A Stand-Alone Forth System

D. B. Brumm
Michigan Technological University
Electrical Engineering Department
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 metacompiler.

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
Z-80 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/O ports. 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.

This 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

The conventional approach to implementing a microprocessor-based controller involves putting the object code into EPROMS in a seemingly unending cycle of edit, assemble (or compile, if a high-level language is used), program the EPROM, and test. The slightest program change or addition requires at least one repetition (usually more) of the entire cycle. This is a painful and time-consuming process.

Debugging and testing such a system is further complicated because the software often does not include any hooks or routines for such purposes. Some simple operations such as the ability to read and modify RAM locations and registers can help somewhat, but much more control over the program is usually needed. An interactive ability is almost essential when some part of the hardware or software does not work as expected. The designer needs to be able to exercise immediate control over various routines or peripherals rather than to only observe system behavior under the control of some inflexible main routine.

Other authors (1,2,3) have pointed out that Forth is nearly ideal for the development of dedicated, real-time controllers. It permits one to build a very versatile stand-alone system upon which the application can be developed quickly and easily while incurring a relatively small speed penalty (compared with optimized machine language). With Forth, routines can be exercised as desired; any set of input parameters can be used for initializing routines; and output parameters can be viewed or changed. In short, the operator is in control of the system rather than vice-versa.

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. Thus it can operate from batteries for extended periods in a wide variety of environments and it can be sealed in a box for protection, if necessary (no fan or air circulation is required).

Metacompiling Forth

One means of generating a new Forth system that is tailored for a specific environment is by using a metacompiler. The general characteristics and uses of Forth metacompilers have been described by Laxen (4). In the context of this paper metacompilation is the process of using a special Forth program (the metacompiler) running on a host computer to convert an input text file (the target source file) into a binary ROM image file. The image is then dumped into one or more EPROMs which are plugged into the target system; the result is an implementation of Forth in ROM on the target machine.

The built-in words and capabilities of the target computer can be altered, enhanced, and supplemented by editing the target source file. This file looks very much like a conventional Forth screen file except that forward references can be used (words can be used in the definition of a new word before they have been defined). There are also special commands available called compiler directives that permit the user to control the metacompiling process (such as to generate headerless code, for example).

The metacompiler used for this work (5) is available from Laboratory Microsystems, Inc. (LMI). It runs on an IBM PC for a Z80 target (target source files for many other microprocessors are available from LMI). The package supplied by LMI (version 2.1) includes target source files for both ROM and disk based systems and a well-written manual.

The image generated for a disk-based system is tailored for use under the CP/M operating system and meets the Forth-83 Standard. This is a generic Forth-83 that is not the same as LMI's Z-80 Forth which is further optimized and extended.

The ROM-based target source file also meets the Forth-83 Standard except for the absence of all words related to disk access and the lack of vocabularies. 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 an image file (on disk) that is ready to dump into EPROMs (or a ROM if it is to be mass produced).

This particular metacompiler permits the creation of new defining words in the metacompilation process as well as words that use them. This is a powerful capability. Data and control structures can be put into ROMs nearly as readily as they can be generated with a conventional Forth system.

The system generated from the ROM 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. An alternative method of adding words is to edit them into the target source file, rerun the metacompiler, and reprogram the EPROMs. Although this is a lengthy process (much like using a conventional compiler), the added words are instantly available when the system is turned on or reset. The results of these two different processes are either permanent or volatile; there is no means of interactivly 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 nonvolatile memory space (called screen memory in this paper) that can be used to implement the virtual memory structure used by most Forth systems. A bubble memory card would probably be quite satisfactory for this purpose in most cases. One particular application of this system is to control a tree harvesting machine, however, which requires reliable operation at low temperatures. Since suitable bubble devices were not yet available when this project started, the other two approaches were used for this implementation. A semiconductor memory system also permits adding screen memory in smaller increments (8 blocks or screens, using 8 K X 8 memory ICs), is simpler to design (especially for one who has not previously used bubble devices), and costs much less to try out. Bubbles would have the advantage of a larger memory space (512 blocks for a 4 M bit unit), but this much space is seldom needed.

The screen memory was implemented as a 64 K byte space accessed through three I/O ports. Two output ports (each 8 bits wide) were used for the address, and a bidirectional port was used for the data. Eight 28-pin sockets connected in the standard JEDEC configuration were used, permitting 8 K byte EEPROMs or CMOS RAMs to be used.

Both nonvolatile memory types were tried. The battery-backed RAMs selected were those with lithium cells inside a molded package (DS1225 by Dallas Semiconductor). No electrical problems have been experienced with these units, but the package is higher and heavier than conventional packages and must be held down with retainers if used in a system subject to vibration. We have also used AMD AM9864 and Xicor X2864 EEPROMs. The read time required for these devices is the same as for the RAMs, but they require considerably more time for writing. This results 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 target source file provided by LMI. Of course, new words for accessing the screen memory were needed. These primitives are shown in screens 2-5. `S@` (screen 1) is the code word that reads a byte from the screen memory array. We have elected to use a percent sign as the initial character in metacompiler equates (such as `%HIBYTE`) to distinguish them from constants. Metacompiler equates act like constants during the meta-compilation process in that they associate a value with a symbolic name, but no headers are built for them in the system that is being compiled. Consequently their names are not recognized by the target system and fewer bytes are used in the target memory.

`S!` (screen 2) writes a byte to the screen memory. An EEPROM (of the types used) will not return the same data written to it 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 fixed 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.

Some EEPROMS have a page write mode in which several bytes can be stored during one write cycle of a few milliseconds. Utilizing this mode would greatly decrease the time required to update a block, but it requires that an entire page be latched into the EEPROM within a certain time interval (16 bytes within 200 microseconds in the Xicor units, for example). The page write mode has not been implemented in this system because of the increased complexity and because the present method has been fast enough for our purposes.

`SCR!` (screen 3) is a high-level equivalent of `S!` but without a wait loop. It could replace `S!` if nonvolatile RAMs are used for screen memory and is included here to demonstrate how the screen memory is accessed for those who may be unfamiliar with Z80 `CODE` definitions. An equivalent Forth definition to replace `S@` is also readily written.

The words `M>S` and `S>M` (screen 3) move multiple-byte strings from RAM to screen memory or vice-versa; they are used directly by `BLK-WRITE` and `BLK-READ` (screen 4), respectively, to perform block transfers. These words are also useful for moving data arrays from or to screen memory.

The screen memory can be segregated into separate sections for programs and data, if desired. The variable `#SCR` (screen 5) contains the number of contiguous blocks (beginning with block 0) that are intended for program storage and that contain only textual material (seven bit ASCII characters). These blocks are called screens and are accessable to words such as `LIST`, `EDIT`, and `LOAD` (except for screen 0 which cannot be loaded). The remaining blocks, if any, can be used for data in any desired format.

Program screens are denoted by a backslash in their first byte (which also makes the first line of these screens a comment). `SET-#SCR` (screen 5) will initialize `#SCR` by searching for the first missing backslash. `?BLK` (screen 5) is then used to prevent `LIST`, `LOAD`, and `EDIT` from gaining access to the blocks intended for data. This obviates crashes and other surprises caused by listing, loading, or trying to edit the data blocks, especially by Forth novices. It enables a friendlier environment to be created, and in a sense permits dividing the screen memory into two separate files.

Since block 0 cannot be loaded, it provides adequate nonvolatile storage space for a small amount of data in a form readable by the user. One example is the storage of a block number for a boot screen to be loaded whenever the system is turned on or reset. The words presented in screens 6-7 implement this function. The number of the desired boot screen is coded as three ASCII characters stored in locations 2-4 in block 0. This permits designating any one of 999 different blocks and provides for a future increase in the size of the screen memory. The three locations are read by `?BOOT` (screen 6) which is executed by `SCR-BOOT` which is, in turn, part of `COLD` (screen 13). If a valid non-zero decimal number is found, then the corresponding screen is loaded. Of course, the boot screen can in turn load other screens so there is virtually no limit to what can be done automatically at turn-on without requiring user action. This feature greatly facilitates tailoring the system for different stand-alone applications without requiring the EPROMs to be reprogrammed. The desired boot screen number can be saved by putting it on the stack and executing `!BOOT-SCR` (screen 7) or by editing the appropriate locations in screen 0. The word `S>T` (definition not shown) types a string fetched from screen memory. It is used by `.BOOT-SCR` (screen 7) which displays the current boot screen number on the terminal as well as by `INDEX` (also not shown) which displays the first line of all screens (i.e., of all blocks from 0 through `#SCR`-1).

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 (6) was included in the EPROMs so it is always instantly available without consuming any screen memory space. This editor is a good example of the power of Forth. It only consumes about 2900 bytes (including a help screen) in an EPROM while implementing sufficient text-massaging capabilities to permit adding or modifying screens with ease.

Words derived from Ericson and Feucht (7) for transferring screens to and from a conventional disk-based system via a serial link were also included in the EPROMs; they require about 490 bytes. This facility permits any disk-based system that has both Forth and an auxiliary serial port to be used for mass storage of Forth screens. The screens can be initially generated using the more powerful capabilities of the larger system, then downloaded to the screen memory as needed. This also permits the screen memory contents to be backed up on a disk and alleviates some difficulties caused by the relatively small capacity of the screen memory system. The 63 block limit (block 0 cannot be loaded) still applies for each application, but the programs required for multiple applications can readily be saved and restored as desired.

Blocks containing arbitrary binary data can be sent through the serial link if both ports are set up to send and receive eight data bits rather than the usual seven. Thus any data generated by the specific application can also be transferred to a more powerful system in this manner. It should be noted however that this protocal has no error checking; a more robust method should be used if the channel is not highly reliable.

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 board and a disk on the same system and permits one to move blocks between the screen memory and the disk very easily and quickly.

The nonvolatile memory ICs can also be unplugged from one system and inserted into another. This permits moving groups of eight blocks, if desired. The groups of blocks can also be rearranged in this manner.

Blocks could also be dumped into 8 K byte EPROMs for semi-permanent storage. This could be useful, for example, for mature code that is seldom used that one does not wish to metacompile into the target EPROMs (thus saving main memory space), for on-board system documentation, or to prevent software alteration or tampering (while preserving the viewability of source code). Since EPROMs are considerably cheaper than the other nonvolatile memory types, installing them in the screen memory has economic advantages under certain conditions.

Operation Without a Terminal

If the final application word 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 some environments where vibration is a problem, where suitable power is not available, or where there is not adequate space available for a terminal. 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 or a jumper. An alternative for some applications is to use a small, battery-operated lap-top computer running a terminal emulator program to replace a conventional terminal.

Hardware Interrupts

Real-time controllers generally require the use of hardware interrupts. Other authors (1,2,8,9) 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 they have incurred a delay while waiting for the current Forth word to finish execution. An interrupt system in which the response is immediate is needed in many real-time systems. Monitoring the speed of a shaft, for example, could require periodic sampling of its position; variable delays can't be tolerated in such a case. The general scheme described by Melvin (8) 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.

The words used for this purpose are shown in screens 8 and 9. An array is defined (INT-VECT in screen 8) that contains two compilation addresses; it simulates the body of a colon definition containing two words. The first cell is used for the compilation address of the Forth word to be executed in response to an interrupt, while the second is for the word that returns the processor to the state that was interrupted. The directive ALLOT-RAM forces the alloted space to be in the target RAM rather than in the ROM image.

The word PUT (screen 8) initializes INT-VECT by storing the compilation address of the desired interrupt response word in the first cell and the compilation address of the CODE word RETURN (screen 8) in the second cell. Executing PUT <NAME> thus sets up the array to execute <NAME> when an interrupt occurs.

%INTSRV (screen 9) is equated to the beginning address of the interrupt service routine for the interrupt mode being used (address 0038H for interrupt mode 1 on the Z-80). Since the assembler routine labelled INTSRV must be located at this address in the EPROM, this screen must be located near the beginning of the target source file, not in the relative location shown here. INTSERV saves the contents of the registers on the parameter stack, initializes the users return stack (with UR0), fetches the compilation address stored in the first cell of INT-VECT, and jumps to NEXT.

This sequence of events serves to set up the virtual Forth machine to execute <NAME> and RETURN as though they were parts of a normal Forth word without disturbing the previous state of the processor. A new return stack is needed because IY (the Z-80 register used for the Forth return stack pointer RP) does not always point to the top of the return stack. Failing to provide for a separate return stack could cause the system to crash when RETURN is executed. Thus after <NAME> is executed, the second cell of the vector directs the program flow back to the CODE word RETURN which pops the saved registers and returns to the interrupted word.

The compiler directive L: identifies INTSERV as a label for the metacompiler that is associated with the current target address (0038H in this case). No header is built for INTSERV and the word is not recognized by the target system. The contents of location 0038H (in the Z-80) must be an executable machine instruction, not the beginning of a name field.

INT-ON and INT-OFF (screen 8) permit enabling and disabling the interrupts. The Z-80 has three different interrupt modes. Mode 1 (the simplest one) is set by INT-ON. Operation in this mode forces a jump to address 0038H for all enabled maskable interrupt requests.

The word installed by PUT (called <NAME> in the previous example) must not use the terminal or otherwise alter any system variables indiscriminately. EMIT increments OUT, for example, which keeps track of the current column on the terminal. Using EMIT within an interrupt service word would upset any formatting being done on the screen.

Nonvolatile Compiled Code

This 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 software required to implement this feature is shown in screens 10-13.

The word COLD, as supplied by LMI, initializes the variables DP and CONTEXT to point to the next available RAM location and the top word in the dictionary, respectively. Since the initial values of these pointers (INIT-DP and INIT-FORTH) are in EPROM, they cannot be changed. The variables WARM-DP and WARM-CONTEXT (screen 10) were added to permit a new word (WARM) (screen 11) to initialize DP and CONTEXT with values previously saved in nonvolatile memory rather than the unalterable ones. When COLD (screen 13) executes, it will execute either (WARM) or (COLD) depending on the setting of a switch as determined by ?WARM (definition not shown).

The values stored in WARM-DP and WARM-CONTEXT are set by FREEZE and THAW (screen 10). FREEZE will cause the current dictionary contents to be preserved and reestablished after a reset while THAW sets the warm values to be the same as the cold ones. FORGET (screen 10) was also modified to prevent losing part of the frozen dictionary.

A switch is used to determine whether (COLD) or (WARM) will be executed rather than a software flag because a crash could potentially corrupt the warm variables without altering the state of the flag, making recovery impossible. This situation is avoided by using an external switch; the user can always recover from a crash by putting the switch in the cold position and pushing the reset button.

Summary

The techniques described here have been implemented in two systems. One is for general laboratory use in an academic setting and has been used primarily for software and hardware development and testing and for student projects. The other is being used by the U. S. Forest Service as a controller for an automated tree harvesting machine (10). Interrupts are used in this second system for monitoring the speeds and positions of several hydraulically-controlled mechanisms. The use of Forth in developing this harvester greatly reduced the software development time and facilitated hardware design and testing; it also permits the software to be altered in the field (more accurately, in the woods).

References

1. Bernier, Gerald E. "Forth Based Controller." *WESCON Conference Record*, 1982, p. 17B/4.

2. Dumse, Randy M., and Duane E. Smith. "High Level Language Solutions for Dedicated Applications." *WESCON Conference Record*, 1982, p. 17B/2.

3. Solley, Evan L. "Sphere: An In-circuit Development System with a Forth Heritage." *1984 Rochester Forth Applications Conference*, p. 25.

4. Laxen, Henry. "Techniques Tutorial: Meta Compiling." *Forth Dimensions*, 4(6):19, 5(2):23, 5(3):31.

5. Duncan, Ray, and Richard Wilton. *LMI Forth-83 Metacompiler.* Laboratory Microsystems, Inc. 1986.

6. Kelly, Mahlon G., and Nicholas Spies. *Forth: A Text and Reference.* Prentice Hall, 1986, chap. 12 and 13.

7. Ericson, Keith, and Dennis Feucht. "Simple Data Transfer Protocol." *Forth Dimensions*, 6(2):32.

8. Melvin, Stephen. "Handling Interrupts in FORTH." *Forth Dimensions*, 4(2):17.

9. Winterle, R. G., and W. F. S. Poehlman. "Asynchronous Words for Forth." *1984 Rochester Forth Applications Conference*, p. 32.

10. Brumm, D. B., and Michael A. Wehr. "A Forth-Controlled Tree Harvester." *Proceedings IECON '86*, 1986, p. 851.

D. B. Brumm received a B.S. from Michigan Technological University and an M.S. and Ph.D. from the University of Michigan, all in electrical engineering. He is currently an Associate Professor of Electrical Engineering at Michigan Technological University. His interests include small computer systems, instrumentation, and fiber optics.

Upendra D. Kulkarni received a B.S. from the University of Bombay and an M.S. from Michigan Technological University, both in electrical engineering. He is currently a Computer Engineer with Information Processing Systems of California. His interests are satellite and weather image processing.

Manuscript received August 1986.

Glossary

!BOOT-SCR n ---
Store boot screen number. Set boot screen to n by storing three ASCII characters into locations 2, 3, and 4 in block 0.

.BOOT-SCR ---
Print boot screen number. Retrieve three ASCII characters from locations 2, 3, and 4 in block 0 and display them on the terminal.

#SCR --- addr
Variable containing the number of contiguous blocks (including block 0) available for source code storage.

%HIBYTE
A metacompiler equate for the address of the output port to which the high byte of the screen memory address must be sent. This word is not in the dictionary of the target system.

%INTSRV
A metacompiler equate for **0038H**, the address at which the Z80 mode 1 interrupt service routine must start. This word is not in the dictionary of the target system.

%LOBYTE
A metacompiler equate for the address of the output port to which the low byte of the screen memory address must be sent. This word is not in the dictionary of the target system.

%MAX#SCR
A metacompiler equate for the maximum number of screens or blocks supported by the target system hardware. Limited to 64 in the system described. This word is not in the dictionary of the target system.

%SCRPORT
A metacompiler equate for the address of the I/O port used for sending data to and from the screen memory. This word is not in the dictionary of the target system.

(COLD) ---
Initialize system pointers for a cold boot. The dictionary will contain only the words compiled into ROM by the metacompiler. **DP** (the dictionary pointer) is set to **INIT-DP** and **CONTEXT** is set to **INIT-FORTH**. **INT-VECT** is initialized to execute **NOOP**, a null word, in response to an interrupt.

(WARM) ---
Initialize system pointers for a warm boot. All words compiled before **FREEZE** was most recently executed will be in the dictionary. Hardware interrupts will be disabled but the contents of **INIT-VECT** will not be disturbed.

?BLK n --- n
Abort with error message if n is outside the range of permitted screen numbers (0 through **#SCR** -1).

?BOOT --- n true
or --- false
Interrogate bytes 2, 3, and 4 in block zero and determine if they are the ASCII codes for a three-digit number greater than zero. If they are then the corresponding number is left on the stack along with a true flag. **?BOOT** does not compare the number with #SCR.

BLK-READ addr n ---
Read block n from the screen memory into a block buffer beginning at addr.

BLK-WRITE addr n ---
Write from a block buffer beginning at addr into block n in the screen memory.

BOOT ---
Portion of initialization sequence that is executed after every reset. Always executed as part of COLD.

COLD ---
Main initialization routine; executed after every reset. This word executes BOOT plus either (WARM) or (COLD). It also issues a sign-on message and loads a boot screen if one is defined.

FORGET ---
FORGET <name> deletes <name> and all words added to the dictionary after <name>. If <name> was compiled before FREEZE was most recently executed, then FORGET aborts and an error message is displayed.

FREEZE ---
FREEZE performs two functions. It saves the current values of DP and CONTEXT in WARM-DP and WARM-CONTEXT, respectively. The current dictionary contents at the time FREEZE is executed will still be resident after a power down condition while any words compiled subsequently will be lost. FREEZE also establishes a fence that protects all current words from FORGET.

INDEX ---
Display the first line of all blocks from 0 through #SCR-1.

INIT-DP ---
Metacompiler label for a location in ROM that contains the first free location in the target RAM. It is used as the initial value for the variable DP when a cold boot is performed. This word is not in the dictionary of the target system.

INIT-FORTH ---
Metacompiler label for a location in ROM that contains the initial value for the variable CONTEXT when a cold boot is performed. This word is not in the dictionary of the target system.

INT-OFF ---
Interrupt off. Disables maskable hardware interrupts.

INT-ON ---
Interrupt on. Enables hardware interrupts (Z80 mode 1 in the system described).

INT-VECT --- addr
Two-element array. Contains the execution address of the word to be executed when an interrupt occurs as well as the address of the word RETURN. The array is initialized by PUT.

INTSRV
Metacompiler label associated with the address in the target system given by %INTSRV. This word is not in the dictionary of the target system.

M>S buf-addr scr-addr n ---
Move n bytes from RAM beginning at buf-addr to screen memory beginning at scr-addr.

PUT ---
Initialize INT-VECT. Used in the form PUT <name> which places the compilation address of <name> into the first element of INT-VECT and the address of RETURN into the second. A

hardware interrupt will then cause <name> to be executed immediately followed by a return to the interrupted word.

RCV n ---
Receive a sequence of consecutive blocks through the auxiliary serial port and save them in screen memory. The protocol described in reference 7 is used; no error checking is done.

RETURN ---
Return from interrupt. A CODE word that restores the CPU register contents, thus restoring the virtual Forth machine to its status before the interrupt occurred. PUT stores the compilation address of RETURN into the second element of INT-VECT.

S! byte addr ---
Store a byte in location addr in the screen memory. The code version given reads back the stored byte until the byte fetched is equal to the byte stored. This permits using self-timed EEPROMs for screen memory and signifies that the byte has been successfully written.

S>M scr-addr buf-addr n ---
Move n bytes from screen memory beginning at scr-addr to RAM beginning at buf-addr.

S>T addr n ---
Similar to the Forth-83 word TYPE. Send n characters from screen memory, beginning with addr, to the terminal.

S@ addr --- byte
Fetch a byte from location addr in the screen memory.

SCR! byte addr ---
A high level version of S! without the capability of using self-timed EEPROMs.

SCR-BOOT ---
Load boot screen if it is greater than zero. The boot screen number is stored as ASCII characters in bytes 2, 3, and 4 of block zero.

SET-#SCR ---
Determine the number of blocks available for source code storage by searching for the first block with a backslash missing from its first location. The number found is stored into the variable #SCR.

THAW ---
Cancels the effects of FREEZE. A warm boot will then be equivalent to a cold one and FORGET can operate on any word compiled into RAM (words put in ROM by the metacompiler cannot be forgotten, of course). THAW sets INIT-DP and INIT-FORTH equal to WARM-DP and WARM-CONTEXT, respectively.

WARM-CONTEXT --- addr
A variable containing the initial value of CONTEXT to be used for a warm boot. WARM-CONTEXT is set equal to the current value of CONTEXT by FREEZE.

WARM-DP --- addr
A variable containing the initial value of DP to be used for a warm boot. All words in the dictionary below WARM-DP remain resident in nonvolatile memory and FORGET is not permitted below this position (acts like FENCE in LMI versions of FORTH). WARM-DP is set equal to the current value of DP by FREEZE.

XMT n1 n2 ---
Send blocks n1 through n2 to another system through the auxiliary serial interface. The protocol described in reference 7 is used; no error checking is done.

```
Screen # 1
   ( s@                                                DBB     860528)

   HEX

   CODE    S@      ( addr -- b ) ( fetch byte from screen memory )
           HL POP                          ( get address from stack )
           A, H LD    %HIBYTE , A OUT              ( output addr )
           A, L LD    %LOBYTE , A OUT
           A, %SCRPORT IN                 ( fetch byte from addr )
           L, A LD    H, # 0 LD    HL PUSH   ( put byte on stack )
           NEXT JP    END-CODE



Screen # 2
   ( s!                                                DBB     860528)

   CODE    S!        ( b addr -- ) ( store byte in screen memory )
           HL POP                            ( get addr from stack )
           A, H LD    %HIBYTE , A OUT              ( output addr )
           A, L LD    %LOBYTE , A OUT
           HL POP                            ( get byte from stack )
           A, L LD    %SCRPORT , A OUT      ( store byte to addr )
                      ( loop until byte is stored -- for EEPROM )
       1$: A, %SCRPORT IN                    ( read byte from addr )
           A, L CP                  ( compare it with byte stored )
           NZ, 1$ JR               ( loop until they are the same )
           NEXT JP         END-CODE



Screen # 3
   ( scr! m>s s>m                                      DBB     860528)

   : SCR!                  ( b addr -- ) ( high-level equiv of S! )
           DUP %LOBYTE P!          ( without wait for data match )
           100 / %HIBYTE P!
           %SCRPORT P! ;
   : M>S                                  ( buf-addr scr-addr n -- )
                        ( move n bytes from RAM to screen memory )
           OVER + SWAP
           DO DUP C@   I S!   1+
           LOOP
           DROP ;
    : S>M   ( scr-addr buf-addr n -- )
           OVER + SWAP         ( move n bytes from scr mem to RAM )
           DO DUP S@   I C!   1+ LOOP DROP ;
```

```
Screen # 4
    ( blk-read  blk-write                                  DBB    860528)

    : BLK-READ                                          ( addr blk ---  )
                    ( read a block from scr memory to buffer )
            B/BUF * SWAP B/BUF S>M ;
    : BLK-WRITE                                         ( addr blk ---  )
                    ( write a block from buffer to screen memory  )
           B/BUF * B/BUF M>S ;



Screen # 5
    ( #scr set-#scr ?blk                                   DBB    860528)

    40 EQU %MAX#SCR          ( maximum possible number of screens )
                                   ( limited to 64 in this system)
                                     ( by hardware configuration )
    VARIABLE #SCR       ( number of screens for prog in SMEM board )
    : SET-#SCR      (  --  ) ( find number of screens and set #SCR )
              ( looks for a missing '\' in first addr of each scr )
            -1 BEGIN
                    1+ DUP B/BUF * S@ 5C <>           ( 5CH = '\' )
                    OVER %MAX#SCR = OR
            UNTIL #SCR ! ;
    : ?BLK    ( n -- n ) ( error if screen number is out of range )
            DUP #SCR @ U< 0=
            ABORT" screen # out of range" ;

Screen # 6
   ( ?boot scr-boot                                      DBB    861030)
   : ?BOOT                  ( -- n t | f )  ( valid boot screen? )
           3 PAD C!                                      ( store count )
           2 ( scr-addr )        PAD 1+ ( buff addr for digits )
           3 S>M ( move 3 bytes )  20 PAD 4 + C! ( append a space)
           PAD NUMBER?          ( -- d f ) ( convert to a number? )
           SWAP DROP                ( convert d to 16 bit number )
                   ( valid boot scr if it is a positive number )
           DUP IF
                 OVER 0> 0=      ( if negative or zero, return 0 )
                 IF 2DROP 0 THEN
               ELSE SWAP DROP
               THEN ;
   : SCR-BOOT                  (  ) ( load boot scr if not zero   )
           ?BOOT     IF CR ." loading boot screen " CR LOAD THEN ;
```

```
Screen # 7
   ( !boot-scr .boot-scr                                DBB     860528)

   : !BOOT-SCR                          ( n -- ) ( set boot scr to n )
         FLUSH ?BLK   0 <# # # # #>  02 SWAP   M>S ;
   : .BOOT-SCR                           ( -- ) ( print boot scr # )
         2 SPACES   2 3 S>T SPACE ;
```

```
Screen # 8
   ( int-vect, return, put, int-on, int-off        DBB     860528)

   VARIABLE INT-VECT  2 ALLOT-RAM    ( array for int serv routine)
   CODE   RETURN                          ( return from interrupt)
          HL POP  DE POP  BC POP            ( restore registers )
          EXX  IY POP  IX POP  HL POP
          DE POP  BC POP  AF POP
          EI   RETI   END-CODE   ( enable interrupts and return )
   : PUT                ( PUT <name> vectors interrupt to <name> )
          [COMPILE] ' INT-VECT !
          ['] RETURN INT-VECT 2+ ! ;
   CODE INT-ON                          ( enable mode 1 interrupts )
          IM1 EI   NEXT JP   END-CODE
   CODE INT-OFF                              ( disable interrupts )
          DI   NEXT JP   END-CODE
```

```
Screen # 9
   ( interrupt service                              DBB     860529)

   FORTH
   %INTSRV HERE -       ( move to beginning of interrupt routine )
   HERE SWAP DUP ALLOT FF FILL             ( fill space with FF )
   ASSEMBLER
   L: INTSRV
          DI   AF PUSH   BC PUSH           ( disable interrupts )
          DE PUSH  HL PUSH                    ( save registers )
          IX PUSH  IY PUSH  EXX
          BC PUSH  DE PUSH  HL PUSH
          IY, UR0 LD             ( initialize user return stack )
          BC, # INT-VECT LD          ( load IP with array addr )
          NEXT JP
   FORTH
```

```
Screen # 10
   ( warm variables, freeze, thaw, forget          DBB     860204)
   VARIABLE WARM-DP                ( dict pointer for warm start )
   VARIABLE WARM-CONTEXT                ( context for warm start )

   : FREEZE                    (  --  ) ( save state, erect fence )
          HERE WARM-DP !  CONTEXT @ WARM-CONTEXT ! ;
   : THAW                           (  --  ) ( restore cold values )
          INIT-DP @ WARM-DP !  INIT-FORTH @ WARM-CONTEXT ! ;
   : FORGET                ( FORGET <name>, forget back to <name> )
          ' DUP >NAME DUP WARM-DP @ U<
                 ABORT" frozen, can't FORGET"
          DP ! >LINK @ CONTEXT ! ;


Screen # 11
   ( (cold (warm                                   DBB     860529)

   : (COLD)                  ( cold boot of nonvolatile variables )
          INIT-DP @ DUP DP ! WARM-DP !       ( set cold pointers )
          INIT-FORTH @ DUP CONTEXT ! WARM-CONTEXT !
          ['] NOOP INT-VECT !             ( init interrupt array )
          ['] RETURN INT-VECT 2+ !               ( to do nothing )
          ." Cold boot " CR ;
   : (WARM)                  ( warm boot of nonvolatile variables )
          INT-OFF                           ( disable interrupts )
          WARM-DP @ DP !                     ( set warm pointers )
          WARM-CONTEXT @ CONTEXT !
          ." Warm boot " CR ;


Screen # 12
   ( boot                                          DB      860529)

   : BOOT  ( initialize system, words executed after every reset )
          EMPTY-BUFFERS FIRST PREV ! FIRST USE !
          INIT-S0 @ S0 !     INIT-R0 @ R0 !
          INIT-UR0 @ UR0 !
          INIT-TIB @ <TIB> !
          BLK OFF    SCR OFF    SPAN OFF    NO_SKIP OFF
          STATE OFF  >IN OFF     DPL OFF     OUT OFF
          SER-INIT                       ( initialize serial ports )
          CONSOLE DECIMAL   CLS ;                ( clear screen )
```

```
Screen # 13
   ( cold                                                DBB     860528)

   : COLD                          ( executed at power-on or reset )
         BOOT                                ( initialize pointers )
         ?WARM                             ( check switch position )
            IF (WARM)                              ( execute warm or )
            ELSE (COLD)                  ( cold boot, depending on )
            THEN                              ( the switch position )
         IDENT                           ( display sign-on message )
         SET-#SCR      ( set number of screens in screen memory )
         SCR-BOOT       ( load boot screen if one is designated )
         ABORT ;                                ( start interpreter )
```