Top-down Design in Forth

Dr. James Basile
Department of Computer Science
Long Island University - C.W. Post Campus

ABSTRACT

Top-down design and coding practices are central to effective software engineering. Forth provides a programming environment which readily can be modified and expanded to suit particular needs. This paper examines the relationship between Forth as a programming environment and the use of top-down design and coding.

1. Introduction

This paper is not intended as a treatise on top-down design. Rather, it is assumed that the reader is familiar with the basic concepts of top-down design and espouses these principles as common programming practice (see [3]). It focuses instead on some techniques that are helpful in implementing top-down design and coding in Forth.

Because Forth is an extensible programming environment, one has an unlimited set of tools to work with when programming in Forth. This is always a double-edged sword: one can either use such freedom to write virtually unmaintainable code or one can design structures, and ultimately code, which reflects the simplicity and clarity of well-designed programs.

One of the advantages of coding in Forth is that one can more or less write the code directly by designing the application from the top down. This minimizes the need for external design tools such as structure charts. The code itself performs that function. The major problem with this approach is that the resulting code is in the wrong order for Forth definitions. Forth must be defined from the bottom up (unless the version being used supports forward referencing, of course). So the basic building blocks of the code must be written first. In the absence of other techniques, this little shortcoming can be resolved by starting at the highest block in a series to be loaded and working backwards.

2. Top-Down Design at Work

Let's look at an example for clarification. Suppose we are designing a terminal emulator. Forth allows us to work up a preliminary design for such a program rather easily. After all, the terminal emulator performs two major functions. It must capture keystrokes and pass them along to the host. It must also monitor the communications port and display the characters

received on the screen. Thus, we might start with the following attempt.

```
: EMULATE
   BEGIN  ?KBD IF   ( key pressed ? ) SEND   THEN
          ?PORT IF  ( something coming in ? ) EMIT   THEN
   0 UNTIL ;
```

We now have a basic design concept and we can begin to analyze its suitability. First, we can refine the design by specifying what each of these words must do.

?KBD	check the keyboard input buffer to see if a character is waiting to be processed. If so, return the character and "true". If not, just return "false".
SEND	similar to EMIT, simply sends a character out on the communications port
?PORT	similar to ?KBD, check the communications buffer to see if a character is waiting to be processed. If so, return the character and "true". If not, just return "false".

Given these specifications for the top level modules, does this design really encompass all that a terminal emulator should do? Let's look a little more closely. The first thing we can observe is that the emulator doesn't handle half duplex communications very well. EMULATE will display the characters received over the communications port but does not display those pressed at the keyboard. We need a word that will echo these characters if the communications are in half duplex mode.

ECHO	If half duplex mode flag is set, display a copy of the character on the top of the stack. If not, do nothing.

The next thing we might want to provide is a little more intelligence for the emulator. Most terminals have some control sequences that can be used to provide capabilities such as cursor positioning. The emulator should mimic these. We probably also want the capability to upload and download files. Since we have added control sequences to the emulator, we might as well provide one that ends the endless loop by setting a flag called DONE (which will be queried at the end of each pass through the loop).

Without cluttering up the design, we can take care of these additional requirements by substituting a slightly more complicated word for EMIT:

DISPLAY	examines the characters to be displayed looking for control sequences. If one is encountered perform it. Otherwise, display the character.

Now, that's a bit easier said than done but this is top-down design at its best. We simply indicate the function of the module in the next level and leave its actual implementation for later.

One last concern might be to ensure that the emulator doesn't lose any characters coming in the communications buffer. To do this, we might set up a Forth task with a queue (see [2]) to monitor the buffer. Then ?PORT can look at that queue instead of the actual port. This port task can be initiated when starting the emulator and suspended when the emulator is exited.

The emulator design has evolved somewhat, but the basic definition is still pretty sound. With a few modifications, it now looks like:

```
: EMULATE    PORT-UP  ( start up communications port )
     BEGIN
          ?KBD IF  ( key pressed )  ECHO SEND   THEN
          ?PORT IF  ( character waiting )   DISPLAY   THEN
     DONE UNTIL     PORT-DOWN ;
```

The design process can now work at refining the details of each of these words. We have a working definition for each word needed along with the overall structure for the program. Now, we can begin to speculate how to design each piece.

For a simple terminal, DISPLAY may just be a CASE statement or a few nested IFs. For a more complicated terminal, DISPLAY might be designed more elegantly, perhaps using a state machine structure. The decision will not alter the overall logic, only the actual definition of DISPLAY.

Similarly, for a time critical application, we might opt to make the port task an interrupt-driven assembly language routine rather than a high-level Forth task. Again, this does not impact the overall logic. It will only affect those words initiating and terminating the port task.

We then can apply these same techniques in refining each word in the program to basic Forth building blocks. For one example of how such an emulator turns out, see [1].

3. Some Observations

As illustrated above, the process of coding in Forth lends itself quite naturally to top-down design. A top level word, defining the overall flow of the program, leads to a definition for each of its component words. This process is continued on each of the component words until each bottom-level component consists of only Forth building blocks. Nonetheless, the programmer can either strengthen the relationship between Forth and top-down design or obliterate it. There are three factors which are central to strengthening the relationship: writing code which is

simple to follow and well-documented, designing useful control and data structures, and recognizing the implications of the stack oriented nature of parameter passing in Forth.

While there may be some variation from language to language, the basics of good coding are no different in Forth than in any other language: identifiers (variable names, word names, etc.) should be chosen to reflect their meaning as best as possible; the logic for control structures should be simplified as much as possible; word definitions should not be too lengthy or too trivial; comments should be meaningful and used as needed.

Because Forth is extensible, the language can be moulded to meet the application. If specialized structures are needed in a particular program, they may be defined. Thus, in most languages the programmer must make his program fit the control and data structures of the language. In Forth, however, the language can be modified to make the programming easier and clearer. For example, specialized CASE statements or array structures can be created.

Finally, because parameter passing between routines is simply implemented on a stack, program refinements and modifications can be accomplished with minimal impact. Particular words may be converted to assembler without impacting program development at all. Similarly, words may change their meaning but have no adverse affect if they maintain proper stack usage. A word of caution is in order here: it is critical that comments for Forth code accurately reflect the stack usage of each word so that the programmer can easily ascertain the impact of a modification to any word in the program.

When these principles are adhered to, writing maintainable Forth code is ensured. There is nothing inherent in Forth that prevents good software engineering practices. In fact, Forth properly used allows easy application of good top-down design and coding.

REFERENCES

[1] Basile, James. "The VT52 - Terminal Emulation in Forth". Proc. 1984 Rochester Forth Conference. Rochester: Institute for Applied Forth Research, 1984.

[2] Leary, Rosemary and McClimans, Donald. "Message Passing with Queues". The Journal of Forth Application and Research, Vol. 1, No. 2. Rochester: Institute for Applied Forth Research, 1983.

[3] Yourdan, Edward. Techniques of Program Structure and Design. Englewood Cliffs: Prentice-Hall, 1975.