

PROGRAMMING THE MICROCONTROLLER

ASSEMBLY LANGUAGE

Assembly language is of higher level than machine language and hence easier to use.

An assembly language code consists of

- a) Program statement lines
- b) Comment lines

A program statement is a line that contains 4 fields in the following format:

```
[<LABEL>]   [<OPCODE MNEMONIC>]   [<OPERANDS>]   [<comments>]
              or
[<LABEL>]   [<DIRECTIVE MNEMONIC>]   [<OPERANDS>]   [<comments>]
```

where [] indicates an optional field that may not be always required. The fields are separated by a tab or space. (Tab is recommended, since it ensures an orderly appearance to your code. For the same reason, when a field is not used, the tab or blank should still to be used, such that the fields of the same type stay aligned in same columns.) When writing <LABEL>, <OPCODE MNEMONIC> or <DIRECTIVE MNEMONIC>, and <OPERANDS>, use upper case characters. When writing <comments>, use lower case.

The <OPCODE MNEMONICS> correspond to the microcontroller opcodes. These mnemonics are found in the Motorola MC68HC11 programming reference guide and related literature.

The <DIRECTIVE MNEMONICS> are native to the Assembly language. A list of directives is given in Table 1. The directives that you will use often are shown in **bold**.

Table 1 Assembler directives

Name of Assembler directive	what it does	Alias for	
END	end program		
DB	define bytes	FCB	
DW	define words	FDB	
DS	define storage	RMB	
EQU	equate		
FCB	form constant byte		
FCC	form constant characters		
FDB	form double bytes		
ORG	set origin		
RMB	reserve memory bytes		
#INCLUDE	include source file		
\$INCLUDE	include source file	#INCLUDE	

The <OPERAND> contains a value, an expression, an address, or a label that the opcodes or the directives need. The operand could be up to 4 bytes long, separated by commas. Some opcodes or directives do not require operands (inherent mode).

The constants used in the <OPERAND> can be hex, decimal, binary, or octal numbers. Table 2 gives the assembler symbols used to this purpose.

Table 2 Assembler symbols for constants

Symbol	Meaning	Example
\$<number>	hex number	\$A1
<number>	decimal number	20
%<number>	binary number	%11001010
@<number>	octal number	@73
'<string>', '<string>	ASCII string	'A' or 'A (the latter does not work with #INCLUDE)

The expressions used in the <OPERAND> can use any of the operators listed in Table 3

Table 3 Assembler symbols for expressions

Symbol	Meaning	Example
-	unary minus	-4
&	binary AND	%11111111&%10000000
!	binary OR	%11111111!%10000000
*	multiplication	3*\$2A
/	division	\$7E/3
+	addition	1+2
-	subtraction	3-1
()	parentheses used for grouping	3*(1+2)

Important conventions used in the <OPERAND> are given in Table 4:

Table 4 Other important conventions

Symbol	Meaning	Example
#	immediate mode (IMM)	#\$A3
;	start of comment line and of comment inside a program statement	LDAA #\$FF ; Load accA
*	alternate sign for start of comment line only	* This is a comment
,X	index X mode (IND,X)	LDAA TFLG1,X
,Y	index X mode (IND,Y)	LDAA TFLG2,Y

The <LABEL> is a very powerful concept that can greatly simplify the programmer's task. The <LABEL> consists of a string of alphanumeric characters that make up a name somehow meaningful to the programmer. The placement of the <LABEL> can be in one of the following positions:

- a) In the first column and terminates with a tab or blank character
- b) In any column and terminates with a colon (:)

There are 3 different usages of the <LABEL>:

- 1) To assign the name inserted in the <LABEL> to a location in a program. The <LABEL> will be assigned the address of that location
- 2) To assign the value of an expression or constant to the name inserted in the <LABEL> using the EQU (equate) or SET directives.
- 3) To define the name of a subroutine (macro). Essentially, this is the same as 1), since an address (the subroutine starting address) is assigned to the label.

When labels are assigned to certain addresses, one can tell the program to go to that address by referring to the label (case 1 above). Alternatively, one can use the contents of a certain address by referring to its label, just like when using variables (case 2 above).

A comment is prefixed by semicolon (;). When the assembler detects an semicolon, it knows that the rest of the line is a comment and does not expect any executable instructions from it. A comment can be a separate line (comment line) or can be inserted in a program statement. A comment line can be also prefixed by an asterisk (*). The comments, either in the comment field or as a separate comment line, are of great benefit to the programmer in debugging, maintaining, or upgrading a program. A comment should be brief and specific, and not just reiterate its operation. A comment that does not convey any new information needs not be inserted. When writing a comment, use lower case characters.

A program written in Assembly language is called *source file*. Its extension is .ASM. When the source file is assembled, two files are generated:

- a) Object file that can be run in the microcontroller. The Motorola object file is in ASCII-HEX format. Its generic name is "S19 file". Its extension is .S19
- b) List file, extension .LST, that contains the original code in Assembly language and the corresponding hex codes resulting from the Assembly process. The list file is used by the programmer to verify and debug his/her coding of the program.

The .ASM files can be opened, viewed, edited and saved in the THRSIM11 application. Alternatively, all three file types (.ASM, .LST, .S19) can be also processed in a text editor, e.g., the Notepad application. Examples of .ASM and .LST files follow.

Addressing Modes

Inherent Mode is implied and requires no programming action.

Immediate Mode means that the number contained in the operand will be immediately used.

Direct and Extended Modes use the number contained in the operand to signify an address where the required information should be retrieved from or deposited to. The Extended mode is automatically used for addresses greater than FF.

Index Mode is used by adding the operand to the value already existing in the Index X or Y, as selected. In this case, the operand acts as an offset.

Relative Mode uses the operand as an offset relative to the present Program Counter value.

MICROCONTROLLER COMMANDS

(Section 6 and Section A of M68HC11 Reference Manual)

The 6811 microcontroller has 145 different commands. These commands can be grouped into several categories. The categories and the commands in those categories are listed below:

- 1) Arithmetic operations:
 - a) Addition:
ABA, ABX, ABY, ADCA, ADCB, ADDA, ADDB, ADDD, INC, INCA, INCB, INS, INX, INY
 - b) Subtraction:
SBA, SBCA, SBCB, SUBA, SUBB, SUBD, DEC, DECA, DECB, DES, DEX, DEY
 - c) Multiplication: MUL
 - d) Division: FDIV, IDIV
- 2) Logical operations: (note: logical operations are carried out on a bit by bit basis)
 - a) Standard logical operations: ANDA, ANDB, EORA, EORB, ORAA, ORAB, COM (Boolean inverse), COMA, COMB
 - b) Operations that shift the location of the bits in the register:
ASL, ASLA, ASLB, ASLD, ASR, ASRA, ASRB, LSL, LSLA, LSLB, LSLD, LSR, LSRA, LSRB, LSRD, ROL, ROLA, ROLB, ROR, RORA, RORB
 - c) Operations that compare two numbers:
BITA, BITB, CBA, CMPA, CMPB, CPD, CPX, CPY
- 3) Branching commands: BCC, BCS, BEQ, BGE, BGT, BHI, BHS, BLE, BLO, BLS, BLT, BMI, BNE, BPL, BRA, BRCLR, BRN, BRSET, BSR, BVC, BVS, JMP, JSR, RTS, RTI, WAI
- 4) Memory/Register Functions
 - a) Move data into / out of memory: LDAA, LDAB, LDD, LDS, LDX, LDY, STAA, STAB, STD, STS, STX, STY
 - b) Change the values in memory/registers: BCLR, BSET, CLC, CLI, CLR, CLRA, CLRB, CLV, COM, COMA, COMB, NEG, NEGA, NEGB, SEC, SEI, SEV
 - c) Transfer data from one register to another: TAB, TAP, TBA, TPA, TSX, TSY, TXS, TYS, XGDX, XGDY
- 5) Stack Pointer Functions: PSHA, PSHB, PSHX, PSHY, PULA, PULB, PULX, PULY
- 6) Misc.: NOP, SWI

Note: Boolean inversion commands: COM, COMA, COMB

SAMPLE PROGRAM IN ASSEMBLY LANGUAGE WITH MCU COMMANDS

PROBLEM STATEMENT

This simple program is an example of addition. It performs the operation:

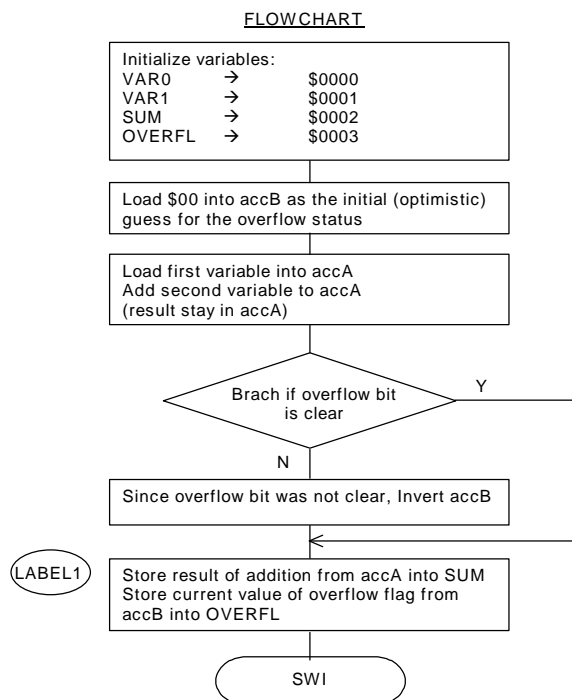
$$\text{VAR0} + \text{VAR1} \rightarrow \text{SUM}$$

In addition, the program checks if an overflow happened during the addition process, and sets the flag OVERFL accordingly.

PROGRAM DESCRIPTION

- The variables are defined in lower memory starting with \$0000, in the order VAR0, VAR1, SUM, OVERFL.
- LDAB with zero is used to reset the initial value of the overflow flag (optimistic!).
- LDAA is used to load VAR0 into accA
- ADDA is used to add accA with VAR1. Result of addition stays in accA
- BVC is used to branch over the next instruction, i.e. to LABEL1, if no overflow occurred
- If an overflow occurred during the addition process, this instruction is reached and COMB is used to invert accB from \$00 to \$FF.
- Label1: STAA is used to store the result of addition from accA into SUM
- STAB is used to store accB (\$00 or \$FF, depending on the logic just discussed) into the overflow flag OVERFL

FLOWCHART



ASSEMBLY (.ASM) CODE

```

* DEMO.ASM
* This simple program adds the contents of
* address $0000 (labeled VAR0) to the contents of
* address $0001 (labeled VAR1) and stores the resulting
* sum at address $0002 (labeled SUM), provided
* the addition process happens without overflow.
* If an overflow occurs during the addition process,
* the overflow flag OVERFL (stored at address $0003)
* is set to $FF; else, it stays $00.

* Include definition of variables for MC68HC11
#INCLUDE      'A:\VAR_DEF.ASM'

* Define program variables
      ORG      DATA
VAR0    RMB    1      ;reserve 1 byte for VAR0
VAR1    RMB    1      ;reserve 1 byte for VAR1
SUM     RMB    1      ;reserve 1 byte for sum
OVERFL  RMB    1      ;reserve 1 byte for overflow flag

* Start main program
      ORG      PROGRAM
      LDAB    #00     ;assume no overflow (optimistic!)
      LDAA    VAR0    ;load VAR0 in accumulator A
      ADDA    VAR1    ;add VAR1 to accumulator A
      BVC     LABEL1  ;jump if no overflow

* We have overflow!
      COMB                    ;Invert accumulator B ($00 to $FF)
LABEL1 STAA    SUM          ;store result of addition
      STAB    OVERFL       ;store accB into overflow flag
      SWI                    ;stop the microcontroller

```

LIST (.LST) OUTPUT RESULTING AFTER ASSEMBLY

```

list#    address  object  label  opcode  operand  comments
          or
          directive

```

DEMO.lst - generated by MiniIDE's ASM12 V1.07b Build 52 [12/29/1999, 16:30:49]

```

1:                                     *12456789012345678901245678901234567 890124567890123456789
2:
3:                                     * DEMO.ASM
4:                                     * This simple program adds the contents of
5:                                     * address $0000 (labeled VAR0) to the contents of
6:                                     * address $0001 (labeled VAR1) and stores the resulting
7:                                     * sum at address $0002 (labeled SUM), provided
8:                                     * the addition process happens without overflow.
9:
10:                                    * If an overflow occurs during the addition process,
11:                                    * the overflow flag OVERFL (stored at address $0003)
12:                                    * is set to $FF; else, it stays $00.
13:

```

```

14:          * Include definition of variables for MC68HC11
1:          * Define variables used by MC68HC11 microcontroller
2:
3:    0000    DATA    EQU    $0000    ;start of data
4:    c000    PROGRAM  EQU    $C000    ;start of program
5:    fffe    RESET    EQU    $FFFE    ;reset vector
6:    1000    REGBAS   EQU    $1000    ;register base
7:
8:    0000    PORTA    EQU    $00
9:    0002    PIOC     EQU    $02
10:   0003    PORTC    EQU    $03
11:   0004    PORTB    EQU    $04
12:   0005    PORTCL   EQU    $05
13:   0007    DDRC     EQU    $07
14:   0008    PORTD    EQU    $08
15:   0009    DDRD     EQU    $09
16:   000a    PORTE    EQU    $0A
17:   000b    CFORC    EQU    $0B
18:   000c    OC1M     EQU    $0C
19:   000d    OC1D     EQU    $0D
20:   000e    TCNT     EQU    $0E
21:   0010    TIC1     EQU    $10
22:   0012    TIC2     EQU    $12
23:   0014    TIC3     EQU    $14
24:   0016    TOC1     EQU    $16
25:   0018    TOC2     EQU    $18
26:   001a    TOC3     EQU    $1A
27:   001c    TOC4     EQU    $1C
28:   001e    TOC5     EQU    $1E
29:   0020    TCTL1    EQU    $20
30:   0021    TCTL2    EQU    $21
31:   0022    TMSK1    EQU    $22
32:   0023    TFLG1    EQU    $23
33:   0024    TMSK2    EQU    $24
34:   0025    TFLG2    EQU    $25
35:   0026    PACTL    EQU    $26
36:   0027    PACNT    EQU    $27
37:   0028    SPCR     EQU    $28
38:   0029    SPSR     EQU    $29
39:   002a    SPDR     EQU    $2A
40:   002b    BAUD     EQU    $2B
41:   002c    SCCR1    EQU    $2C
42:   002d    SCCR2    EQU    $2D
43:   002e    SCSR     EQU    $2E
44:   002f    SCDR     EQU    $2F
45:   0030    ADCTL    EQU    $30
46:   0031    ADR1     EQU    $31
47:   0032    ADR2     EQU    $32
48:   0033    ADR3     EQU    $33
49:   0034    ADR4     EQU    $34
50:   0039    OPTION   EQU    $39
51:   003a    COPRST   EQU    $3A
52:   003b    PPROG    EQU    $3B
53:   003c    HPRI0    EQU    $3C
54:   003d    INIT     EQU    $3D
55:   003e    TEST1    EQU    $3E
56:   003f    CONFIG   EQU    $3F

```

```

list#    address  object  label  opcode  operand  comments
          or
          directive

57:      *12345678901234567890123456789012345678901234567890123456789
15:      #INCLUDE      'A:\VAR_DEF.ASM'
16:
17:      * Define program variables
18:      ORG      DATA
19:      VAR0    RMB      1      ;reserve 1 byte for VAR0
20:      VAR1    RMB      1      ;reserve 1 byte for VAR1
21:      SUM     RMB      1      ;reserve 1 byte for sum
22:      OVERFL  RMB      1      ;reserve 1 byte for overflow flag
23:
24:      * Start main program
25:      ORG      PROGRAM
26:      c000    c6 00      LDAB     #00      ;assume no overflow (optimistic!)
27:      c002    96 00      LDAA     VAR0     ;load VAR1 in accumulator A
28:      c004    9b 01      ADDA     VAR1     ;add VAR2 to accumulator A
29:      c006    28 01      BVC     LABEL1   ;jump if no overflow
30:      * We have overflow!
31:      c008    53          COMB          ;Invert accumulator B ($00 to $FF)
32:      c009    97 02    LABEL1  STAA     SUM     ;store result of addition
33:      c00b    d7 03          STAB     OVERFL  ;store accB into overflow flag
34:      c00d    3f          SWI          ;stop the microcontroller

```

Symbols:

```

data      *0000
labell    *c009
overfl    *0003
program   *c000
sum       *0002
var0      *0000
var1      *0001

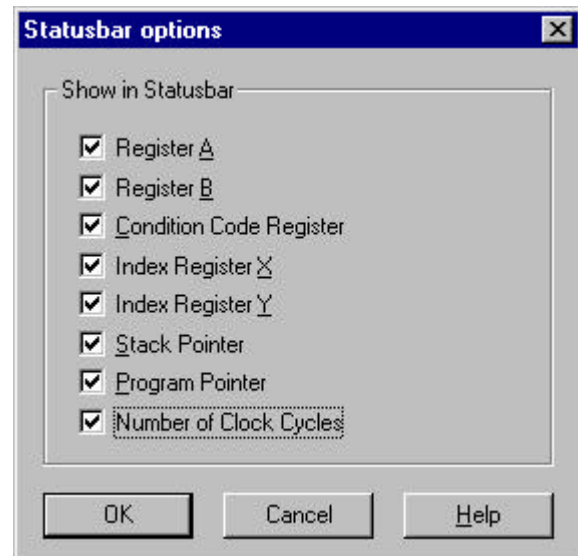
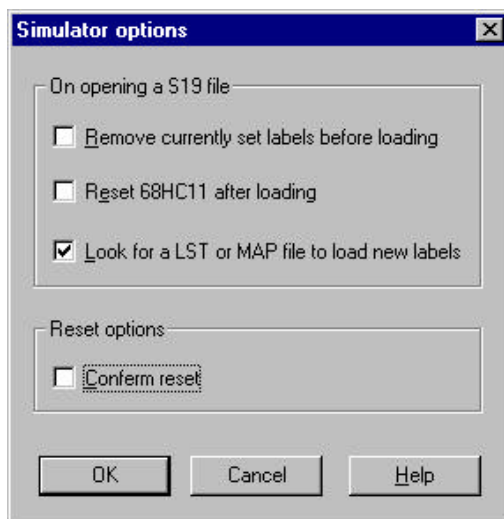
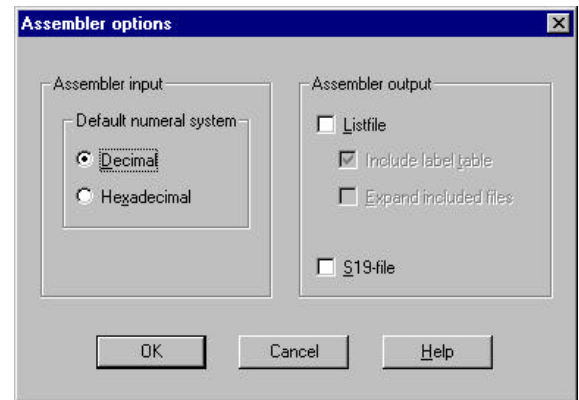
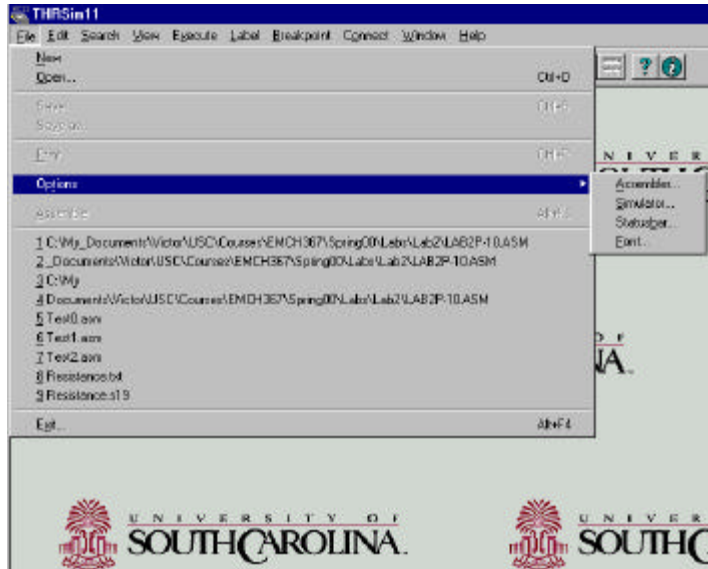
```


THRSIM11

You need to install this software on your PC.

THRSIM11 OPTIONS SETUP

Before you run the simulator first time on a certain PC, set the Options as shown in the following windows:



Immediately after opening the THRSim11 program, close the Commands window. You will not use in this course, unless otherwise specified.

GETTING STARTED WITH PROGRAMMING

Take a formatted empty floppy disk write on the label:

EMCH 367

LASTNAME, Firstname

Email address

Contact telephone #

This way, if you loose the disk, there is a good chance that you might have it recovered.

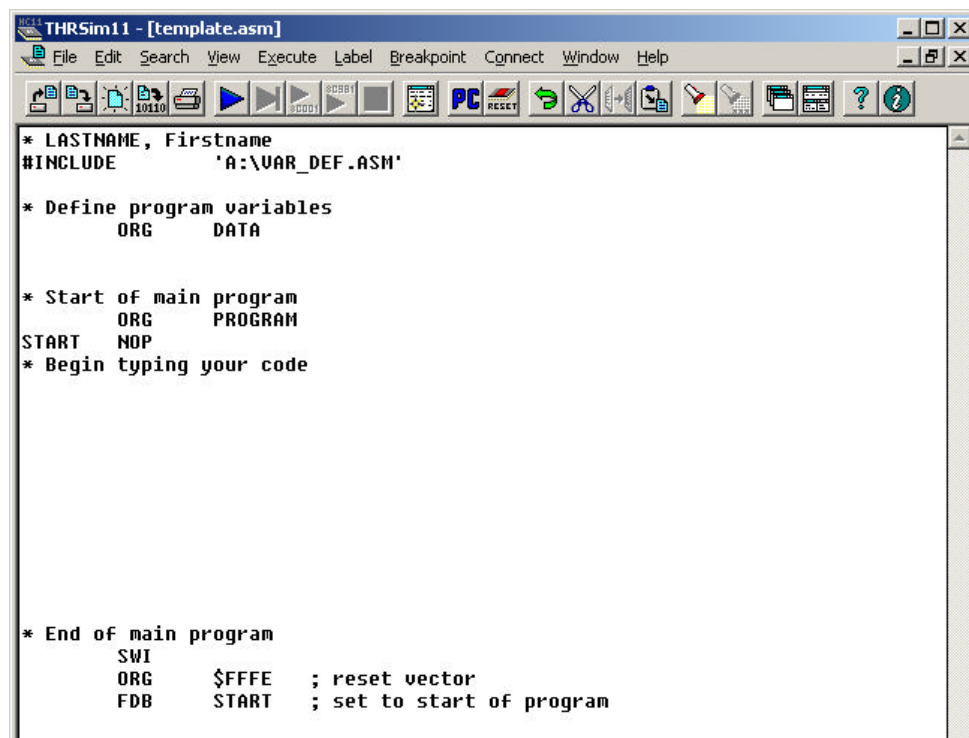
Download the template.asm file and place it on the floppy disk. This template will always be a good to start your programming.

Download the file VAR_DEF.ASM and place it in the root of the directory structure on your floppy disk. (This will allow the programs to find it when executing the instruction #INCLUDE 'A:VAR_DEF.ASM'.

Download example files from the course website onto this disk. (For safety, make copies into your folder or PC.)

USING THE TEMPLATE.ASM FILE

An .asm template file is available on the course website. This template has the required instructions to make your program interface properly with the simulator. When generating a new program, open the template.asm file, save it under the new name you want to create (remember to save on a secure area, preferably your floppy disk), and then start typing in your program in the indicated area.



```

THRSim11 - [template.asm]
File Edit Search View Execute Label Breakpoint Connect Window Help
* LASTNAME, Firstname
#include 'A:\VAR_DEF.ASM'
* Define program variables
ORG DATA
* Start of main program
ORG PROGRAM
START NOP
* Begin typing your code
* End of main program
SWI
ORG $FFFE ; reset vector
FDB START ; set to start of program

```

After you type and save your program (save as often as you can, use Ctrl S for productivity), assemble the program and test run it.

SCREEN/WINDOW CAPTURE

To capture the image of a window or of the complete screen:

- Press **Alt + PrintScreen** to capture the image of the window that is currently active.
- Press **PrintScreen** to capture the image of the entire screen.

The captured image can be viewed on the clip board.

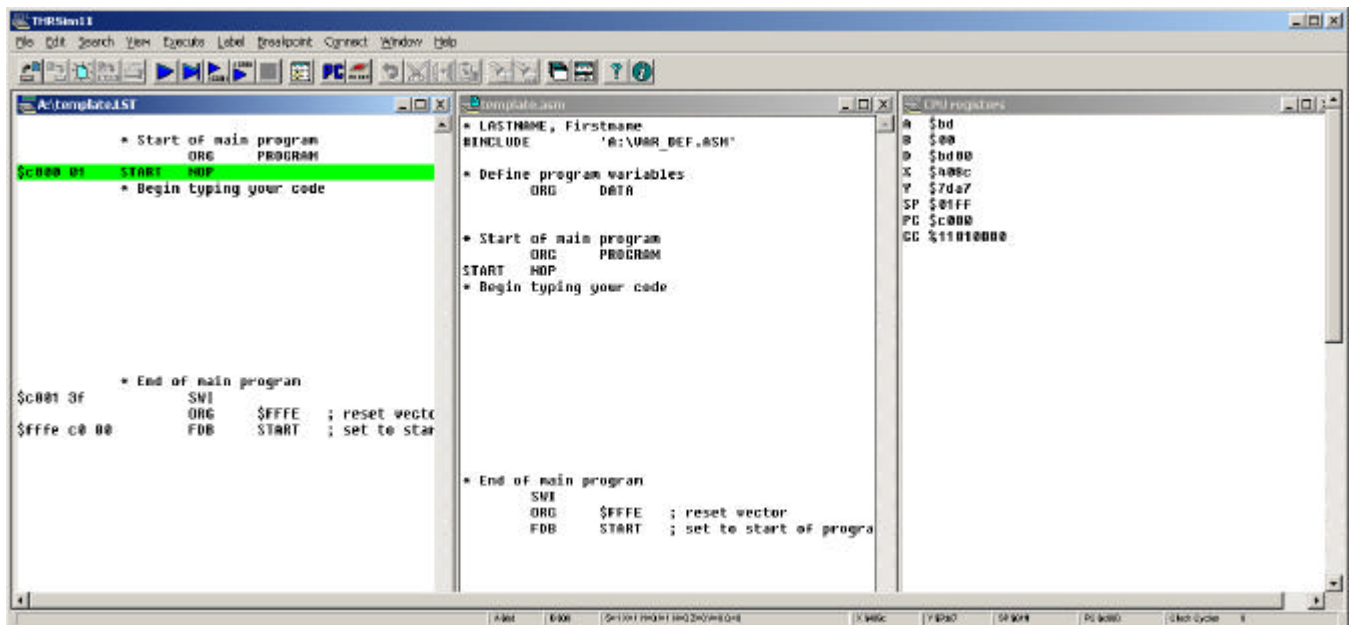
To paste the captured image into a document:

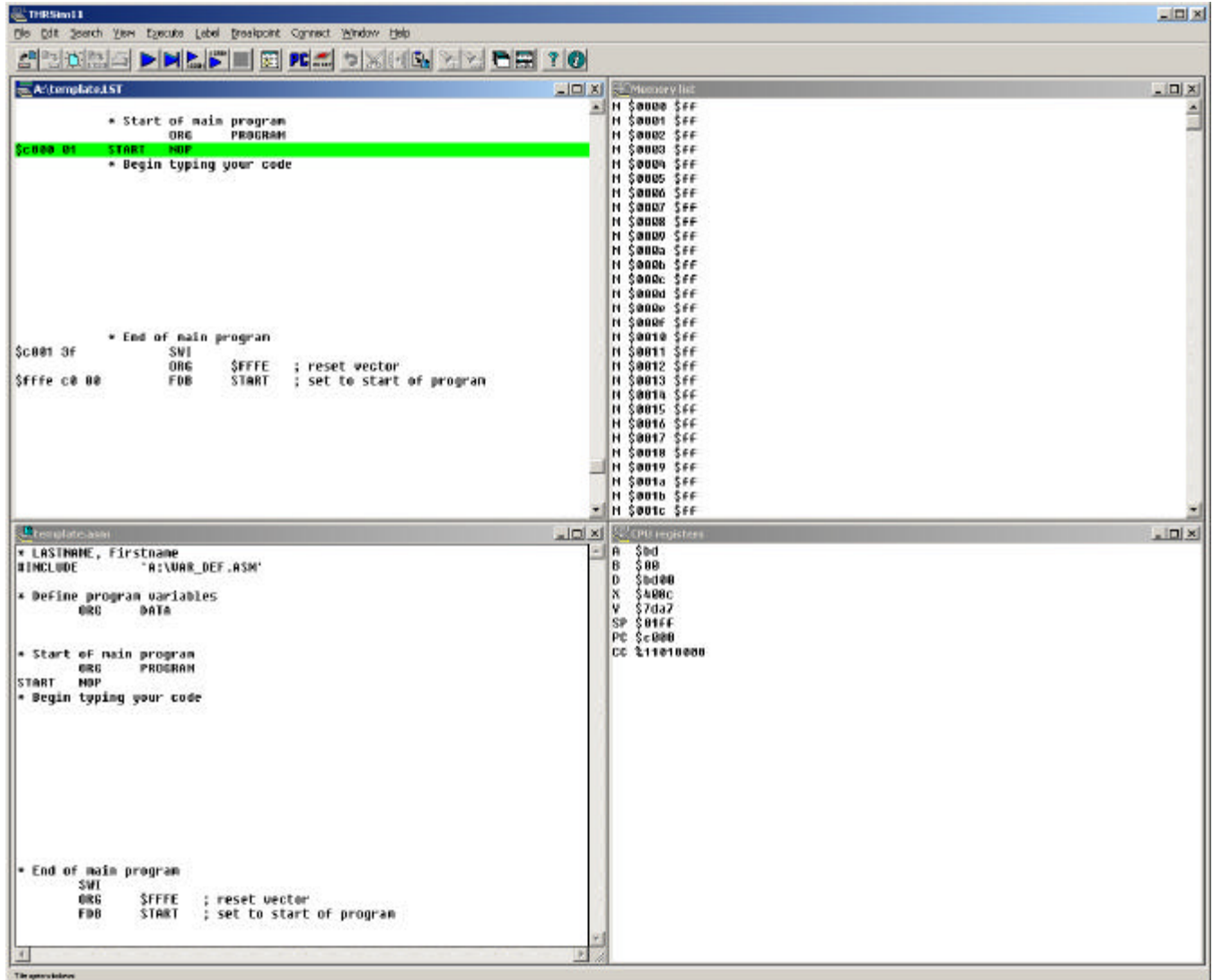
- In the document window, on the **Edit** menu, click **Paste**. Alternatively, use **Ctrl + V**.

Note: In most cases, you will need to capture just the active window, using **Alt + PrintScreen**.

DEFAULT WINDOWS

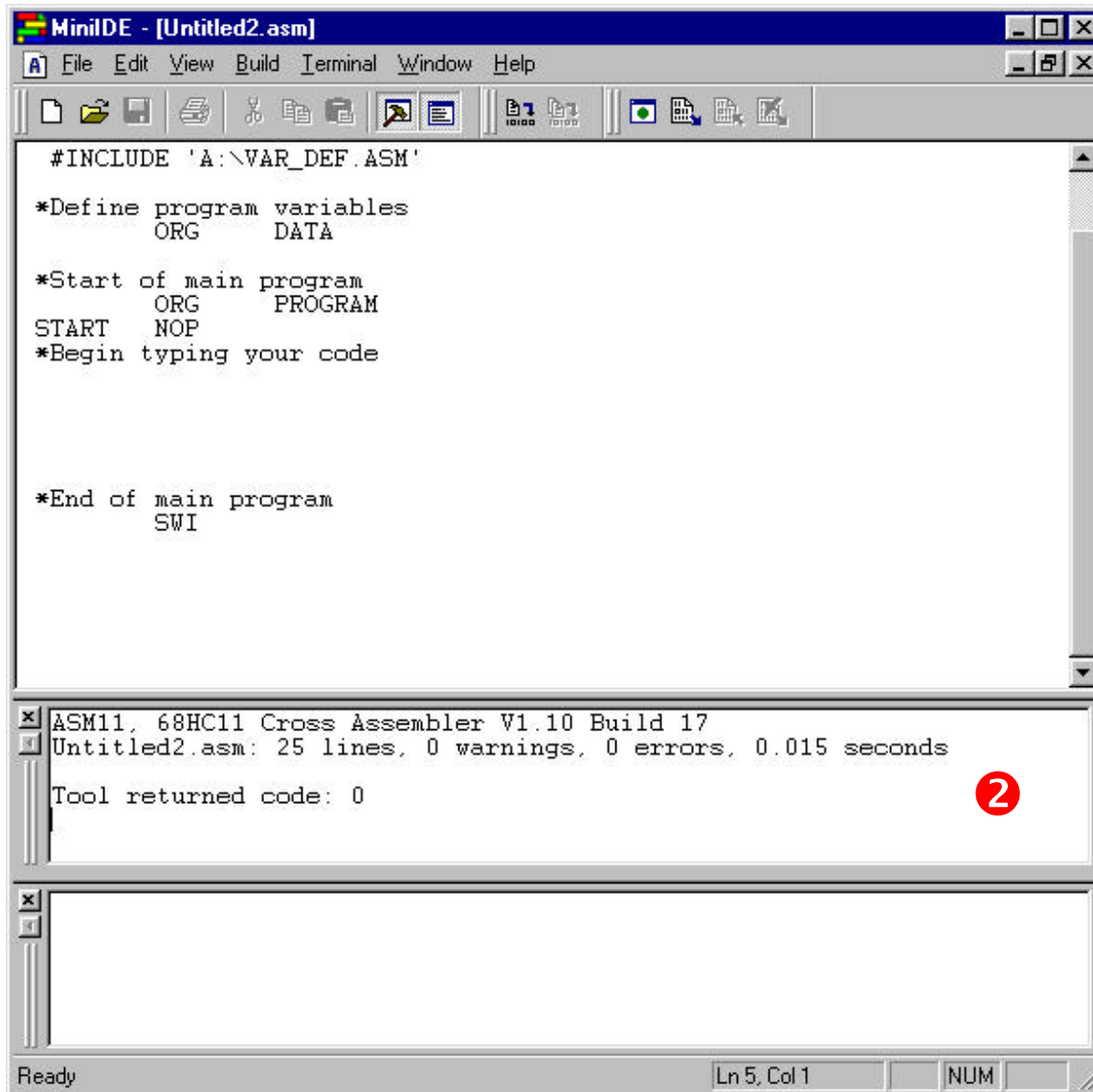
Default windows are either (*.LST, *.asm, and CPU registers), or (*.LST, *.asm, Memory list, and CPU registers), as shown below. In the memory list, standard labels are shown. However, they can be removed if you use the pull down menu command Label/Remove all.





MINIIDE EMULATOR

MiniIDE is an integrated development environment running under Windows 95/98/Me/NT/2000. It was developed by MGTEK in Germany. It is a tool for developers of embedded software who write software in assembler for Motorola's 68HC11 and 68HC12 microcontroller. MiniIDE incorporates an editor and a serial communication terminal. A command-line cross assembler, which is seamlessly integrated in the IDE, is also included.



With MiniIDE, user can edit compile and download program to microcontroller, then debug program interactively. As shown above, a user can edit ASM program in editor window 1; then compile the program, if there are syntax errors, warning messages will be shown in output window 2; at last, download the program and interact with the microcontroller in terminal window 3 to debug and run the program.

In this course, MiniIDE is used to download codes into the MCU Evaluation Board (EVB). In this context, it acts as a terminal program.

You do not need to install this software on your PC.

PROGRAMMING FLOW CHART

The programming flow chart is shown in the figure below. First, the source code is written in Assembly language on the THRSim11 simulator. The simulator assembles the .asm code and generates a list file (*.LST). The simulator is then used to step through the program and debug it until it performs the intended functionality. All this can be done remotely, in the computer room, or on a personal computer. Once the program has been debugged, it can be taken on a floppy disk to the EMCH 367 lab (A 235). The MCU evaluation board (EVB) hardware is accessed through the MiniIDE emulator software installed on the lab computers. MiniIDE reads the .asm file from your floppy disk and transforms it into machine language executable code (*.S19). This code is downloaded to the MCU. After downloading the code into the MCU, you can make the MCU run your code using the MiniIDE interface screens. The MiniIDE also generates a list file (.LST) that can be used during debugging.

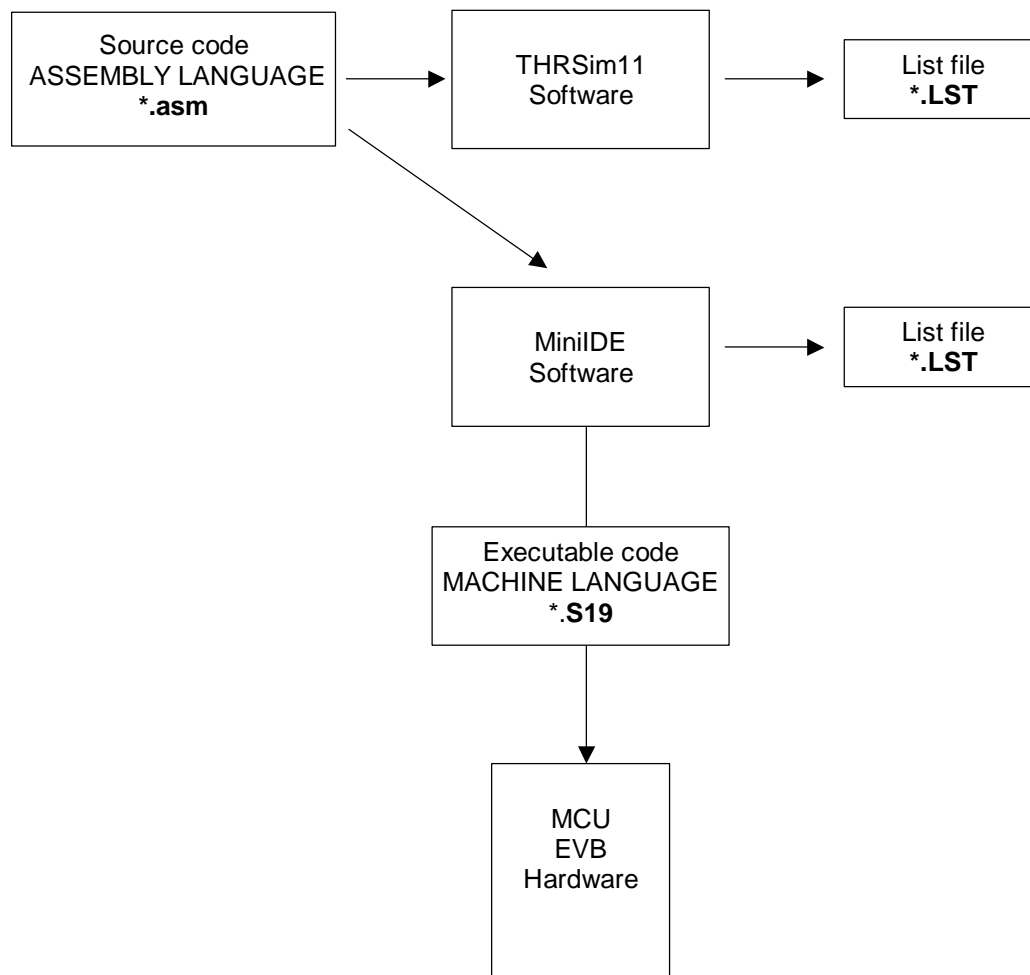


Figure 1 Flowchart of typical programming steps used in the EMCH 367 course.

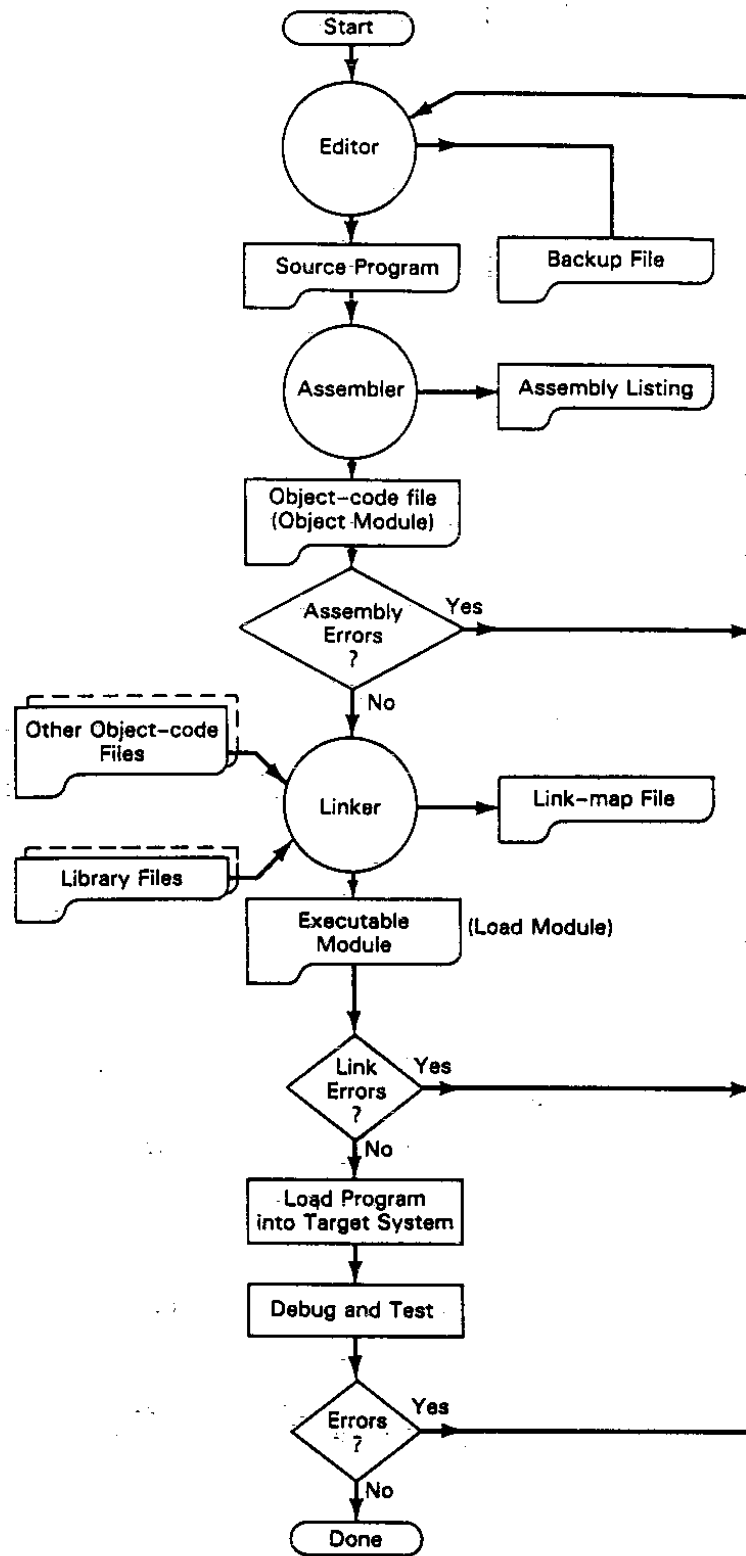


Figure 2 Flowchart of typical programming steps in a generic programming environment.

BINARY AND HEX NUMBERS

Note: To quickly grasp the use of binary and hex arithmetic, use your binary/hex pocket calculator and the website <http://homepage.ntlworld.com/interactive/BinaryAddition.html>

The binary number system is a base-2 numbering system. In binary representation, any value is represented using a combination of 1's and 0's. For example: $14_{10} = 1110_2$ in binary. The subscript 10 on the first number indicates that the number 14 is represented in the decimal (base 10) system. The subscript 2 on the second number indicates that 1110 is represented in the binary (base 2) system.

The binary representation is also called "digital". "Digit" also means finger, and you can imagine a numbering representation in which you use your 8 digits to for number containing 1's and 0's. The ability to represent numbers in terms of 1's and 0's is important because it is the easiest most unambiguous way to represent and communicate information. In a computer, a 1 is represented by a "high" voltage (5V) and a 0 by a "low" voltage (~0V). The binary system is the backbone of all digital computers and other high-tech applications.

THE BINARY SYSTEM

To understand how the binary system works, let's first examine how the conventional base-10 system works. The base-10, or decimal, system constructs numbers using increasing powers of 10. For example, the number 135_{10} is constructed using 3 powers of 10: 10^0 , 10^1 , and 10^2 . These numbers correspond to 1, 10, and 100. The number 135 is constructed as:

$$1 \times 100 + 3 \times 10 + 5 \times 1 \quad \text{or} \quad 1 \times 10^2 + 3 \times 10^1 + 5 \times 10^0$$

The equivalent of number 135_{10} in base two is 10000111_2 . This is constructed as:

$$1 \times 128 + 0 \times 64 + 0 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1$$

or

$$1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

It can be seen that the only significant difference between the two systems is the base number.

Each one or zero in the binary representation is called a "bit". A collection of eight bits is called a "byte" and, in a somewhat humorous note, a collection of four bits is called a "nibble". The bit associated with the highest power of two is called the Most Significant Bit (MSB); the bit associated with the lowest power of two is the Least Significant Bit (LSB).

1

bit

1	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

byte

(1 byte = 8 bits)

1	0	0	1
---	---	---	---

Hex number (nibble)

(1 nibble = 4 bits)

1	0	0	1	0	0	0	1
---	---	---	---	---	---	---	---

2 nibbles = 1 byte

DECIMAL TO BINARY CONVERSION:

Because most people are more comfortable using, and thinking in, the decimal system, it is important to know how to convert from the decimal to the binary system. This is most easily achieved through a series of divisions by two and by tracking the resulting remainders. Let's consider our example of 132_{10} :

$132 \div 2 =$	66	Remainder	0
$66 \div 2 =$	33	Remainder	0
$33 \div 2 =$	16	Remainder	1
$16 \div 2 =$	8	Remainder	0
$8 \div 2 =$	4	Remainder	0
$4 \div 2 =$	2	Remainder	0
$2 \div 2 =$	1	Remainder	0
$1 \div 2 =$	0	Remainder	1

$$132_{10} = 10000100$$

The remainder 1 resulting from the **last** division is the MSB, while the **first** remainder is the LSB of the conversion. From this example we see that the decimal number 132 is equal to the binary number 10000100.

The conversion from binary to decimal is done in the same manner as the first example, by adding together power of two values of the non-zero bits.

HEXADECIMAL (HEX) NUMBERS

As one might have already surmised, binary numbers quickly become long and hard to remember. For this reason, it is more convenient to convert the binary values into hexadecimal numbers (hex). Hexadecimal numbers are base 16 numbers. This requires six additional characters to represent the values 10, 11, 12, 13, 14, and 15. These values will be represented by the letters A, B, C, D, E, and F. The counting order in hex is: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. The reason hex notations are used is that it allows for a one-to-one correspondence between the 16-bit binary nibble and a single hexadecimal value. If the binary number is broken down into nibbles, and each nibble is replaced with the corresponding hexadecimal number, the conversion is complete. Consider 132_{10} . The binary number is 10000100. It can be broken down into two separate nibbles: 1000 and 0100. Convert each nibble into the corresponding hex value (8 and 4, respectively), and the hex equivalent of 132_{10} is 84_{16} . This is much more convenient to remember. For example, the hex number A23E3 is easily converted to 10100010001111100011 in binary without using any difficult calculations.

To convert decimal to hex numbers it is easiest to convert the decimal number to binary and then convert the binary to hex. In addition to these methods, there is a conversion chart in the back of the **Programming Reference Guide** for the conversion from decimal to hex.

BINARY ARITHMETIC

The rules for addition of binary numbers are straightforward:

$0 + 0 = 0$, $0 + 1 = 1$, and $1 + 1 = 0$ with a carry of 1, i.e. $1 + 1 = 10_2$.

For example:

$$\begin{array}{r}
 1001010010 \quad + \\
 \underline{0100110001} \\
 1110000011
 \end{array}
 \qquad
 \begin{array}{r}
 1010010100 \quad + \\
 0100010010 \\
 \underline{0001010001} \\
 1111110111
 \end{array}$$

NEGATIVE NUMBERS IN THE COMPUTER (2'S COMPLEMENT NUMBERS)

Until now, we have discussed only positive numbers. These numbers were called "unsigned 8-bit integers". In an 8-bit byte, we can represent a set of 256 positive numbers in the range 0_{10} - 255_{10} . However, in many operations it is necessary to also have negative numbers. For this purpose, we introduce "signed 8-bit integers". Since we are limited to 8-bit representation, we remain also limited to a total of 256 numbers. However, half of them will be negative (-128_{10} through -1_{10}) and half will be positive (0_{10} through 128_{10}).

The representation of signed (positive and negative) numbers in the computer is done through the so-called 8-bit 2's complement representation. In this representation, the 8th bit indicates the sign of the number ($0 = +$, $1 = -$).

The signed binary numbers must conform to the obvious laws of signed arithmetic. For example, in signed decimal arithmetic, $-3_{10} + 3_{10} = 0_{10}$. When performing signed binary arithmetic, the same cancellation law must be verified. This is assured when constructing the 2's complement negative binary numbers through the following rule:

To find the negative of a number in 8-bit 2's complement representation, simply subtract the number from zero, i.e. $-X = 0 - X$ using 8-bit binary arithmetic.

Example 1: Use the above rule to represent in 8-bit 2's complement the number -3_{10}

Solution: Subtract the 8-bit binary representation of 3_{10} from the 8-bit binary representation of 0_{10} using 8-bit arithmetic (8-bit arithmetic implies that you can liberally take from, or carry into the 9th bit, since only the first 8 bits count!).

BINARY		DECIMAL
00000000	-	0_{10} -
<u>00000011</u>		<u>3_{10}</u>
11111101		-3_{10}

Note that, in this operation, a 1 was liberally borrowed from the 9th bit and used in the subtraction!

Verification We have establish that $-3_{10} = 1111101_2$. Verify that $-3_{10} + 3_{10} = 0_{10}$ using 8-bit arithmetic.

BINARY		DECIMAL	
11111101	-	-3_{10}	-
<u>00000011</u>		<u>3_{10}</u>	
00000000		0_{10}	

Note that, in this operation, a carry of 1 was liberally lost in the 9th bit!

Example 2: Given the binary number 00110101, find it's 2's complement.

Solution: Subtract the number from 00000000, i.e.

BINARY	HEX	DECIMAL
00000000 -	00 -	0_{10} -
<u>01110101</u>	<u>75</u>	<u>106_{10}</u>
10001011	8B	-106_{10}

Verification: $01110101 + 10001011 = (1)00000000$. Since the 9th bit is irrelevant, the answer is actually 00000000, as expected

The rule outlined above can be applied to both binary and hex numbers.

Example 3: Given the hex number 6A, find its 8-bit 2's complement.

Solution: Subtract the number from 00_{16} using 8-bit arithmetic:

HEX		DECIMAL	
00	-	0_{10}	-
<u>6A</u>		<u>106_{10}</u>	
96		-106_{10}	

Verification: $6A_{16} + 96_{16} = (1)00$. Since the 9th binary bit is irrelevant, the answer is actually 00_{16} , as expected

Example 4: $11001010_2 \rightarrow CA_{16} \rightarrow 202_{10}$.

NUMERICAL CONVERSION CHART FOR UNSIGNED 8-BIT BINARY INTEGERS

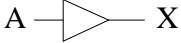
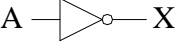
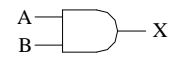
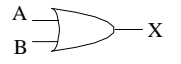
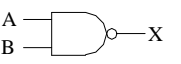
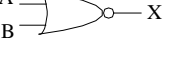
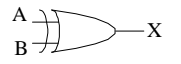
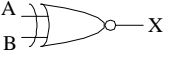
Decimal (base 10)	4-bit binary (base 2)	Hex (base 16)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

NUMERICAL CONVERSION CHART FOR 2'S COMPLEMENT SIGNED 8-BIT BINARY INTEGERS

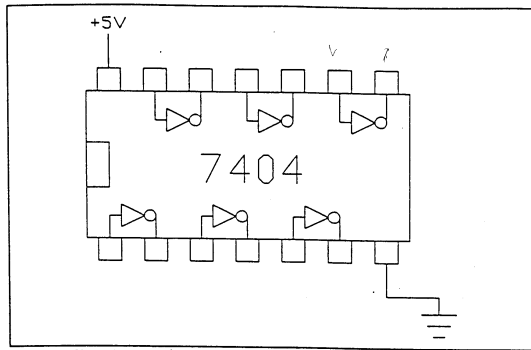
Decimal	8-bit 2's complement signed binary	Hex
+127	0111 1111	7F
...
+16	0001 0000	10
+15	0000 1111	0F
+14	0000 1110	0E
+13	0000 1101	0D
+12	0000 1100	0C
+11	0000 1011	0B
+10	0000 1010	0A
+9	0000 1001	09
+8	0000 1000	08
+7	0000 0111	07
+6	0000 0110	06
+5	0000 0101	05
+4	0000 0100	04
+3	0000 0011	03
+2	0000 0010	02
+1	0000 0001	01
0	0000 0000	00
-1	1111 1111	FF
-2	1111 1110	FE
-3	1111 1101	FD
-4	1111 1100	FC
-5	1111 1011	FB
-6	1111 1010	FA
-7	1111 1001	F9
-8	1111 1000	F8
-9	1111 0111	F7
-10	1111 0110	F6
-11	1111 0101	F5
-12	1111 0100	F4
-13	1111 0011	F3
-14	1111 0010	F2
-15	1111 0001	F1
-16	1111 0000	F0
...
-128	1000 0000	80

LOGIC GATES AND BOOLEAN ALGEBRA

LOGIC GATES

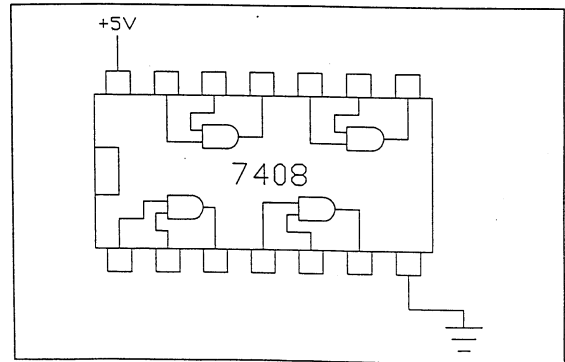
Circuit	IC #	Symbol	Boolean Function
Buffer	7407		$X = A$
NOT (Inverter)	7404		$X = \bar{A}$
AND	7408		$X = A \cdot B$
OR	7432		$X = A + B$
NAND	7400		$X = \overline{A \cdot B}$
NOR	7402		$X = \overline{A + B}$
Exclusive OR XOR	7486		$X = A\bar{B} + \bar{A}B$ $= A \oplus B$
Comparator			$X = A\bar{B} + \bar{A}B$ $A = B$

Inverting gate



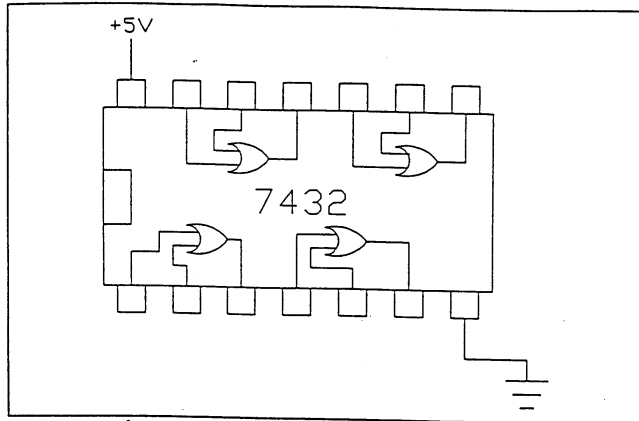
7404: Hex Inverting Gates

AND gate



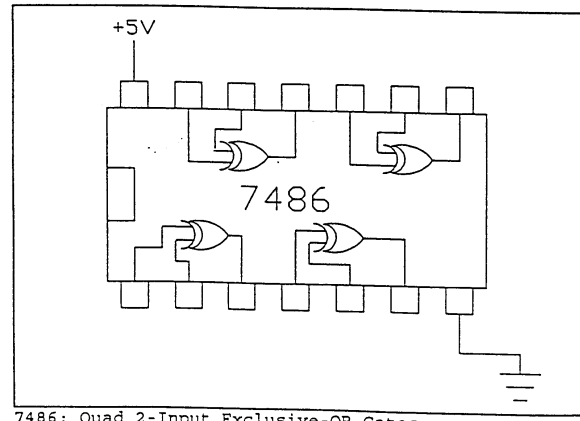
7408: Quad 2-Input AND Gates

OR gate



7432: Quad 2-Input OR Gates

XOR gate



7486: Quad 2-Input Exclusive-OR Gates

BOOLEAN ALGEBRA

In formulating mathematical expressions for logic circuits, it is important to have knowledge of Boolean algebra, which defines the rules for expressing and simplifying binary logic statements. The basic Boolean laws and identities are listed below. A bar over a symbol indicates the Boolean operation NOT, which corresponds to inversion of a signal.

Fundamental Laws

OR	AND	NOT
$A+0=A$	$A\cdot 0=0$	
$A+1=1$	$A\cdot 1=A$	
$A+A=A$	$A\cdot A=A$	
$A+\bar{A}=1$	$A\cdot\bar{A}=0$	$\bar{\bar{A}}=A$ (double inversion)

Commutative Laws

$$A+B=B+A \quad (2)$$

$$A\cdot B=B\cdot A$$

Associative Laws

$$(A+B)+C=A+(B+C) \quad (3)$$

$$(A\cdot B)\cdot C=A\cdot(B\cdot C)$$

Distributive Laws

$$A+(B\cdot C)=(A+B)\cdot(A+C) \quad (4)$$

Other Useful Identities

$$A+(A\cdot B)=A \quad (5)$$

$$A\cdot(A+B)=A \quad (6)$$

$$A+(\bar{A}\cdot B)=A+B \quad (7)$$

$$(A+B)\cdot(A+\bar{B})=A \quad (8)$$

$$(A+B)\cdot(A+C)=A+(B\cdot C) \quad (9)$$

$$A+B+(A\cdot\bar{B})=A+B \quad (10)$$

$$(A\cdot B)+(B\cdot C)+(\bar{B}\cdot C)=(A\cdot B)+C \quad (11)$$

$$(A\cdot B)+(A\cdot C)+(\bar{B}\cdot C)=(A\cdot B)+(\bar{B}\cdot C) \quad (12)$$

DeMorgan's Laws are also useful in rearranging of simplifying longer Boolean expressions or in converting between AND and OR gates:

$$\overline{A+B+C+\dots}=\bar{A}\cdot\bar{B}\cdot\bar{C}\cdot\dots \quad (13)$$

$$\overline{A\cdot B\cdot C\dots}=\bar{A}+\bar{B}+\bar{C}+\dots \quad (14)$$

If we invert both sides of these equations and apply the double NOT law from Equation (1) we can write DeMorgan's Laws in the following form:

$$A+B+C+\dots=\overline{\bar{A}\cdot\bar{B}\cdot\bar{C}\cdot\dots} \quad (15)$$

$$A\cdot B\cdot C\dots=\overline{\bar{A}+\bar{B}+\bar{C}+\dots} \quad (16)$$

This page is left intentionally blank

CONDITION CODE REGISTER (CCR)

S	X	H	I	N	Z	V	C
---	---	---	---	---	---	---	---

- S** = Stop bit
Allows user to turn the microcontroller stop function on or off.
- X** = XIRQ mask
Used to disable interrupts from the XIRQ.
- H** = Half carry bit
Indicates a carry from bit 3 during addition. Only updated by ABA, ADD, and ADC. It is used by the DAA in BCD operations (setting a hexadecimal number to decimal).
- I** = Interrupt mask
Global interrupt mask. Allow user to turn on/off interrupts.
- N** = Negative bit
Set to 1 when the result of an operation is 1 in the MSB.
Set to 0 when the result of an operation is 0 in the MSB.
- Z** = Zero bit
Set to 1 when the result of an operation is 00_{16} .
Set to 0 when the result of an operation is anything other than 00_{16} .
- V** = oVerflow bit
Set to 1 when a 2's complement overflow has occurred due to a specific operation.
 $7E_{16} + 04_{16} = 82_{16}$, 10000010_2
Note: The 1 in the MSB indicates that an overflow occurred. The addition yielded a number larger than $7F_{16}$, which is the maximum positive value that a 2'S compliment number is allowed.
- C** = Carry bit
Set to 1 when a carry or borrow has occurred in the MSB. In addition operations, it is set if there was a carry from MSB. In subtractions, it is set if a number with a larger absolute value is subtracted from a number with a smaller absolute value. It is also used in multiplication and division.

BUFFALO COMMANDS

The monitor BUFFALO program is the resident firmware for the EVB, which provides a self-contained operating environment. It interacts with the user through predefined commands. The BUFFALO command line format is as follows:

```
><command>[<parameters>](RETURN)
```

where:

```
>          EVB monitor prompt.
<command>  Command mnemonic.
<parameters> Expression or address.
(RETURN)   RETURN keyboard key
```

NOTES:

- 1) The command line format is defined using special characters that have the following syntactical meanings:

```
< >      Enclose syntactical variable
[ ]       Enclose optional fields
[ ]...    Enclose optional fields repeated
```

These characters are **NOT** entered by user, but are for definition purpose only.

- 2) Fields are separated by any number or space, comma, or tab characters.
- 3) All input numbers are interpreted as hexadecimal.
- 4) All input commands can be entered either upper or lower case lettering.
- 5) A maximum of 35 characters may be entered on a command line.
- 6) Command line errors may be corrected by backspacing or by aborting the command (CTRL-X/Delete).
- 7) After a command has been entered, pressing (RETURN) a 2nd time will repeat the command.

Some of the frequently used BUFFALO commands are listed alphabetically in Table 1.

COMMAND	DESCRIPTION
ASM [<address>]	Assembler/disassembler
BF <address1> <address2> <data>	Block fill memory with data
CALL [<address>]	Execute subroutine
G [<address>]	Execute program
HELP	Display monitor commands
MD [<address1> [<address2>]]	Memory Display
MM [<address>]	Memory Modify
MOVE <address1> <address2> [<destination>]	Move memory to new location
RM [p,y,x,a,b,c,s]	Register modify
T [<n>]	Trace \$1~\$ff instructions

Next few pages are detailed description and examples for each command.

ASM

Assembler/Disassembler

ASM [<address>]

where: <address> is the starting address for the assembler operation.

Assembler operation defaults to internal RAM if no address is given. Each source line is converted into the proper machine language code and is stored in memory overwriting previous data on a line-by-line basis at the time of entry.

The syntax rules for the assembler are as follows:

- (a.) All numerical values are assumed to be hexadecimal.
- (b.) Operands must be separated by one or more space or tab characters.

Addressing modes are designated as follows:

- (a.) Immediate addressing is designated by pre-ceding the address with a # sign.
- (b.) Indexed addressing is designated by a comma. The comma must be preceded a one byte relative offset and followed by an X or Y designating which index register to use (e.g., LDAA 00,X).
- (c.) Direct and extended addressing is specified by the length of the address operand (1 or 2 digits specifies direct, 3 or 4 digits specifies extended). Extended addressing can be forced by padding the address operand with leading zeros.
- (d.) Relative offsets for branch instructions are computed by the assembler. Therefore the valid operand for any branch instruction is the branch-if-true address, not the relative offset.

Assembler/disassembler subcommands are as follows.

- / Assemble the current line and then disassemble the same address location.
- ^ Assemble the current line and then disassemble the previous sequential address location.
- (RETURN) Assemble the current line and then disassemble the next opcode address.
- (CTRL)-J Assemble the current line. If there isn't a new line to assemble, then disassemble the next sequential address location. Otherwise, disassemble the next opcode address.
- (CTRL)-A Exit the assembler mode of operation.

EXAMPLE

```
>ASM C000
C000 STOP $FFFF
>LDAA #55
86 55
C002 STOP $FFFF
>STAA C0
97 C0
C004 STOP $FFFF
>LDS 0,X
AE 00
C006 STOP $FFFF
>BRA C500
Branch out of range
C006 STOP $FFFF
>BRA C030
20 28
C008 STOP $FFFF
>(CTRL)A
```

EXAMPLE

DESCRIPTION

```
Immediate mode addressing, requires #
before operand.

Direct mode addressing.

Index mode, if offset = 0 (,X) will not
be accepted.

Branch out of range message.

Branch offsets calculated automatically,
address required as conditional branch
operand.

Assembler operation terminated.
```

DESCRIPTION

```

>ASM C000          Enter assembler/disassembler mode.
C000 CLR $0800
>LDY #C200        First byte where data is stored.
                18 CE C2 00      IMM mode
C004 TEST
>LDX #C400        Point to data to be fetched.
                CE C4 00      IMM mode
C007 TEST
>LDAA 102E       Clear RDRF bit if set.
                B6 10 2E      EXT mode
C00A TEST
>LDAA 0,X        Get first data byte.
                A6 00        INX mode
C00C TEST
>STAA 102F       Store data in SCI data register.
                B7 10 2F      EXT mode
C00F INX
>LDAA 102E       Read SCI status register.
                B6 10 2E      EXT mode
C012 TEST
>ANDA #80        Send data byte.
                84 80        IMM mode
C014 TEST
>BEQ C00F        Wait for empty transmit data register.
                27 F9        REL mode
C016 BITB $80F6
>LDAA 102E       Read SCI status register.
                B6 10 2E      EXT mode
C019 BVS $C01B
>ANDA #20        Extract RDRF bit fram status register.
                84 20        IMM mode
C01B STX $00FF
>BEQ C016        Branch true = SCI RDR not full.
                27 F9        Branch false = SCL RDR full.
C010 STX $4065
>LDAA 102F       Read data from SCI RDR.
                B6 10 2F      EXT mode
C020 STAA $00,Y
>STAA 0,Y        Store data byte.
                18 A7 00      INY mode
C023 STX $00FF
>INX            Increment fetch pointer.
                08          INH mode
C024 TEST
>INY            Increment storage pointer.
                18 08      INH mode
C026 ASRB
>CPX #C41F      Done sending data?
                8C C4 1F      IMM mode
C029 ASLD
>BEQ C02E
                27 03
C02B STX SOOFF
>JMP C00C       No, get next data byte.
                7E C0 0C      EXT mode
C02E MUL
>BRA C02E       Yes, stop here.
                20 FE        REL mode
C030 ILLOP
>(CTRL)A       Exit assembler/dissembler mode.

```

BF

Block Fill**BF <address1> <address2> <data>**

where:

<address1>	Lower limit for fill operation.
<address2>	Upper limit for fill operation.
<data>	Fill pattern hexadecimal value.

EXAMPLE**DESCRIPTION**

<u>>BF C000 C030 FF</u>	Fill each byte of memory from C000 through C030 with data pattern FF.
<u>>BF C000 C000 0</u>	Set location C000 to 0.

CALL*Execute Subroutine***CALL [<address>]**

where: <address> is the starting address where user program subroutine execution begins.

EXAMPLE**DESCRIPTION**

<u>>CALL C000</u>	Execute program subroutine.
P-C000 Y-DEFE X-F4FF A-44 B-FE C-DO 5-004A	Displays status of registers at time RTS encountered (except P register contents).

G(GO)*Execute Program***G [<address>]**

where: <address> is the starting address where user program execution begins.

EXAMPLE**DESCRIPTION**

<u>>G C000</u>	Execute program subroutine.
P-C000 Y-DEFE X-F4FF A-44 B-FE C-DO 5-004A	Displays status of registers at time RTS encountered (except P register contents).

HELP*Help Screen*

HELP

Display monitor commands

MD

Memory Display

MD [<address1> <address2>]

Display a block of user memory beginning at address 1 and continuing to address 2.

EXAMPLE

```
>MD C000 C00F
```

```
C000 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

MM

Memory Modify

MM [<address>]

Examine/Modify contents in user memory at specified address in an interactive manner

EXAMPLE

```
>MM C700
C700 44 66(RETURN)
>MM C000
C000 55 80 C2 00 CE C4
```

DESCRIPTION

Display memory location C700.
Change data at C700
Examine location \$C000.
Examine next location(s) using (SPACE BAR).

MOVE

Block Move

MOVE <address1> <address2> [<dest>]

where: <address1> Memory starting address.
<address2> Memory ending address.
[<dest>] Destination starting address (optional).

Copy/move memory to new memory location. If the destination is not specified, the block of data residing from address1 to address2 will be moved up one byte.

EXAMPLE

```
>MOVE E000 E7FF C000
>MOVE C000 C0FF
```

DESCRIPTION

Move data from locations \$E000-\$E7FF to locations \$C000-\$C7FF.
Move data from locations \$C000-\$C0FF to locations \$C001-\$C100.

RM*Register Modify***RM [p,y,x,a,b,c,s]**

The RM command is used to modify the MCU program counter (P), Y index (Y), X index (X), A accumulator (A), B accumulator (B), Condition Code Register (C), and stack pointer (S) register contents.

EXAMPLE

```
>RM
P-C007 Y-7982 X-FF00 A-44 B-70 C-C0 S-0054
P-C007 C020
```

```
>RM X
P-C007 Y-7982 X-FF00 A-44 B-70 C-C0 S-0054
X-FF00 C020
```

DESCRIPTION

Display P register contents.

Modify P register contents.

Display X register contents.

Modify X register contents.

T*Trace***T[<n>]**

Where: <n> is the number (\$1~\$FF) of instructions to execute.

Monitor program execution on an instruction-by-instruction basis. Execution starts at the current program counter (PC).

EXAMPLE

```
>T
Op-86
P-C002 Y-DEFE X-FFFF A-44 B-00 C-00 S-0048
```

```
>T2
Op-B7
P-C005 Y-DEFE X-FFFF A-44 B-00 C-00 S-004B
Op-01
P-C006 Y-DEFE X-FFFF A-44 B-00 C-00 S-004B
```

DESCRIPTION

Single trace

Register contents after execution.

Multiple trace (2)

DEBUGGING TIPS

MICROCONTROLLER PROBLEMS

- Is the processor plugged into the PC serial port?
- Is the processor plugged into the power supply?
- Is the power supply turned on?
- Is the serial port plugged into the correct connector?

HARDWARE PROBLEMS

- Does the component have power? - Check all voltages
- Are the chips oriented correctly - notch in the correct direction?
- Do the chips straddle the gap in the center of the board?
- Make sure all chips have power (not just input & output lines).
- Verify the direction of diodes and electrolytic capacitors.
- Verify the power at intermediate locations - use 5 or 0 volts from the supply instead of chip input to check various conditions.
- Verify that the PC ports are giving the expected output signals.
- Verify chip and transistor pins with the pin diagrams.
- Are there any "open" lines, no voltage connection instead of zero volts?
- Verify resistor codes and capacitor values.

SOFTWARE PROBLEMS

- Is the correct program currently in memory?
- Is the correct starting location being used (G ????)
- Verify the program with ASM.
- Use trace (T) to step through and verify branches, jumps and data.
- Compare memory locations with expected information after the program stops.
- Insert SWI at a key location to allow verification of branch, memory and accumulator values.
- Do branches and jumps have the correct offsets?
- Have RET and RTI commands been reversed somewhere?
- For serial communications, has TE or RE been set?
- For serial communications, has TDRE or RDRF been reset?
- For parallel port C, has 1007 been set for input or output?
- Has the interrupt mask been cleared (CLI)?
- Has the stack pointer changed substantially?

Use the BUFFALO commands to do step-by-step (Trace, T) and Break-Point (BR) execution of the program. Press F1 for details of the BUFFALO commands.

REGISTERS INFORMATION

REGISTER AND CONTROL BIT SUMMARY

	Bit 7	6	5	4	3	2	1	Bit 0
\$1000	Bit 7	-	-	-	-	-	-	Bit 0
\$1001								
\$1002	STAF	STAI	CWOM	HNDS	OIN	PLS	EGA	INVB
\$1003	Bit 7	-	-	-	-	-	-	Bit 0
\$1004	Bit 7	-	-	-	-	-	-	Bit 0
\$1005	Bit 7	-	-	-	-	-	-	Bit 0
\$1006								
\$1007	Bit 7	-	-	-	-	-	-	Bit 0
\$1008			Bit 5	-	-	-	-	Bit 0
\$1009			Bit 5	-	-	-	-	Bit 0
\$100A	Bit 7	-	-	-	-	-	-	Bit 0
\$100B	FOC1	FOC2	FOC3	FOC4	FOC5			
\$100C	OC1M7	OC1M6	OC1M5	OC1M4	OC1M3			
\$100D	OC1D7	OC1D6	OC1D5	OC1D4	OC1D3			
\$100E	Bit 15	-	-	-	-	-	-	Bit 8
\$100F	Bit 7	-	-	-	-	-	-	Bit 0
\$1010	Bit 15	-	-	-	-	-	-	Bit 8
\$1011	Bit 7	-	-	-	-	-	-	Bit 0
\$1012	Bit 15	-	-	-	-	-	-	Bit 8
\$1013	Bit 7	-	-	-	-	-	-	Bit 0
\$1014	Bit 15	-	-	-	-	-	-	Bit 8
\$1015	Bit 7	-	-	-	-	-	-	Bit 0
\$1016	Bit 15	-	-	-	-	-	-	Bit 8
\$1017	Bit 7	-	-	-	-	-	-	Bit 0
\$1018	Bit 15	-	-	-	-	-	-	Bit 8
\$1019	Bit 7	-	-	-	-	-	-	Bit 0
\$101A	Bit 15	-	-	-	-	-	-	Bit 8
\$101B	Bit 7	-	-	-	-	-	-	Bit 0
\$101C	Bit 15	-	-	-	-	-	-	Bit 8
\$101D	Bit 7	-	-	-	-	-	-	Bit 0
\$101E	Bit 15	-	-	-	-	-	-	Bit 8
\$101F	Bit 7	-	-	-	-	-	-	Bit 0

REGISTER AND CONTROL BIT ASSIGNMENTS

	Bit 7	6	5	4	3	2	1	Bit 0	
\$1020	OM2	OL2	OM3	OL3	OM4	OL4	OM5	OL5	TCTL1
\$1021			EDG1B	EDG1A	EDG2B	EDG2A	EDG3B	EDG3A	TCTL2
\$1022	OC1I	OC2I	OC3I	OC4I	OC5I	IC1I	IC2I	IC3I	TMSK1
\$1023	OC1F	OC2F	OC3F	OC4F	OC5F	IC1F	IC2F	IC3F	TFLG1
\$1024	TOI	RTI	PAOVI	PAI			PR1	PRO	TMSK2
\$1025	TOF	RTIF	PAOVF	PAIF					TFLG2
\$1026	DDRA7	PAEN	PAMOD	PEDGE			RTR1	RTR0	PACTL
\$1027	Bit 7	-	-	-	-	-	-	Bit 0	PACNT
\$1028	SPIE	SPE	DWOM	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
\$1029	SPIF	WCOL		MODF					SPSR
\$102A	Bit 7	-	-	-	-	-	-	Bit 0	SPDR
\$102B	TCLR		SCP1	SCP0	RCKB	SCR2	SCR1	SCR0	BAUD
\$102C	R8	T8		M	WAKE				SCCR1
\$102D	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCCR2
\$102E	TORE	TC	RDRF	IDLE	OR	NF	FE		SCSR
\$102F	Bit 7	-	-	-	-	-	-	Bit 0	SCDR
\$1030	CCF		SCAN	MULT	CD	CC	CB	CA	ADCTL
\$1031	Bit 7	-	-	-	-	-	-	Bit 0	ADR1
\$1032	Bit 7	-	-	-	-	-	-	Bit 0	ADR2
\$1033	Bit 7	-	-	-	-	-	-	Bit 0	ADR3
\$1034	Bit 7	-	-	-	-	-	-	Bit 0	ADR4
\$1035									Reserved
\$1036									Reserved
\$1037									Reserved
\$1038									Reserved
\$1039	ADPU	CSEL	IRQE	DLY	CME		CR1	CR0	OPTION
\$103A	Bit 7	-	-	-	-	-	-	Bit 0	COPRS
\$103B	ODD	EVEN		BYTE	ROW	ERASE	EELAT	EPPGM	PPROG
\$103C	RBOOT	SMOD	MDA	IRV	PSEL3	PSEL2	PSEL1	PSEL0	HPRIO
\$103D	RAM3	RAM2	RAM1	RAM0	REG3	REG2	REG1	REG0	INIT
\$103E	TILOP		OCCR	CBYP	DISR	FCM	FCOP	TCON	TEST1
\$103F					NOSEC	NOCOP	PROMON	EEON	CONFIG

PARALLEL PORTS

(Section 7 of the M68HC11 Reference Manual)

Parallel communication is communication that occurs simultaneously on many lines -- thus the word, parallel. It is used most often when the communicating devices are local to one another. For the MC6811, there are two parallel ports to which the user has direct access: Port B and Port C. Since MC6811 is an 8-bit microcontroller, each of these parallel ports has 8 bits. That is, each of the parallel ports has eight separate wires coming out of the microcontroller, one wire for each bit of data.

The two parallel ports are configured differently. Parallel Port B is restricted to output- only applications. Parallel Port C can be used for either input or output. Moreover, in Parallel Port C, not all bits have to be the same type of communication. For example, the first four bits of Parallel Port C (PC0 - PC3) can be set to read input, while the last four bits of Parallel Port C (PC4 - PC7) can be set to send output information.

To use these parallel ports, a program must load and store specific numbers to special memory locations. These memory locations are referred to as control registers. There are three different control registers, which are related to Parallel Port operation, one related to Parallel Port B, and two related to Parallel Port C.

As Parallel Port B is output only, there is only one thing, which needs to be specified: the output data. This will be a signal of either 5V or 0V for **each line** in the Parallel Port. A 0 corresponds to 0V; a 1 corresponds to 5V. To send desired data out Parallel Port B, store the two-digit hexadecimal number corresponding to the eight bits of data that you wish to output into memory location \$1004. **This one action specifies the output voltage on the eight separate output lines.**

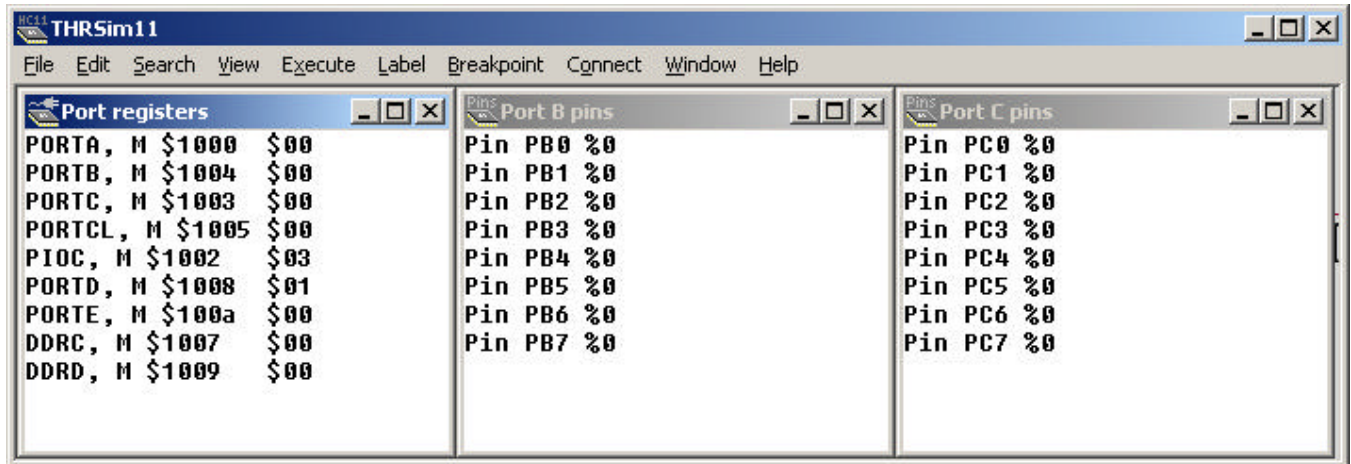
For Parallel Port C, two aspects of parallel communication must be specified. These are the data direction for each pin (whether a pin is input or output) and the actual data for each pin. The data direction for each pin is specified by storing a two-digit hexadecimal number corresponding to the data direction of each individual pin into memory location \$1007. A 0 corresponds to input; a 1 corresponds to output. The specific data for Parallel Port C is in memory location \$1003. If the pin is output, then the value in that bit location indicates the voltage currently sent out that pin. The behavior of Parallel Port C in output is the same as Parallel Port B. Changing the value of the bit changes the value of the output voltage. If the pin is input, the value in that bit location indicates the voltage currently being measured on that pin. Writing to an input pin has no effect.

DO NOT SEND AN INPUT SIGNAL INTO A PIN SPECIFIED FOR OUTPUT!!! THAT WILL FRY THE CHIP!!!

THRSim11 SIMULATION OF PARALLEL COMMUNICATION

The specific windows that need to be open during the THRSim11 simulation of parallel communication are:

- Port registers
- Port B pins
- Port C pins



THE THRSIM11 IO BOX

The THRSim11 IO box is used, among others, to perform the simulation of Port B and Port C functions.

Port B, which is only an output port, is simulated as the eight LEDs PB0, PB1, ..., PB7. When a logical 1 signal is sent to a Port B pin, PBx, the corresponding LED lights up (becomes red).

Port C pins (PC0, PC1, ..., PC7) can be selected as either input or output using the DDRC register bits in your program. When selected as input (DDRCx = 0, x = 0, 1, ..., 7), the switches are used to send signals into the MCU along the PCx line. When selected as output (DDRCx = 1), the switches flip up and down according to the value on that PCx line. (up = 1, down = 0)

The THRSim11 IO box.

This box contains:

- Switches connected to the 68HC11 pins on PC0-PC7.
- LED's connected to the 68HC11 pins on PB0-PB7.
- LED's connected to the 68HC11 pins on PD0-PD5.
- STRA switch + LED connected to the 68HC11 pin STROBE A.
- 4 x 20 rows LCD display. (When controlling this display use the ['simlcd.inc'](#) include file in your program.)

A switch can be toggled by clicking on it with the mouse or by typing 0 to 7 for PC0 to PC7 or A for STRA.

The switches are normally used when port C is configured as input port (DDRCx = 0). When port C is configured as output port (DDRCx = 1) the switches represent the value written to port C. Real switches will not follow this behavior but will cause a short circuit :-).

ASCII AND BCD CODES

Ascii: 1 character / byte
(7 bits)

MSD LSD

		ASCII CHARACTER SET (7-Bit Code)							
		000	001	010	011	100	101	110	111
MS Dig.	LS Dig.	0	1	2	3	4	5	6	7
0000	0	NUL	DLE	SP	0	@	P	'	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	\$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(8	H	X	h	x
1001	9	HT	EM)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[k	{
1100	C	FF	FS	,	<	L	\	l	
1101	D	CR	GS	-	=	M]	m	~
1110	E	SO	RS	.	>	N	^	n	~
1111	F	SI	US	/	?	O	_	o	DEL

Decimal	BCD			
	8s	4s	2s	1s
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

BCD: 2 ch / byte
(4 bits)

Decimal 9 1
 ↓ ↓
BCD 1001 0001
 Decimal-to-BCD conversion

BCD 0111 0010
 ↓ ↓
Decimal 7 2
 BCD-to-decimal conversion

SERIAL COMMUNICATIONS

(Section 9 of the M68HC11 Reference Manual)

Serial communications are used when one bit is sent at a time. All the data is transferred on one line; the bits are transferred sequentially, making serial communication much slower than parallel communication. The data is specified by holding each bit at a certain voltage for a certain period of time. The data is usually sent in character format using the 7-bit ASCII (American Standard Code for Information Interchange) code. It specifies a 7 bit binary code for commonly used characters. To put the 7-bit ASCII into an 8-bit byte, one fills the 8th bit with 0.

The data byte being sent is bracketed by two bits, the start bit (0V) and the stop bit (5V). An idle line has a voltage of 5V. Each data byte is prefixed by a 0V start bit. The data bits are then sent from the least significant bit to the most significant bit. At the end, a 5V stop bit is added. All bits are held for the same amount of time. The time is specified by the BAUD rate (bits/sec).

MC6811 has the capacity to receive and transmit data through the serial communication interface. The selection of receive and/or transmit modes is done by setting to 1 the RE and TE bits in the Serial Communication Control Register #2 (SCCR2) (memory location \$102D, bits 2 and 3). Simultaneous selection of both receive and transmit modes is permitted, since MC6811 has separate lines for reception and transmission (RxD and TxD through port D pins PD0 and PD1, respectively).

In the receive mode, the Receive-Data-Register-Full (RDRF) indicates when serial communications data has been received (RDRF=1). RDRF is bit 5 of the Serial Communication Status Register (SCSR) at memory location \$102E. When serial communications data is received, it gets placed in the Serial-Communication-Data-Register (SCDR) (memory location \$102F). As a user, you would normally check RDRF until found equal to 1, then load the data from SCDR into an accumulator. This sequence of reading RDRF=1 and loading data from SCDR will trigger the clearing of RDRF (i.e., will make RDRF=0). For this reason, it is called "clearing sequence". In this way, MC6811 becomes ready for the reception of the next serial communication data.

Transmission of data from MC6811 also uses the Serial-Communication-Data-Register (SCDR). Before placing new data in SCDR for transmission, one must first make sure that SCDR is empty, i.e., it has finished transmitting previous data. This verification is done by checking the value of Transmit Data Register Empty (TDRE) bit (memory location \$102E, bit 7). If TDRE = 0, then MC6811 is still transmitting data through the serial communication interface. If TDRE = 1, then transmission has finished, and the data register is empty and ready to receive new data for transmission. When data is stored into SCDR for transmission, MC6811 automatically adds the start and stop bits to the data, sends the data out through the serial communication interface, and, after transmission is complete, makes TDRE=1. The clearing sequence for TDRE consists in reading TDRE=1 followed by storing of data into SCDR. Subsequently, MC6811 starts serial communication transmission of the data placed in SCDR.

Interrogating the value of specific bits in SCSR (RDRF, TDRE, etc.) can be done in a number of ways. One way could be to AND the contents of SCSR with the appropriate mask and use a BEQ instruction to loop back if the result is zero (i.e., if the interrogated bit is not yet set). For RDRF (bit 5), the mask is #20. For TDRE (bit 7), the mask is #80. However, there are also other ways of branching in correlation with the status of specific bits (e.g., instructions BRCLR, BRSET, etc.). Feel free to experiment!

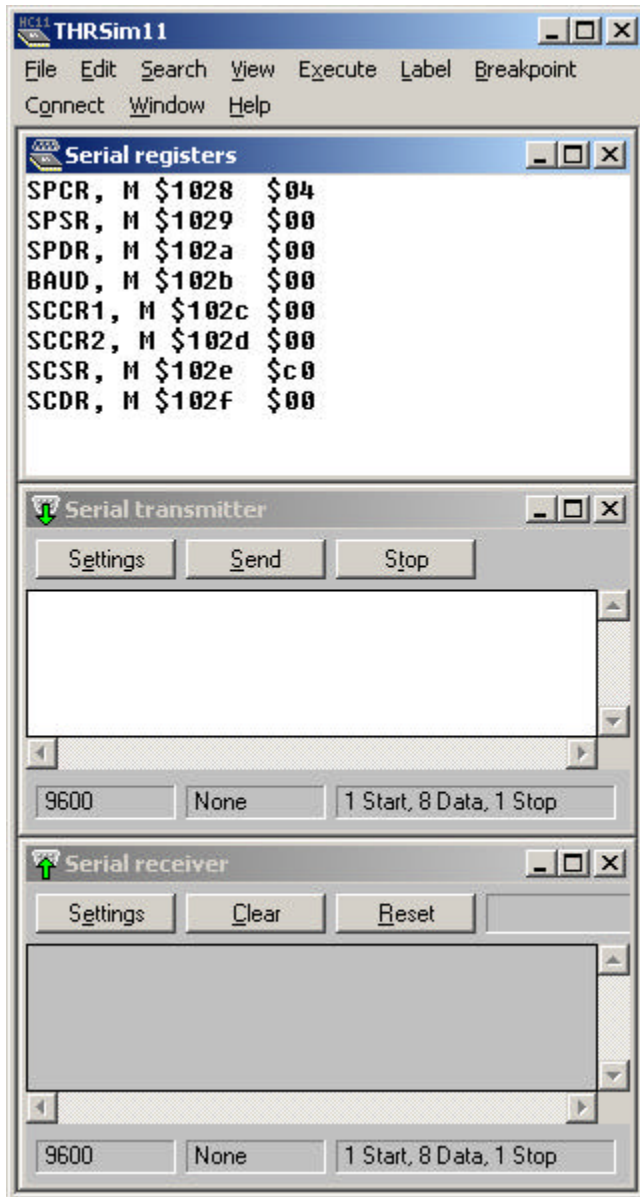
Serial communication is critical to the operation of modern computers. This is how keyboards communicate with the computer, and how you will control your programs during labs and project.

NOTE: Please, see Section 9 of the M68HC11 Reference Manual for more detailed information on serial communication.

THRSIM11 SIMULATION OF SERIAL COMMUNICATION

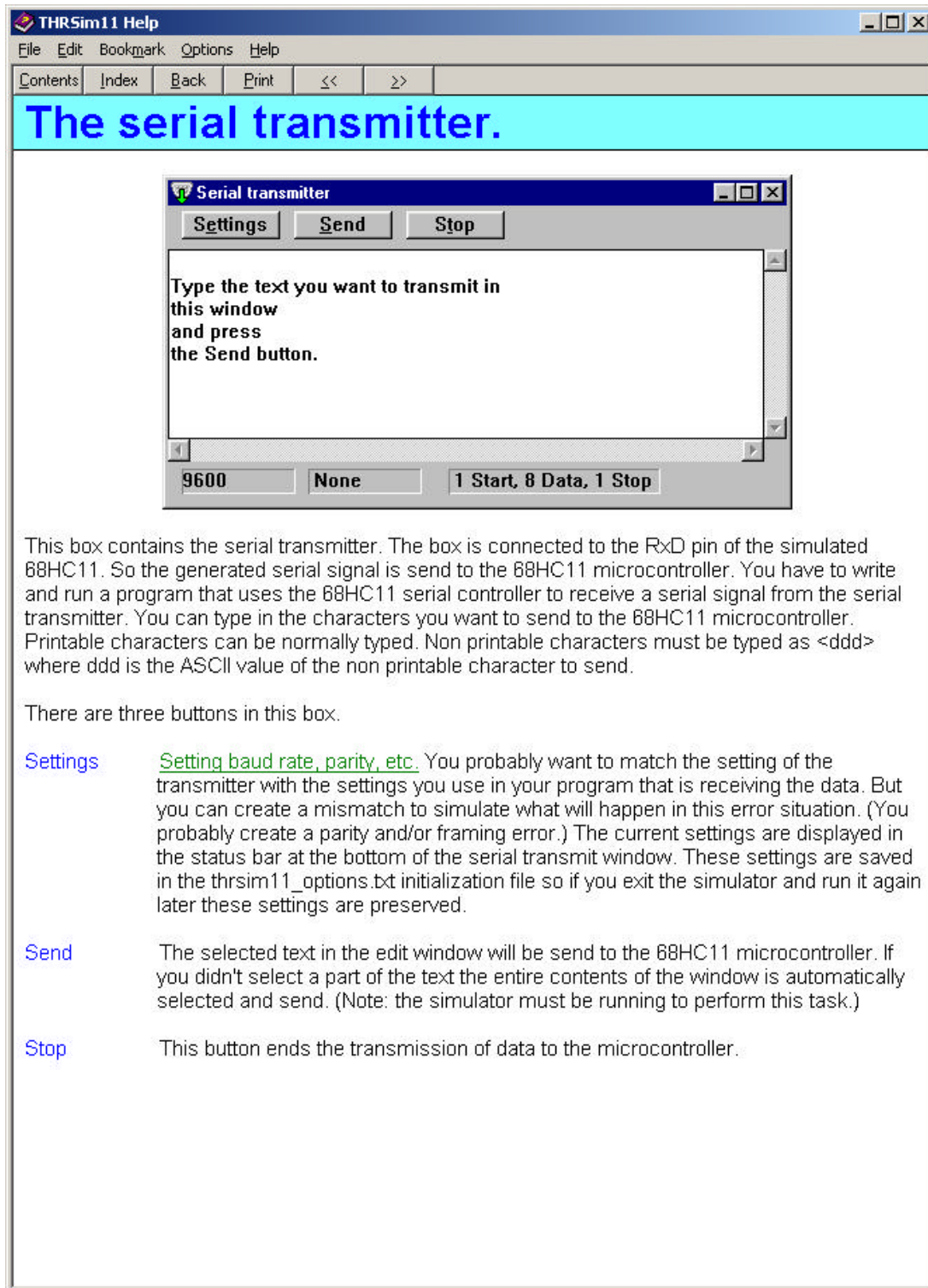
The specific windows that need to be open during the THRSim11 simulation of serial communication are:

- Serial registers
- Serial transmitter
- Serial receiver



THE THRSIM11 SERIAL TRANSMITTER

The THRSim11 serial transmitter simulates the PC keyboard in the lab. It sends characters to the MCU. During simulation, with your program running, type a character in the transmitter and press the Send button. The MCU should receive it and react according to your instructions.



This box contains the serial transmitter. The box is connected to the RxD pin of the simulated 68HC11. So the generated serial signal is send to the 68HC11 microcontroller. You have to write and run a program that uses the 68HC11 serial controller to receive a serial signal from the serial transmitter. You can type in the characters you want to send to the 68HC11 microcontroller. Printable characters can be normally typed. Non printable characters must be typed as <ddd> where ddd is the ASCII value of the non printable character to send.

There are three buttons in this box.

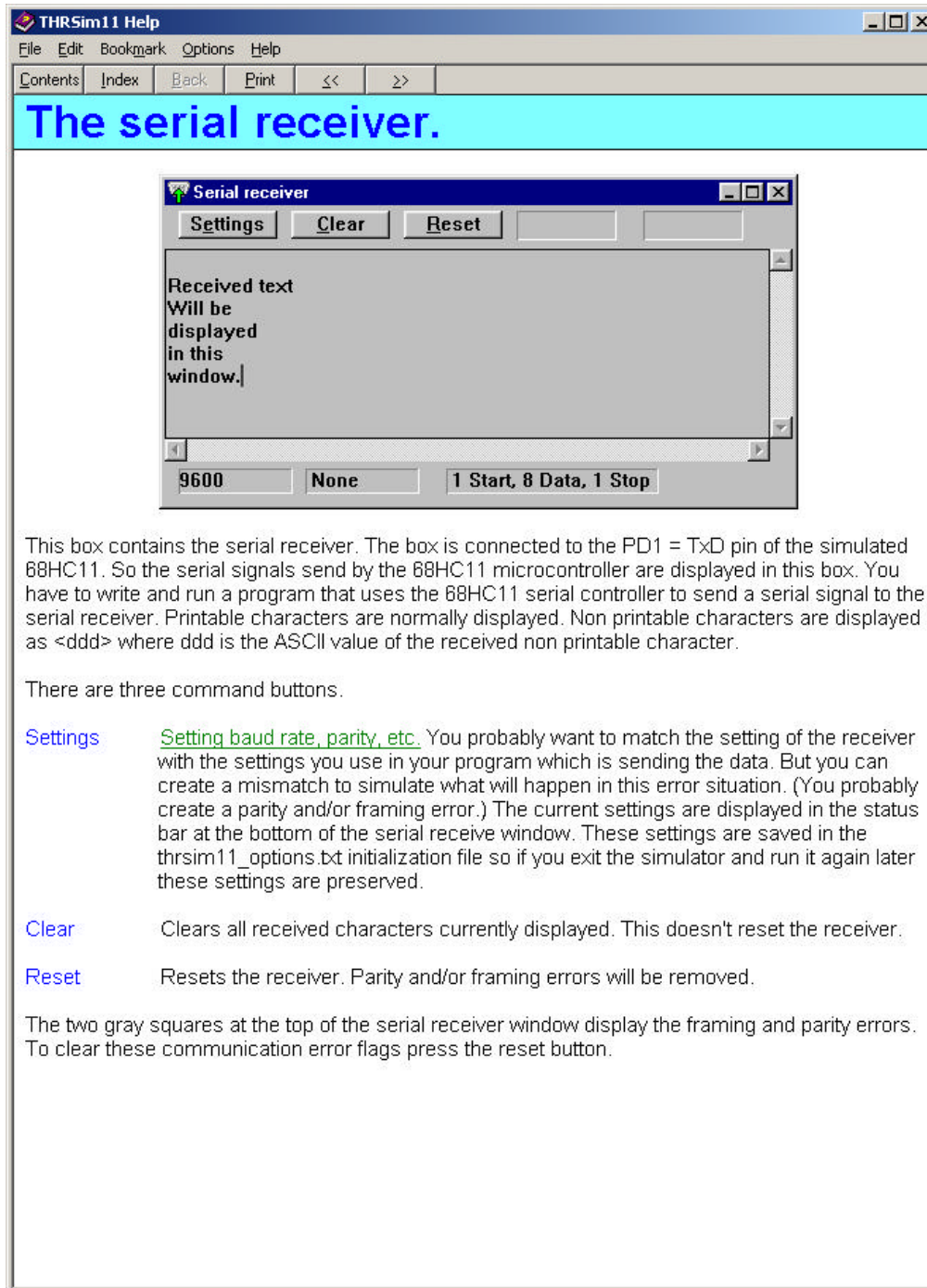
Settings [Setting baud rate, parity, etc.](#) You probably want to match the setting of the transmitter with the settings you use in your program that is receiving the data. But you can create a mismatch to simulate what will happen in this error situation. (You probably create a parity and/or framing error.) The current settings are displayed in the status bar at the bottom of the serial transmit window. These settings are saved in the thrsim11_options.txt initialization file so if you exit the simulator and run it again later these settings are preserved.

Send The selected text in the edit window will be send to the 68HC11 microcontroller. If you didn't select a part of the text the entire contents of the window is automatically selected and send. (Note: the simulator must be running to perform this task.)

Stop This button ends the transmission of data to the microcontroller.

THE THRSIM11 SERIAL RECEIVER

The THRSim11 serial receiver acts like the PC monitor in the lab. It receives signals sent by the MCU. With your program running, and the serial receiver window open, you should see a character displayed in the receiver window every time the MCU transmits a character while executing your program.



This box contains the serial receiver. The box is connected to the PD1 = TxD pin of the simulated 68HC11. So the serial signals send by the 68HC11 microcontroller are displayed in this box. You have to write and run a program that uses the 68HC11 serial controller to send a serial signal to the serial receiver. Printable characters are normally displayed. Non printable characters are displayed as <ddd> where ddd is the ASCII value of the received non printable character.

There are three command buttons.

Settings [Setting baud rate, parity, etc.](#) You probably want to match the setting of the receiver with the settings you use in your program which is sending the data. But you can create a mismatch to simulate what will happen in this error situation. (You probably create a parity and/or framing error.) The current settings are displayed in the status bar at the bottom of the serial receive window. These settings are saved in the thrsim11_options.txt initialization file so if you exit the simulator and run it again later these settings are preserved.

Clear Clears all received characters currently displayed. This doesn't reset the receiver.

Reset Resets the receiver. Parity and/or framing errors will be removed.

The two gray squares at the top of the serial receiver window display the framing and parity errors. To clear these communication error flags press the reset button.

TIMER FUNCTIONS

(Section 10 of the M68HC11 Reference Manual)

Timer functions allow the microcontroller to determine "time" by counting the number of machine cycles between events. The timer is based on the Timer Counter register (TCNT, \$100E - \$100F). The timer counter register increments once every machine cycle. Once the timer counter register reaches #FFFF, the next machine cycle causes the register to "overflow" (go from #FFFF to #0000). To let the user know that this has happened, the microcontroller sets a flag, TOF, the Timer Overflow Flag (bit 7 of \$1025). 1 implies that there has been a timer overflow; 0 implies that there has not been a timer overflow. To use TOF as a counting tool, you must clear TOF. Here, clearing TOF is obtain by writing a 1 to it (unusual, but true for all timer flags: see Section 10.2.4 on page 10-14 in the Reference Manual). **When clearing a flag, it is important that you do not interfere with the other bits in the register!**

The timer is also linked to external lines, allowing the microcontroller to record the value of the timer counter when an input voltage changes. These functions are called input capture functions. They detect a signal transition. At the time that the signal transition is detected, the input capture function automatically records the value in the timer counter in a separate memory location and sets a flag, ICxF, to let the user know that there has been an input capture. Value 1 implies that there has been an input capture; 0 implies that there has not. Each flag is cleared by writing a 1 to the flag in the control registers. The type of signal transition that causes an input capture is determined by the edge bits, EDGxB and EDGxA. Because these two bits act together, there are four different modes for each input capture: disabled; low-to-high detection; high-to-low detection; and both low-to-high and high-to-low detection. MC6811 has three individual input captures. All act in the same way, with separate memory locations, EDG bits, and ICF's.

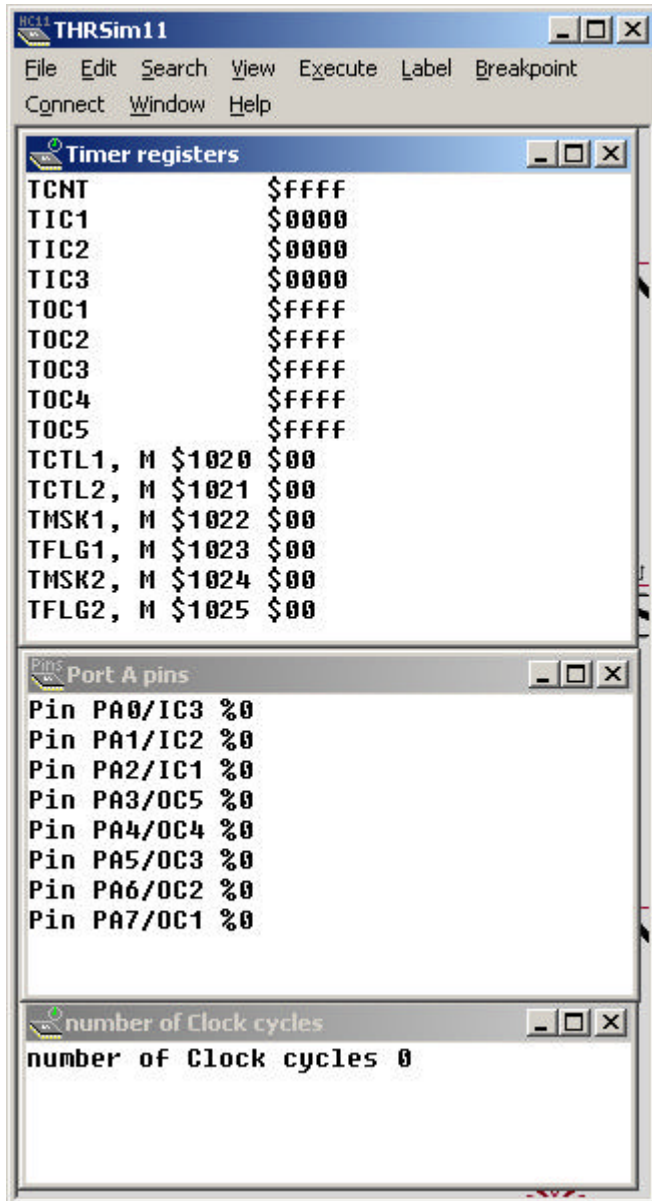
Another timer function is the output compare function. When the value in the timer counter register reaches the value in the output compare register, the microcontroller sends a signal out on the selected pin. In essence, the microcontroller schedules when to send the signal out. There are four commonly used output compares on the MC6811. They are OC2, OC3, OC4, and OC5. As the timer is a two byte register, each of the output compare registers is a two-byte register. To set a value for output compare, simply store the two-byte number to the output compare registers. Once the timer counter reaches the value in a timer output-compare register, an OCxF (output compare flag) is set to let the user know that an output compare has occurred. 1 indicates that output compare has occurred; 0 indicates that output compare has not occurred. To clear an output compare flag, write a 1 to OCxF. The signal sent out of the microcontroller on output compare is controlled by two bits acting together, the OMx and OLx bits. The four available options are: (i) disabled; (ii) send out 0V; (iii) send out 5V; and (iv) toggle the output voltage. Each of the timer output-compare functions has output compare registers, OM and OL bits, and output compare flags in the control registers.

The timer counts and measures events in terms of machine cycles. In Lab 3, you measure the clock speed of the microcontroller. In essence, you calculate a conversion factor between machine cycles and real time. Using the timer functions of the microcontroller and the conversion factor that you derive, you can use the microcontroller for data acquisition involving time measurement.

THRSim11 SIMULATION OF TIMER FUNCTIONS

The specific windows that need to be open during the THRSim11 timer functions simulation are:

- Timer registers
- Port A pins
- Number of Clock cycles



ANALOG-TO-DIGITAL CONVERSION

(Section 12 of the M68HC11 Reference Manual)

An analog-to-digital converter (A/D) takes an analog voltage, such as those produced by many electronic measuring devices, and converts it to a digital value. MC6811 has an 8-bit analog-to-digital converter. The range of measurement is from 0V to 5V. This allows the microcontroller to interface with such devices as potentiometers, cermets, thermocouples, LVDT's, etc. MC6811 has many different ways that analog-to-digital conversions can be made, as there are 8 separate lines (or channels) that the A/D can utilize. All of the options are controlled by one control register, ADCTL, in \$1030. The results of the A/D conversions are stored in four separate memory locations, \$1031, \$1032, \$1033, and \$1034 -- ADR1, ADR2, ADR3, and ADR4, respectively.

There are two different modes that MC6811 can use to take data. These are determined by the value of SCAN, bit 5 in \$1030. If SCAN = 1, then the microcontroller continuously scans for data along the A/D lines. Every time a new measurement is made, the data is stored in the appropriate memory location. If SCAN = 0, then four conversions are made, one on each specified line. The results of these four conversions are stored in the specified memory locations. As soon as all four conversions are completed, the A/D stops making conversions.

The lines specified to take data are determined by bits CD - CA, bits 3 - 0 in \$1030. The meanings of these bits are specified by MULT, bit 4 in \$1030. If MULT = 0, then four consecutive conversions are performed on the same data line. The results of the conversions are stored in ADR1 - ADR4. CD - CA specify the single line for all four conversions. Table 12 - 1 shows the values of CD - CA for each input line. If MULT = 1, then one conversion is made on each of four separate lines. The results are stored in ADR1 - ADR4. Only CD and CC have any effect in determining which four lines take the data. The four lines and the location of the A/D data are shown in Table 12 - 1.

To start the A/D conversions, write the value to \$1030 that configures SCAN, MULT, CD, CC, CB, and CA for the desired data acquisition. This action automatically clears the Conversion Complete Flag, CCF in ADCTL (bit 7 of \$1030). CCF is set when four A/D conversions are completed. If SCAN = 1, CCF is set after the first four conversions are completed and remains set until a subsequent write to ADCTL (\$1030). There is no interrupt for CCF. As such, polling operations must be used to monitor CCF. Once the microcontroller has completed the conversions, CCF is set. The data in ADR1 - ADR4 represents valid conversion values. It takes 128 machine cycles to make four eight-bit conversions. At 2 MHz, this is an impressive data acquisition rate.

There are many types of A/D conversion techniques. MC6811 uses a successive approximation technique. Some other types of A/Ds are the counter, integrative and flash A/Ds.

THRSim11 SIMULATION OF ANALOG TO DIGITAL CONVERSION

The specific windows that need to be open during the THRSim11 simulation of analog to digital conversion are:

- AD converter registers
- Sliders E port

