Programming the 87C750: A Beginner's Notes

Michael A. Covington
Artificial Intelligence Center
The University of Georgia
Athens, Georgia 30602–7415
mc@uga.edu

Release 0.1, September 7, 1994

Contents

1 Introduction

These are brief notes about my early experiences programming the Philips 87C750 microcontroller, starting in the summer of 1994. I'm writing them up in the hope that other experimenters will find them useful. This is not a complete description of the 87C750; I presume the reader has access to a full reference manual.

2 Why the 87C750?

The 87C750 is designed to be small and cheap. It fits in a single 24-pin, 0.3-inch-wide DIP with a minimum of support circuitry. At present, the one-time-programmable version costs about $6 in single quantities, but Philips expects to get the price down to $1 (qty. 1000) as production increases.

At present the 87C750 is not yet the cheapest microcontroller (that honor still belongs to the PIC family), nor is it the easiest to use from the hardware standpoint (the serial-programmable 68HC11s probably win that prize). But it is readily available in small quantities (which 68HC11s don't seem to be), takes up less space on the circuit board, and has a machine language much more familiar to PC and Z80 programmers.

3 The CPU

The 87C750 is an 8-bit processor derived from the Intel 8051; the main differences are that it has only 64 bytes of RAM and 1024 bytes of PROM, and one 16-bit timer. Books about the 8051 architecture, such as Ayala's *The 8051 Microcontroller,* are highly applicable to the 87C750.

The 87C750 uses a Harvard architecture. That means program memory is separate from data memory. More conventional (von Neumann) computers have a single memory for both program and data. The IBM PC (8086, etc.) is in between the two, because although program and data memory are physically the same, it is standard practice to divide it into separate code and data segments.

(Don't worry; there *is* an instruction, MOVC, that lets you read data from program memory. That means you can store data in the PROM.)

The CPU clock runs at 1/12 the crystal frequency. Most instructions take 2 CPU cycles; some instructions take only one. Here are some useful timing data:

Crystal (MHz)	CPU cycle (μs)	CPU cycles per second
3.579545	3.352	298,295
4.0	3.0	333,333
12.0	1.0	1,000,000

The first of these is the nominal 3.58-MHz color TV crystal, very cheap and readily available in the United States. It is also close to the lowest frequency at which the 87C750 can run, and hence suitable for long-period timing applications.

An unusual property of the 87C750 is that the registers have addresses, just like memory. In fact, the general-purpose registers R0–R7 *are* the first 8 bytes of data memory (though they can be mapped to other locations). The 64 bytes of data memory comprise addresses 00h-4Fh, and the other registers have higher addresses: the accumulator is E0h, the stack pointer is E0h, and so on.

As a result, there are two ways of referring to the accumulator (A register) in assembly language. Some instructions pertain to the accumulator specifically, and refer to it as A, such as:

```
MOV A,#10       ; store 10 in accumulator
```

This is different from the usual move-a-number-to-an-address instruction. On the other hand, there is no PUSH A instruction. Does that mean you can't push the accumulator value? Not at all; you just have to refer to it by its address, like this:

```
PUSH 0E0H       ; push contents of accumulator
```

Better yet, you can take advantage of the fact that the assembler equates ACC with the address of the accumulator, and do this:

```
PUSH ACC
```

Similarly, in some instructions B is the name of a register, and in others, it is a symbol that has been equated with the address of the B register. This is not as confusing as it sounds.

By default, the stack begins at the 9th byte of RAM (just after register R7), but you can put it elsewhere by initializing SP (Stack Pointer) to a different value. Also, to avoid using the stack, you can swap the contents of any memory address with those of the accumulator by using the XCH instruction, like this:

```
XCH A,10   ; exchange A with contents of memory location 10
```

After working on the data in the accumulator, you can XCH A,10 again to put the result back into location 10 and restore the previous contents of A.

4 Support Circuitry and Interface Details

A working 87C750 circuit needs only the following components and connections:

- +5V at pin 24;
- Ground at pin 12;

- A crystal (3.5 to 12 MHz, 40 MHz for the high-speed version) across pins 10 and 11;
- 20-pF capacitors from pins 10 and 11 to ground;
- A 10-μF capacitor from the reset pin (9) to V+.

All the other pins are input-output ports.

There are three I/O ports, P0, P1, and P3. (P2, from the 8051, got left out, and P0 is only three bits wide.) Each port is a CPU register and has an address; in addition, you can refer to the individual bits as P0.0, P0.1, and so on.

All three ports are bidirectional; that is, you can write all 1's to them, then externally pull some or all of the bits down to 0 and read back the data in its changed condition. At reset, all ports are indeed set to all 1's.

Port 0 is open-drain and can sink 10 mA, enough to drive an LED connected through a 470-ohm resistor to V+. (Use much larger pull-up resistors, such as 10kΩ, if you want to interface with logic circuits rather than LEDs.) Ports 1 and 3 have pull-up resistors built in, and can also sink 10 mA or source about 0.1 mA. The total sink current, however, must not exceed 26 mA per port or 67 mA for the entire chip; don't try to light LEDs on all the ports simultaneously.

5 The DS-750 Kit, And Other Tools

You cannot program the 87C750 with an ordinary EPROM programmer because it requires a special serial control sequence to put it into programming mode. The cheapest 87C750 programmer that I know of is the CEIBO DS-750, sold by Philips for $47.50 during the summer of 1994 and $98 afterward. (The promotional price is obviously an allusion to Motorola's $68.11 deal on the 68HC11 a while back; I'm glad they didn't choose $877.50! But why not make the permanent price $87.50 for its mnemonic value?)

The DS-750 consists of a small board that attaches to the serial port of your PC, and a program that will disassemble, simulate, and trace 87C750/1/2 programs and program them into the microcontrollers. It also supports a limited form of in-circuit emulation which I haven't tried out.

The DS-750 software is called C750D. Version 1.20B came packed with my kit. An update, version 1.20D, is available from the Philips BBS (1-800-451-6644, 1-408-991-2406) and includes the ability to read back the contents of a programmed microcontroller and to check that a microcontroller is blank without programming it.

C750D version 1.20D isn't free of bugs. It requires a *lot* of free memory — over 600,000 bytes, which hardly gives you room for DOS — and it crashes at unpredictable times, often hanging the machine, if its memory desires aren't satisfied. I strongly suspect that the excessive memory requirement is due to a program error. Crashes often seem to occur when I hit "run" without preceding it with "program reset."

The DS-750 kit doesn't include an assembler, but you can get the Metalink ASM51 assembler free from the aforementioned Philips BBS. Be sure to get MODELS2.ZIP, which includes the define file for the 87C750.

Finally, you will need an ultraviolet EPROM eraser. Mine is made out of a 4-watt germicidal lamp bulb and the parts from a small battery-operated fluorescent lamp. It takes 1 hour to reliably erase an 87C750.

6 The Assembly Language

One quirk of the assembly language — the fact that the accumulator has two names — has already been mentioned. Another point to note is that bare numbers are presumed to be *addresses* unless you prefix them with #. Contrast these instructions:

```
MOV A,#10  ; store in A the number 10

MOV A,10   ; store in A the contents of data memory location 10

MOV A,@R1  ; store in A the contents of the data memory location
           ; whose address is in R1
```

Numbers are decimal unless suffixed with H. Hexadecimal numbers must not begin with a letter; just as on the PC, you write 0FFH rather than FFH. The assembler also takes binary numbers, such as 1001B.

Caution: The assembler will not complain if you try to store a 16-bit number in an 8-bit register, like this:

```
MOV A,1234H   ; wrong!
```

As a veteran PC programmer, I have trouble remembering that the 87C750's registers are only 8 bits wide, and I've made this mistake too many times.

The assembly–language program should begin with ORG 0 (to signify that the code should start at the beginning of program memory) and must end with END. ORG 0 is the default, but including it explicitly makes it clear that you are looking at the beginning of the program.

7 How to End the Program

If you're new to microcontrollers (as I was), you'll pause for a moment and wonder — how is a program supposed to end? There is no operating system to exit to.

Basically, you have three choices. Some microcontroller programs go through a main loop, over and over. Some of them idle in a short endless loop, waiting either for an external interrupt, or for a human being to come along and turn off the power. In that case, you can end on the instruction:

```
SJMP $
```

where $ stands for the location of the current instruction; it's equivalent to:

```
SLEEP: SJMP SLEEP
```

which might be more understandable.

Or the microcontroller can power itself down, by setting bit 1 of the power control (PCON) register (which, oddly, is not bit-addressable), like this:

```
ORL  PCON,#2H       ; power off the chip
```

When you do this, the clock stops running and the chip shuts off. Alternatively, you can put the chip in idle mode:

```
ORL  PCON,#1H       ; go into idle mode
```

In this mode, the clock oscillator, on-chip timer, and interrupt section of the chip continue to run, but the rest is turned off, to wake up again when any enabled interrupt occurs or when a hardware reset signal is received. Note that in this situation the reset signal will make the program resume where it left off, rather than resetting the processor.

8 Simple Program: Beep!

At last we're ready to write and run a program. Here it is:

```
$MOD750                 ; code is for 87c750
$TITLE(Beep!)
$DEBUG
; Simplest program for 87c750:
; produce 1-second 1000-Hz beep in speaker, then shut down.

     ORG  0             ; start at memory location 0

     MOV  R0,#100       ; For 2000 inversions (1000 cycles),
L0:  MOV  R1,#20        ; use 20-step loop inside a 100-step loop.

L1:  CPL  P0.0          ; toggle bit 0 of port 0
     MOV  R2,#73        ; innermost loop to kill time
L2:  DJNZ R2,L2         ; count down until R2=0
```

```
     DJNZ R1,L1          ; inner loop until R1=0
     DJNZ R0,L0          ; outer loop until R0=0

     SETB P0.0           ; final state of bit 0 of port 0
     ORL  PCON,#2H       ; power off the chip
     END                 ; tells assembler program is over
```

Here $DEBUG tells the assembler to produce a .DBG file, which can be loaded by C750D just like a .HEX file except that it also contains symbolic variable names (if any) and labels.

This program drives a speaker that is connected to pin 8 (P0.0) through a 330-ohm resistor; the other side of the speaker goes to V+. The beep isn't loud, but it's enough to show that the microcontroller is working.

Prominent here is the DJNZ (decrement and jump if not zero) instruction, which is similar to LOOP in 8086 assembly language except that it can decrement and test any address, not just a specific register. Here we use R0, R1, and R2 as loop counters.

The CPL statement toggles P0.0 back and forth between 1 and 0. Each CPL takes 1 cycle and is followed by a MOV (2 cycles) and then a 73-iteration loop comprising one DJNZ statement (2 cycles per iteration). Thus, P0.0 toggles once every 149 cycles, or once every 499.5 ms given a 3.58-MHz clock.

9 Precise Time Delays

The next program contains two reusable subroutines for producing precise time delays measured in CPU cycles:

```
$MOD750                  ; code is for 87c750
$TITLE(Simple timing routines - M. Covington 1994)
$DEBUG

     ORG   0              ; start at memory location 0

; Demonstration: Toggle p0.0 with a period of 1028 cycles
;   = 162.127 Hz with 4-MHz crystal.

START:
     CPL   P0.0       ; 1 cycle
     MOV   A,#2       ; 1 cycle
     ACALL LONG_WAIT  ; 2*256*2 = 1024 cycles
     SJMP  START      ; 2 cycles

SHORT_WAIT:
     ; Takes exactly 2*A (>= 4) machine cycles,
     ; including the ACALL that calls it.
     PUSH  ACC
     SUBB  A,#5
     NOP
     DJNZ  ACC,$
     POP   ACC
     RET

LONG_WAIT:
     ; Takes exactly 2*256*A machine cycles (A >= 1),
     ; including the ACALL that calls it.
     PUSH  ACC
     DEC   A      ; if A was 1, skip the loop
     JZ    LW2
     ; Delay 2*256*(A-1) cycles
LW1: PUSH  ACC
     NOP
     MOV   A,#252
     ACALL SHORT_WAIT
     POP   ACC
     DJNZ  ACC,LW1
     ; Now delay 2*256 cycles, minus the overhead
LW2: MOV   A,#250
     ACALL SHORT_WAIT
     POP   ACC
     RET

     END
```

In each case A contains the argument of the subroutine, which consumes 2×A cycles (`SHORT_WAIT`) or 2×256×A (`LONG_WAIT`). The main program is an endless loop that simply produces a square wave of precisely known frequency, at P0.0.

10 The Onboard Timer

A totally different way to do timing is to use the on-board timer, which, when running, is incremented once per machine cycle, regardless of what else the CPU is doing. What's more, the on-board timer is a 16-bit register, and can be told to reload itself automatically from another 16-bit register every time it rolls over to zero. In addition, it can trigger an interrupt every time it rolls over and reloads itself.

Here is a program that uses the on-board timer to maintain a real-time clock, as a number stored at location hex 3F, equal to the number of eighths of a second that the program has been running, up to 256/8 = 32 seconds.

The longest interval that can be timed is 65,536 cycles, equal to 0.2197 second with a 3.58-MHz crystal. To time 1/8 second, the timer is automatically reloaded each time it rolls over, so that it will count up from hex 6E5A instead of from zero. The reload value comes from a 16-bit register that is treated as two 8-bit registers called RTL and RTH ("reload timer low" and "reload timer high"). Thus the timer counts:

...6E5A...6E5B...6E5C... ... ...FFFF...(1)0000...6E5A...

and there are a total of 37,287 states of the timer, so that a full cycle takes 0.1250002 second.

When the timer rolls over, it raises an interrupt which causes execution to jump to address hex 0B. Thus, the program begins with a jump instruction to jump past the interrupt service routine. Notice that three bits have to be set to get everything going: ET0 to enable timer interrupts, EA to enable the interrupt section of the CPU as a whole, and RT to start the timer running.

For demonstration purposes this program copies the clock value to port 3 every time it is updated. Thus, the eight bits of port 3 produce square waves with periods of 1/4, 1/2, 1, 2, 4, 8, 16, and 32 seconds.

```
$MOD750
$TITLE(REALTIM1 - Simple real-time-clock demonstration)
$DEBUG

; This program maintains a real-time clock, in eighths of a second,
; at address 3FH (the highest byte in data memory).
; It also copies the clock byte to P3 every time it is updated.
; Thus, P3.2 is a 1-Hz square wave.

; For 3.58-MHz crystal.

CLOCK   EQU     3FH     ; name for location of clock data

        ORG     0       ; code segment
        SJMP    START

;;; Timer interrupt service routine
        ORG     0BH
ISR:    INC     CLOCK
        MOV     P3,CLOCK
        RETI
;;; End of interrupt service routine

START:  MOV     RTH,#6EH        ; counting from 6E5AH to 10000H
```

```
        MOV     RTL,#5AH        ; to time 1/8 sec at 3.58 MHz
        SETB    ET0             ; enable timer interrupt
        SETB    EA              ; enable interrupts in general
        SETB    TR              ; run the timer

;;; Now do anything you want.  We run an idle loop:
        SJMP    $

        END
```

11 More To Come...

This is an unfinished document. More information will be added in later releases.