STEP VM in the **s_halt** state. A STEP client must call **awmpStep::reset()** or **awmpStep::load()** before it can resume execution of a halted STEP VM. The sequence "0 halt" signifies a successful termination of a STEP program, while any "non-0 halt" sequence returns the application-specific, non-0 exception code.

3.5. STEP public primitives

This section documents STEP primitive operations that STEP programs can invoke. Each entry (row) in the primitive tables below contains four fields (columns):

- name of the primitive as entered into a STEP program
- equivalent C++ #define from awmpstep.h
- top-of-stack contents before the operation, with top-of-stack to the right
- top-of-stack contents after the operation, with top-of-stack to the right

The pause and halt instructions discussed in the last section would appear like this:

Table 1: control primitives

primitive	C++ name	stack before	stack after
pause	STEP_PAUSE		
halt	STEP_HALT	halt_status	

The binary arithmetic primitives work comparable to their C counterparts on 2 STEPWORD operands. Each binary primitive expects to find operands *x* and *y* on the data stack, with *y* on top of the stack and *x* beneath it. Each pops the two operands, evaluates the equivalent of the C expression "*x op y*" where *op* is the operator; the operation pushes the result to the data stack.

Table 2: binary arithmetic primitives

primitive	C++ name	stack before	stack after
+	STEP_ADD	x y	(x+y)
-	STEP_SUB	x y	(x-y)
*	STEP_MULT	x y	(x*y)
/	STEP_DIV	x y	(x/y)
%	STEP_MOD	x y	(x%y)
&	STEP_BITAND	x y	(x&y)
|	STEP_BITOR	x y	(x|y)
^	STEP_BITXOR	x y	(x^y)
<<	STEP_SHL	x y	(x<<y)
>>	STEP_SHR	x y	(x>>y)
&&	STEP_LOGAND	x y	(x&&y)
||	STEP_LOGOR	x y	(x||y)

Table 2: binary arithmetic primitives

primitive	C++ name	stack before	stack after
==	STEP_EQ	x y	(x==y)
!=	STEP_NEQ	x y	(x!=y)
<	STEP_LT	x y	(x<y)
>	STEP_GT	x y	(x>y)
<=	STEP_LE	x y	(x<=y)
>=	STEP_GE	x y	(x>=y)

Rotate primitives **rrot** and **lrot** are STEPWORD-size right-rotate and left-rotate operations. If STEPWORD is 16 bits wide then these are the same as **rrot16** and **lrot16**. For a 32 bit STEPWORD they are 32-bit rotates. **rrot16** and **lrot16** are dedicated 16 bit rotates. For a 16-bit STEPWORD they use the whole word; for a 32-bit STEPWORD they rotate bit 15 out of/into bit 0, zeroing any upper bits above bit 15.

Table 3: rotate primitives

primitive	C++ name	stack before	stack after
rrot	STEP_RROT	x bits	(right-rotate x by bits positions)
lrot	STEP_LROT	x bits	(left-rotate x by bits positions)
rrot16	STEP_RROT16	x bits	(right-rotate x by bits positions, only use bottom 16 bits, zero top bits if any)
lrot16	STEP_LROT16	x bits	(left-rotate x by bits positions, only use bottom 16 bits, zero top bits if any)

There are two C-like unary arithmetic primitives, **minus** and ~. The former computes unary - while the latter computes the one's complement. There are also unary operations **sign** and **abs**. Primitive **sign** returns the sign bit of its operand (i.e., 1 means negative, 0 means 0 or positive), and **abs**

returns absolute value.

Table 4: unary arithmetic primitives

primitive	C++ name	stack before	stack after
minus	STEP_MINUS	x	(-x)
~	STEP_COMPLEMENT	x	(~x)
sign	STEP_SIGN	x	(x<0)
abs	STEP_ABS	x	((x<0)?-x:x)

There is a set of “peephole optimized” unary primitives that are really binary operators with a constant second operand collapsed into the operation. For example “0==” compares the top of the stack to 0, as though two distinct operations “0” and “==” appeared. All of these eliminate one of two primitive fetch-decode-execute steps for common binary operations.

Table 5: optimized constant binary arithmetic primitives

primitive	C++ name	stack before	stack after
0==	STEP_ZEQ	x	(x==0)
0!=	STEP_ZNEQ	x	(x!=0)
0<	STEP_ZLT	x	(x<0)
0>	STEP_ZGT	x	(x>0)
0<=	STEP_ZLE	x	(x<=0)
0>=	STEP_ZGE	x	(x>=0)
1+	STEP_ADD1	x	(x+1)
1-	STEP_SUB1	x	(x-1)
2+	STEP_ADD2	x	(x+2)
2-	STEP_SUB2	x	(x-2)

Table 5: optimized constant binary arithmetic primitives

primitive	C++ name	stack before	stack after
2*	STEP_MULT2	x	(x*2)
2/	STEP_DIV2	x	(x/2)
1<<	STEP_SHL1	x	(x<<1)
1>>	STEP_SHR1	x	(x>>1)

The data stack manipulation primitives duplicate, rearrange and drop elements on the data stack. Fetch and store primitives move values between variable and array storage in the dictionary and the data stack.

Table 6: data stack manipulation primitives, also fetch and store

primitive	C++ name	stack before	stack after
dup	STEP_DUP	x	x x
dup2	STEP_DUP2	x	x x x
dupi	STEP_DUP_I	data_stack i	(data_stack[i], where “0 dupi” is same as dup, “1 dupi” is same as over, “2 dupi” is same as over2, etc. “i dupi” duplicates the i’th entry from the top of stack, where 0 is the top, and i itself does not count as a stack entry.)
swap	STEP_SWAP	x y	y x
swap2	STEP_SWAP2	x y z	z y x
over	STEP_OVER	x y	x y x
over2	STEP_OVER2	x y z	x y z x
drop	STEP_DROP	x	
drop2	STEP_DROP2	x y	

Table 6: data stack manipulation primitives, also fetch and store

primitive	C++ name	stack before	stack after
@f	STEP_FETCH	addr	dictionary[addr]
@s	STEP_STORE	x addr	pops both, sets dictionary[addr] to x

There are two types of IO commands, text IO and IO port IO. The former is useful for debugging print commands, while the latter is a place holder. Classes derived from awmpStep supply their own, application-specific IO primitives.

Table 7: IO primitives

primitive	C++ name	stack before	stack after
prints	STEP_PRINTS	addr	prints pops addr and prints out the string referenced by addr, where the string is stored as an array of ASCII values, ending with 0
printi	STEP_PRINTI	x	printi pops x and prints out x according to the numeric output base (see below)
crlf	STEP_CRLF		crlf ignores the data stack; it prints out a newline
decimal	STEP_DECIMAL		decimal ignores the data stack; it sets printi to base 10
hex	STEP_HEX		hex ignores the data stack; it sets printi to base 16
octal	STEP_OCTAL		octal ignores the data stack; it sets printi to base 8
inp	STEP_INP	addr	(byte at ioport[addr])
outp	STEP_OUTP	x addr	pops both, sets ioport[addr] to byte value of x
inpw	STEP_INPW	addr	(16-bit word at ioport[addr])
outpw	STEP_OUTPW	x addr	pops both, sets ioport[addr] to 16-bit word value of x

The miscellaneous primitives include operations to query the state of the STEP machine, along with some other operations.

Table 8: Miscellaneous primitives

primitive	C++ name	stack before	stack after
sizeof_word	STEP_SIZEOF		(sizeof(STEPWORD))
ds_depth	STEP_DS_DEPTH		(depth of data stack)
ts_depth	STEP_TS_DEPTH		(depth of thread stack)
dict_len	STEP_DICT_LEN		(used length of dictionary)
cmds_len	STEP_CMDS_LEN		(used length of commands buffer)
noop	STEP_NOOP		noop does nothing

3.6. STEP private primitives

The private primitives give STEP machine-level support for the compiler's data-defining and control-construct keywords already discussed. A STEP program cannot invoke these directly.

Table 9: Private primitives

primitive	C++ name	stack before	stack after
constant	STEP_CONST		pushes inline word to data stack
variable	STEP_VAR		pushes inline variable address to data stack
array	STEP_ARRAY		pushes inline array address to data stack
goto	STEP_GOTO		does not use stack, jumps to inline addr
goto0	STEP_GOTO0	test-value	pops test-value from data stack, when 0 jumps to inline addr
gotox	STEP_GOTOX	index	pops index from data stack and gets inline index limit, calls proc from inline table

Table 9: Private primitives

primitive	C++ name	stack before	stack after
return	STEP_RETURN		does not use stack, pops thread stack into ip
bpt	STEP_BPT		does not use stack, enters/ resumes breakpoint

3.7. STEP source interpreter and compiler

The STEP source interpreter and compiler is in executable file teem/PLATFORM/step/**steptest**, where PLATFORM is one of *solaris*, *sunos* or *win32*. Environment variable STEPHOME must be set to teem/src/step or another directory containing file stepsgs.tcl. The STEP compiler searches for stepsgs.tcl, which contains the basic compiler system, and optional file stepextr.tcl, which contains primitive operation extensions for an awmpStep-derived class, by looking first in the current directory, and then in the $STEPHOME directory, and finally in the $HOME directory. The compiler searches for stepsgs.tcl and stepextr.tcl at distinct stages, and they can appear in different directories. While stepsgs.tcl must appear in one of the search directories, stepextr.tcl is optional.

Steptest can be invoked in a number of ways:

- steptest

Invoked without file arguments, steptest reads and interprets stdin. The user must type **...** (three dots) to tell the STEP reader to send its accumulated input to STEP's incremental compiler and loader.

- steptest [-ofile] file1 file2 ... filen
- steptest [-xofile] file1 file2 ... filen

These two variants create object file "file" from source files file1 through filen. Option "-xo" also executes commands immediately after each source file is compiled. Any of the input files may be "stdin," in which case stdin is read at that point in compilation. In this way a setup file can be compiled and run, and then the user can interact with STEP interpretively via stdin. The object file is optional. Without it the source files are simply compiled and executed immediately.

- steptest -lfile

STEP loads and executes object file "file" that was previously created using option "-ofile" or "-xofile."

4. STEP debugging using TEEM

Once a programmer has compiled a STEP program, that program can be executed and debugged under the TEEM debugger environment [6]. TEEM linked to the STEP VM class is available in teem/PLATFORM/step/stepteem, where PLATFORM is one of *solaris*, *sunos* or *win32*. It is invoked simply by running *stepteem* with environment variable STEPHOME set as described in Section 3.7.

4.1. An example STEP program under TEEM

In file teem/src/step/test/selsort.stp there is an example selection sort program used in a STEP regression test. This section looks at running selsort.stp under TEEM.

4.1.1. Selection sort example program

Here is the text of selsort.stp. There are three proc definitions for “selsort,” “dumparray” and “main,” along with global table “table1.” Finally there is a call to “main” from the commands buffer. Each proc definition line in the program includes a comment that shows:

```
// stack-before-call      ==>      stack-after-call
```

Most command lines also show the top of stack contents after executing that line. A very helpful technique in reading and writing code is to refer to the intended “top of stack trace” in the preceding line. By looking at the stack comment in the preceding line, a STEP programmer can determine the correct stack manipulation primitives to use (swap, over, drop, dup, etc.) to get the correct operands on top of the stack. A program reader can follow the effect of instructions by reading commands in relation to the top of stack comment in the preceding line.

```
// selsort.stp -- selection sort test of “if” and “while”
proc selsort     // array-addr count ==> (array sorted, nothing on stack)
        // Forth doesn't have function-static variables but STEP does,
        // here is a test of them
        variable my_array
        swap my_array @s // stick array address in my_array variable
        // The outer loop keeps "count-1 i" on the stack, iterating through
        // i = 0 through i = count-2. i selects the next element.
        // The inner loop pushes "count j" on top of "count i," i.e.,
        // "count i count j" where j = i+1 through count-1. j inspects values
        1- 0            // count i ("count" is original "count" - 1)
        while over over >       // count i
        do
                over over 1+    // count i count j (j = i+1 through  n)
                while over over >=
                do
                        over2 my_array @f + @f // count i count j array[i]
                        over my_array @f + @f // count i count j array[i] array[j]
                        if over over >  // count i count j array[i] array[j] >
                        then
                                // store array[i] at j
                                swap over2 my_array @f + @s
                                        // count i count j array[j]
                                // store array[j] at i
                                3 dupi my_array @f + @s // count i count j
```

```
                else
                    drop2                   // count i count j
                endif
                1+                          // increment j
            endwhile
            drop2                           // count i
            1+                              // increment i
        endwhile
        drop2                               // inputs args. discarded
    endproc
```

Selsort works by selecting the minimum remaining table element for array[i], where count is the array length and i iterates from 0 through count-2. Each i inspects remaining elements array[j], where j iterates from i+1 through count-1. The last element is at count-1. The “then” portion of the order test swaps out-of-order elements, while the “else” portion discards from the data stack a pair that is already correctly ordered.

The “over over” idiom seen above is a typical command sequence for performing a binary test (e.g., >) without destroying the operands. Each “over” copies one of the operands to the top of the stack, and the binary test pops these operand copies and pushes the test result.

```
    proc dumparray // array-addr count ==> (array printed, nothing on stack)
        0                   // array-addr count index
        while over over >
        do
            over2 over + @f printi crlf // array-addr count index
            1+                          // bump the index
        endwhile
        drop2 drop
    endproc
```

Proc dumparray simply prints out the array. Proc main calls dumparray to print table1, then calls selsort to sort it, and again calls dumparray.

```
    // make table1 global
    table table1 45 27 -100 100 0 -1 1 5 endtable
    proc main
        "table1 before the sort\n" prints
        table1 8 dumparray
        "table1 after the sort\n" prints
        table1 8 selsort
        table1 8 dumparray
    endproc
```

```
main    // this is an immediate command in the commands buffer
```

Here is the output seen from executing “teem/PLATFORM/steptest selsort.stp.”

```
[D:/dteem/win32/step] steptest selsort.stp
stepCompile selsort.stp, 167 words to dictionary in this pass, 167 total
stepCompile selsort.stp, 1 words to command buffer in this pass
stepCompile max thread stack is 2 on main
table1 before the sort
45
27
-100
100
0
-1
1
5
table1 after the sort
-100
-1
0
1
5
27
45
100
[D:/dteem/win32/step]
```

Each pass through the STEP compiler reports the size of the dictionary, the size of the commands buffer, and the maximize size of the thread stack as indicated by the deepest possible sequence of proc calls from a proc. When a recursive proc is present the compiler reports maximum thread stack depth as an unknown.

4.1.2. Running selection sort under TEEM

There is a demo of STEP and TEEM under directory teem/PLATFORM/step.demo. From within that directory with the STEPHOME environment variable set to teem/src/step, the following steps will compile selsort.stp and invoke TEEM.

```
[D:/dteem/win32/step/demo] ../steptest -oselsort.e $STEPHOME/test/selsort.stp
stepCompile d:/dteem/src/step/test/selsort.stp, 167 words to dictionary in this pass, 167 total
stepCompile d:/dteem/src/step/test/selsort.stp, 1 words to command buffer in this pass
stepCompile max thread stack is 2 on main
```

```
D:/dteem/win32/step/demo] ../stepteem
pdemo note 25005: current pssr set to pdemo
```

At this point stepteem has sourced the file .teemrc, which contains the following commands:

```
pssr new step pdemo
pdemo mload selsort.e
source stepdemo.tcl
proc lui {} {
        global env
        source $env(LIBTEEM)/lui
}
```

At this point typing “lui” brings up TEEM’s Lightweight User Interface. The file .teemdefaults contains LUI window and breakpoint configuration for STEP instance pdemo. When LUI comes up it displays the windows of Figure 2. Executing “Reset” and “Resume” takes execution to a breakpoint near the top of proc selsort.

Figure 2 shows the state of the STEP program after reaching the breakpoint and clicking “Resume” twice. The source window is on the left. Selsort’s outer loop has run twice. The “Watch 1” window contains calls to “dds,” a Tcl proc from file stepdemo.tcl, which was sourced in .teemrc above. Tcl proc dds takes an offset into the data stack and returns the value at that offset. The “Watch 1” window uses calls to “dds” to display the contents of the data stack. In Figure 2 the i index from proc selsort is on top of the data stack with a value of 2; beneath it is selsort’s count-1, a value of 7 here.

The “Watch 2” window uses Tcl proc “dts” from stepdemo.tcl to display the proc names in the STEP thread stack. A memory display window in the upper right shows the table1 array. In Figure 2 the first two elements have already received the correct values of -100 and -1. The final, “Register” window on the lower right shows the STEP virtual machine registers.

Tcl procs dds and dts from stepdemo.tcl appear below.

```
# stepdemo.tcl -- some LUI windows for STEP
proc dds {offset} {
        # debug data stack, where offset of 0 is top of stack,
        # -1 is below that on stack, etc.
        if {$offset > 0} {
                return "?"
        } else {
                set memoff [expr [fxpr @rd ds] + $offset]
```

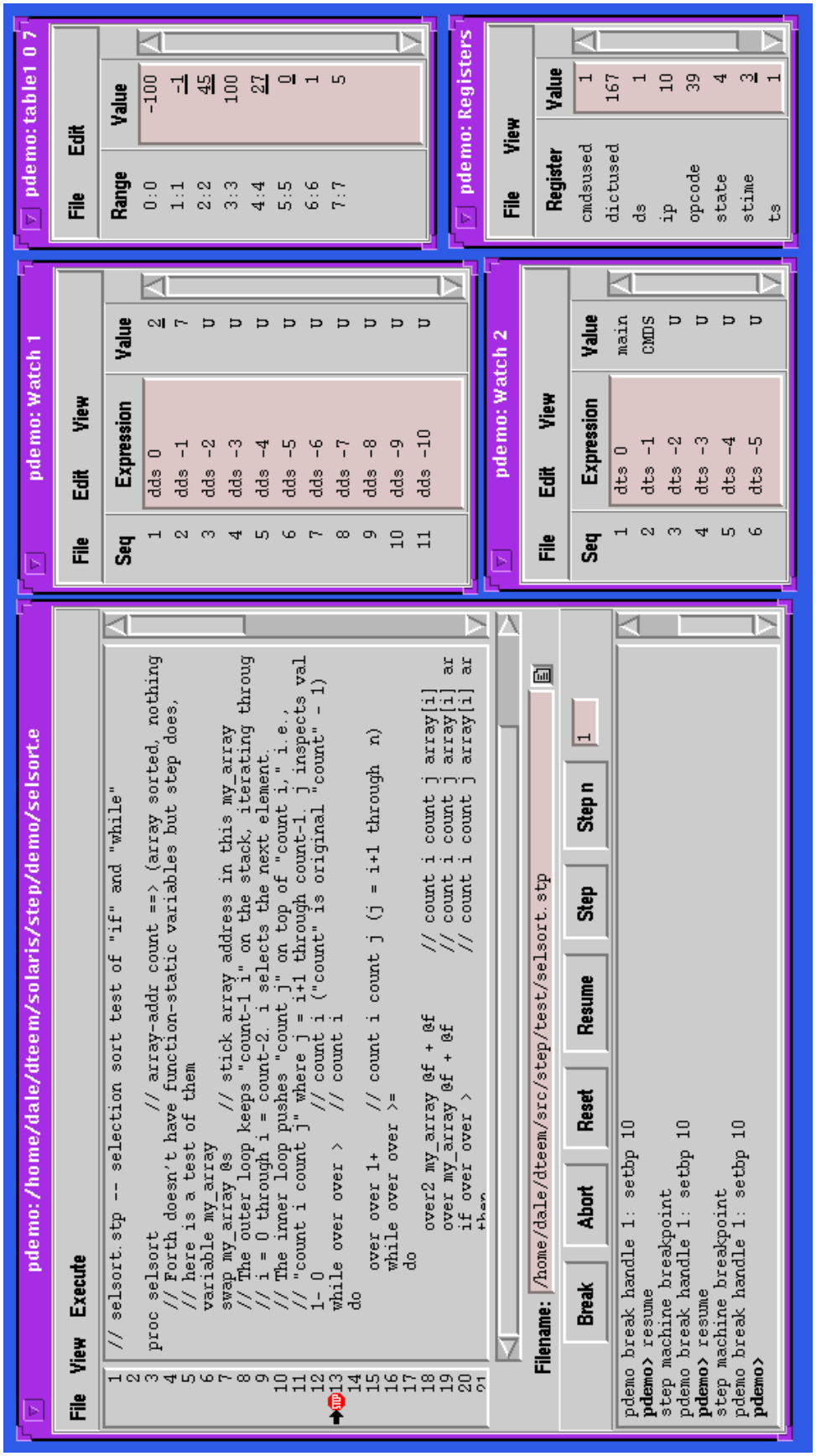

Figure 2:

TEEM debugging session for STEP running selsort.e

```
                if {$memoff > -1} {
                        return [fxpr @rd dstack($memoff)]
                } else {
                        return U
                }
        }
}
proc dts {offset} {
        # debug thread stack, where offset of 0 is top of stack,
        # -1 is below that on stack, etc.
        if {$offset > 0} {
                return "?"
        } else {
                set memoff [expr [fxpr @rd ts] + $offset]
                if {$memoff > -1} {
                        return [dts_label [fxpr @rd tstack($memoff)]]
                } else {
                        return U
                }
        }
}
proc dts_label {addr} {
        global dts_mload_number dts_mapping
        set iname [pssr]
        set loadnum [mload -N]
        if {(! [info exists dts_mload_number($iname)])\
                || ($dts_mload_number($iname) != $loadnum)} {
                # get symbol table info. when it changes
                set dts_mload_number($iname) $loadnum
                set dts_mapping($iname) {}
                foreach labpair [? L] {
                        if {[string compare [lindex $labpair 1] load-label] == 0} {
                                set linkpair [fxpr [lindex $labpair 0](0:0)]
                                if {[string compare [lindex $linkpair 0] dict]==0} {
                                        set laddr [lindex $linkpair 1]
                                        lappend dts_mapping($iname) \
                                                [list [lindex $labpair 0] $laddr]
                                }
```

```
                        }
                    }
                }
                if {$addr < 0 || $addr >= [fxpr @rd dictused]} {
                    return CMDS ; # return pointer to commands buffer
                } else {
                    set retval ??
                    set retnum -1
                    foreach lpair $dts_mapping($iname) {
                        # sequential search of memory range
                        set lval [lindex $lpair 1]
                        if {$addr >= $lval && $lval > $retnum} {
                            set retnum $lval
                            set retval [lindex $lpair 0]
                        }
                    }
                    return $retval
                }
            }
```

4.2. Learning about TEEM HDM by putting STEP under TEEM

Five C++ class definitions appear in files stepdbg.h and stepdbg.cxx in directory teem/src/step. These class definitions have helped to provide examples and tests of the use of TEEM infrastructure in attaching a TEEM debugger to a virtual machine that was not designed using TEEM's modeling classes. Class awmpStep does not using modeling infrastructure classes because it needs to run as fast as possible, and it needs to run on MSDOS and Win16. TEEM is not supported on MSDOS or Win16. Nonetheless we will be able to attach a TEEM debugger to a STEP VM on MSDOS or Win16 by treating the STEP VM as a hardware processor. This exercise has been the first opportunity to attach TEEM to a remote, emulated processor. For now the STEP VM must be running in the same UNIX or Win32 process as the TEEM debugger, but when HDM networking is available, we will be able to debug remote STEP VMs across TCP/IP using TEEM.

The remaining subsections of Section 4.2 give an overview of the classes used to connect the STEP VM to TEEM.

4.2.1. class awmpSTEPSgnl : public awmpArithSgnl

Class awmpSTEPSgnl defines the **fxpr** arithmetic of STEP under TEEM. Consult reference [7] for a discussion of class awmpArithSgnl and its derived classes used to support processor-driven debugger arithmetic. Fxpr is TEEM's command-level processor-driven arithmetic expression evaluator.

Before the advent of awmpSTEPSgnl there was only one class derived from awmpArithSgnl, namely class awmpDynaSgnl. The latter performs multiple-precision fixed-point arithmetic for fixed-point DSPs. Class awmpSTEPSgnl performs STEPWORD arithmetic, where STEPWORD

is a typedef of a C integral type, currently a long integer. Besides supporting STEP debugging, awmpSTEPSgnl now serves as a demonstration and test of TEEM's processor-driven approach to debugger arithmetic.

4.2.2. class awmpSTEPAbstractDebugger

The networked HDM package supported for cmd16xx works by distributing a number of cmd16xx-specific C functions across TCP/IP. This platform-neutral remote procedure call (RPC) mechanism relies on a utility called *netgen* to generate networked function distribution code. Netgen generates function-specific host code to pack function arguments into a networked data stream as well as function-specific host code to unpack return results. Netgen also generates function-specific target code to unpack networked function arguments, code to call the correct networked function, and code to pack and return networked results. Netgen works with a set of global C application functions.

Netgen for TEEM will expand to support networked distribution of a C++ class. Figure 3 gives an outline of the approach to be taken by *netgen++*. The Abstract Class at the top of the class hierarchy consists entirely of pure virtual functions that define the behavior of the distributed class. Function signatures for member functions that are to be distributed must meet the requirements for function signatures under netgen [8].

The Concrete Class of Figure 3 defines the functions that implement the behavior of the Abstract Class.

The Networked Class, on the other hand, defines networked member functions that distribute function calls over a network. Netgen++ will generate most or all of the Networked Class code. A client module that calls an Abstract Class function need not be concerned with the function's implementation as a Concrete or Networked function.

The Networked Helper Class does not participate in an inheritance relationship with these other classes. Instead a Networked Helper Class object constitutes a run-time wrapper for a Concrete Class object that is accessed over a network. A Networked Helper Class object unpacks networked data streams, calls member functions, and packs return data streams for its Concrete Class object. The Networked Helper Class acts as a calling client on the target networked environment. Netgen++ will generate most or all of the Networked Helper Class code.

Class awmpSTEPAbstractDebugger is an example Abstract Class. It provides access to a STEP object that may be resident in the current process or that may exist on another machine across a network. Its definition appears in Figure 4.

4.2.3. class awmpSTEPDebugger : public awmpSTEPAbstractDebugger

Class awmpSTEPDebugger is the Concrete counterpart to awmpSTEPAbstractDebugger. Its constructor takes a pointer to an actual awmpStep object. It works by calling corresponding awmpSTEP member functions.

At present there is no networked counterpart to awmpSTEPDebugger. Part of the test phase of TEEM's networked HDM will consist of generation and testing of a Networked Class for STEP debugging.

4.2.4. class awmpSTEPPssr : public awmpDriverPssr

The Tcl level of TEEM interacts with models, or more specifically, with a set of awmpDriverPssrs

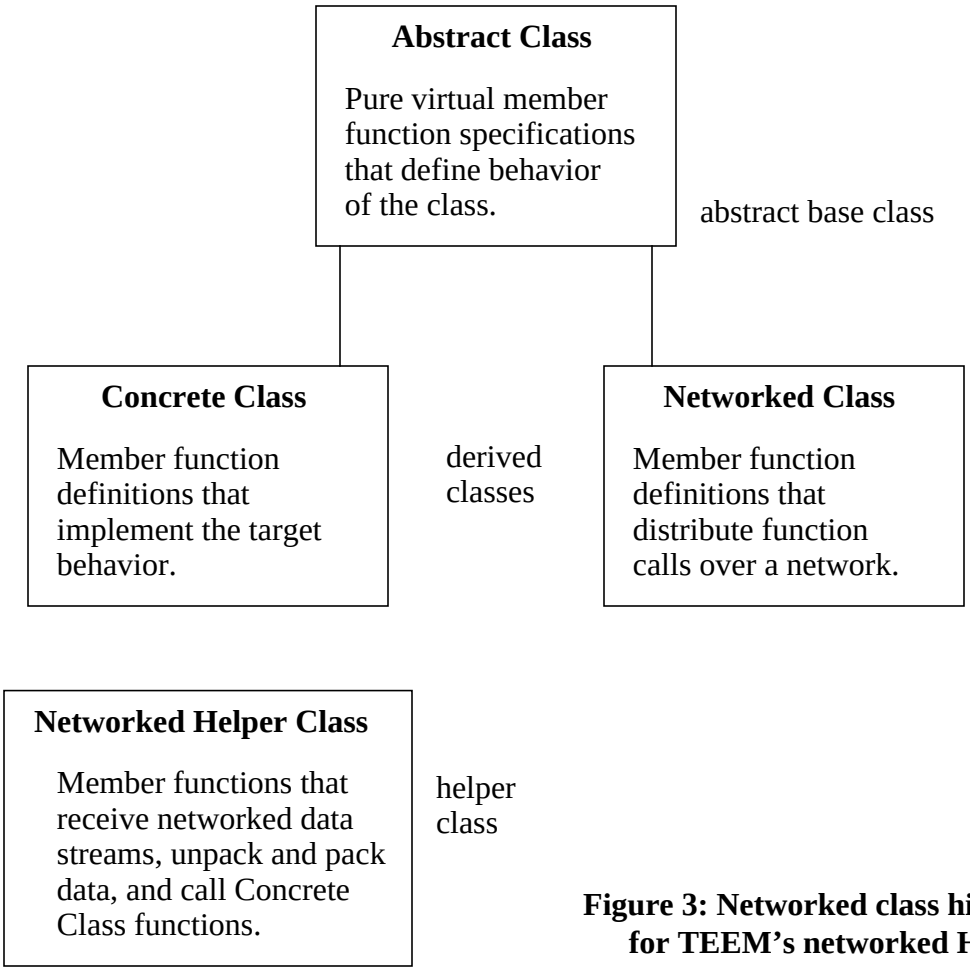

Figure 3: Networked class hierarchy for TEEM's networked HDM

and awmpInterfacePssrs [9]. When TEEM runs in hardware development mode (HDM), a model acts as a data repository for its corresponding hardware processor. Class awmpSTEPPssr is an awmpDriverPssr within TEEM that acts as a repository for the state of an awmpSTEP object. It contains modeling register and memory objects that can store and provide user access to STEP VM state. Class awmpSTEPPssr works only in hardware mode; it does not provide any simulation capability.

4.2.5. class awmpSTEPHDM : public awmpTclHDM

Class awmpTclHDM is a class within the TEEM framework that distinguishes between simulation mode and hardware mode operation. For a number of operations that work differently in software and hardware mode, for example **reset** and **resume**, awmpTclHDM calls into a model when in simulation mode, and it communicates with a hardware processor when in hardware emulation mode.

Some parts of awmpTclHDM's behavior are processor-core-specific. Class awmpTclHDM has existed since January of 1996, but its processor-core-specific interfaces were incomplete, and there had been no hardware processor available to TEEM for testing awmpTclHDM.

The STEP VM appears as a hardware processor to TEEM, and so it was possible to bring the Tcl-

```
class awmpSTEPAbstractDebugger {
public:
   // This class gives direct or networked access to an awmpSTEP
   // object for debugging. This class merely specifies the direct
   // or indirect access via abstract functions. See the equivalent
   // functions in class awmpSTEP, awmpstep.h for documentation.
   virtual ~awmpSTEPAbstractDebugger(); // needed for derived destructors
   virtual void regUpload(STEPWORD *ds_size, STEPWORD *ds_top,
         STEPWORD *ts_size, STEPWORD *ts_top, STEPWORD *dict_size,
         STEPWORD *dict_end, STEPWORD *cmds_size, STEPWORD *cmds_end,
         STEPWORD *ipval, STEPWORD *op_code, STEPWORD *stepstate) = 0 ;
   virtual STEPWORD regDownload(STEPWORD ds_size, STEPWORD ds_top,
         STEPWORD ts_size, STEPWORD ts_top) = 0 ;
   virtual STEPWORD memUpload(STEPWORD memID, STEPWORD start,
      STEPWORD maxwords, STEPWORD *destination, STEPWORD *realwords) = 0 ;
   virtual STEPWORD memDownload(STEPWORD memID, STEPWORD start,
      STEPWORD words, STEPWORD *source) = 0 ;
   virtual STEPWORD setbp(STEPWORD addr) = 0 ;
   virtual STEPWORD clrbp(STEPWORD addr) = 0 ;
   virtual STEPWORD load(STEPWORD *memoryImage,
      STEPWORD memoryImageLength,
         STEPWORD appendmode, STEPWORD execute) = 0 ;
   virtual STEPWORD resume(STEPWORD steps) = 0 ;
   virtual void reset(void) = 0 ;
   virtual void sigint(void) = 0 ;
} ;
```

Figure 4: awmpSTEPAbstractDebugger is an Abstract Class

HDM interface of awmpTclHDM two steps closer to completion by first fleshing out the processor-core-specific interface functions, and then building some tests. Class awmpSTEPHDM adds the STEP-specific hooks to awmpTclHDM.

4.3. STEP-TEEM conclusion

It has taken about a week to write this documentation. Two weeks for the STEP virtual machine, SGS and debugger and another week for documentation is not half bad.

The fact that it is possible to construct an environment like this in so short a time should say something about the effectiveness of TEEM for the rapid support of heterogeneous processor types, and about the effectiveness of Tcl as a SGS implementation language.

STEP's main application is target hardware control for networked HDM. That is by no means STEP's only potential application. The STEP VM is available for other applications that wish to extend and use its capabilities. A client can readily extend STEP through the addition of primitives and the definition of procs. The SGS and debugging tools will work for any class derived from awmpStep in a modular, application-specific way.

5. References

1. E. Rather, L. Brodie and C. Rosenberg, *Using FORTH*, Hermosa Beach, CA: FORTH, Inc., 1980.

2. Brent B. Welch, *Practical Programming in Tcl and Tk*, Second Edition, Upper Saddle River, NJ: Prentice Hall, 1997.

3. R. G. Loeliger, *Threaded Interpretive Languages, Their Design and Implementation*, Peterborough, NH: Byte Books, 1981.

4. "Choosing Forth," Products and Services from FORTH, Inc., forthinc@forth.com, 1995.

5. D. Parson, *Threaded Intermediate Code*, Master of Science Thesis, Lehigh University, 1986.

6. D. Parson, P. Beatty and B. Schlieder, "A Tcl-based Self-configuring Embedded System Debugger," *The Fifth Annual Tcl/Tk Workshop '97 Proceedings*, Boston, MA: USENIX, July, 1997, p. 131-138.

7. D. Parson, "C Debugger Architecture for TEEM: A Position Paper," Bell Labs internal memo, June 25, 1997.

8. D. Parson, "DSP16xx Networked Software Tools for Remote Access to PC-based Hardware Development Resources," Bell Labs internal design specification, final revision, October 18, 1994.

9. D. Parson, "Architecture for AT&T Signal Processor Simulation Models, Debugger Environment and Hardware Development Tools," Bell Labs internal architecture specification, April 5, 1995.

Dale E. Parson