Frank Sergeant

A 3-INSTRUCTION FORTH FOR EMBEDDED SYSTEMS WORK

Illustrated on the Motorola MC68HC11

by Frank Sergeant

ABSTRACT

How many instructions does it take to make a Forth for target development work? Does memory grow on trees? Does the cost of the development system come out of your own pocket? A 3-instruction Forth makes Forth affordable for target systems with very limited memory. It can be brought up quickly on strange new hardware. You don't have to do without Forth because of memory or time limitations. It only takes 66 bytes for the Motorola MC68HC11. Full source is provided.

Literary Justifications

I come to Forthifize the 68HC11 not to praise it.

```
Had we but memory enough and time,
    this wastefulness, Programmer, were no crime.
```

```
The embedded system box is a private place and fine,
    but none should there, I think, megabyte ROMs enshrine.
```

Do You Want Forth on the Target Board?

Yes. In a FORML paper, it seems reasonable to assume you do want Forth on the target board, if possible, if you can afford it. Without Forth, maybe without a monitor ROM of any sort, you might find yourself in this position: You have to write some code, burn it into an EPROM, guess why it doesn't work, make changes to your code, burn another EPROM … and do this over and over until something begins to work. Finally you get an LED to flash and your non- programmer friends wonder why you are so excited over such a little thing! You say there's got to be an easier way and start to wish you could afford expensive development systems.

But how do you **put** Forth on the target? If you write your own, you might find yourself in that same situation as you burn EPROMs and wonder why the Forth doesn't run. If you buy one, it might not work with your hardware configuration, it might be too expensive, it might not include full source. You might not have **time** to do all this before starting on your real project. The end product might not be able to afford the PCB space, ROM, and RAM that a Forth requires.

Let's Design the Simplest Possible Forth

Might there be a way to make a very small Forth? How much of the burden can we offload to the host system and what is the minimum that must be done in the target? We are used to the idea of the host providing text editing and source storage services. Let's see if it can also provide all the high-level code while the target provides only the primitives.

The absolute minimum the target must do, it seems to me, is fetch a byte, store a byte, and call a subroutine. Everything else can be done in high-level Forth on the host. (I'm assuming the host and target are connected over a serial line, and that sending and receiving serial bytes will be included as part of accomplishing the three primitive functions.) With those three primitives, memory can be inspected and altered, new routines can be downloaded, and the new routines can be executed. Let's call this a 3-instruction Forth. "Everything else being the same," a 3-instruction Forth should be pretty easy to write and pretty easy to debug. Let's start with a pseudocode description:

```
1. Initialize the processor.
2. Initialize the serial port.
3. Repeat the following forever:
   Get a byte from the serial port.
   If byte = 01  [fetch]
      A. Get address from the serial port.
      B. Fetch the byte from that address.
      C. Send the byte to the serial port.
   Else If byte = 02  [store]
      A. Get address from the serial port.
      B. Get a byte from the serial port.
```

```
   C. Store the byte at that address.
Else If byte = 03  [call]
   A. Get address from the serial port.
   B. Jump to the subroutine at that address.
End If.
```

Step 1 and 2 include whatever housekeeping tasks are needed to set up the processor and the serial port such as initializing the stack and selecting various modes. These steps are particularly simple on the ’HC11. When it wakes up in its bootstrap mode, an internal ROM sets up the stack, turns on the serial port, and waits for a program to be downloaded over the serial line. On the ’HC11, step 1 consists of establishing addressability of the internal registers and ports. This is done by loading Register X with the base address of the register space. The ’HC11’s serial out line wakes up in the “wire-or” mode. The only thing we have to do for Step 2 is to un-wire-or it. Doing so eliminates the necessity of putting a pullup resistor on that pin.

In the main part of the routine (step 3), the functions of getting a byte or getting an address from the serial port are required in a number of places, thus they are good candidates for subroutines. Indeed, getting an address can make use of the get a byte subroutine. A byte is sent back to the host in only one place, so there is no point in making that into a subroutine.

Philosophy?

Yes, we could eliminate that last instruction and still be able to set and examine all of the target’s RAM and registers. Unfortunately we wouldn’t be able to test actual code on the target. With only two instructions (fetch and store) we could exercise the devil out of the hardware, but we really need that third instruction (call) in order to extend the Forth. Thus, I would not be entirely happy to call it Forth with only the first two instructions.

Is it fair to call it a Forth even with all three instructions? After all, it lacks a data stack, headers, either the inner or the outer interpreter (I vacillate on this). It relies on the host (but, then, which target Forth boards connected to a PC over a serial line do not rely on their hosts to at least some extent?). It is extensible and even the main loop can be extended to test for codes in addition to 01, 02, and 03. So, is it a Forth or not? I lean toward “yes.” It has the Forth attributes of simplicity, economy, and (I hope) elegance. What, though, about linking together calls to earlier routines in order to build later routines? The most obvious approach is to build simple assembly language routines on the host and download them to the target one at a time and test them. Then download higher level assembly routines that call those earlier ones. This is the most memory efficient approach. If you do have extra RAM available on the system then you can use the 3-instruction Forth to build and test a full Forth, and then use it. Either way, starting with a 3-instruction Forth you get to work in an interactive, Forth-like environment.

Can We Afford It?

“But wait,” you say, “the 68HC11 has only 192 bytes of onboard RAM (or 256 or 512, depending on the version) – surely it’s not big enough for a Forth or any other monitor!” No, not for a conventional monitor or a conventional Forth. However, it’s got memory to spare for a 3-instruction Forth, and that’s all we need. This 3- instruction Forth for the ’HC11 only takes 66 bytes on the target. It was created with Pygmy Forth running on a PC. Following the source code listing is a listing of the actual 66 bytes of ’HC11 machine language, so you should be able to use it with any Forth running on any host. An old slow PC is adequate; no fast, expensive development system is required.

Once that tiny program is running you can use it to do everything else you need. This technique works with all micros, but it is especially easy with the Motorola 68HC11 family. The first reason is that I’m including a listing of the program right here. If you are not using an ’HC11 then you’ll have to write an equivalent program for another micro. The second reason is the ’HC11’s lovely serial boot loader mode. This eliminates the EPROM burning stage and lets you download the monitor over the serial line.

The 68HC11 gives you another bonus. In the bootstrap mode, the internal ROM initializes the chip and turns on the built-in serial port. With another micro you’ll need to write your own code to do this step. If the micro does not have an on-board serial port you’ll need to add one, either externally with a serial or parallel port chip, or with on- board I/O lines.

Distributed Development

The 3 instructions are

```
XC@       fetch a byte
XC!       store a byte
XCALL     jump to a subroutine
```

This is a “tokenized” Forth. The host sends a single byte code to the 68HC11. The code 01 stands for XC@, the code 02 stands for XC!, the code 03 stands for XCALL, and any other code is ignored. When the host wants to read a byte it first sends the 01 code to indicate an XC@ instruction and then sends two more bytes to tell the 68HC11 which address should be read. Upon receiving the 01 code the 68HC11 executes the XC@ instruction, which collects the two byte address, reads the byte at that address, and sends the value to the host. The XC! instruction works similarly. The host sends the 02 code followed by two bytes for the address and one byte for the value to be stored. For the XCALL instruction the host sends the 03 code followed by two bytes for the address. Using the XC! instruction you can download short subroutines for testing, or long programs if the target has enough RAM, or you can program the onboard EEPROM or EPROM if present. Once the routine is downloaded, the host can send the XCALL instruction to cause the 68HC11 to jump to the newly downloaded routine. If the routine ends with a return instruction (RTS) then control will automatically return to the 3-instruction monitor.

Because all of the 68HC11's status, control, and I/O registers are memory mapped, you can use XC@ & XC! to read and set them as if they were regular memory. On other micros, you might need more than the 3 instructions if you need to set registers or I/O ports that aren't memory mapped. In that case you might need a 5 or 6-instruction Forth.

Let's Your Forth and My Forth Do Lunch

"Wait another minute," you say, "I'm not going to sit there and type <value> <address> XC! over and over in order to download a program!" No, of course not. You also run Forth on the host and let the host's Forth do all that work for you. You type high-level commands and the host's Forth breaks all the work down into the 3 simple commands the target Forth understands. This is distributed embedded systems development. Some work is done on the host and some on the target. It allows you the full power of Forth and the full power of an interpreter without requiring extensive resources on the target.

By the power of an interpreter I mean you can examine and set registers and memory interactively from the keyboard. This allows you to exercise the hardware directly. This is so much faster and more convenient than writing your whole program, burning an EPROM, and wondering why it doesn't work. This 3-instruction Forth approach gives you the power of a development system for free if you are using the 68HC11 and for cheap if you are using any other micro.

Sample Session

If you can type 003F XC@ . and understand that you've just displayed the value of the 68HC11's byte at address $003F then you know just about everything you need to know.

Here's a sample session.

Let's pretend I've wired the 68HC11 so that port A pin 01 controls an LED. I can't remember whether a 0 or a 1 turns it on and I'm not sure whether it works. I look in the data book and find that address zero is the port A data address. Although the LED only uses a single bit of port A I don't have anything else connected so I'll just set the whole byte on or off and see what happens.

```
HEX
0 0 XC!      ( the LED goes on)
FF 0 XC!     ( the LED goes off)
```

Now I want to see something more entertaining so I'll write a little Forth program to make the LED flash. A non-Forther might think of this as defining keyboard macros.

```
DECIMAL
: LED-ON ( -)  0 0 XC! ;
: LED-OFF  ( -) $FF 0 XC!  ;
: DELAY ( -)   500 MS    ;
```

```
: FLASHES ( # -)   FOR  LED-ON DELAY LED-OFF DELAY NEXT ;
25 FLASHES   ( for about 25 seconds of entertainment)
  or
10000 FLASHES ( for nearly 3 hours of boredom)
```

MS stands for miliseconds and kills time, thus 500 MS kills about 1/2 a second. If you don't have that word in your Forth system, you can define it with something like

```
: MS  FOR    2000 FOR  NEXT     NEXT   ;
```

then type 10000 MS and see if it takes about 10 seconds. If not, retype the definition of MS with a larger or smaller number than 2000 until you get it close enough.

Note that all of the above definitions are in the host's dictionary, not the target's. They take up no room at all in the target's memory. Only the 3 instructions (which we invoke with the host instructions XC@, XC!, & XCALL) are present in the target's memory.

The Code

The following code loads under Pygmy Forth running on an PC to create a memory image on the PC of the 3- instruction Forth to be run on the 68HC11. It assumes a 68HC11 assembler has already been loaded on top of Pygmy.

```
( name the 'HC11's serial communication registers)
    $2E CONSTANT SCSR ( addr of 'HC11's SCI Status Register)
    $2F CONSTANT SCDR ( addr of 'HC11's SCI Data Register)

( provide labels for the two subroutines)
    VARIABLE 'GET-BYTE
    VARIABLE 'GET-ADDR

 ( how many bytes long is the 3-instruction Forth?)
    VARIABLE MONITOR-LENGTH

( define two 'HC11 assembly language macros)
    : GET-BYTE, ( -)  'GET-BYTE @ BSR,  ;
    : GET-ADDR, ( -)  'GET-ADDR @ BSR,  ;

( create the 3-instr Forth)
  CODE MONITOR         ( do NOT execute this word on the PC)
     CLRA, CLRB, XGDX,  ( now X points to base of registers)
     CLRA, $28 ,X STA,  ( un wire-or port D)

    ( define 2 subroutines)
     NEVER, IF,  ( branch around the 2 subroutines)

       ( get-byte)  HERE 'GET-BYTE !
       HERE   SCSR  ,X   $20 BRCLR, ( wait for incoming char)
                    SCDR  ,X LDA, ( read serial port)
                             RTS, ( byte is in register A)

       ( get-addr)  HERE 'GET-ADDR !
                         GET-BYTE,
                              TAB, ( 1st byte in Reg B)
                         GET-BYTE, ( 2nd byte in Reg A)
                             XGDY, ( transfer to Reg Y)
```

```
                                RTS, ( addr is in register Y)
     THEN,

    ( start the endless main loop)
     BEGIN,
       GET-BYTE,   ( get command code of either 1, 2, or 3)
       1 #, CMPA, 0=, IF,
         ( instruction XC@ was requested)
                         GET-ADDR,
                         0 ,Y LDA,
        HERE  SCSR  ,X   $80 BRCLR, ( wait for Transmit)
                                    ( Data Register Empty)
                      SCDR  ,X  STA, ( send byte to PC)
       ELSE,
       2 #, CMPA, 0=, IF,
         ( instruction XC! was requested)
                         GET-ADDR,
                         GET-BYTE,
                         0 ,Y STA,
       ELSE,
       3 #, CMPA, 0=, IF,
         ( instruction XCALL was requested)
                         GET-ADDR,
                         0 ,Y JSR,
       THEN,  THEN,  THEN,
     NEVER, UNTIL,  ( branch back to beginning of main loop)
  END-CODE

( calculate how long the 3-instruction Forth is)
  HERE  ' MONITOR -  MONITOR-LENGTH !
```

In order to load the above you must have an ’HC11 assembler running on your PC. You can download such an assembler from the Forth roundtable on GEnie – do a search in the files section for 68HC11. However, the whole point of the above code is to create the 66 byte image of the 3-instruction Forth. Here is a hex dump of those 66 bytes:

```
0000   4F 5F 8F 4F A7 28 20  F  1F 2E 20 FC A6 2F 39 8D
0010   F7 16 8D F4 18 8F 39 8D  EF 81  1 26  D 8D F0 18
0020   A6  0 1F 2E 80 FC A7 2F  20 16 81  2 26  9 8D DF
0030   8D D6 18 A7  0 20  9 81   3 26  5 8D D2 18 AD  0
0040   20 D5
```

It would be a simple matter to comma those 66 bytes directly into your host Forth, and thus not require the ’HC11 assembler. Although, surely you’d want the assembler for other work you’d do on the ’HC11, wouldn’t you? Something like

```
HEX
CREATE MONITOR
  4F C, 5F C, 8F C, 4F C, A7 C,  ...  20 C, D5 C,
```

would serve to build the memory image. Or, you could toss all the numbers on the stack, copy them to the return stack (to retrieve the first bytes first), and do all the comma- ing in a loop, as in

```
DECIMAL
VARIABLE MONITOR-LENGTH    66 MONITOR-LENGTH !
: COMMA-THEM-IN ( ... -)
  66 FOR POP SWAP PUSH PUSH NEXT  (   ) ( rs: ... )
           ( now the bytes are on the return stack)
  66 FOR POP POP C, PUSH NEXT  (   ) ( rs:   )
```

```
                    ( now the bytes are in memory)  ;

HEX
4F 5F 8F 4F A7 28 20  F  1F 2E 20 FC A6 2F 39 8D
F7 16 8D F4 18 8F 39 8D  EF 81  1 26  D 8D F0 18
A6  0 1F 2E 80 FC A7 2F  20 16 81  2 26  9 8D DF
8D D6 18 A7  0 20  9 81   3 26  5 8D D2 18 AD  0
20 D5  ( put the 66 bytes on the data stack)

CREATE MONITOR
    COMMA-THEM-IN
```

Downloading

Either way you do it, the next step is to download that image to the ’HC11. The following words are defined on the host to allow the 3-instruction Forth to be downloaded to the ’HC11.

```
.OUT  ( a # -) copies a string to the 'HC11.
?EMIT ( c -)   emits a character if it is displayable,
               else prints its ASCII value.
.OUTE ( a # -) copies the string to the 'HC11
               and displays the 'HC11's echo.
?HEX  ( -)     emits hex value of any waiting
               serial characters.
DL    ( -)     downloads the 3-instruction Forth
               to the 'HC11.

: .OUT ( a # -)  FOR DUP C@ SER-OUT 1+ NEXT DROP ;

: ?EMIT ( c -) DUP $20 $7F BETWEEN IF EMIT ELSE . THEN ;

: .OUTE ( a # -)
  FOR DUP C@ SER-OUT  SER-IN ?EMIT  1+ NEXT DROP ;
: ?HEX ( -) BASE @ HEX
     BEGIN SER-IN? WHILE SER-IN 3 U.R REPEAT BASE ! ;

: DL ( -)  ( download the 3-instruction Forth)
  $FF SER-OUT  ( tell 'HC11 to pay attention)
  ['] MONITOR ( a)  MONITOR-LENGTH @  ( a #)
  .OUT 1000 FOR NEXT ?HEX     ;
```

Using the Three Primitives

Once the 3-instruction Forth has been downloaded to the ’HC11, those primitives can be used from the host. Here are some examples.

```
FLIP  ( hhll - llhh) converts Intel byte order to
                     Motorola order, and vice-versa.
ADDR-OUT ( a -) sends a 16-bit address to the 'HC11 as
                two bytes.
XC@   ( a - c) fetches byte from the 'HC11's address a.
XC!   ( c a -) stores byte to the 'HC11's address a.
XCALL ( a -)   makes 'HC11 jump to a subroutine at
               address a.
XDUMP ( a - a+16) dumps 16 bytes starting at 'HC11's
                  address a.
```

```
XDU   ( a # - a') shorthand to repeat XDUMP for as
                  many lines as you wish.

: FLIP ( hhll - llhh) DUP $100 * SWAP $100 U/ OR ;

: ADDR-OUT ( a -)  DUP SER-OUT   FLIP SER-OUT   ;

: XC@ ( a - c) 1 SER-OUT   ADDR-OUT  SER-IN  ;

: XC! ( c a -) 2 SER-OUT   ADDR-OUT  SER-OUT  ;

: XCALL ( a -) 3 SER-OUT  ADDR-OUT  ;

: XDUMP ( a - a')
  HEX CR DUP 4 U.R 2 SPACES ( a)
  DUP ( a a) 2 FOR
                8 FOR DUP XC@ 3 U.R 1+ NEXT
                SPACE
              NEXT DROP  2 SPACES
  ( a) 2 FOR
           8 FOR DUP XC@ DUP $20 $7F WITHIN NOT
                 IF DROP $2E THEN EMIT 1+
             NEXT SPACE
        NEXT ;

: XDU ( a # - a') FOR ?SCROLL XDUMP NEXT  DROP  ;
```

Bibliography

- Harold M. Martin, “Developing a Tethered Forth Model,” SIGForth Newsletter, Vol. 2, No. 3.
- Brad Rodriguez, “A Z8 Talker and Host,” The Computer Journal, issue 51, July/August, 1991.

Frank Sergeant:

http://pygmy.utoh.org

Contact information: frank@pygmy.utoh.org