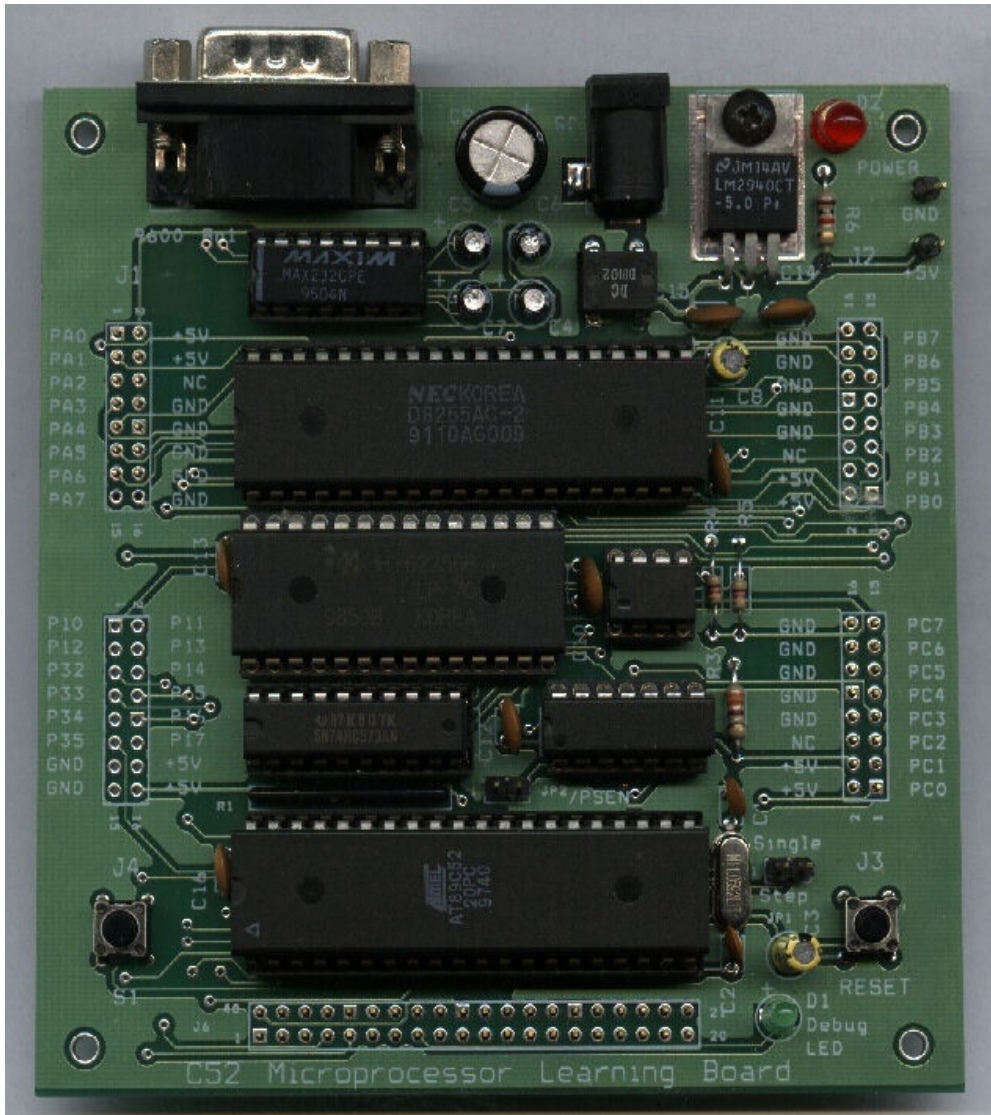


C52EVB V2.0

“The 89C52 Microprocessor Learning Board”

REFERENCE MANUAL



Wichit Sirichote, kswichit@kmitl.ac.th

Microprocessor Lab, Department of Applied Physics, Faculty of Science,
King Mongkut's Institute of Technology Ladkrabang, Bangkok 10520, THAILAND

10 April 2002

Copyright © 2002 by Wichit Sirichote

PREFACE

My intention to write this document is to provide reference materials for those who interested in building a microprocessor learning board at home and to practice myself writing a technical manual using a foreign language. The C52EVB has been designed for using as a microprocessor learning tool for my student since 1999. The new version, C52EVB V2.0 has a nice feature for both learning and using it as a dedicated controller. The feature is the use of an external eeprom for program saving and later booting. I modified the extra package of PAULMON2 monitor program by adding eeprom service functions. The PAULMON2 monitor program has an excellent idea of using program header. The header enables user to add his/her own command, startup code to the available space of the 2nd page code space. So when combine the boot loader and command for saving binary image into eeprom, the new C52EVB can then be used as a learning tool as well as a dedicated controller easily.

The materials I provided in my homepage are in public domain. Everybody can get and build the board for education purpose. The source code of my modified on EXTRA package can be downloaded and later modified without the need of permission.

I like sharing knowledge with others. The projects appeared in my page are my hobby, just for fun. I am very happy when updating new project. It is beyond my imagination that how fast and how far of my document reaches everybody. I thank to the Internet for connecting people around the globe, giving friendship to them. We are all having, rather short period for our day left living, so spend your time for whatever you like to do. For me is to build a microprocessor project and post it on the Internet.

Wichit Sirichote

CONTENTS

INTRODUCTION

What's C52EVB?	4
----------------	---

CIRCUIT DESCRIPTION

Block Diagram	6
CPU	6
Memory	7
Serial I/O	8
Parallel I/O	9
Serial EEPROM	9
I/O Connectors	11
Jumper	12

SOFTWARE

PAULMON2	13
EXTRA package	13
Program Header	14
Boot Loader	26

PROGRAMMING

Monitor Commands	29
Machine Code	31
ASM51	34
IAR C compiler	37
ZAP EEPROM	42

APPENDIX

I.	BOM
II.	Schematic
III.	PCB Layout

INTRODUCTION

What's C52EVB?

A microprocessor learning board having built-in monitor program is the best tool for studying the architecture of a given microprocessor. While running a monitor program, we can practice enter a machine code in hexadecimal number and let the CPU executes it. We can examine memory, CPU registers and I/O pin. The 8051 family is quite popular and cheap microprocessor. There are lot of information and applications on the Internet. The 89C52 chip is software compatible to the 8051, Intel 8-bit Microcomputer.

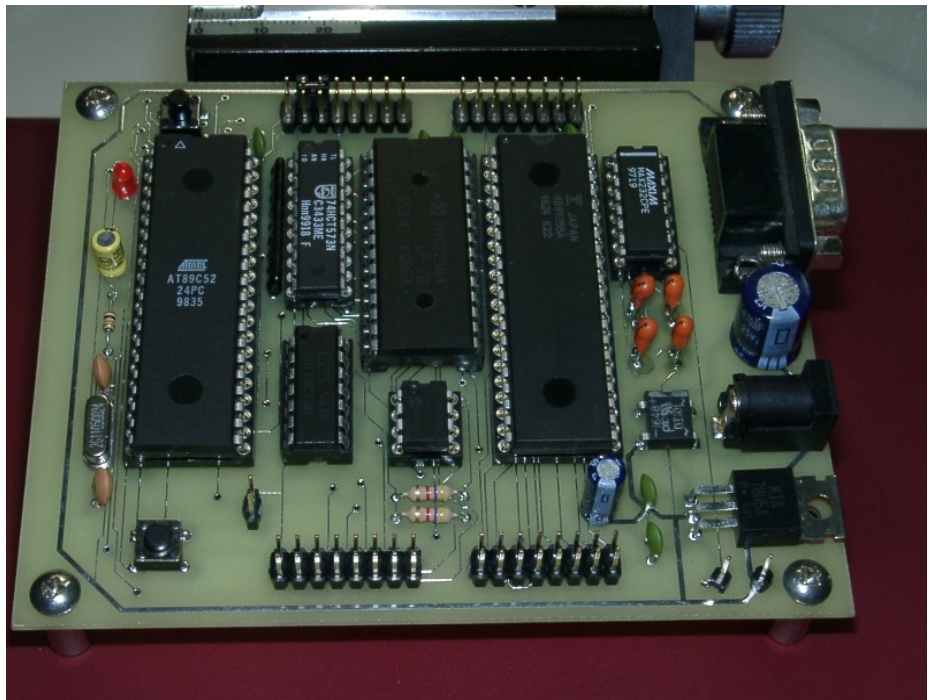


Figure 1: Prototype of C52EVB V2.0

The C52EVB has been designed for study Assembly and C language programming at the Department of Applied Physics, KMITL since 1999. One day I brought the I2C serial eeprom, 24LC256, 32kB! The chipmaker says we can get erase/write cycle with minimum at 1,000,000 cycles! And the retention time is about 40Yrs! So I got the idea to use this serial eeprom for program saving and booting when power up. The EXTRA memory space of PAULMON2 monitor program enables me to put the eeprom ZAP functions. I designed the new version of C52EVB to provide such eeprom ZAP functions. Under program development, student will have 32KB for program testing. After the program works fine, they can save the binary image into eeprom with maximum of 32KB! Same amount of SRAM. I put

the eeprom boot loader as a startup code in PAULMON2. So when power up the board, after serial port has initialized, the boot loader will check a jumper. If it set, it will retrieve binary image from eeprom, write to SRAM and jump to start address running the program.

The prototype of C52EVB V2.0 depicted in Figure 1 features,

- CPU: 89C52, CMOS 8-bit microcomputer @11.0592MHz,
- MEMORY: on-chip 8kB FLASH code memory and 256 bytes RAM, external 32KB SRAM for both code and data spaces,
- EEPROM: 24LC256, 32KB serial eeprom, 1,000,000 erase/write cycles, retention time 40Yrs.
- I/O pins: P1,P3 of 89C52, 24-bit I/O pins using 8255 PPI,
- On-chip Timers: three 16-bit timers, timer0, 1, and 2.
- Debug LED: single dot LED connected to P1.7,
- RS232 Level Converter: MAX232,
- Serial Interface: Asynchronous 9600 8n1,
- Monitor Program: 8KB PAULMON2 including external eeprom boot loader,
- I/O Connectors: four 16-pin I/O connectors and one 40-pin CPU connector,
- Programming Language: hexadecimal machine code, ASM51 Cross-assembler, High level language.

With the external eeprom boot loader enabled by jumper, student can use the board for study programming in the LAB. The C52EVB can also easily be used as a dedicated controller for final year senior project as well. By making the peripheral boards and put it on a CPU board.

The source code, hex file, schematic and gerber files for PCB layout can be download at www.kmitl.ac.th/~kswichit/C52evbv2/C52evbv2.html.

P1 and some I/O pins from P3 are provided for experiment through the 16-pin connector.

EA (External Access) is tied to logic '1', thus when power up the board, the CPU will fetch first byte from internal code memory, i.e. runs the PAULMON2 monitor program.

Memory

The C52EVB has two kinds of memory, on-chip and external memory. The 89C52 chip has 8kB FLASH code memory and 256-byte data memory. The 8kB code memory holds the PAULMON2 monitor program. The 256 bytes data memory is for data storage and stack usage. As mentioned in CPU section, EA is tied to logic '1', so when reset the CPU, it will start fetch the code from internal code memory at address 0x0000.

The external 32kB SRAM however is configured for both code and data memory. While the program monitor was running, the CPU would see the 32kB SRAM as a data memory. So the machine code can then be written using `MOVX @dptr,a`, say.

This SRAM is also mapped into program memory. Let see at the OE signal of the 32kB RAM. The `/RD` and `/PSEN` signal are logical AND together providing output enable signal. Thus when CPU jump from the monitor program to run the code in SRAM, the CPU will generate instruction fetch cycle producing `/PSEN` signal to enable the external RAM. So the machine code that stored in SRAM can then be executed.

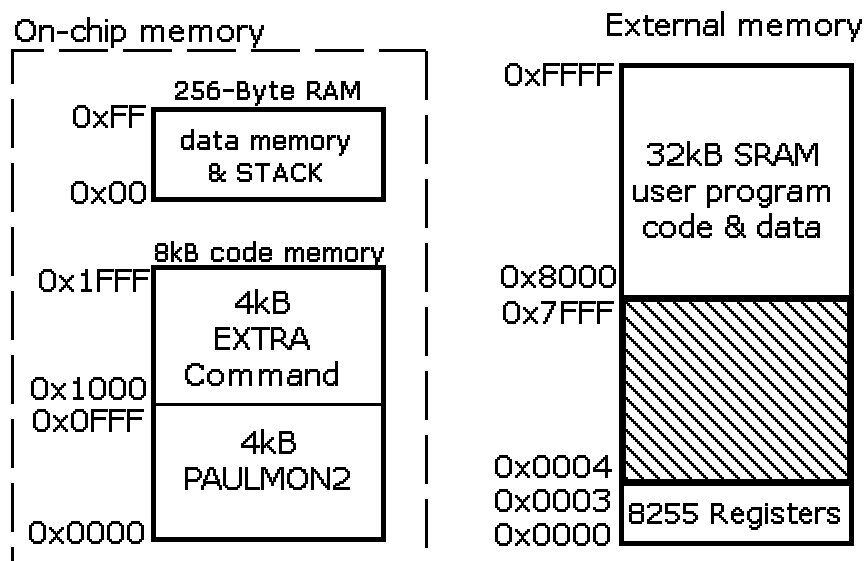


Figure 3: Memory MAP

The address line needed for 32kB RAM is A0-A14. A15 is tied to the inverter to provide memory decoder signal that ties to the CS pin of the RAM. So this RAM will be enabled only when the CPU fetches address in the range of 0x8000 to 0xFFFF. Figure 3 shows a memory map of C52EVB V2.0.

Serial I/O

Since the C52EVB board has no display and keyboard. We must use a PC running terminal emulation program and use PC's keyboard and display for communicating with the monitor program. The 89C52 chip has internal UART. After powerup the board, the monitor program initializes serial port to 9600 baud, 8 data bit, 1 stop bit. The MAX232 converts TTL logic for TxD and RxD pins to RS232C logic. For MARK or logic '1', MAX232 will convert to approx. -9V and for SPACE or logic '0', it will be approx. +9V. The digital baseband of 89C52 serial port is NRZ. Most of the terminal emulation says VT100 or TVI320 can be used to connect the C52EVB board easily. The cable that used to connect PC's COM port and C52EVB board is a null cable.

PC's COM Port	DB9		DB9	C52EVB
TxD	3	-----	2	RxD
RxD	2	-----	3	TxD
RTS	7	-----	8	CTS
CTS	8	-----	7	RTS
Signal Ground	5	-----	5	Signal Ground
DSR	6	-----	4	DTR
DTR	4	-----	6	DSR

Figure 4: Null cable wiring between COM port and C52EVB

PC runs terminal emulation program, VT100.

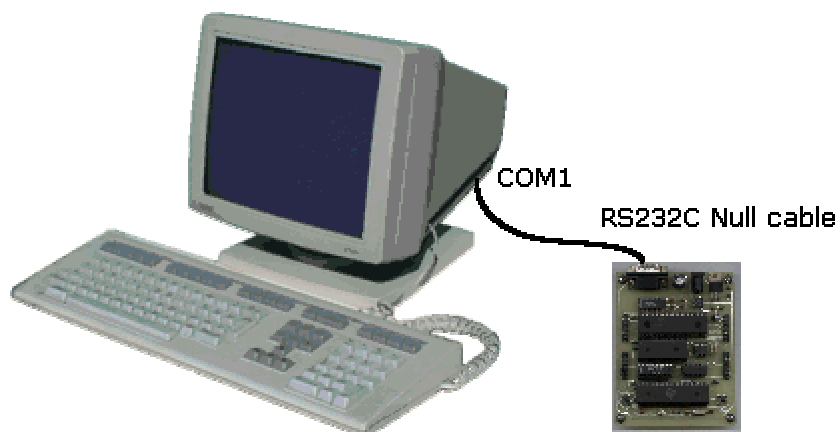


Figure 5: Using terminal to communicate with C52EVB

Parallel I/O

As can be seen from Figure 3, the space available for 8255 interfacing is the shade area of external data memory, from 0x0000 to 0x7FFF. So I use A15 to provide chip select signal directly. This means as long as A15 is low, the 8255 chip will be selected. To make it easier to remember, I choose address ranging from 0x0000-0x0003 to access the 8255's internal registers.

The method is commonly known as memory mapped I/O. A0 and A1 is used to select PORT A, B, C and control register. As shown in Figure 2, we have three 8-bit parallel ports for experiment, PORT A, B and C. The signal /RD and /WR are still borrowed from memory control bus. So to access these registers, we must use MOVX with dptr as a pointer. See example below;

8255's Register	Address
PORT A	0x0000
PORT B	0x0001
PORT C	0x0002
Control Register	0x0003

Figure 6: The memory mapped I/O address of 8255 chip

EXP. To set all ports to be output port,

```
MOV  A,#80H    ; the control word 80H is for all output port
MOV  DPTR,#0003 ; address of 8255's control register
MOVX @DPTR,A  ; write the content of A to control register
```

With C language, we may use a pointer to access memory.

```
// Declare pointer variable, p
```

```
xdata char xdata *p;
```

```
p = 0x0000;    // set p to 0x0000
```

```
*(p+3) = 0x80; // write control word 0x80 to control register
```

```
*p = 0x00;    // write 0x00 to PORT A
```

```
*(p+1) = 0xAA; // write 0xAA to PORTB
```

Serial EEPROM

A serial eeprom is a kind of nonvolatile memory. When data was written, it will be retained without the need of power supply. The C52EVB has a 256kBit eeprom, 24LC256. The interfacing bus, I2C needs two I/O pins, bi-directional data line SDA and shift clock signal SCL. As shown in Figure 2, SDA is connected to P3.4 and SCL to P1.0. Since there is no I2C hardware on the 89C52 chip, so

we implement I2C signaling by software method. Later we will see the EXTRA package resided in the 2nd 4kB-monitor program holds the I2C driver routines. The I2C bus uses only two signal lines, however a number of I2C chips can tie to the same bus. Each device has its own SLAVE address. So when the MASTER wants to talk to which SALVE, the MASTER must issue the SLAVE address at the beginning of I2C protocol.

The 24LC256 has SLAVE address shown below;

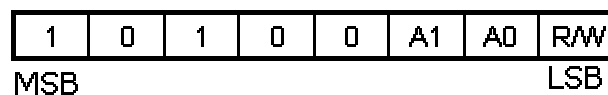


Figure 7: 24LC256 SLAVE address

A0 and A1 are hardware device address inputs. For our C52EVB we have only one chip, so A0 and A1 are tied to GND. The MASTER will issue this SLAVE address for 24LC256 chip every time they want to write with 0xA0 and to read with 0xA1.

A sample protocol to write a byte to the eeprom is shown in Figure 8.

Write data 0x02 to eeprom address 0x7F00

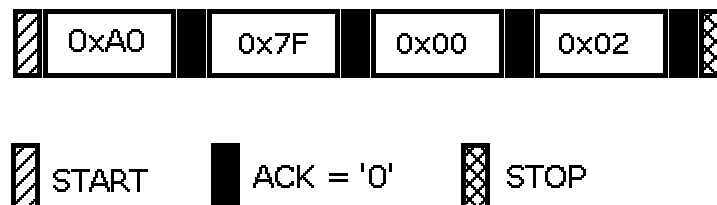


Figure 8: Byte write sequence

Suppose the MASTER want to write a byte to eeprom. It first issues START condition followed with SLAVE address with write operation, i.e. 0xA0. Every time the SLAVE got a byte sent by the MASTER, it will respond with ACK bit or logic '0' to tell the MASTER that it recognizes commanding. The word address will then be issued, say 0x7F and 0x00. And then followed with the byte to be written, 0x02. The STOP condition issued by the MASTER will end the write operation. At this time the eeprom enters an internally-timed write cycle. All inputs are disabled and the eeprom will not respond until the write cycle is complete. The chip that we used will take about 10ms with +5V power supply.

For read operation, it needs more bytes.

Read data from eeprom address 0x2000

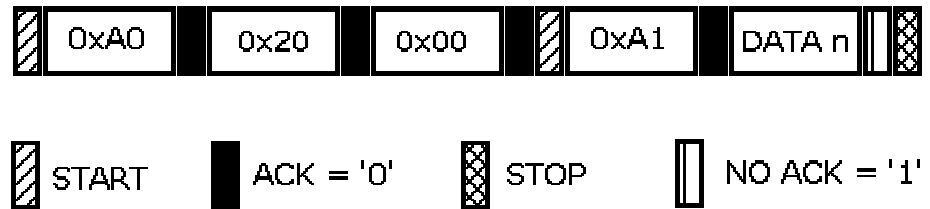


Figure 9: Random read sequence

The MASTER must send eeprom address to be read at the beginning of the package. Before read a byte, the MASTER must issue START bit and read command, 0xA1, then the SALVE will send the byte on the bus. Notice that for read operation, the SLAVE responds with NO ACK or logic '1' at the end of package.

Above is the sample operation of I2C protocol for the serial eeprom. There are special techniques for both operations that shorten the access time as well, for example PAGE write, CURRENT address read, SEQUENTIAL read, say. See data sheet for such special techniques.

I/O connectors

There are four connectors on the C52EVB. Each connector has 16 pins. The layout shown in Figure 7 is the same as seen on the board. J1 is for 8255 PORT A, J2 for PORT B and J3 for PORT C. We may notice the 1st pin on the board is a square pad, not a circle one. J4 provides P1 and some bit from P3 of the 89C52 chip. Note that P1.0 and P3.4 is used for I2C bus.

P1.7 drives small dot LED with sink current. See schematic for detail. Each connector also provides +5V power supply for the peripheral board.

J6 is prepared for accessing 40-pin CPU signal. All 40 pins are tied pin-to-pin directly to the CPU. The example of using this connector is for Philips 8051RD2 chip.

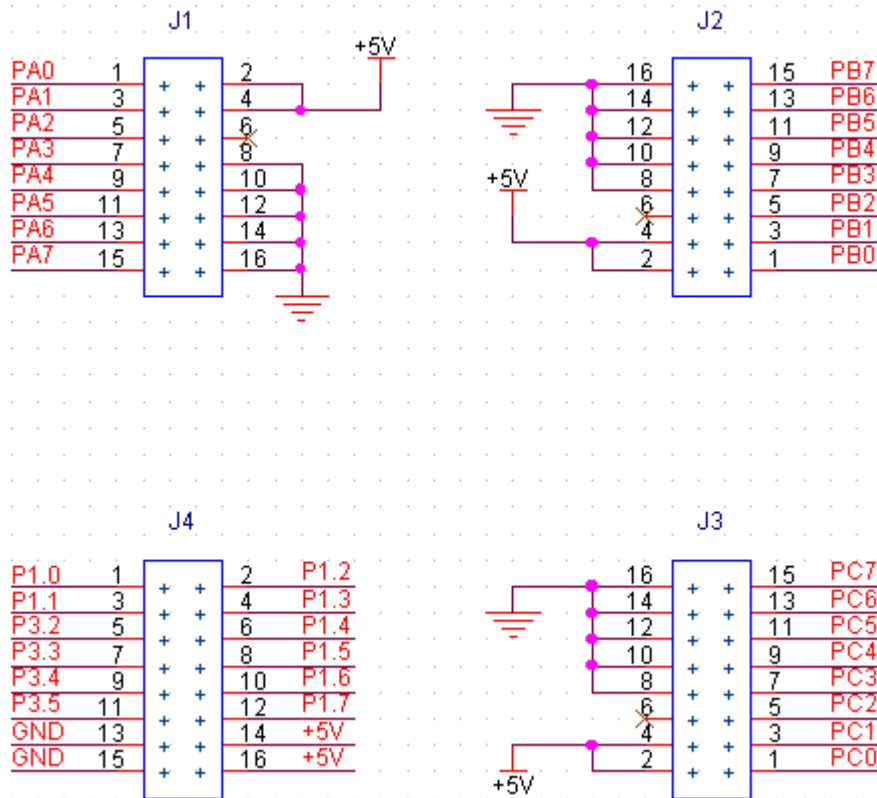


Figure 10: I/O Pin connectors

Jumper

The C52EVB has two jumpers, JP1 and JP2. JP1 is for enable single step running. This jumper ties INT1 or P3.3 to GND when enables single step. In monitor mode we can use this jumper for single step setting. We can move it to J4 and tie P3.5 to GND to enable eeprom boot loader. See example in Figure 11.

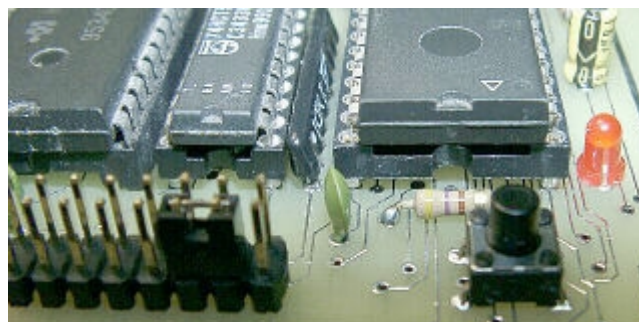


Figure 11: Tie P3.5 to GND to enable boot loader

JP2 is for 8051RD2-loader setting. It ties /PSEN signal to GND.

SOFTWARE

PAULMON2

The PAULMON2 is a free monitor program for 8051 single board computer. It was developed by Paul Stoffregen. The source code and document can be reached via URL below;

<http://www.ece.orst.edu/~paul/8051-goodies/>

The binary image of the PUALMON2 can be put in 4kB code memory. Paul provides a very nice idea of putting the user commands, startup code in the 2nd page code space. The C52EVB uses on chip 8kB code memory of 89C52 CPU for monitor program. So with the 89C52 chip, the C52EVB has the 4kB space for the extra command service routines.

EXTRA package

As mentioned earlier, the extra space is resided in 2nd page of 8kB, or the address from 0x1000 to 0x1FFF. Paul has provided very useful commands in this space already, i.e.

L-List
S-Single-Step

The command L is used to disassemble the machine code into mnemonic. And the command S is used to enable single step running that executes the instruction one by one and to examine the internal registers, memory, says.

For the detail how they work, you may study from Paul's assembly code directly.

Here I will show you how can I put my command for eeprom services. Actually the PAULMON2 has a fancy editor, it covered nearly 4kB, so I have to cut it, otherwise I cannot put my code in the 89C52 chip. Let see my code that services eeprom function.

I will describe the main concept of the eeprom ZAP functions. I will insert my description at the portion that I think you should know.

The service function is located at 0x1A00.

0xA5,0xE5,0xE0,0xA5 ;this line is a signature bytes used to tell the PUALMON2 what to do, here is the user installed command.

"Z" is a single letter command that PUALMON2 will recognize when we press Z key.

```

;=====
;   ZAP eeprom service routines
;
;   Wichit Sirichote Feb 5, 2002
;=====

.org     locat+0xA00           ; i.e. 0x1A00
.db     0xA5,0xE5,0xE0,0xA5   ;signature
.db     254,'Z',0,0           ;id (254=user installed command)
.db     0,0,0,0               ;prompt code vector
.db     0,0,0,0               ;reserved
.db     0,0,0,0               ;reserved
.db     0,0,0,0               ;reserved
.db     0,0,0,0               ;user defined
.db     255,255,255,255       ;length and checksum (255=unused)
.db     "ZAP EEPROM",0

```

Program Header

Above code is a 64-byte constant reserved for program header. The actual address of the executable code begins at 0x1A40. When we press Z key the CPU will jump to here.

```

.org     locat+0x0A40           ;executable code begins here

loop_eeprom:  mov     dptr,#eeprom_menu
              lcall  pstr
              mov   a,#'>'
              lcall  cout
              lcall  cin
              cjne  a,#'1',next2
              acall  save_pgm
              sjmp  loop_eeprom

next2:  cjne  a,#'2',next3
        acall  eeprom_dump
        sjmp  loop_eeprom

```

```

next3: cjne a,#'3',next4
       acall boot_eeprom
       sjmp loop_eeprom

next4: mov  r6,#0x00
       mov  r7,#0x80    ; return to start address of SRAM 0x8000
       ljmp newline_h

eeprom_menu: .db 13,10,"24LC256 EEPROM functions 1 SAVE, 2 DUMP, 3
              LOAD",13,10,0

```

The service function will continue polling the key that enters by user. It will recognize for ASCII code of '1', '2', and '3'. If user press another key, the CPU will get back to PAULMON2 and reload current prompt address with 0x8000 by using r6 and r7.

See the last line for command description; key 1 is SAVE, 2 DUMP and 3 for LOAD.

IC START

I usually put the low-level driver routines before putting the high-level interface routines. Below routines are I2C driver code.

```

;===== eeprom drivers =====
; Generate a START condition & TX char in A

ICstrt: SETB    P3.4                ;SDA = 1
        SETB    P1.0                ;SCL = 1
        NOP
        NOP
        CLR     P3.4                ;SDA = 0 - START
        NOP
        NOP
        CLR     P1.0                ;SCL = 0 - Ready for first bit
        RET

```

This routine generates START condition. While the SCL is settled to logic high, a negative going pulse or transition from logic high to low, of SDA is used to signal the devices connected to I2C bus ready to begin a transaction.

I used two NOP instructions to provide approx. 2µs settled of SCL before transition of SDA. It works fine with 24LC256 eeprom.

```

; Send 8 bit character in A

ICsend: MOV      R7,#8          ;8 bits data
send1:  RLC      A             ; Get next bit to send
        MOV      P3.4,C       ;Write bit to data line

        MOV      P1.7,C       ; use P1.7 as a I2C functioning

        SETB     P1.0         ;Toggle CLOCK high
        NOP
        NOP
        NOP
        CLR      P1.0         ;Toggle CLOCK low
        DJNZ     R7,send1     ;Send all bits

        SETB     P3.4         ;Release DATA line
        SETB     P1.0         ;9'th clock
        NOP
        NOP
        NOP
        NOP

        MOV      C,P3.4       ;Get ACK bit
        CLR      P1.0         ;Return clock LOW
        RET

```

ICsend

The I2Csend routine sends 8-bit data in accumulator by using RLC A instruction. It started with MSB first by rotating the accumulator left through carry bit and copy a carry bit to SDA pin. The data is valid when SCL is logic high. I provide 4-NOP for SCL to settle. Then make SCL to low before sending next bit. The loop counter R7 loaded with 8 is for 8 times shifting. When the sender complete 8-bit shifting, it will release the SDA pin to logic high or to change it to be input port in order to get the ACK bit from the receiver. The receiver will pull this pin to logic low for ACKnowledge. See at the end of the routine, this ACK bit will read back into carry bit, so the main code will know whether the receiver get the 8-bit data correctly or not.

Notice that I used P1.7 to copy a data bit to be sent, since the P1.7 is connected to an onboard debug LED, thus we can see the I2C functioning from this LED.

The I2Csend routine is used to send the SLAVE address, address or command to the device.

```

;
; Read 8 bits of data into ACC
;
ICread: MOV      R7,#8

read_bit: SETB   P1.0
          NOP
          NOP
          NOP
          NOP

          MOV     C,P3.4

          MOV     P1.7,C ; also for reading I2C indicator

          RLC     A
          CLR     P1.0
          DJNZ   R7,read_bit
          RET

```

ICread

Similarly for the I2Cread routine, the MASTER will generate the shift clock to the device and shift each bit by pulsing SCL to logic high and copy it into carry flag which in turns rotating through accumulator 8 times. The results of 8-bit data will be in accumulator.

```

;
; Send an ACK to the device
;
ICack: CLR      P3.4 ; Zero DATA
        SETB    P1.0 ; Raise CLOCK
        NOP
        NOP
        NOP
        NOP
        CLR     P1.0 ; Lower CLOCK
        SETB    P3.4 ; Release DATA
        RET

```

```

; Generate STOP condition
;
ICstop: CLR      P3.4          ;Zero DATA
        SETB     P1.0          ;Raise CLOCK
        NOP
        NOP
        NOP
        NOP
        SETB     P3.4          ;Raise DATA
        RET

```

ICstop

This routine generates STOP condition by producing a transition of SDA from low to high while the SCL settled to logic high. Again I used 4-NOP for SCL to settle before rising the SDA.

From above low level driver routine, we can now use them to form a high level interface routine.

Before writing the high level interface routine, we must know the I2C device address. Here is the device address for 24LC256.

I2C Device Address

```

.equ device_addressWR, 0xA0
.equ device_addressRD, 0xA1

```

We see that there are two values, i.e. for WRITE operation the R/W bit in the device address will be low (0xA0) and high for READ operation (0xA1).

```

; Read byte from EEPROM
; Entry: DPTR holds address to be accessed
; Exit: ACC
;
eeread: ACALL    ICstrt      ; Start transfer
        MOV     A,#device_addressWR
        ACALL   ICsend      ; Write
        MOV     A,DPH
        ACALL   ICsend
        MOV     A,DPL
        ACALL   ICsend

```

```

    acall    ICstrt
    mov     a,#device_addressRD
    acall    ICsend

    ACALL   ICread           ; Get data byte
    acall    ICstop
    ret

```

eeread

This subroutine enters with data pointer register, DPTR that holds eeprom address. The byte that read is exit through accumulator. The read operation is straightforward, begins with START, devices address for write command, and physical address of eeprom. DPH holds high order byte and DPL for low order byte. The first portion uses write operation to the eeprom in order to set up the physical address. To read a byte for that address we must send START followed with read command, then read a byte and finally sends STOP.

```

; Write byte to eeprom
; Entry: DPTR address to be written
;       ACC byte to be written
;
eewrite: mov  b,a
         acall ICstrt

         mov  a,#device_addressWR
         acall ICsend

         mov  a,dph
         acall ICsend

         mov  a,dpl
         acall ICsend

         mov  a,b
         acall ICsend

         acall ICstop
         acall complete_poll
         ret

```

eewrite

The data to be written is entered in accumulator. The DPTR holds the address. The routine first saves the accumulator into register B. The transaction begins with sending START, write command, eeprom address and then a byte to be written. And finally STOP.

Before exit the routine, we see that I put the call instruction to routine `complete_poll`. Since the eeprom when enters a write cycle, even the job will be done internally but if the eeprom chip still in writing cycle, we can not make another operation. I used a simple routine to poll the ACK bit whether the write cycle is completed or not.

```
; Write complete polling
; exit when complete

complete_poll:
    ACALL ICstrt
    MOV   A,#device_addressWR
    ACALL ICsend
    JC    complete_poll
    ret
```

complete_poll

The method is done by polling the ACKnowledge bit, or bit 9 after we send a write command byte. The ICsend routine will read bit 9 and save into carry bit. See the JC instruction, the routine will exit only when ACK is LOW or the write cycle is completed. The datasheet said that the self-timed write cycle will be 5ms typically.

```
; hex dump from external eeprom address 0000H-7FFFH
; DPTR: start address to be displayed

eeprom_dump:
    mov     dptr,#read_location
    lcall   pstr
    lcall   ghex16      ; get start address
    lcall   newline
    mov     r2, #16     ;number of lines to print
eedump1:  lcall   newline
    lcall   phex16     ; print location
    mov     a,#' '
    lcall   cout       ; send one blank
    mov     r3, #16    ;r3 counts # of bytes to print
eedump2:  acall   eeread
    inc     dptr
    lcall   phex       ;print each byte in hex
    mov     a,#' '
    lcall   cout
    djnz   r3,eedump2
    djnz   r2,eedump1
    lcall   newline
    ret
```

```
read_location: .db 13,10,"eeprom address [0000-7FFF] > ",0
```

eeprom_dump

The eeprom_dump routine performs a dump of internal content of eeprom the same manner as PUALMON2 does. The DPTR register was loaded with 16-bit start address. R2 is a counter for outer loop counting and R3 is for inner loop counting. As shown in the source code, it will dump 16-line of 16-byte for each line. Here is a sample;

```
24LC256 EEPROM functions 1 SAVE, 2 DUMP, 3 LOAD
>
eeprom address [0000-7FFF] > 0000

0000 00 80 A8 FC 4D 02 20 52 00 00 00 A6 93 01 00 00
0010 02 80 11 00 00 00 00 00 00 00 00 43 8C DC 05 08
0020 32 75 81 08 90 F0 01 E4 F0 43 89 01 D2 AF D2 A9
0030 D2 8C 12 80 28 02 19 54 74 00 90 F0 00 F0 E5 08
0040 12 82 79 7B 0A 12 82 70 12 82 C6 45 F0 70 03 02
0050 80 45 02 80 2E 74 00 F5 08 E5 90 64 80 F5 90 90
0060 F0 00 E0 04 90 F0 00 F0 14 74 41 75 F0 83 C0 E0
0070 C0 F0 90 F0 00 E0 75 F0 00 C0 E0 C0 F0 90 F0 00
0080 E0 75 F0 00 C0 E0 C0 F0 90 F0 00 E0 75 F0 00 C0
0090 E0 C0 F0 74 04 75 F0 00 12 80 94 7F F8 12 82 5A
00A0 02 80 2E 22 23 F4 25 81 F8 12 83 17 12 80 E0 12
00B0 83 17 02 80 AF 78 FB 12 82 4E 86 82 08 86 83 E0
00C0 60 2D 12 80 BE A3 80 F7 78 FB 12 82 4E E6 30 99
00D0 FD C2 99 F5 99 B4 0A 17 74 0D 80 F2 E4 30 98 0C
00E0 30 98 FD C2 98 E5 99 B4 0D 02 74 0A 75 F0 00 22
00F0 12 82 27 86 82 08 86 83 08 E4 FE FF E0 70 06 12

24LC256 EEPROM functions 1 SAVE, 2 DUMP, 3 LOAD
>
```

save_pgm

The save_pgm routine is quite long, so I will break the entire routine into small portion and insert the description for each portion. Briefing, the save_pgm will help save a specified start address and a number of byte from 32kB SRAM to eeprom. Later we can use Load function to load a saved program to running again in SRAM.

```
; save program
; save code from 8000H to eeprom
;     the number of byte to be saved was input by user
;
; test with 9000H.....
;
; eeprom address      content
; 0000H              start address low
```

```

; 0001H          start address high
; 0002H          number of byte low
; 0003H          number of byte high
; 0004H          XX reserved for future
; 0005H          XX
; .....         ..
; 0010H          first byte
; 0011H          ..
; 7FFFH          nn

.equ ptr2,0xFC   ; 16-bit eeprom address pointer
.equ ptr1,0xFE   ; 16-bit SRAM address pointer

```

Let begin with detail of save area header as shown above. I reserved a first 16 bytes for program header. My first version as shown has used a simple format. The physical eeprom address 0x0000 and 0x0001 save the start address of SRAM for low and high byte respectively. The address 0x0002 and 0x0003 save the number of byte for low and high byte. The actual area that saves binary image was started at 0x0010. The space available for binary image saving will then be from 0x0010 to 0x7FFF or 32768-16 bytes.

Above directives .EQU were used to define the address of 89C52 internal RAM 0xFC and 0xFE for temporary usage as a 16-bit pointer.

```

save_pgm:
    mov  dptr,#save
    lcall pstr
    lcall ghex16
    jnc  no_esc1
    ret

no_esc1:
    mov  a,dpl
    mov  r2,dph

    push dph
    push dpl

    mov  dptr,#0
    acall eewrite
    mov  dptr,#1
    mov  a,r2
    acall eewrite    ; save start address to 0000-0001

```

```

pop  dpl
pop  dph

```

The save_pgm starts print the message to screen asking for the start address on SRAM to be saved. Then get a 16-bit address value entering by user and save it to eeprom at 0x0000-0x0001.

```

mov  r0,#ptr1
mov  a,dpl
mov  @r0,a
inc  r0
mov  a,dph
mov  @r0,a

```

Save DPTR's content to pointer1 or ptr1.

```

mov  dptr,#save2
lcall pstr
lcall getnum ; get number of byte to be saved

```

Ask for enter the number of byte using getnum. User can enter the number of byte in decimal, say 9710 bytes.

```

cpl  a
mov  dpl,a
mov  a,b
cpl  a
mov  dph,a

inc  dptr ; got two's complement of number of bytes

```

Convert it into two's complement number.

```

lcall newline

push dph
push dpl

mov  a,dpl
mov  r2,dph
mov  dptr,#2
acall eewrite
mov  dptr,#3

```

```

mov  a,r2
acall eewrite ; save number of byte to 0002-0003

pop  dpl
pop  dph

```

Then save it to eeprom at 0x0002 and 0x0003.

```

mov  r0,#ptr2
mov  a,#0x10
mov  @r0,a ; eeprom start address is 0010H
inc  r0
mov  a,#0
mov  @r0,a

```

Prepare a pointer for saved address at 0x0010.

The two's complement value of the number of byte to be saved still with DPTR.

```

loop_write: ; loop for writing

    push dph ; save number of byte or loop counter
    push dpl

;----- loop body -----

    mov r0,#ptr1
    acall get_dpتر ; get source pointer1,transfer to DPTR

    movx a,@dpتر ; read byte from 32K SRAM
    mov r3,a ; save to r3

    mov r0,#ptr2
    acall get_dpتر ; get destination pointer
    mov a,r3
    acall eewrite ; write to eeprom

    mov r0,#ptr1
    acall inc_ptr ; next source
    mov r0,#ptr2
    acall inc_ptr ; next destination

;-----
    pop dpl
    pop dph

```

```

    mov a,#13
    lcall cout
    lcall pint16u    ; show counter on screen

    inc dptr        ; next counter
    mov a,dph
    orl a,dpl      ; check if both are zero
    jnz loop_write ; repeat if not zeor

    mov dptr,#done2 ; done print string "done"
    lcall pstr

    ret

save:  .db 13,10,"Save Program start address> ",0
save2: .db 13,10,"Number of bytes [0-32768] > ",0
save3: .db 13,10,"Saving...",0
done2: .db 13,"Done..",0

```

After getting two parameters, i.e. start address and number of byte, above portion then is a loop that writes the binary image into eeprom. The loop was counted with incrementing DPTR and was repeated until DPH and DPL are zero. Below routines are utility that helps incrementing pointer and transfer to DPTR.

```

; increment ptr
; entry: r0

inc_ptr:
    mov a,#1
    add a,@r0
    mov @r0,a
    inc r0
    mov a,#0
    addc a,@r0
    mov @r0,a
    ret

; get DPTR

get_dptr:
    mov a,@r0
    mov dpl,a
    inc r0
    mov a,@r0
    mov dph,a
    ret

```

Boot Loader

The saved program or binary image can then be retrieved from eeprom and be written to SRAM using monitor command or hardware setting. With monitor command under eeprom ZAP function, we can invoke by entering number 3 for LOAD eeprom. Using hardware invoking, P3.5 will be detected if the logic is low, it will perform boot loading. The hardware boot loader is put in monitor program as a startup program header (**253=startup**).

```
-----;
;                                     ;
;           EEPROM Boot Loader       ;
;   detect P3.5 if P3.5 == 0 then run EEPROM Booting   ;
;-----;

.org      locat+0x800
.db       0xA5,0xE5,0xE0,0xA5        ;signature
.db       253,255,0,0                ;id (253= startup)
.db       0,0,0,0                    ;prompt code vector
.db       0,0,0,0                    ;reserved
.db       0,0,0,0                    ;reserved
.db       0,0,0,0                    ;reserved
.db       0,0,0,0                    ;user defined
.db       255,255,255,255            ;length and checksum (255=unused)
.db       "EEPROM booting",0

.org      locat+0x0840                ;executable code begins here

        JNB P3.5,boot
        ret

boot:    acall boot_eeprom
```

The boot_eeprom can then be called by either one, P3.5 or monitor command.

Briefing, the boot_eeprom will begin retrieve the start address and the number of byte to prepare for loading. Then read binary image and save into SRAM when finish it will jump to SRAM at the start address running the program.

```
=====
; load program from eeprom, write to RAM and jump to start address
; in SRAM

boot_eeprom:

        mov dptr,#loading
        lcall pstr
```

```

; get start address from eeprom 0000H and 0001H save to ptr1
  mov dptr,#0
  acall eeread
  mov r0,#ptr1
  mov @r0,a
  mov dptr,#1
  acall eeread
  mov r0,#ptr1+1
  mov @r0,a

```

First, print the message "Loading..." on screen. Then read two byte from eeprom address 0x0001 and 0x0002, save them to pointer1.

```

; Initialize address counter for eeprom to ptr2

  mov a,#0x10
  mov r0,#ptr2
  mov @r0,a
  inc r0
  mov a,#0
  mov @r0,a

```

Initialize the address counter ptr2 with 0x0010.

```

; get a number of byte from eeprom 0002H-0003H

  mov dptr,#2
  acall eeread
  mov r2,a          ; save acc
  mov dptr,#3
  acall eeread
  mov dph,a
  mov dpl,r2

```

Retrieve a number of byte to be read from eeprom address 0x0002 and 0x0003. Save it to DPTR.

```

loop_read:
  push dph          ; save counter
  push dpl

;=====loop body=====

  mov r0,#ptr2
  acall get_dptr    ; get eeprom address

```

```

acall eeread

mov  r3,a          ; save to r3

mov  r0,#ptr1
acall get_dptra
mov  a,r3
movx @dptra,a     ; write to SRAM

mov  r0,#ptr1
acall inc_ptr     ; next destination
mov  r0,#ptr2
acall inc_ptr     ; next source

;-----
pop  dpl
pop  dph

inc  dptra
mov  a,dph
orl  a,dpl
jnz  loop_read

```

Again above loop will read the binary image from 0x0010 with a number of byte counted by DPTR. The loop will repeat until DPTR is zero.

```

mov  dptra,#0
acall eeread
mov  r2,a
mov  dptra,#1
acall eeread
mov  dph,a
mov  dpl,r2

push dpl
push dph
ret          ; run application program

```

```
loading: .db 13,10,"Loading...",0
```

The start address is read again to DPTR. The start address of SRAM is push into STACK, then RET will change a Program Counter with the top of stack and execution will transfer to program in SRAM.

PROGRAMMING

Monitor Commands The monitor program provides 11 standard commands and 3 user installed commands. The board needs a PC that runs terminal emulation, say VT100 with 9600 8n1 to be a user interface device.

Below is a list of PUALMON2 standard commands.

Standard Commands

- ?- This help list
- M- List programs
- R- Run program
- D- Download
- U- Upload
- N- New location
- J- Jump to memory location
- H- Hex dump external memory
- I- Hex dump internal memory
- E- Editing external ram
- C- Clear memory

And here is a user installed commands.

User Installed Commands

- L- List
- S- Single-Step
- Z- ZAP EEPROM

Details of using PAULMON2 monitor commands can get from <http://www.ece.orst.edu/~paul/8051-goodies/>

Here I show you how to use some basic command that needed to get started.

When power up, you have to press ENTER key.

```
PAULMON2 Loc:8000 >
```

We get the command prompt. The location shows the current address pointer, e.g. location 0x8000.

First command is to dump content of SRAM with key 'h'.

```
PAULMON2 Loc:8000 > Hex dump external memory
```

```
8000: 02 80 03 75 81 44 90 F0 00 85 83 A0 12 94 09 EC      u D p      l
8010: 4D 60 6A 78 45 80 03 76 00 18 B8 42 FA 78 42 80      M`jxE v 8BzxB
8020: 03 76 00 18 B8 42 FA 78 20 80 03 76 00 18 B8 20      v 8Bzx v 8
8030: FA 90 F0 D7 AE 83 AF 82 90 F0 B6 12 80 86 60 05      z pW. / p6 `
8040: E4 F0 A3 80 F6 90 F0 00 A8 82 90 F0 00 A9 82 E8      dp# v p ( p ) h
8050: C3 99 50 05 76 00 08 80 F6 90 80 E4 12 80 8F 90      C P v v d
8060: 80 E8 12 80 8F 90 80 EC 12 80 8F 90 80 F0 12 80      h l p
8070: AD 90 80 F6 12 80 AD 90 80 FC 12 80 AD 75 D0 00      - v - | -uP
8080: 12 93 DD 02 81 02 EF 65 82 70 03 EE 65 83 22 E4      ] oe p ne "d
8090: 93 F8 74 01 93 F9 74 02 93 FE 74 03 93 F5 82 8E      xt yt ~t u
80A0: 83 E8 69 70 01 22 E4 93 F6 A3 08 80 F4 E4 93 FC      hip "d v# td |
80B0: 74 01 93 FD 74 02 93 FE 74 03 93 FF 74 04 93 F8      t }t ~t t x
80C0: 74 05 93 F5 82 88 83 12 80 86 70 01 22 E4 93 A3      t u p "d #
80D0: A8 83 A9 82 8C 83 8D 82 F0 A3 AC 83 AD 82 88 83      ( ) p#, -
80E0: 89 82 80 E3 20 20 8F 93 42 42 8F 93 45 45 8F 93      c BB EE
80F0: F0 00 8F 93 8F 93 F0 D7 8F 93 8F 93 F0 D7 90 1F      p pW pW
```

```
PAULMON2 Loc:8100 >
```

The hex dump will display 256 bytes of data from 0x8000 to 0x80FF. Right hand side shows the printable ASCII representation. For nonprintable, it shows blank instead. See below prompt, the next current pointer will be 0x8100. So if we press key 'h' again it will display from address 0x8100.

We can see the data resided in the on-chip RAM as well. Using command 'i' or internal RAM display.

```
PAULMON2 Loc:8100 > Hex dump internal memory
```

```
00: 00 00 00 00 43 00 00 81 B4 08 08 55 AA AA AA AA
10: 55 55 55 55 AA AA AA AA 5D 55 55 55 AA AA A8 AA
20: 31 00 00 37 00 00 00 08 00 84 20 04 00 02 8F B3
30: 00 DF 01 80 07 07 84 01 43 72 0F A6 0B 00 0B 37
40: FB 00 00 00 24 83 80 08 94 18 93 66 92 1F 92 8D
50: 8B 31 F2 8D F0 14 FB 8A 78 89 55 55 AA AA AA AA
60: 55 55 55 55 AA AA AA AA 55 55 55 45 AA AA AA AA
70: 55 55 55 5D AA AA AA AA E7 36 AF FA AA BA AA AA
80: 55 55 55 55 AA AA AA AA 55 55 55 D5 AA AA AA AA
90: 55 55 55 55 AA AA AA AA 55 55 55 55 AA AA AA AA
A0: 55 55 55 55 AA AA AA AA 55 45 55 55 AB AA AA AA
B0: 55 55 55 55 AA A8 AA AA 55 55 15 55 AA AA AA AA
C0: 55 55 55 55 AA AA BA AA 55 55 55 55 2B BA AA AA
D0: 55 55 55 55 AA AB AA AA 55 55 55 75 AA A2 AA AA
E0: 55 55 75 55 BA AA EA AA 57 55 55 55 AA AA AA AA
F0: D5 55 55 55 AA AA AA AA 55 75 55 55 53 24 43 A4
```

```
PAULMON2 Loc:8100 >
```

It shows the content of 256 bytes RAM on-chip the CPU. See the address left hand side, the address is an 8 bit wide.

Let clear the memory content of external SRAM to zero using command 'c'.

```
PAULMON2 Loc:8000 > Clear memory

First Location: 8000
Last Location: 80FF
Are you sure?

PAULMON2 Loc:8000 > Hex dump external memory

8000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

PAULMON2 Loc:8100 >
```

Machine Code

The machine code is the code that CPU can execute directly. The monitor program helps us to enter the machine code using hexadecimal number. Here is a sample machine code that we will enter into a memory, e.g. 74, 05, 24, 01, b2, 97, 01, 02.

See example of entering machine code into SRAM below.

```
PAULMON2 Loc:8000 > Editing external ram

Address> 8000
8000: (00)  New Value: 74
8001: (00)  New Value: 05
8002: (00)  New Value: 24
8003: (00)  New Value: 01
8004: (00)  New Value: B2
8005: (00)  New Value: 97
8006: (00)  New Value: 01
8007: (00)  New Value: 02
8008: (00)  New Value:      Editing complete, this location unchanged

PAULMON2 Loc:8008 >
```

Now using a nice command for disassembling the machine code into mnemonic. Command 'l' will convert a machine code into 8051 instruction or mnemonic.

See what we have entered?

```
PAULMON2 Loc:8008 > New location

New memory location: 8000

PAULMON2 Loc:8000 > List

8000: 74 05      MOV     A, #05
8002: 24 01      ADD     A, #01
8004: B2 97      CPL     P1.7
8006: 01 02      AJMP   8002
8008: 00          NOP
8009: 00          NOP
800A: 00          NOP
800B: 00          NOP
800C: 00          NOP
800D: 00          NOP
800E: 00          NOP
800F: 00          NOP
8010: 00          NOP
8011: 00          NOP
8012: 00          NOP
8013: 00          NOP
8014: 00          NOP
8015: 00          NOP
8016: 00          NOP
8017: 00          NOP
```

```
PAULMON2 Loc:8018 >
```

We have entered a machine code byte by byte from address 0x8000. The first two bytes, 74 and 05 are disassembled into 8051 instruction that moves an immediate 8-bit value into accumulator A. Similarly for the rest.

Another useful command for learning is command 's' single step running. We can let CPU execute our machine code instruction by instruction and examine the content of registers, or memory.

Before using command 's' we have to put a jumper to JP1 to tie INT1 to GND.

```
PAULMON2 Loc:8000 > Single-Step
```

```
Jump to memory location (8000), or <ESC> to exit:
```

```
Now running in single step mode: <RET>= step, ?= Help
```

ACC	B	C	DPTR	R0	R1	R2	R3	R4	R5	R6	R7	SP	Addr	Instruction
00	00	0	8000	00:00:00:00:00:00:00:00	0A	14DD:	JMP	@A+DPTR						
05	00	0	8000	00:00:00:00:00:00:00:00	0A	8000:	MOV	A, #05						
06	00	0	8000	00:00:00:00:00:00:00:00	0A	8002:	ADD	A, #01						
06	00	0	8000	00:00:00:00:00:00:00:00	0A	8004:	CPL	P1.7						
06	00	0	8000	00:00:00:00:00:00:00:00	0A	8006:	AJMP	8002						
07	00	0	8000	00:00:00:00:00:00:00:00	0A	8002:	ADD	A, #01						
07	00	0	8000	00:00:00:00:00:00:00:00	0A	8004:	CPL	P1.7						
07	00	0	8000	00:00:00:00:00:00:00:00	0A	8006:	AJMP	8002						

On the left hand it shows the content of registers. See the bold line the content of ACC changed from 00 to 05 with instruction MOV A,#05 was executed. The instruction ADD A,#01 adds accumulator with 1. When the CPU execute CPL P1.7 instruction, notice the LED on board will be lit. The help command for single stepping also uses key '?'.
The help command for single stepping also uses key '?'.

```
Single Step Command:
```

```
<RET> run next instruction
```

```
<SP> run next instruction
```

```
'?' print this help
```

```
',' print register names
```

```
'R'  print special function registers
'H'  hex dump internal ram
'S'  skip next instruction
'A'  change the Acc value
'Q'  quit single step
```

ASM51

To better understanding CPU architecture, we have to learn how to program the CPU with machine dependent language. Assembly language is hardware based language that totally enabling the user to control all CPU functioning. We can access CPU register, memory and i/o with a simple instruction and require less memory space. ASM51 is the assembler that helps translate a symbolic program into machine code. Here is a sample assembly program that prints “hello world” and counting number from 1 to 10.

```
$mod52
#include (mypaulm2.equ)

eos    equ 0

        dseg at 30h

i:      ds 1
n:      ds 1

        cseg at 8000h

        mov n,#0
        mov i,#10

loop:   mov dptr,#hello
        call pstr
        inc n
        mov a,n
        call pint8u
        djnz i,loop

        jmp monitor

hello:  db 13,10,'hello world ',eos
        end
```

The first two lines are assembler directives. \$MOD52 is the file that contains direct address of the 8052 registers. The second line \$INCLUDE(MYPAULM2.EQU) tells the assembler to include this

file to the source program. It will let the source program to refer the symbols that are the utility routines, e.g. CALL PSTR, CALL PINT8U. The source program can be edited by using a simple text editor program. The ASM51 program will translate it to standard INTEL HEX file. The hex file contains the machine code in hexadecimal number. PAULMON2 has a command that can receive INTEL HEX file and convert them into binary code and write to the SRAM.

See above BOLD line, it was a directive that tells the assembler to put the machine code from address 0x8000 which is the start address of C52EVB SRAM.

Here is a simple DOS command line.

```
C:\ASM51>ASM51 hello<
```

We then get the HEX file, hello.hex.

```
:1080000075310075300A9080191200380531E5315C
:1080100012004DD530F00219540D0A68656C6C6F72
:0880200020776F726C642000F0
:00000001FF
```

Now PAULMON2 has a command 'd' or download INTEL HEX file into SRAM.

```
PAULMON2 Loc:8000 > Download
Begin ascii transfer of Intel hex file, or ESC to abort
....
Download completed
Summary:
 4 lines received
40 bytes received
40 bytes written
No errors detected
PAULMON2 Loc:8000 >
```

We see that after pressing command 'd' the monitor will wait us to send the hex file. With a Hyperterminal we can use send text file in Transfer menu.

Let see what we have downloaded into SRAM? Using command 'l' to list the instruction.

```
PAULMON2 Loc:8000 > List
8000: 75 31 00      MOV      31, #00
8003: 75 30 0A      MOV      30, #0A
8006: 90 80 19      MOV      DPTR, #8019
8009: 12 00 38      LCALL   0038
800C: 05 31          INC      31
800E: E5 31          MOV      A, 31
8010: 12 00 4D      LCALL   004D
8013: D5 30 F0      DJNZ    30, 8006
8016: 02 19 54      LJMP    1954
8019: 0D            INC      R5
801A: 0A            INC      R2
801B: 68            XRL     A, R0
801C: 65 6C        XRL     A, 6C
801E: 6C            XRL     A, R4
801F: 6F            XRL     A, R7
8020: 20 77 6F      JB      2E.7, 8092
8023: 72 6C        ORL     C, 2D.4
8025: 64 20        XRL     A, #20
8027: 00            NOP
8028: 00            NOP

PAULMON2 Loc:8029 >
```

Let compare the listing above to the source code and hex file, you can then get understand very easy.

Now to run the program we can use command 'j' or jump to user program.

```
PAULMON2 Loc:8029 > Jump to memory location
Jump to memory location (8029), or ESC to quit: 8000
running program:
hello world 1
```

```
hello world 2
hello world 3
hello world 4
hello world 5
hello world 6
hello world 7
hello world 8
hello world 9
hello world 10
Accumulator = 0A
PSW = 20

PAULMON2 Loc:8000 >
```

IAR C Compiler

For the application program, it is easier and a good design to develop a complicated program using high level language. There are a number of high level languages suitable for microcontrollers, e.g. C, BASIC, and FORTH. C language is quite popular and easy to use with microcontroller. Here I describe some trick of how to use a given c compiler with the C52 board. Briefing, the compiler will translate the c source code into the machine code. Similarly to using ASM51, the machine code will be the INTEL hex file that can be downloaded to the 32kB SRAM with command 'd'. However with a C compiler the compiler first translate the c program to assembly code. And the assembler will translate it to the relocatable object code. The relocatable object has no absolute address. To get the executable machine code for a given microcontroller board, the object code or machine code is then linked to another module and is placed at the real memory address. As we know from c programming, in c source code there is no statement that declares hardware address. In order to make the object code to be an executable code for a given microcontroller board, we must know memory segmentation, and the real memory address.

Now let take a look a sample c program below.

```
// Sample c program

#include <stdio.h>           1
#include <io51.h>           2

idata int counter, tick;    3

void main(void)             4
{
    counter = 0;           // clear counter

    TMOD |= 0x01;         // logical OR TMOD with 0x01           5
    ET0 = 1;              // enable timer0 interrupt
    EA = 1;                // enable all interrupt
    TR0 = 1;              // run timer0

    for (;;)              // run block below forever
    {
        switch(tick){
            case 5: P1|= 0x80; break;
            case 50: printf("\n hello worlds %d",counter++); break;
            case 100: P1 &= ~0x80; tick = 0; break;
        }
    }
}

// service routine for timer0 interrupt every 10ms (11.0592MHz)

interrupt [0x0B] void Timer0_int (void)           6
{
    tick++;
    TH0 |= 0xDC;    // reload timer0 with 0xDC00
}
```

Notice: Bold numbers are markers for description.

We see that even the one, who has never been using c before, would get what result of program running will be. The main() function is just to initialize timer registers, variable and do checking the value of tick to pass control according to what tick will be. The variable tick is declared as a global variable so every function can refer and use it. See the timer interrupt service routine, it is just to increment tick by one every 10ms.

Let take a look more detail in the source program. Line **1** and **2** tell the compiler to include the header file for standard i/o and 8051 register addresses. Line **3** declares the 16-bit variables name counter and tick. The modifier `idata` (internal data memory) forces the compiler to reserve both variables using on chip data memory. Line **4** is `main()` function that will be called after startup code has initialized STACK and some memory. We see that line **5** shows how to access CPU registers, just using assignment operator, e.g. `TMOD |= 0x01` or `TMOD = TMOD | 0x01`. For a direct bit address, the c compiler allows us to use assignment statement to set or clear bit directly. The body of `main()` function is a for loop that runs forever checking the value of tick and pass control to each statement. Let say every 50ms, makes P1.7 to '1'. Every 500ms, prints "hello worlds" and count value to terminal screen. And every 1000ms or one second, makes P1.7 to '0', LED that connected to P1.7 with sink current driving will be lit and tick will be cleared. Line **6** is an interrupt service routine for timer0 interrupt. We see that we can put the interrupt service routine in c source code directly.

There is no hardware address associated to above c program. So how can we get the object code that can run on our C52 board? There must be the way to tell the compiler what is our hardware address. With a small c compiler e.g. Micro-C compiler for 8051, we may modify the startup code using assembler directive, say `ORG` or `CSEG AT` to declare our memory addresses.

Here we will see how to provide such modification for IAR C compiler. As mentioned earlier the process of c compiler that generates the executable object code by the linker program. The IAR linker has method to allow user to customize their hardware memory. The method is to override the default linker command file with `.XCL` extension by user command file. The command file tells the linker where to place the object code for data, code, interrupt vector, etc.

Let see the example of my modification of the command file.

```

// sample mylnk8051.xcl

-c8051

// If you have register independent code use: -D_R=0
// (or 8,16,24) to choose the register bank used at startup
-D_R=0

// Setup "bit" segments (always zero if there is no need
// to reserve
// bit variable space for some other purpose)

-Z(BIT)C_ARGB,BITVARS=0

// Setup "data" segments. Start address may not be less
// than start of register bank + 8. Space must also
// be left for interrupt functions with the "using" attribute.

-Z(DATA)B_UDATA,B_IDATA,C_ARGD,D_UDATA,D_IDATA=20

// Setup "idata" segments (usually loaded after "data")

-Z(IDATA)C_ARGI,I_UDATA,I_IDATA,CSTACK

// Setup "xdata" segments to the start address of external
// RAM.
// Note that it starts from 1 since a pointer to address zero
// is regarded as NULL.
// Note that this declaration does no harm even if you use
// a memory
// model that does not utilize external data RAM

-Z(XDATA)P_UDATA,P_IDATA,C_ARGX,X_UDATA,X_IDATA,ECSTR,RF_XDATA,
XSTACK=f000

// Setup all read-only segments (PROM). Usually at zero

-Z(CODE)INTVEC,RCODE,D_CDATA,B_CDATA,I_CDATA,P_CDATA,X_CDATA,
C_ICALL,C_RECFN,CSTR,CCSTR,CODE,CONST=8000

// See configuration section concerning printf/sprintf
-e_small_write=_formatted_write,

// See configuration section concerning scanf/sscanf
-e_medium_read=_formatted_read

```

For the external data memory(XDATA), I set to **f000** or 0xF000. And for the code memory (CODE), I set to **8000** or 0x8000. The memory model was set to COMPACT. After compile the sample.c

and link it with our command file, we will get the hex file suitable for downloading into our C52 board. See how big the machine code is.

```
PAULMON2 Loc:8000 > Download

Begin ascii transfer of Intel hex file, or ESC to abort

.....
.....
.....
.....
Download completed

Summary:
 219 lines received
 3026 bytes received
 3026 bytes written
No errors detected

PAULMON2 Loc:8000 > Jump to memory location

Jump to memory location (8000), or ESC to quit:

running program:

hello worlds 0
hello worlds 1
hello worlds 2
hello worlds 3
hello worlds 4
hello worlds 5
hello worlds 6
hello worlds 7
hello worlds 8
hello worlds 9
hello worlds 10
hello worlds 11
hello worlds 12
hello worlds 13
hello worlds 14
hello worlds 15
hello worlds 16
```

The number of machine code is 3,026 Bytes! Not surprise for c programmer. And it works fine with our C52 board.

ZAP EEPROM

Let use ZAP EEPROM function to save above program to eeprom. We enter eeprom function by command 'z'. Let write the binary code using function save with key '1'. Enter start address 8000 and the number of byte 3026 bytes.

```
PAULMON2 Loc:8000 > ZAP EEPROM

24LC256 EEPROM functions 1 SAVE, 2 DUMP, 3 LOAD
>
Save Program start address> 8000
Number of bytes [0-32768] > 3026
Done..
24LC256 EEPROM functions 1 SAVE, 2 DUMP, 3 LOAD
>
```

If you want to see the content of eeprom, try with command DUMP using key '2'.

```
24LC256 EEPROM functions 1 SAVE, 2 DUMP, 3 LOAD
>
eeprom address [0000-7FFF] > 0

0000 00 80 2E F4 4D 02 20 52 00 00 00 A6 93 01 00 00
0010 02 80 0E 75 81 44 90 F0 00 85 83 02 88 34 75 81
0020 52 90 F0 00 85 83 A0 12 88 4E EC 4D 60 6A 78 53
0030 80 03 76 00 18 B8 4F FA 78 4F 80 03 76 00 18 B8
0040 4F FA 78 20 80 03 76 00 18 B8 20 FA 90 F0 00 AE
0050 83 AF 82 90 F0 00 12 80 91 60 05 E4 F0 A3 80 F6
0060 90 F0 00 A8 82 90 F0 00 A9 82 E8 C3 99 50 05 76
0070 00 08 80 F6 90 80 EF 12 80 9A 90 80 F3 12 80 9A
0080 90 80 F7 12 80 9A 90 80 FB 12 80 B8 90 81 01 12
0090 80 B8 90 81 07 12 80 B8 75 D0 00 12 87 D6 02 81
00A0 0D EF 65 82 70 03 EE 65 83 22 E4 93 F8 74 01 93
00B0 F9 74 02 93 FE 74 03 93 F5 82 8E 83 E8 69 70 01
00C0 22 E4 93 F6 A3 08 80 F4 E4 93 FC 74 01 93 FD 74
00D0 02 93 FE 74 03 93 FF 74 04 93 F8 74 05 93 F5 82
00E0 88 83 12 80 91 70 01 22 E4 93 A3 A8 83 A9 82 8C
00F0 83 8D 82 F0 A3 AC 83 AD 82 88 83 89 82 80 E3 20

24LC256 EEPROM functions 1 SAVE, 2 DUMP, 3 LOAD
>
```

Look at the content of eeprom memory, **00 80** is start address of code 0x8000 and **2E F4** is two's complement (0xF42E) of 3026. The first instruction was saved at eeprom address 0x0010, **02 80 0E** this instruction is LJMP 0x800E. You may quit zap eeprom function and use command 'l' to disassemble the machine code.

Now try enters command LOAD with key '3'.

```
24LC256 EEPROM functions 1 SAVE, 2 DUMP, 3 LOAD
>
Loading...
hello worlds 0
hello worlds 1
hello worlds 2
hello worlds 3
hello worlds 4
hello worlds 5
hello worlds 6
hello worlds 7
hello worlds 8
hello worlds 9
hello worlds 10
hello worlds 11
hello worlds 12
hello worlds 13
hello worlds 14
```

The binary image or machine code will be retrieved and written to SRAM when complete the CPU will jump to 0x8000 start running the program.

To enable boot loader when power up the board, we can also tie P3.5 to GND.

APPENDIX

- I. BOM
- II. Schematic
- III. PCB Layout

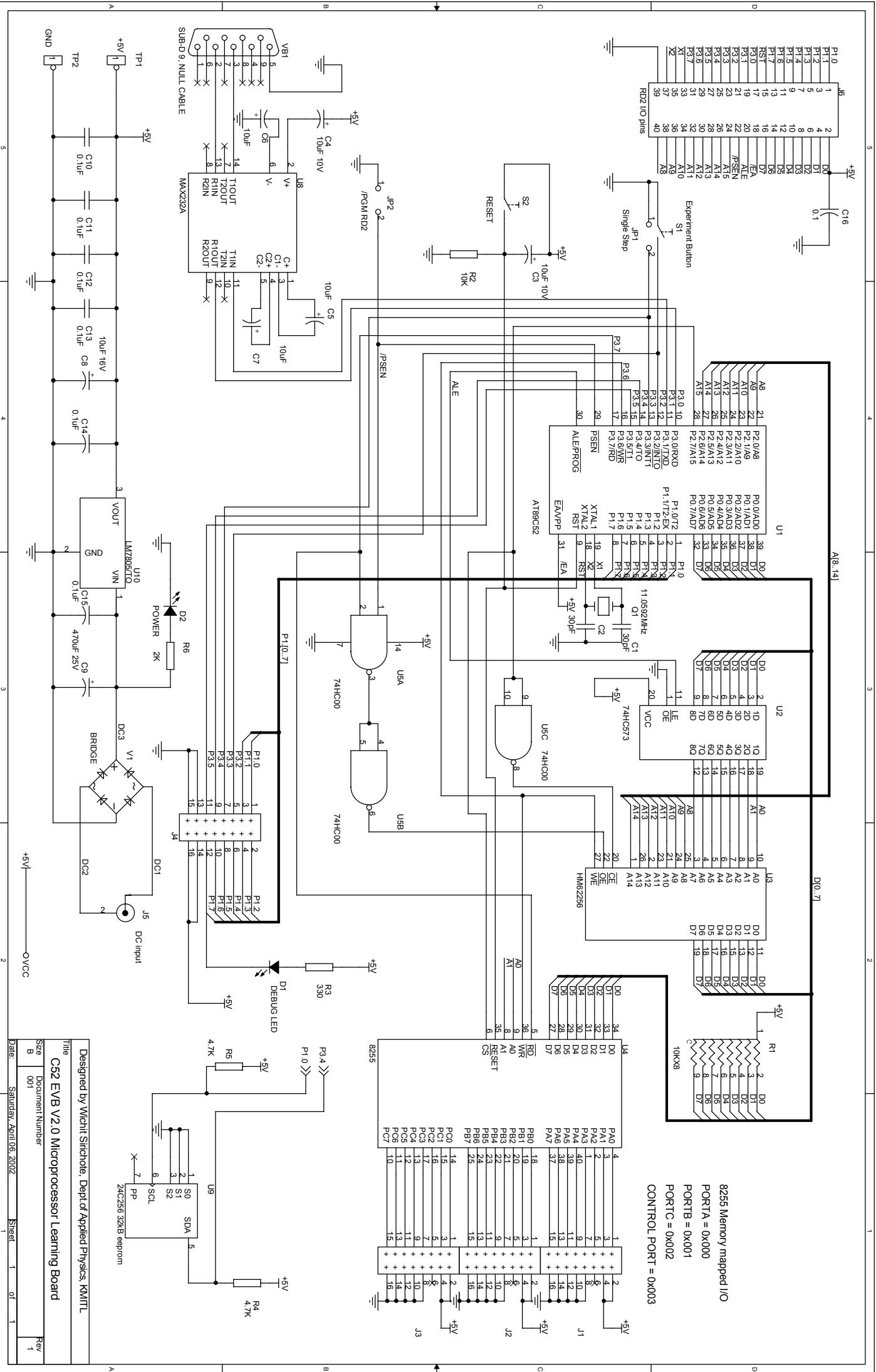
C52 EVB V2.0 Microprocessor Learning Board

Revised: Saturday, April 06, 2002

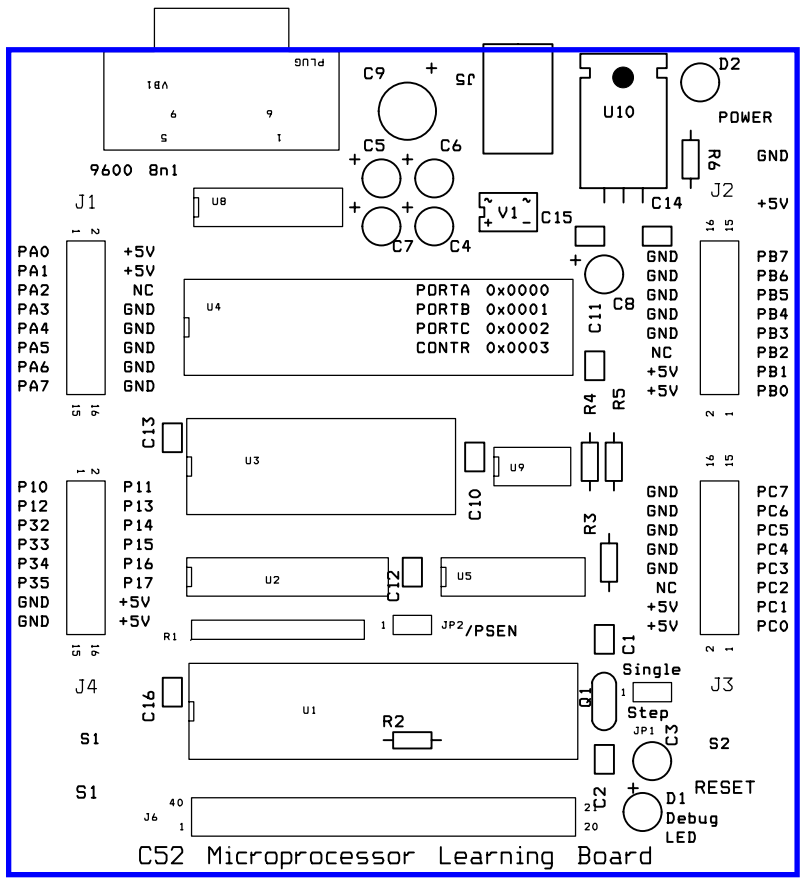
001 Revision: 1

Bill Of Materials April 6,2002 21:40:38 Page1

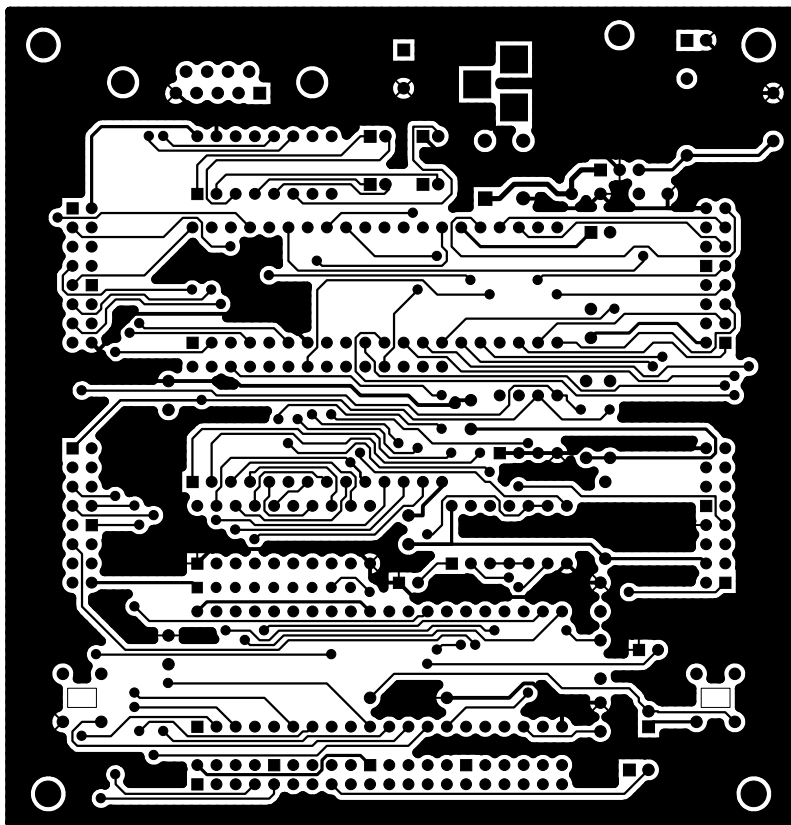
Item	Quantity	Reference	Part
1	2	C1,C2	ceramic cap 30pF
2	2	C3,C4	electrolytic cap 10uF 10V
3	3	C5,C6,C7	electrolytic cap 10uF
4	1	C8	electrolytic cap 10uF 16V
5	1	C9	electrolytic cap 470uF 25V
6	4	C10,C11,C12,C13	ceramic cap 0.1uF
7	2	C14,C15	ceramic cap 0.1uF
8	1	C16	ceramic cap 0.1uF
9	1	D1	small dot LED for debug P1.7
10	1	D2	small dot LED for POWER
11	1	JP1	Single Step jumper
12	1	JP2	Philips RD2 program mode setting
13	4	J1,J2,J3,J4	Expansion I/O connector 16-pin
14	1	J5	DC input
15	1	J6	Philips RD2 40-pin expansion connector
16	1	Q1	Crystal 11.0592MHz
17	1	R1	R pack 10K x 8
18	1	R2	10K
19	1	R3	330
20	2	R5,R4	4.7K
21	1	R6	2K
22	1	S1	Experiment Button
23	1	S2	RESET
24	1	TP1	+5V
25	1	TP2	GND
26	1	U1	AT89C52
27	1	U2	74HC573
28	1	U3	HM62256
29	1	U4	8255
30	1	U5	74HC00
31	1	U8	MAX232A
32	1	U9	24C256 32kB eeprom
33	1	U10	LM7805
34	1	VB1	SUB-DB 9
35	1	V1	BRIDGE diode
36	1	PCB	Double Side Printed Circuit Board



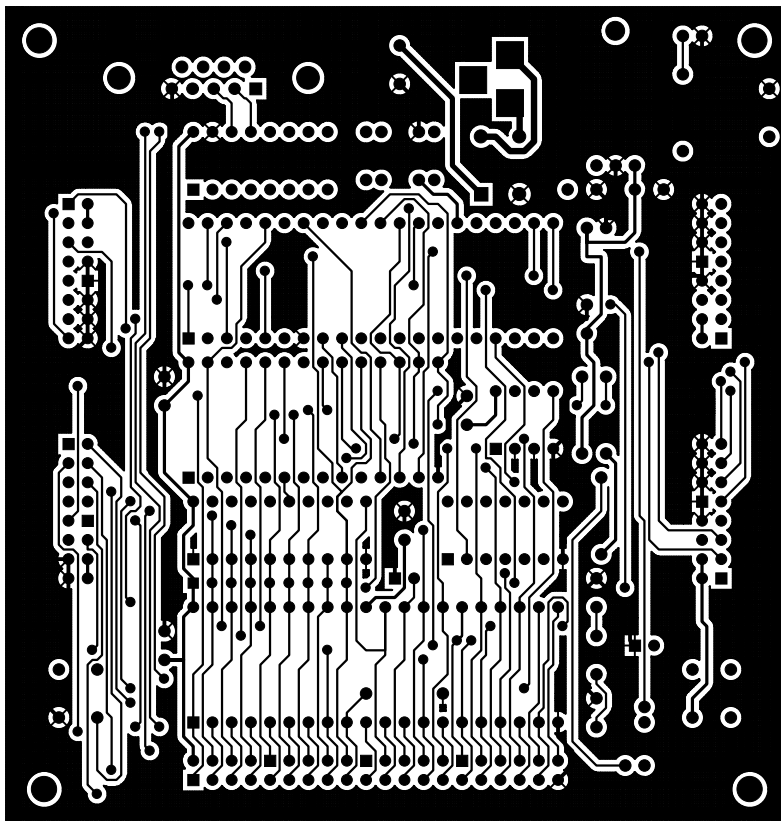
Title		Designed by Wiclit Sircinote, Dept of Applied Physics, KMVITL	
C52 EVB V2.0 Microprocessor Learning Board			
Size	Document Number		
B	001		
Date:	Saturday, April 06, 2002	Sheet	1 of 1
Rev	1		



Component Placement Layout



TOP LAYER



BOTTOM LAYER