

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

FEATURE ARTICLE

Jamie Pollock

Building An Embedded Timing System

Although development systems are robust and have a great diversity of hardware and software, a few applications and some wire wrap can go a long way. Jamie used an MC68HC11 series microcontroller to design a timing system that will fit into even the tightest of budgets.



Single chip solutions using the MC68HC11 series microcontrollers can be cost effective for many applications. For the beginning designer or hobbyist, most development systems are beyond practical consideration due to their high cost.

Although development systems are robust with a great diversity in hardware and software, a few application notes and some wire wrap can go a long way. The system here can be built for under \$20, excluding the LCD.

Besides cost, time-to-market is also important. If you can't see progress in your efforts, it becomes difficult to maintain good motivation. Projects like LCD-displayed temperature, a lawn-sprinkler controller, or a simple stopwatch are inexpensive projects with great learning potentials.

In the petroleum industry, the viscosity of a product is important. The viscosity baths I use hold six tubes. Two baths are needed, one at

100°C and another at 40°C, to characterize the fluids.

If I try to test six products then I need 12 timers. This can become confusing and more than once I've had to restart the test because of carelessness.

This project will create five independent stopwatches on a 40-character LCD display. The controls for each timer will be directly below the LCD screen. A start/stop button and a clear button for each timer will be sufficient for measuring viscosity time.

The display will show seconds and milliseconds because those increments are necessary for the calculation of viscosity. Four characters are allotted for the seconds and two for the milliseconds. The timers will rollover at 9999.99 or 2.77 hours. A typical viscosity test takes about 300 s.

The minimum configuration for the MC68HC11 consists of three parts—the microcontroller, a crystal oscillator and a low voltage reset IC. First, the controller must be chosen. For the simple stopwatch project, I chose the MC68HC11F1CFN3 primarily because it has enough I/O ports to easily get the job done.

Second, a 12-MHz crystal oscillator is needed to give an internal clock of 3 MHz. (The last digit in the controller part number is its maximum speed). The '6811 divides the external oscillator by four to generate the internal clock.

Third, a MC34064 (low-voltage reset) is needed to properly initialize the controller. This is specific to the '6811 line, other microcontrollers don't require this.

Finally, pull-up resistors are needed on the XIRQ, IRQ, and any input ports. The extras include an OPTREX DMC 40218 40 × 2 character LCD and a homemade keypad. The LCD was

taken from a dead typewriter and the buttons for the keypad were taken from an old printer. Recycling old parts is extremely important for hobbyists.

Because this project is a single-chip solution, the controller will be used in a special mode called bootstrap mode. This mode allows the controller to execute a 256-byte program in ROM at startup, which Motorola created.

The Motorola code initializes the serial communications interface allowing an RS-232 transceiver like the MC145407 to connect the controller to a PC port.

Bootstrap mode provides a means to program the RAM. This code receives a byte from the RS-232 port and places it into RAM in sequential order, starting at location \$0000. The code uses a timer to exit the loop. Once the timeout is activated, the controller jumps to the beginning of RAM executing the downloaded program. Bootstrap code listings are available on the Internet for most of the '6811 line. Studying the listing is essential to understanding the bootstrap operation.

PROBLEM SOLVING

A couple of obstacles are created with a simple program like this, but if the RAM can be mine, then so can the world! First, the bootstrap fills the RAM from the address \$0000 up. This seems like a good idea until one realizes that the vector table for bootstrap mode points to the block \$00C4 to \$00FD, right in the beginning of the 1-KB on-chip RAM (\$0000 to \$03E8).

Second, the EEPROM is only 512 bytes. Because I plan to store my program in EEPROM and have the controller execute my code at power-up, I need to have a block of 512 bytes of RAM available in one chunk. I also need to use interrupts so a patch will be created.

The first attempt to resolve the interrupt address conflict was to use a free program from Motorola called PCBUG11. This is a great tool for working with the '6811. If the controller doesn't have external RAM, then the software uses up all available RAM.

Listing 1—This small piece of code enables the use of the onchip RAM and the bootstrap vectors. The code was adapted directly from the Motorola bootstrap code for the 'F1 available online.

```

BEGIN EQU $0000 ; 'F1 RAM starts at $0000
REG EQU $1000 ; in bootstrap and reg are $1000

SCSR1 EQU $2E
SCDRL EQU $2F
RAM EQU $0100
; I send my second set of code into external RAM

DELAY EQU 854
TOC1 EQU $16
TOC2 EQU $18
HPRIO EQU $3C
EEPRM EQU $0D80
INIT EQU $3D

ORG BEGIN
LDX #REG

LDD DELAY
STD TOC1,X ; store 1200 bps delay here for timeout
LDY #RAM ; not actually using a output compare!

STY TOC2,X
BRCLR SCSR1,X #$80 * ; send FF ready signal
STAA SCDRL,X

WAIT
BRCLR SCSR1,X #$20 * ; wait for character
LDAB SCDRL,X
CMPB #$FF ; compare it to FF, I'm ready
BNE WAIT

LDY TOC1,X ; just get the delay value
WAIT2
BRSET SCSR1,X #$20 NEWONE ; wait for a character on SCI
DEY ; count down for a timeout
BNE WAIT2 ; timeout after no more code
JMP RAM ; Done! Jump to my program

NEWONE
LDY TOC2,X
LDAB SCDRL,X ; get incoming byte from receiver
STAB 0,Y ; put it into RAM
BRCLR SCSR1,X #$80 * ; wait for transmit ready
LDAB 0,Y
STAB SCDRL,X ; send it to terminal
INY ; get next address
STY TOC2,X
BRA WAIT1 ; go wait for another

```

My solution is to write a small bootloader that imitates the original bootstrap code but places the program at \$0100 then times out and jumps to \$0100 to execute my code (see Listing 1).

The 6811F1 has 1 KB of RAM. If I start my code at \$0100, there are 744 bytes available in one block. This gives me 512 bytes with 256 bytes for the stack. There is unused RAM below \$00C4 that I will use for variables. Now I'm ready to download and execute my assembled 'S19 file from RAM. A simple BASIC program will

communicate with the controller. The terminal program must do several different tasks. First, it must establish communications and send the first piece of code—the bootloader.

The controller accepts the code in binary sequential format. The 'S19 file must first be translated to remove the header, address, and the checksum. I have modified the Motorola application note AN 1260 for making this change to the bootloader and to the project 'S19 files.

Bootload.bas is the final PC terminal program. It sends the bootloader

Listing 2—Protecting the EEPROM by not needlessly erasing the bytes was important in development. I ended up writing to the EEPROM more times than expected.

```

BEGIN EQU $0000 ; 68HC11F1
REG EQU $1000 ; uses 'EEPROM SAVER' code
SCSR1 EQU $2E ; which decides if the EEPROM BYTE
SCDRL EQU $2F ; needs to be erased before programming.
PORTF EQU $05 ; Since A 1 can be turned into A 0
DELAY EQU 854 ; without erasing this saves EEPROM
TOC1 EQU $16 ; life by not needlessly erasing bytes.
TOC2 EQU $18 ; An asterisk is sent each time
TOC3 EQU $1A ; an erase cycle is skipped!
HPRIO EQU $3C
EEPRM EQU $FE00
INIT EQU $3D
RAMEND EQU $03FF
EEBYTE EQU $1C
PGBYTE EQU $1D

ORG BEGIN
LDAA #$00
STAA $1035
LDX #REG
LDS #RAMEND
LDD DELAY
        STD TOC1,X ; store 1200-bps delay here for timeout
        LDY #EEPRM ; not actually using a output compare!
STY TOC2,X
BRCLR SCSR1,X $80 * ; send FF ready signal
STAA SCDRL,X
WAIT
BRCLR SCSR1,X $20 * ; wait for character
LDAB SCDRL,X
        CMPB #$FF ; compare it to FF, I'm ready !
        BNE WAIT

        LDY TOC1,X ; JUST GET THE DELAY VALUE
WAIT2
BRSET SCSR1,X $20 NEWONE ; wait for a character on SCI
        DEY ; count down for a timeout
        BNE WAIT2 ; timeout after no more code
        JMP EEPRM ; Done! Jump to my program

NEWONE
LDY TOC2,X
CLRA
LDAB #$2A
        STD TOC3,X ; flag for skipping erase
        LDAA SCDRL,X ; get incoming byte from receiver
STAA PGBYTE,X
        LDAB 0,Y ; get value in EE
        STAB EEBYTE,X ; compare value in EEPROM
        CMPB PGBYTE,X ; to programming value
        BEQ EQUAL ; jump around bit check if equal
BRA CHECK
EQUAL
CLRA
LDAB #$24
BRA SEND ; send a special character to the termi-
nal
ERASE
LDAB #$16 ; set erase - byte - EELAT
STAB $103B
LDAB #$FF
        STAB 0,Y ; store $FF
        LDAB #$17 ; set erase- byte - EELAT - EPGM
STAB $103B
BSR DLY
CLR $103B ; turn off EEPROM prog
LDD #$0000
STD TOC3,X

PGMEE
LDAA PGBYTE,X
CMPA EEBYTE,X
BEQ RDMEM ; skip programming if they are the same!
LDAB #$02
STAB $103B ; set EELAT

```

(continued)

code then prompts the user for the name of the 'S19 file to be sent. *Bootload.bas* processes the 'S19 files and places the raw code into an array. The array is then dumped to the controller, echoing the response to the PC screen for verification. When the code finishes sending the *Bootload.bas* program, it becomes a terminal emulator for debugging.

EEPROM PROGRAMMING

Next, the problem of programming the EEPROM exists. Because the bootloader program was written to relocate our program in RAM, a few modifications enable us to relocate our program into EEPROM. PCBUG11 could easily be used to perform this function. I felt that understanding the function of the EEPROM made it necessary to write the code myself.

Listing 2 is a robust version of the bootloader with some special EEPROM modifications. This code helps extend the EEPROM life by choosing which bytes need to be erased or programmed. The EEPROM in the '6811 has a finite programming lifetime and this code utilizes Motorola's recommendations for extending the EEPROM life to its maximum.

Basically, the code decides first if the Source and the Target byte are the same. If they are, then it skips the erasing and programming sections for the byte. It's amazing how many times this actually occurs during development.

If the Target and Source are not the same, the code decides if the Target byte needs to be erased first (set to \$FF). An EEPROM location in which the only changes are turning 1s into 0s doesn't need the byte erased first. If the byte doesn't need to be erased, the code jumps to the programming section.

Finally, if the Source and Target byte are sufficiently different, the erase code is executed followed by the programming code. Special registers are modified during erasure and programming. A good explanation of these functions and some example code are given in the '6811 reference manual.

This method is good for developing the program. But, for the final product, the EEPROM will be entirely erased then programmed.

SOME ASSEMBLY...

Up to this point we should be able to run our code in RAM using interrupts, and program the EEPROM for testing the final product. Writing code for the 512-byte EEPROM requires the use of assembly language. I use the Motorola free assembler (*AS11.EXE* available on the Internet).

Using assembly language can be frustrating at first, but having complete control of the hardware with compact code is wonderful. I began this project knowing very little assembly language.

At the beginning of the code, the variables need to be cleared. One way to do this is to use a statement like CLR {variable} that takes up a considerable amount of code to clear 39 bytes. Instead, I use a loop relying on the indexed addressing associated with the X register. The code clears the 16-bit D register then begins a loop that stores this value according to where the X register is pointing. Next, the X register is incremented twice and its value is checked against my ending value. This loop can clear as much space as needed with seven statements. The hardware is set up next. In order to use interrupts, there must be a Vector table somewhere telling the controller what to do when an interrupt occurs. Bootstrap mode has the Vector table set up, so I only have to know where each vector points. The Bootstrap code from Motorola lists the Vector table.

The Vector table lists the Timer Output Compare 1 as \$00DF. Placing the opcode \$7E tells the controller to jump to the location given by the next two bytes. The other two interrupts are programmed similarly. The next part of the initialization is setting the ports for input or output.

Port C and Port F are both cleared before they are set as outputs. These two ports are needed for the LCD. Port C is used as an 8-bit data bus to the LCD and three Port F pins are used for the control pins. The LCD is

Listing 2—continued

```

STAA 0,Y
LDAB #$03
STAB $103B                ; set EELAT - EPGM
BSR DLY
                        CLR $103B                ; turn off EEPROM prog

                        BRCLR SCSR1,X $80 *    ; wait for transmit ready
LDD TOC3,X
  CMPB #$00                ; check to see if ERASE ran
  BNE SEND                ; send * if erase did not have to run

RDMEM
LDAB 0,Y
SEND
  STAB SCDRL,X            ; send it to terminal
  INY                    ; get next address
STY TOC2,X
BRA WAIT1                ; go wait for another
ERASE1
  BRA ERASE                ; branches only reach 128 bytes either
way
                        ; so ERASE1 is a stepping stone

CHECK
                        ; check to see if erasing is necessary

B7
BRCLR PGBYTE,X $80 B6
BRCLR EEBYTE,X $80 ERASE1
B6
BRCLR PGBYTE,X $40 B5
BRCLR EEBYTE,X $40 ERASE1
B5
BRCLR PGBYTE,X $20 B4
BRCLR EEBYTE,X $20 ERASE1
B4
BRCLR PGBYTE,X $10 B3
BRCLR EEBYTE,X $10 ERASE1
B3
BRCLR PGBYTE,X $08 B2
BRCLR EEBYTE,X $08 ERASE1
B2
BRCLR PGBYTE,X $04 B1
BRCLR EEBYTE,X $04 ERASE1
B1
BRCLR PGBYTE,X $02 B0
BRCLR EEBYTE,X $02 ERASE1
B0
BRCLR PGBYTE,X $01 BB
BRCLR EEBYTE,X $01 ERASE1
BB
BRA PGMEE

DLY
PSHY                    ; EEPROM programming delay
LDY #3000
BK1
DEY
BNE BK1
PULY
RTS

```

initialized by software. Five sets of codes are sent to the LCD. These codes determine the options for the LCD controller, like 8-bit data bus and the font. Finally, the interrupt flags are cleared and the CLI opcode is executed. CLI instruction clears the 'I' bit in the Condition Code register. This enables all masked interrupts.

Masked interrupts can be enabled and disabled through software. Only the interrupts with flags that have

been cleared will operate immediately.

Next, we need to talk about the interrupt service routines (ISR). In order to achieve a stable timebase, the use of interrupts is necessary to ensure that the millisecond counter will be updated regardless of what the microcontroller is doing.

The main interrupt is the Timer Output Compare 1 (TOC1). This interrupt compares a 16-bit register to

the 16-bit microcontroller clock. If these registers match, the TOC1 ISR is called.

Several things happen inside the TOC1 ISR. The most important thing that happens is the clearing of the TOC1 flag and incrementing the TOC1 register. This example adds 30,000 counts to the TOC1 register. Basically, every 30,000 clock counts the TOC1 ISR is called and the timers are incremented. Next, the individual active timers are incremented and the LCD display is updated. The LCD is updated inside this ISR because I have plenty of time before the next cycle. This arrangement ensures that the display accurately shows the correct time value. If the LCD update was elsewhere, the counter would be able to increment during a write to the LCD.

The next ISR is for the keypad. The keypad uses a special pin called the Pulse Accumulator (PACC). One feature of the PACC is acknowledging pin logic changes on Port A pin 7. This function could have also been done with one of the input capture pins.

The keypad is made by placing a pull-up resistor on each input pin. Port G pins 0–4 are for the Start/Stop function and Port A pins 0–4 are for clearing the timers. The switches are then attached to ground. This gives ten switches for control of five timers.

Diodes are placed between each switch and the PACC pin to prevent the switches from interfering with each other. The PACC also has a pull-up resistor.

The keypad can now be operated on an interrupt basis, which is better than the polling method because a key press might slip through the cracks while one of the ISRs is running.

Finally, we need a time-out counter to debounce the switches. TOC2 can be used to prevent one key press from being registered as multiple key presses.

Enabling the TOC2 ISR inside the PACC ISR does this. The TOC2 ISR is called when the timeout has expired. The PACC interrupt flag is then cleared to enable the keypad again. The TOC2 interrupt is not enabled until another key has been pressed.



Photo 1—Under the LCD is a socketed 68HC11F1, and the other supporting components. The dial is a contrast control which come assembled to the LCD.

The controller is entirely run off of interrupts after the initialization is completed. The main routine consists of WAI (wait for interrupt) and a loop back to the same instruction.

INTERFACING THE LCD

Interfacing the LCD can pose certain challenges. With the Optrex display, the technical manual for the LCD controller is needed. The manual for the Hitachi HD44780 can be found on the Internet. Complete descriptions of all LCD functions are explained in the manual, but a good understanding of the LCD is necessary for reliable operation.

At the beginning of the program, the LCD is initialized. This consists of placing data on the LCD bus and toggling the E pin on the LCD. The code sets up the LCD for an 8-bit data bus, then chooses two lines and a 5 × 7 font. Next, the display is turned on (in software), then auto-increment and cursor are picked. After these control commands are accepted, the busy flag is in operation.

The busy flag enables the LCD to receive data as quickly as possible. The use of delay loops as opposed to busy-flag checking reduces the efficiency of the code. Busy-flag checking allows the LCD to be updated inside the TOC1 ISR.

Listing 3 is the routine that writes characters to the LCD. This code receives the RAM location for the character in register Y and continues to send characters until the RAM location contains a 0. This arrangement has multiple uses. First, a string of text can be sent, or the result of a Hex to Decimal conversion (derived from the Motorola 6811 manual) can

Listing 3—The LCD subroutine requires the Y register to point to the beginning of the message, with a \$00 denoting the end.

```

LCD
LDX #REG
    LDAA 0,Y          ; get character
    CMPA #0          ; looks for A 0 to terminate mes-
sage
    BEQ DONE
    STAA PORTC,X     ; put data on Port C
    BSET PORTF,X $01 ; E high
    BCLR PORTF,X $01 ; E low
    LDAB #$00        ; Port C input
    STAB DDRC,X
    BSET PORTF,X $04 ; set R/W to 1
    BRCLR PORTC,X $80 * ; wait for busy flag to clear

    LDX #10
    WAIT
        DEX          ; wait for LCD ADDR to increment!
        BNE WAIT    ; wait loop ~ 120 cycles

    LDX #REG
    BCLR PORTF,X $04 ; set R/W to 0
    LDAB #$FF
        STAB DDRC,X ; Port C output

        INY          ; point to next character
    BRA LCD
DONE
    BSET PORTF,X $02 ; set R/S to 1
    RTS              ; end display

```

be sent. The routine first sends the data to Port C then toggles the E pin.

Next, Port C is changed to an input port and the busy flag bit is monitored until it clears. Once the busy flag has cleared, the LCD autoincrements its display address. A small delay loop of 120 cycles (by trial and error) allows the LCD to update. This routine can send the seconds, a decimal point, the milliseconds, and a space, five times. The HOME command, which returns the cursor to position 1 without clearing the display, is then sent.

Control commands require more time to complete so the DLY1 delay subroutine is called after a Control command has been sent. This allows the character-write loop to operate at its fastest speed.

Photo 1 shows the completed project. Now all that is left is to build a case to protect the components.

With the use of the Internet and freeware, many obstacles for developing microcontroller solutions are eliminated. Wire wrapping prototype boards with few components can allow experimentation on the smallest of budgets. Realistically, \$20 and a little time can produce stand-alone single chip products for many different applications.

Jamie Pollock is a graduate in the field of chemistry from Wichita State University. Currently, Jamie works for BG Products Inc. doing quantitative microcontroller applications as a hobby. You may reach him at jpollock@ks.freei.net.

SOURCES

MC68HC11

Motorola
(512) 328-2268
Fax: (512) 891-4465
www.mot-sps.com

DMC40218

Optrex
(313) 471-6220
Fax: (313) 471-4767

Circuit Cellar, the Magazine for Computer Applications.
Reprinted by permission. For subscription information,
call (860) 875-2199, subscribe@circuitcellar.com or
www.circuitcellar.com/subscribe.htm.