# A Stand-alone Forth System

by

D. B. Brumm
Michigan Technological University
Electrical Engrg. Dept.
Houghton, MI 49931

Upendra D. Kulkarni
Information Processing Systems of California
70 Glenn Way
Belmont, CA 94002

ABSTRACT

A general purpose, diskless microprocessor system operating in Forth has been implemented. It behaves just like a normal disk-based system. The Forth kernel, contained in EPROMs, was generated with the Laboratory Microsystems metacomplier.

This system has the following features:

- Forth 83 standard (except for vocabularies)
- nonvolatile source code storage
- nonvolatile retention of compiled code
- interrupt support (written in Forth)
- stand-alone operation
- built-in editor
- Z80 STD bus
- low power consumption
- no mechanical devices

The source code storage area consists of up to 64 K bytes of nonvolatile memory on a separate board which is accessed through I/Oports. New source code can be entered into the screen memory by using the built-in editor, downloading through a serial link, or plugging the board into a disk-based STD bus system. Any block can be designated as a boot screen, permitting any sequence of words to be executed automatically at power-on.

The use of CMOS chips for the main memory permits the nonvolatile retention of compiled code as well, so that any application words are ready to run immediately after turning the system on.

The system is being used to control an automatic tree harvesting machine designed by the U.S. Forest Service; it can easily be adapted for other tasks.

## INTRODUCTION

That Forth is nearly ideal for the development of dedicated, real-time controllers has been pointed out by other authors (1,2). It permits one to build a very versatile stand-alone system upon which the application can be developed and tested interactively in the actual hardware environment in which it must run. Forth is also faster than most other interpreters, an essential property for many real-time applications.

Conventional Forth implementations require some sort of mass storage, usually a magnetic disk. The mass storage is needed to provide a mechanism for editing and testing new routines and for saving the results of the work. An alternative to disks must be found, however, if their use is prevented by excessive vibration, dust and dirt, low-power requirements, or temperature extremes.

This paper describes a Forth system in ROM that implements mass storage with non-volatile semiconductor memory. It appears to the user to be a conventional (but small) disk-based system. Compiled code can be saved in nonvolatile memory, an application word can run automatically at power-on or reset, and the service routine for hardware interrupts can be written in high-level Forth. The system is constructed entirely with CMOS logic to achieve low power, high noise immunity, large power supply tolerances, and a wide operating temperature range.

## METACOMPILING FORTH

A metacompiler marketed by Laboratory Microsystems, Inc., (LMI) and running on an IBM PC host was used to generate this stand-alone Z8Ø Forth system. The definitions of words such as KEY, EMIT, and ?TERMINAL in the source file are edited as required by the target hardware, and any routines needed for system-specific initialization are added to COLD (the cold boot routine). From this source file the metacompiler then produces a disk file that is ready to dump into ROMs. The resulting system basically meets the Forth-83 standard except for the absence of all words related to disk access and the lack of vocabularies.

The system generated from the source file as supplied by LMI permits new words to be added to the dictionary by keying them in from the terminal. Of course, the new definitions are lost when the power is turned off or the system is reset. There is no means of interactively creating, testing, and saving new definitions as one normally does with a disk-based system. The hardware and software additions presented here overcome these problems and yield a system that is still ROM based while permitting the usual interactive debugging and software development associated with a disk-based system.

## HARDWARE

Magnetic bubbles, battery-backed CMOS RAM, or EEPROMs could readily be used for mass storage. Using semiconductor memory chips permits adding screen memory in smaller increments (8 blocks, using 8 K byte chips), is simpler to design, and costs much less to try out. Bubbles have the advantage of a larger memory space, if needed.

The screen memory was implemented as a 64 K byte array accessed through I/O ports. Eight 28-pin sockets connected in the standard JEDEC configuration were used, permitting 8 K byte EEPROMs or CMOS RAMS to be used interchangeably. The EEPROMs require considerably more time for writing, resulting in a noticeable delay when a block is saved.

## SOFTWARE

Adding the required block support words was fairly straightforward; they were essentially copied from the disk-based source file provided by LMI. Of course, new words for accessing the screen were needed. Primitive code words S@ and S! were written to read and write one byte from and to the screen memory, respectively. The EEPROM's used will not return the same data byte written to them until the end of the internally-timed write cycle. Thus S! writes a byte, then reads it back continually until the byte returned matches the one written. This approach (rather than using a timing loop) permits the interchangeable use of nonvolatile RAMs as well as faster or slower EEPROMs with no software changes; extra time for writing is taken only if it is required. More powerful words that read and write a block at a time were then defined in terms of these simple primitives.

Three locations in block Ø have been set aside for storing a boot screen number. These locations are read by COLD; if a valid non-zero decimal number is found, then the corresponding block is loaded. This permits the execution of any desired set of words automatically at turn-on without requiring user action and greatly facilitates tailoring the system to different stand-alone applications without requiring the EPROMs to be reprogrammed.

SCREEN GENERATION AND MAINTENANCE

Several ways of generating, maintaining, and editing the screen memory contents have been implemented. A small screen editor based on the one described by Kelly and Spies (3) was included in the EPROMs so it is always instantly available without consuming any screen memory space.

Words derived from Ericson and Feucht (4) for transferring screens to and from a conventional disk-based system via a serial link were also included in the EPROMs. The screens can be initially generated using the more powerful capabilities of the larger system, then downloaded to the screen memory as needed. This capability permits the screen memory contents to be backed up on a disk and alleviates any difficulty caused by the relatively small capacity of the screen memory system.

The screen memory was constructed on a single board that plugs into the backplane. Thus the entire board can be moved to any disk-based system that uses the same bus (STD in this case). This gives instant access to both the screen memory and a disk on the same system and permits one to move screens between the two very easily and quickly.

OPERATION WITHOUT A TERMINAL

If the final application program is written such that the terminal is not needed, i.e., no keyboard or video display I/O is used, then it can be run with the terminal unplugged. This may be necessary in environments that are unfriendly to terminals. If the automatic running of an application program upon power-up without terminal control is desired, then all terminal I/O must be disabled, including the sign-on message. This can be done by using a flag to enable all terminal I/O routines, with the state of the flag being determined by COLD from the position of a switch.

HARDWARE INTERRUPTS

Real-time controllers generally require the use of hardware interrupts. Others (1,2,5,6) have described methods of implementing interrupts in a Forth system. Some approaches have either required that the entire interrupt service routine be written in machine language or that the system wait for the current Forth word to finish execution before responding. The general scheme described by Melvin (5) permits the interrupt service routine to be written (and tested) directly in high-level Forth while still achieving an immediate response to the interrupt request.

An array is defined that contains two compilation addresses; it simulates the body of a colon definition containing two words. The first cell contains the compilation address of the Forth word to be executed as an interrupt service routine while the second is for the word that returns the processor to the state that was interrupted. When an interrupt occurs the processor jumps to a particular address. A small amount of machine code at this address saves the state of the processor, loads the IP with the address of the first cell in the array and jumps to NEXT. The desired word is then executed followed by the word that returns the processor to its previous state.

NONVOLATILE COMPILED CODE

The system as described so far can function almost like a disk-based system in most respects. It cannot save a file containing a compiled application, however. The application must be loaded from screen memory (or typed in) each time the system is reset or turned on. All that is needed to permit compiled code to remain viable after the power has been off is to use nonvolatile RAM for the memory in which the compiled code is stored and to provide for the proper initialization of two pointers. The hardware modification is easily achieved by replacing one or more of the RAM chips in the main memory space with battery-backed CMOS memory chips.

The variables DP and CONTEXT must be initialized to point to the next available RAM location and the top word in the dictionary, respectively. The desired initial values can be saved in nonvolatile memory and stored in the proper locations by COLD.

REFERENCES

1. Bernier, Gerald E., "Forth Based Controller." WESCON Conf. Record, 1982, p. 17B/4
2. Dumse, Randy M., and Duane E. Smith. "High Level Language Solutions for Dedicated Applications." WESCON Conf. Record, 1982, p. 17B/2
3. Kelly, Mahlon G., and Nicholas Spies. "Forth: A Text and Reference." Prentice Hall, 1986, chap. 12 and 13
4. Ericson, Keith, and Dennis Feucht. "Simple Data Transfer Protocol." Forth Dimensions, Vol. 6, No. 2, p. 32
5. Melvin, Stephen. "Handling Interrupts in FORTH." Forth Dimensions, Vol. 4, No. 2, p. 17
6. Winterle, R. G., and W. F. S. Poehlman. "Asynchronous Words for Forth." 1984 Rochester Forth Conference, p. 32