A MICRO BASED ASW TRAINER IN FORTH

Robert H. Davis

ABSTRACT

The author's group at the Naval Surface Weapon Center is engaged in the development of a low cost desktop trainer for NAVY ASW Acoustic Sensor Station Operators. Each device is based on a Zenith 150 microcomputer, with the software being developed using ROHDA-FORTH, the author's personal implementation of the FORTH language. Brief descriptions of the system being simulated and the Forth based simulation are given. The paper then focuses on the productivity aspects of using FORTH in the development effort. Particular areas discussed are the value of very high level coding (using the ROHDA-FORTH Data Structures and Symbolic Function compilers) to expedite development of all coding for an initial pass, and use of trace facilities to achieve extremely rapid code development. Selected examples of coding from the project will be used to illustrate the use of the high level programming constructs and the trace facilities used in the software development.

DISCUSSION

The Navy's AntiSubmarine Warfare (ASW) systems depend heavily on information derived from the spectral analysis of underwater acoustics. The operation of the spectral analysis equipment and the interpretation of the outputs of that equipment require considerable expertise. Personnel turnover is rapid and the training necessary to develop the required expertise has been chronically inadequate. Interactive training devices have been expensive and in short supply. The author and two other engineers at the Naval Surface Weapons Center in White Oak, Maryland, have undertaken the development of a low-cost desktop training device for initial training based on the Zenith 100/150 microcomputer. The software has been developed in FORTH using ROHDA-FORTH, the author's personal implementation of the FORTH language.

The primary output of the ASW acoustic processor is a spectral 'gram', which consists of a time history of Spectral Power Density estimates displayed as successive scans of an intensity versus frequency display. Modern processors display such grams on CRT monitors, and therefore a micro with reasonable graphics capability makes a fine basis for a low cost simulator. The simulator display provides a 12 minute 'gram' (120 scans) with 512 frequency bins per scan. The graphic display includes frequency cursors (controllable by the student), and alphanumeric time, cursor frequency, frequency limits and mode readouts. The simulation scenario allows two acoustically radiating 'targets' in a typical ocean environment. Motion dynamics and acoustic signature dynamics are modeled to allow a problem to evolve in real time.

Of the three engineers, one (the author) had considerable experience in FORTH, one had minimal FORTH experience, and one had no previous FORTH experience. However, all three are very capable assembly language programmers, and in addition had the benefit of having programmed similar simulators on other computers (Data General Novas, Navy AN/UYK-20s, and VAXs). With this rather atypical set of initial conditions, any rigorous attempt to compare productivity relative to other programmers, languages or tasks is not easy. Some subjective observations may be made, however, both in loose comparisons with the rest of the world and in the perceptions of the programming team.

The time available for producing a deliverable prototype system (software running and about 99% debugged but minimal documentation) was approximately 90 days. The software developed for the deliverable prototype consisted of approximately 70 FORTH screens, each with an average of a dozen lines of code, for something on the order of a thousand lines of (FORTH) source code. The compiled code occupied around 24000 bytes (not including data arrays), so it is estimated that something on the order of 4000 to 5000 FORTH operators were coded. If one uses the source lines of code (SLOCs) as the software industry is want to do, one arrives at a productivity of 100+ debugged SLOCs per man month. This appears to be, if anything, a little low by industry norms. In FORTH, however, each operator is more or less equivalent to a line of code in other languages, so one might venture an 'equivalent' productivity measure of 400+ SLOCs per man month, which looks a little more impressive.

Neither of the above 'productivity' numbers really tells the whole story. A similar but considerably less sophisticated program was written in FORTRAN on a VAX 11/780. This effort resulted in about 1500 lines of FORTRAN in about 1.5 man years, for a productivity of a little under 100 lines of code per man month. Several critical differences exist, however. The display programming for the VAX program was essentially already available and the program ran as a single process - no interrupt handling was necessary. For the micro system, the first two weeks were spent developing interrupt driven multiprocessing for ROHDA-FORTH. The VAX-FORTRAN program used floating point throughout, while the micro system uses a mixture of single and double integer fixed point, and a double mixed fixed point (16 bits of integer, 16 bits of fraction) which is supported in ROHDA-FORTH. An added advantage of the FORTH implementation is that, as is often done, a slightly simplified FORTH interpreter is made available at the foreground (non-interrupt driven) level. This permits interactive display and modification of program variables, and greatly facilitates interactive control. Obviously, nothing equivalent to this was available in the VAX-FORTRAN program. Last, but not least, the VAX program (even though a simpler system running on a much more powerful computer) required bumping other users for demonstrations to insure against unacceptable interferences for real-time operation. Thus it can be reasonably claimed that a more powerful and sophisticated simulation program was developed in less time with FORTH in spite of considerably less powerful hardware and a supposedly inferior programming environment.

ROHDA-FORTH is an almost 79-Standard system with most FIG-FORTH words available plus several powerful extensions. Although the lion's share of credit for an impressively productive programming effort goes to the FORTH concept itself, many of the ROHDA-FORTH extras were found very helpful. In particular, a PASCAL like Data Structures compiler, a Function compiler with local variables, and a recursive single step TRACE facility played key roles.

To the author, the only useful feature of the PASCAL language is the ability to define complex data structures and to then let the compiler worry about where particular elements are located. The ROHDA-FORTH Data Structures compiler provides a capability similar to PASCAL in that arbitrarily complex structures may be defined and then accessed symbolically. The compiler computes as much as possible of the address of the desired element and compiles the minimum runtime code to effect the access. As an example, delayed action commands were implemented using a data structure consisting of an array of records, each of which contained a double variable TIME and a COMMAND string. The structure was defined as follows:

```
 DINTDT << TIME >>  ( define a dbl. int. datatype TIME )
 80 STRDT  COMMAND  ( and 80 char. string datatype COMMAND )
 RECDT TCREC << TIME COMMAND >>   ( record of 1 each )
 100 ARRDT TCARR TCREC  ( finally an array datatype TCARR
                               of 100 'TIME COMMAND' records )
```

Memory for the array is then reserved in HEAP (which consumes all remaining available RAM after a 64Kbyte Dictionary area) :

```
 ALLOCATE TIMED-COMMAND TCARR
```

The 19th TIME can then be obtained by:

```
 TIMED-COMMAND 19 TIME
```

which would leave a double integer on the stack, or, if within a loop, the 'I'th string can be obtained by:

```
 TIMED-COMMAND I COMMAND
```

which would leave an address pointer to the first character of the 'I'th COMMAND string followed by the character count 80.

The Data Structures compiler took the labor out of managing an involved program database. Similarly, the Function compiler greatly eased the pain of stack-based programming. It has been pointed out by others that most of the difficulty in FORTH programming is the necessity for the programmer to mentally keep track of where things are on the stack. ROHDA-FORTH provides a Function compiler which allows the programmer to declare the input and output stack configurations symbolically, and to declare temporary local variables. Coding then proceeds using the stack as in a normal FORTH word, but with inputs, outputs and intermediate results accessed symbolically with the local symbols. As an example, the following is a single pole recursive filter with stack inputs parameter A, old state Y', and new input X, and output Y, and coded using the Function compiler:

```
 : 2PFIL FUNCT( A Y' X > Y )
      X Y' - M* Y' + TO Y  DROP F;
```

where the compiler word FUNCT(initiates the local variable declaration phase ended by), and F; is the Function compiler equivalent to ; for a normal definition. Notice that any FORTH words including stack manipulation operators may be used in the body of the definition, but that the stack must be clean when finished.

Both Data Structures and Functions provide runtime checks for critical errors. Runtime array computations check for indices out of bounds, and runtime Function support checks for stack residue errors. These obviously extract a small penalty in speed, but greatly aid debugging. If necessary, two runtime words can be redefined to eliminate all runtime checks once the program is completely debugged.

Last but not least is the value of an interactive TRACE facility. ROHDA-FORTH provides a recursive single step debugger which allows one to TRACE and high-level definition and to optionally descend and TRACE any other high-level definition encountered during a TRACE session. As most of the FORTH community knows, the value of such a tool to quickly pinpoint bugs is immense. For the Micro Simulator effort, the average debug time for high-level definitions was probably about 15 minutes.

In summary, the development of a desktop training device using the FORTH language on a personal computer provides another example of applying the power and efficiency of FORTH to real problems. Although direct comparisons of productivity with other languages is virtually impossible without comparing apples to oranges, all three of the programmers, FORTH freaks or not, agree that development time and effort was drastically less than previous similar efforts. The Department of Defense could benefit greatly by serious consideration of FORTH as an alternative to ADA, but Bureaucratic inertia being what it is, it will probably take a club the size of a battleship to change its direction.