Forth programming language

From Wikipedia, the free encyclopedia
with additions from the OpenBIOS Project and Carsten Strotmann
compiled for 21C3.

10th December 2004

Contents

1 FORTH

Forth is a computer programming environment. It was initially developed by Chuck Moore at the US National Radio Astronomy Observatory (NRAO) during the 1960s, formalized as a pro-

gramming language in 1977, and standardized by ANSI in 1994. It features both interactive execution of commands (making it suitable as a shell for systems that lack a more formal operating system), as well as the ability to compile sequences of commands into threaded code for later execution.

Its name is derived from Mr. Moore's belief that it was a "fourth-generation computer language" but it was developed on a computer whose file system allowed only five-letter identifiers.

1.1 Overview

Forth offers a standalone programming environment consisting of a stack oriented interactive incremental interpreter/compiler. Programming is done by extending the language with 'words' (the term used for Forth subroutines), which become part of the language once defined. Forth is usually implemented with an inner interpreter tracing indirectly threaded machine code, which yields extremely compact and fast high-level code that can be compiled rapidly.

A character-oriented screen/block mechanism and standard editor written in Forth, provide a file mechanism for creating and storing Forth source code. A typical Forth package will consist of a pre-compiled kernel of the core words, which the programmer uses to define new words for the application. The application, once complete, can be saved as an image, with all new words already compiled. Generally, programmers will extend the initial core with words that are useful to the sorts of applications that they do, and save this as their working foundation. Due to the ease of adding use small machine code definitions to the language and using those in an interactive high-level programming environment, the Forth language has been popular as a development language for embedded systems and as a vehicle for instrument control.

The structure of the inner interpreter is similar to modern RISC processors and processors that use Forth as machine language have been produced (free examples are the b16 processor and μCore). Because of the modular extensible nature of Forth, which allows, e.g., for object-oriented programming, many high-level applications, such as CAD systems were written in Forth.

Forth is used in the OpenFirmware boot ROMs used by Apple and Sun. It is also used by FreeBSD as the first stage boot controller.

1.2 Forth from a programmer's perspective

Forth relies heavily on explicit use of the stack data structure and Reverse Polish Notation (or RPN, also used on advanced calculators from Hewlett-Packard). This notation is also called postfix notation because the operator comes after its operands, as opposed to the more common infix notation where the operator comes between its operands. The rationale for postfix notation is that it is closer to the machine language the computer will eventually use, and should therefore be faster to execute.

For example, one could get the result of a mathematical expression this way:

```
25 10 * 50 + .
300
```

This command line first puts the numbers 25 and 10 on the implied stack; the "*" command multiplies the two numbers on the top of the stack and replaces them with their product; then the number 50 is placed on the stack, and the "+" command adds it to the previous product; finally, the "." command prints the result to the user's terminal. Even the language's structural features are stack-based. For example:

```
: FLOOR5 DUP 5 < IF DROP 5 ELSE 1 - THEN ;
```

This code defines a new word (again 'word' is the term used for a subroutine) called "FLOOR5" using the following commands: "DUP" simply duplicates the number on the stack; "<" compares the two numbers on the stack and replaces them with a true-or-false value; "IF" takes a true-or-false value and chooses to execute commands immediately after it or to skip to the "ELSE"; "DROP" discards the value on the stack; and "THEN" ends the conditional. The net result is a function that performs similarly to this function (written in the C programming language):

```
int floor5(int v) {
  if (v < 5)
     return 5;
  else
     return v - 1;
}
```

A terser Forth definition of FLOOR5 that gives the same result:

```
: FLOOR5 6 MAX 1 - ;
```

Forth became very popular in the 1980s because it was well suited to the small microcomputers of that time: very efficient in its use of memory and easy to implement on a new machine. At least one home computer, the British Jupiter ACE, had Forth in its ROM-resident OS. The language is still used in many small computerized devices (called embedded systems) today for reasons of efficiency in memory use, development time, and execution speed.

Forth is also infamous as being one of the first and simplest extensible languages. That is, programmers can easily adapt the features of the language to the problem. Unfortunately, extensibility also helps poor programmers to write incomprehensible code. The language never achieved wide commercial use, perhaps because it acquired a reputation as a "write-only" language after several companies had product failures caused when a crucial programmer left.

Responsible companies using the language, such as FORTH Inc, address the problem by having internal cultures that stress code reviews.

1.3 Facilities of a FORTH system

Most Forth systems include a specialized assembler that produces executable words. Assembly language words usually end in a macro called "NEXT" which indexes the address interpreter to the next word, and executes it.

Classic Forth systems use no operating system. Instead of storing code in files, they store it as source-code in disk blocks written to physical disk addresses. This is more convenient than it sounds, because the numbers come to be familiar. Also, Forth programmers come to be intimately familiar with their disks' data structures, just by editing the disk. Forth systems use a single word "BLOCK" to translate the number of a 1K block of disk space into the address of a buffer containing the data. The Forth system automatically manages the buffers.

Classic Forth systems are also multitasking. They use a special word, "PAUSE" to save all the important registers to the current stack, locate the next task, and restore all the registers. Tasks are organized as a scratchpad, an area for variables used by the task, and the stacks for that task. The customary way to search for an executable task is to jump to the schedule, which is a linked list consisting of jump instructions. When a software interrupt instruction replaces the jump, the task begins to run. This system is remarkably efficient. In a Forth programming class, ten users have been supported on an 8MHz PDP-11, with each user operating out of less than 4K of RAM and sharing a single floppy disk. In a telephone system, a thousand tasks (one per phone) were supported on a small NOVA minicomputer.

1.4 Structure of the language

The basic data structure of FORTH is a "dictionary," that maps "words" to executable code or other named data structures. The general structure of the dictionary entry consists of a head and tail. The head contains the name, the indexing data, a flag byte, a pointer to the code associated with the word, and sometimes another, optional pointer to the data associated with the word. The tail has data.

The flag byte in the head of the dictionary entry distinguishes words with "compile time" behavior. Most simple words execute the same code whether they are typed on a command line, or embedded in code. Compile-time words have special meanings inside Forth code. The classic examples of compile time words are the control-structures. All of Forth's control structures, and almost all of its compiler are implemented as compile-time words.

When a word is purely executable, the code pointer simply points at the code. When a word is a variable, or other data structure, the code pointer points at code shared with other variables of that type, and a data pointer points at the data area for that specific type of variable.

1.5 Computer programs in FORTH

Words written in Forth usually compile into lists of addresses of other words, which saves very large amounts of space. The code executed by these words is an "address interpreter." The

address interpreter does just enough work to be able to execute the lowest level of words, which are written in assembly language.

1.6 Implementation of a FORTH System

Forth uses two stacks for each executing task. The stacks are the same width as the index register of the computer, so that they can be used to fetch and store addresses. One stack is the parameter stack, used to pass data to words. The other stack is the linkage stack, used to nest words, and store local variables. There are standard words to move data between the stacks, and access variables.

A Forth interpreter looks up words one at a time in the dictionary, and executes their code. The basic algorithm is to search a line of characters for a non-blank, non-control-character string. If this string is in the dictionary, and it is not a compile-time word (marked in the flag byte), the code is executed. If it is not in the dictionary, it may be a number. If it converts to a number, the number is pushed onto the parameter stack. If it does not convert, then the interpreter prints the string, followed by a question mark, and throws away the rest of the line of text.

A Forth compiler produces dictionary entries. Other than that, it tries to simulate the same effect that would be produced by typing the text into the interpreter.

The great secret to implementing Forth is natively compiling it, so that it compiles itself. The basic scheme is to have the compiler defined in terms of a few words that access a code area. Then, one definition of the words compiles to the normal area of memory.

Another definition compiles to disk, or to some special memory area. How does one adapt the compiler? One recompiles it with the new definitions. Some systems have even defined the low-level words to communicate with a debugger on a different computer, building up the Forth system in a different computer.

One mainstream use of the Forth programming language is in Open Firmware, a BIOS-replacement system developed by Sun Microsystems and used in Sun, Apple Computer.

2 Open Firmware ?

Open Firmware provides a novel capability, virtually unheard of before its invention in 1988 at Sun. This new capability is writing hardware independent boot code, firmware and device drivers.

2.1 A Brief History of Open Firmware

As mentioned above, Open Firmware was invented at Sun for release in 1988 to prevent a maintenance and support nightmare with the then unprecedented wide choice of hardware and software configurations the new product lines required. Open Firmware, then called Open Boot,

prevented the nightmare by allowing one version of the Boot Rom to run on any configuration of hardware and software, even supporting boot-time operations on third-party plug-in cards.

The idea worked so well, other major players in the computer market, such as IBM and Apple, got in on the act as well. The existence of a comprehensive IEEE standard for Open Firmware, IEEE-1275, makes this possible.

2.2 Hardware Independent Boot Code? Get Real!

This is such a new capability, it defies belief. Open Firmware provides this capability by a careful application of the FORTH philosophy. Just as FORTH has always presented its users with unique capabilities through a careful combination of

1. pushdown stack
2. dictionary and
3. interpreter,

so Open Firmware works because of its combination of

1. standardized tokens=FCode,
2. User Interface~= outer interpreter
3. Device Interface ~= inner interpreter and
4. Client Interface

to allow the OS to call Open Firmware services with the calling conventions and bindings of a high-level language, such as C.

2.3 The Tasks of Boot Code

The basic tasks of boot code are largely

1. Boot-time Drivers and
2. building a device tree,

which the Operating System then uses to discover what devices are available to it and how to use them. The particular format of the device tree is operating system dependent, but all device trees have a great deal in common. That commonality can be expressed in a common language independent of the operating system. The format of an Open Firmware device tree is such a common language. In a typical installation, the operating system uses Client Interface calls to translate the Open Firmware device tree into the Operating system's own format.

2.4 Why FORTH for all This?

Forth programmers have known for years that Forth provides a virtual machine, consisting of a data stack, a return stack and the registers IP, W, RP and SP. It is amazing how much computing can be done with such a simple machine. This machine needs only a small supplement to become an excellent virtual machine for all the tasks of boot code. This supplement is the hardware dependent portion of the Open Firmware code, whether in the Fcode on a plug-in card, or in the Host's Open Firmware interpreter.

3 OpenBIOS, free OpenSource OpenFirmware not only for PC

3.1 What is OpenBIOS

OpenBIOS is a free portable firmware implementation. The goal is to implement a 100% IEEE 1275-1994 (Referred to as OpenFirmware) compliant firmware. Among it's features, Open Firmware provides an instruction set independent device interface. This can be used to boot the operating system from expansion cards without native initialization code. It is OpenBIOS' goal to work on all common platforms, like x86, Alpha, AMD64 and IPF. Additionally OpenBIOS targets the embedded systems sector, where a sane and unified firmware is a crucial design goal.

Open Firmware is found on many servers and workstations and there are several commercial implementations from SUN, Apple, IBM, CodeGen and others.

Even though OpenBIOS has made quite some progress with it's several components, there's a lot of work to be done to get OpenBIOS booting an operating system. The basic development environment is functional, but some parts of the device initialization infrastructure are still incomplete. Our development environment consists of a forth kernel (stack based virtual machine), an fcode tokenizer and detokenizer (assembler/disassembler for forth bytecode drivers)

3.2 Why and where is Forth used?

Although invented in 1970, Forth became widely known with the advent of personal computers, where its high performance and economy of memory were attractive.

These advantages still make Forth popular in embedded microcontroller systems, in locations ranging from the Space Shuttle to the bar-code reader used by your Federal Express driver.

Forth's interactive nature streamlines the test and development of new hardware. Incremental development, a fast program-debug cycle, full interactive access to any level of the program, and the ability to work at a high ?level of abstraction? all contribute to Forth's reputation for very high programmer productivity.

These, plus the flexibility and malleability of the language, are the reasons most cited for choosing Forth for embedded systems.

3.3 FCode

FCode is a Forth dialect compliant to ANS Forth, that is available in two different forms: source and bytecode. FCode bytecode is the compiled form of FCode source.

3.4 Why bytecode

Bytecode is small and efficient. And an evaluator (bytecode virtual machine) is almost trivial to implement. For example, putting a 1 to the stack only takes one byte in an FCode bytecode binary. This is especially valuable on combined system or expansion hardware roms where the available space is limited.

4 References

- Forth Interest Group Website - http://www.forth.org/
- German Forth Gesellschaft e.V. - http://www.forth-ev.de/
- OpenBIOS Project - http://www.openbios.info/
- b16 Processor - http://www.b16-cpu.de/
- μCore Project - http://www.microcore.org/