TAPS-8 CODE

The TAPS-8 program is contained in an EPROM on the controller card. This code may be changed to accommodate changes in data collection or to correct errors by a more-or-less simple process, described in these notes.

Most of the TAPS-8 code is written in FORTH. A text file containing the latest program is provided on this CD. FORTH is somewhat unique in two respects.

First, the code, which consists of ASCII text characters, is compiled as it is loaded. The process is akin to a threaded interpreter and eliminates the need for separate compilers and linkers; the code can be loaded and run immediately. I am going to assume that you know, or can learn, FORTH. There are books available; I learned FORTH from a book and by writing and testing code.

Second, the program is composed of "words" that are defined in terms of previously defined words. Each word comprises a sort of subroutine that can be called, or executed, by subsequent words. The called word can itself consist of calls to other, previously-defined words. The CPU chip itself contains the basic FORTH dictionary or set of primitive commands. The only constraint on word definition is that all words called have been previously defined. Note that this hierarchy suggests that a FORTH program be read from the bottom up.

As an example, this is the MAIN word in the TAPS-8 operating program:

```
: main
    init                    ( setup parms, SIO, etc.
    ram-map                 ( check NVRAM for data
    read-bat disp-bat       ( display battery voltage
    startup                 ( wait for 2 minutes for inputs
    runrun                  ( ops code
;
```

The colon is FORTH for 'begin word definition'. The semicolon is FORTH for 'end word definition'. "main" is the name of the word. In this case, all of the succeeding words are previously-defined words. For example, "init" is the code that performs CPU and controller card initialization, setting digital port values and directions, setting up the serial interface, etc. An open paranthesis denotes comments that are ignored during program load. The operating code for TAPS is commented to describe the functions of each word and, sometimes, of each line in the code.

Some words in the code are written in machine code. For example, the code that causes the CPU to enter stop mode and cease operations (used when the batteries have become depleted) is

```
hex
code-sub stop               ( enter CPU stop mode, clocks off
  07 c,                     ( tpa
  847f ,                    ( and #$7f
```

```
  06 c,                         ( tap
  cf c,                         ( stop
  39 c,                         ( rts
end-code
decimal
```

In this case, "code-sub" is FORTH for "begin a machine code segment" and "end-code" is FORTH for "end a machine code segment". The name of the word is "stop". The FORTH command "," stores a 16-bit word into memory while "c," stores an 8-bit word. The command "hex" means to interpret numbers as hexadecimal for conversion to binary. There is no assembler in this version of FORTH so the assembly language, shown here in the comments, has been assembled externally and the machine code equivalents stored in memory directly.

Note that this code segment involves both direct commands and interpreter instructions. "code-sub" tells the interpreter what is to come and how to handle it but the rest of the instructions consist of commands that are executed as they are read in. For instance, "07 c," causes the hex value $07 to be converted to a binary word and the low byte of that word stored in memory at the location of the code pointer.

Recall from the TAPS-8 Manual that the basic memory map of the 68HC11 is 64 Kb in size. The lower 32 Kb contains on-board registers and the RAM in U06. The upper 32 Kb contains the operating program in EPROM (U07) and the Forth code in ROM on the cpu chip. A complete memory map is shown below. All addresses are in hexadecimal.

FROM	TO	CONTAINS
0000	0100	FORTH variables & stack
0101	01FF	Text Input Buffer
0200	103B	misc FORTH variables
103D	7EFF	RAM
7FF8	7FFF	RTC registers (in RAM)
8000	AFFF	PROGRAM (EPROM) space
B000	B3FF	Registers & Ports
B400	B5FF	PROGRAM (EPROM) space
B600	B7FF	EEPROM
B800	BFFF	PROGRAM (EPROM) space
C000	C0FF	External memory/latches
C100	DFFF	PROGRAM (EPROM) space
E000	FFFF	FORTH ROM

The current operating code for TAPS-8 is contained in the file *T8V201.4TH* located in the folder *CODE LISTINGS* on the CD.

PROGRAM LOGIC

A text version of the program flow is shown on the next page.

After initialization (see the word *init*), the program will wait up to two minutes for some user input (*startup*). Normally, this time would be used to set the operating parameters for the upcoming deployment (see the TAPS-8 Installation document on this CD; see *command*). If user input is received, the input is processed (*doit*) and then the two minute timer is reset.

When this timer expires, the program enters a low-power sleep state (see *runrun*). Current draw in this state is about 200 μA (if nothing is connected to the host serial port). A timer on the external real-time-clock (RTC) causes interrupts at 1 minute intervals. The CPU briefly wakes and checks the time on this clock to determine if it is time to take data. If not, the CPU goes back to sleep.

When it is time to take data, the CPU powers up the acoustic system and starts taking data. The value of the noisefile interval counter is decremented and compared to zero; if this counter has reached zero, an INHIBIT flag is set and the counter is reset to the starting value. If the INHIBIT line is set to the transmitter, the next data set will consist of receiver noise files only. The acoustic system will function just as in a real data set but no transmissions will occur.

Data are taken by the word *get-data*. The first data set is the small-volume data set (*get-smallv-data*) taken at all eight frequencies. Pings are generated on each channel in sequence and the echo data digitized and stored in working memory (RAM). The CPU sends frequency codes from a stored table to the direct digital synthesizer (DDS) to set the transmit frequency (times 4; *set-freq*) and the local oscillator (LO) frequency. The digital gain stage in the receiver is set from values stored in the *gains* array (*set-if-gain*). A flag is set to select the shorter (336 μs) pulse length (*set-pulse*). A TVG voltage is produced by reading a value from a stored TVG table and sending this value to a digital-to-analog converter (DAC; *set-tvg*). The actual ping is generated by a machine-code subroutine (*tr*).

Detected envelopes from the receiver are digitized by a 12-bit ADC on the controller card. The raw data are moved from the input buffer to working buffers in RAM (*moveSVdata*) until all data are collected.

The same process occurs for the large-volume data except that the pulse length is changed to 762 mS. See *get-largev-data* and the routines called by it.

When data collection is done, the acoustic system is turned off and echo processing begins. The raw data are processed into result accumulators. For the small-volume data, there is one accumulator per frequency for echo intensities

COLD START

Initialize parms from EEPROM
Check clock is running -- start it if it isn't
    Compare external RTC to ram-clock and reset if needed
Wait two minutes for inputs
    Service inputs if found; reset two minute timer
Go to sleep (and notify outside world)

NORMAL OPS

Timer pops and time = time for next data?
Read RTC in low-mem RAM and store data
Update external RTC
Power up the boards
Set the first frequency
Small-volume Sv loop:
    Loop on NPINGS1
        Loop on 8 FREQS
            Set freq
            Set post-IF gain
            Point at proper TVG table start & set TVG
            Call TR
            Call MOVE-SV-DATA to move part of echo to RAM
        END
    END
Large-volume Sv loop:
    Loop on NPINGS2
        Loop on 4 FREQS
            Set freq
            Set post-IF gain
            Point at proper TVG table start & set TVG
            Call TR
            Call MOVE-LV-DATA
        END
    END
Power off to boards
Process the small-volume Sv data
    ΣI and ΣI^2
Move to NVRAM
Process the large-volume Sv data
    ΣA, ΣI, ΣI^2 in range bins
Move to NVRAM
Go to sleep

Program Logic

(squared-amplitudes) summed over 5 samples and the 16 pings. This is accomplished by *process-smallv-data* and the routines called by it.

There are three accumulator sets for the large-volume data -- for echo amplitudes, intensities, and squared-intensities. The raw data are stored in working RAM like the small-volume data until all data have been collected. Only four frequencies are used for this data set, consistent with the larger scatterers targeted by this data set. The data are accumulated into 2 meter range bins as well as by frequency, thus the amplitude accumulator for frequency 1 consists of seven 2-byte values, the intensity accumulator consists of seven 4-byte values, and the squared-intensity accumulator consists of seven 8-byte values.

Following data processing, the accumulators are moved en masse to the non-volatile data RAM. Each data set fits in a 512 byte data block; each NVRAM can hold up to 4096 data blocks. The data format is commented in the code as well as in the Matlab processing programs previously supplied.

Data are moved to NVRAM by the word *move-block*. This routine takes care of storing a byte in the NVRAM and incrementing the address in a 512 count loop. The memory is broken down into 4096 of these blocks and a block counter is kept in RAM (*nvcntr1* - *nvcntr4*). The proper counter to check, and the number of the NVRAM currently being used, is held in *which-nvram*. After the data block is written, a zero is written to the starting byte of the next block (the reason will be seen below).

As described in the TAPS-8 manual, access to the NVRAM data memory chips (up to 4 may be installed) is though a memory-mapped letter-box arrangement. The address in the NVRAM (21 bits or 3 bytes) must be set one byte at a time by writing to memory addresses $C000 through C002 (high to low bytes). A byte is written to (or read from) NVRAM1 by accessing $C003. NVRAM2 is accessed by a read or write to $C004, etc. The process for reading or writing a random byte is thus to write the address, one byte at a time, to $C000-2. Then the byte is read or written to the appropriate address $C003-6.

In the code, routines have been written to set the memory pointers (*set-mem*), increment the memory pointer by one (*inc-ram*), and clear the memory pointers (zero the address; *clear-mem*).

After a deployment, the data in NVRAM can be dumped to the host computer. This is done by the word *dump-ram*. This word checks the block counter for each NVRAM (*nvcntr1-4*) and, if >0, outputs that many blocks of data in ASCII-hex format. Each NVRAM is dumped separately with warnings to open and close capture files as needed.

Clearing out the data memory is done by setting the counters to zero, setting the address to zero, and writing a zero to the first byte of the first NVRAM.

During initialization, the subroutine *ram-map* is called. This routine checks each NVRAM for data. If the first byte is zero, the NVRAM is considered empty and all higher-numbered NVRAMs are considered empty. If the first byte is non-zero, the appropriate block counter is incremented and the address bumped by 512. This continues until a zero byte is encountered or the counter reaches 4096, at which point the next NVRAM is tested. Thus, clearing the first byte of the first NVRAM clears all of memory so far as the code is concerned.

Of course, if you want for some reason to recover the data in NVRAM, so long as no new data has been written (putting a zero byte in some other data set), all one has to do is write a non-zero byte in the first data byte of the first NVRAM. Then run *ram-map* again and the pointers and counters will be reset.

There once was an UN-ERASE command to accomplish these steps but apparently it was removed at some point. The method to do this should be obvious from this description, however.

TEST MODES

The code contains a number of test routines to allow hardware testing. One accesses the test mode by typing CTRL-X. This brings up the test menu from which one can select a particular test to conduct. Tests available include:

RETURN
TAKE A DATA SET
DUMP RAM DATA SPACE
TEST NVRAM / DESTRUCTIVE
ADC TEST
TURN BOARDS ON
SELECT CHANNEL
TRANSMIT TEST
RECEIVE TEST
TURN BOARDS OFF

One types the number of the desired test and it is executed.

One must be thoughtful in using these tests (they were developed for ME to use in hardware test and are not particulary user-friendly). For example, to transmit on a selected channel (one hopes the transducers are immersed when one does this), you need to do a couple of things. TURN BOARDS ON will apply power to all the cards. SELECT CHANNEL will let you setup the DDS with the proper frequencies. TRANSMIT TEST will start TAPS pinging. A screen message tells you to press any key to stop the test. At this point, you may want to TURN BOARDS OFF or SELECT CHANNEL if you want to test another channel.

RECEIVE TEST was designed to let one measure the receiver gain under some standard conditions. For example, the TVG gain is set to an effective range of 1 meter. You are asked for a receiver IF GAIN code (0 is a gain of 1X, 1 is a gain of 2X, 2 is a gain of 4X, 3 is a gain of 8X, and 4 is a gain of 16X). You must supply a source of acoustic input to the transducer (or electrical input to the receiver itself) at the proper frequency and level. You must also manually open the T/R switch by installing a jumper on the receiver card. Then the output must be observed with an oscilloscope at the appropriate test point. Again, when finished you should press any key, remove the hardware jumper, and, probably, TURN BOARDS OFF or SELECT CHANNEL.

Any time you exit test mode, if you have turned the boards on, TURN BOARDS OFF.

LOADING A PROGRAM

The TAPS-8 Operating Program was developed using the actual controller card as a test bed. The EPROM is replaced with a 32Kb RAM chip and the program loaded into the RAM and tested. When a working program is developed, the code image from the RAM is extracted and an EPROM containing this code is burned.

Remove the EPROM and insert a 32 Kb RAM chip. Note the two jumpers above and below the EPROM chip. Each contains a 2-pin shorting plug that is shorting the center pin to one end pin. Move each of these shorting plugs to short the other end pin to the center (check the schematic, *TAPS8 CPU*, to see how these shorting plugs should be arranged for EPROM and RAM usage). This will enable the RAM chip for reading and writing.

It is generally prudent to disconnect the switched power to the external cards while loading or testing a program, at least until the program is believed to be working properly.

Connect a host computer to the serial port (9600, N-8-1) with a terminal program running. Connect power to the TAPS and install the shorting plug (or, if you are dealing with the controller card alone on the bench, apply 24VDC to the power pins on the power jack and clip a jumper between the shorting plug pins). You should see a line of text like

>MaxForth 3.5 ….

on the screen. Pressing <CR> should return a line like

>OK

This proves the 2-way connection between the host and the TAPS controller.

Downloading a program requires paced line transfers -- something that modern terminal programs no longer offer. We typically use BitCom, an old DOS program, to download program files. On newer Windows systems, a program available from New Micros Inc. called NMITerm can be used. This program is somewhat slower but quite reliable. This program is found on the website www.newmicros.com under the DOWNLOADS tab.

Whatever program is used, the goal is to send the ASCII text program file to the TAPS controller card in such a way as to let the CPU interpret each line, store the resultant machine code in memory, and respond with "OK<CR><LF>". In BitCom, we use the <LF> character to tell us that it's ok to send the next line. I presume MNITerm works much the same way. On the screen of either terminal program, you will see the code scroll down with each line ending with "OK". Errors will be signaled with a "?" character along with some semi-cryptic error message. In MAXTerm, downloading will stop and you will be given the opportunity to continue downloading or to stop and correct the error. In BitCom, you will need to stop downloading manually by pressing a function key.

One small note: The FORTH embedded in the CPU expects the input data to be in upper case. I find this very annoying when I write code, so I have set a converter in BitCom to always turn lower case letters into upper case during downloads and keyboard entry. This feature is not available in MAXTerm so, when I'm forced to use it, I use the editing features of a text processor to highlight and then change case on the entire file before I download it. Remember to set the shift lock when you type words or commands directly into the CPU.

When the program has completely loaded, the CPU will do … nothing. It is waiting for more commands to be entered via the serial port. Some commands that you might use include "WORDS". (Remember about the upper case!) This will produce a (rather large) display of every FORTH word that exists in the FORTH dictionary and the address in memory where that word is stored. This listing contains the primitive FORTH vocabulary as well as the words you have defined. I often turn capture on just before I type WORDS and save this listing for later use.

Keep the memory map in mind if you make major changes to the program. At various spots in the TAPS-8 operating program you will find code like this:

```
create skip2              ( $C000-C1FF are memory-mapped I/O
hex
c200 here -
allot
decimal
```

This code was inserted to skip over the section of memory that was re-mapped to form an external I/O port. The proper place to put it was found by trial and error -- when the code attempts to load in a region that is re-mapped, the load will cause an error. One then backs up, inserts some code like this example to skip over this section, and loads again.

TESTING THE PROGRAM

One nice feature of FORTH is that each defined word stands alone. This means that each word can be tested individually as it is written, if desired.

How does one run a word, you might ask. One types it's name and hits <CR>! For example, here is a word from the RTC routines in the TAPS-8 operating code:

```
: get-time2
    read ctrl-reg c!         ( halt updates to clock registers
    year-reg c@ year-mask and
     bcd2int year2 c!
    month-reg c@ month-mask and
     bcd2int month2 c!
    day-reg c@ day-mask and
     bcd2int day2 c!
    hour-reg c@ hour-mask and
     bcd2int hour2 c!
    min-reg c@ min-mask and
     bcd2int minute2 c!
    sec-reg c@ sec-mask and
     bcd2int second2 c!
    0 ctrl-reg c!
;
```

This code stops the RTC briefly by storing a code byte (READ) in a control register (CTRL-REG) and then reading out year, month, day, hour, minute, and second data from the appropriate registers in the RTC. These bytes are stored in variable locations named YEAR2, MONTH2, etc. Then the RTC is restarted by storing a zero in the control register. If you type

GET-TIME2

and <CR>, this word will be executed (it might be wise to execute INIT first to set up ports and such). In this case, the RTC in the NVRAM chip will be read and the results stored in RAM somewhere. If you want to see what the data look like, you could type

YEAR2 C@ .

and <CR>. This fetches (C@) the byte value from the variable location YEAR2 and prints it (. is the FORTH print command) to the screen. After you are done, this line might look like

```
YEAR2 C@ . 6 OK
```

where the year is 6 (2006). Similarly, you can inspect the contents of the month, day, hour, minute, and second variables by fetching and printing them to the screen.

Other code may require inspection by watching level changes on a digital line (such as transmit pulses or digital gain settings) or measuring something like the voltage level from the DAC or the frequency output of the DDS. It is assumed that some study of the code and the controller schematic will be sufficient to gain familiarity with the functions of both the code and the hardware.

BURNING AN EPROM

The ultimate goal of any program change is to produce a working code image that can be burned into an EPROM. A program that will automatically run when power is applied. This involves two steps: copying the code image and installing a special byte sequence at a special location.

The operating code for TAPS-8 includes the following statement near the beginning of the program:

```
8004 dp !
```

This statement causes the dictionary pointer (DP) to be set to the hex value, $8004. The EPROM space begins at $8000 so this command forces code to start loading 4 bytes beyond the beginning of this space. The program code ends with the statements

```
hex

a55a 8000 !                    ( set auto-run flag in EPROM
' main cfa 8002 !

decimal
```

The first part of this code places the byte sequence $A55A at address $8000. The next part places the address (code field address or cfa) of the word MAIN at $8002. The FORTH code in the CPU looks at special addresses for special code words -- when it finds $A55A at $8000, it reads the next word as an address and resumes execution at that address. Thus, on power-up, our code will begin executing.

It is usual to comment out the last code snippet when testing new code so that execution is solely at the control of the programmer. When the program is

tested and ready to move to EPROM, a final load is done with this code active. This will produce a code image that is 'ready to run'. Note that the last lines of the operating code output an address (HERE). You will need to note this address as it is the end of the code image in RAM.

Note from the memory map that the lower memory ($0000-7FFF) is used to hold variables and data. This means we could use some of it to hold a program to download the code image if we wished. A program, HEXDUMP2.4TH, has been provided to do this semi-automatically.

Without disturbing the program just loaded into the RAM chip in high memory, load the program HEXDUMP2.4TH. This program will load into low memory at $1000. Then type

8000 xxxx HEX_DUMP

where xxxx is the value of HERE you noted above. DO NOT HIT <CR> YET!

Now enable a capture file to save the program code image to disk. This output will be an ASCII text file of data in Motorola S-Format. When the capture file is enabled, press <CR> and lines of text will begin to scroll down the screen. When the last line is output, close the capture file.

The file will contain lines that look like the following:

```
:20802000FFFEFEAC8004FE22FD4E8004FA7DFEB6FE800200FE800201FE800202FE800203F0
:20804000FE800204FE800206FE800208FE80020AFE80020CFE80020EFE80020FFE800211F4
:20806000FE800213FE800215FE800217FE800219FE80021BFE80021FFE800221FE800225F4
:20808000FE800229FE80022BFE80022DFE80022FFE800231FE800233FE800235FE800237F3
:2080A000FE800239FE80023BFE80023DFE80023DFE80023FFE800241FE800245FE800249F2
:2080C000FE80024BFE80024DFE80024FFE800251FE800257FE800259FE80025BFE80025DF1
:2080E000FE80025FFE800261FE800262FE800263FE800264FE800265FE800266FE800267F1
:20810000FE800268FE80026AFE80026CFE800270FE800272FE800278FE80027AFE800280F1
:20812000FE800284FE800286FE800288FE80028AFE80028CFE80028EFE800290FE800292F0
```

The format of each line is a colon, the number of bytes ($20 or 32), the load address ($8020 for the first line), the data bytes, and a 4-byte checksum. Now comes a tricky part. Note that the first line of this sample code came from address $8020. We actually want to load it in address $0020 for our EPROM since it's addresses start at zero (mapping it to $8000 is done by the CPU and glue logic, not the EPROM itself!). Somehow, we have to change all of the :208x addresses to :200x addresses. Hurrah for search and replace!

In the primitive text editor we usually use, there is a command to replace all instances of :208 with :200 automatically. WORD and similar word processors can do this as well. Of course, then you need to replace all the :209's with :201, the :20A's with :202, etc. Until you come to the end of this hex code listing. There you will find two different lines that might look like:

```
:1BC10000FEAC94A08128FE22FC24FD05000A9D7DBF54FD170004BF39FEB604F4
:00000001FF
```

The last line is an end-of-record mark and should be left alone. The next-to-last line is the one that we need to inspect and (carefully) change. Note that the load address for this line is $C100. We need to change the C to 4 (remember, 8->0, 9->1, A->2, B->3, C->4, …).

Finally, edit the file to remove any stray blank lines and whatever trash accumulated at the end of the file. Save this file as FILENAME.HEX (you pick the FILENAME, of course). This file is now readable by most EPROM burner programs and will properly align the data beginning at the start of the EPROM.

Burn the EPROM (32Kb) and install in place of the RAM chip used to load the program and test it. Remember to swap the jumpers back to their original locations. Apply power and you should see the program start operation as programmed.

Nothing can possibly go wrong at this