**About**

6800IDE is a freeware windows based IDE for Motorola's 6800/6811 processor. Designed for educational purposes, it includes an assembler and an emulator for the 6800/6811 with builtin debugging support such as user breakpoints, execution trace, internal register display and a Hex/Bin/Dec number convertor.

The 6800/6811 family of processors fueled the early home computing explosion and its derivatives were the processors of choice for many personal computers including Apple, Commodore64, Nintendo etc, and numerous gaming consoles. Its direct decendents are still widely used today as embedded processors.

The 6800/6811 is an 8 bit, dual accumulator processor with flexible memory addressing modes. Compared to other processors, it's compact, and highly orthogonal instruction set makes it easier to program.

**Screen Shots**

**Development / Debug Mode**

**Output Display Mode**

**Programming Reference**

**Specifications**

Memory Mapped IO Support

* Data Input Ports
  + Keyboard data at port $FFF0,1
  + Mouse cursor X coordinate at port $FFF2
  + Mouse cursor Y coordinate at port $FFF3
* Data Output
  + Display Buffer at $FB00 to $FF38
  + 20 lines of 54 characters

Interrupt Vectors

* \_RST @ 0xFFFE,F
* NMI @ 0xFFFC,D
* SWI @ 0xFFFA,B
* IRQ @ 0xFFF8,9

Execution Trace Support:

* Current Program line Highlight
* Current Memory Location Highlight

User Break point support:

* Disabled
* Program Line No.
* PC Address value
* SP Address value
* X Register value
* Accumulator A value
* Accumulator B value
* Memory location changed value

Number Base Conversion:

* Hexadecimal <=> Decimal
* Hexadecimal <=> Binary
* Decimal <=> Binary

**Introductory Tutorial**

[6800.pdf](http://www.hvrsoftware.com/6800.pdf) [Adobe Acrobat PDF - 604.55 KB]

**Programming Reference**

Assembly statements contain the following fields:  [Label] Operation [operand] [comment]

Label

Can be used to define a symbol, to skip the field use a blank or tab

Operation

Defines the opcode or directive. (Not case sensitive)

Operand

Contains an address or the data for the instruction. (Ignored with inherent addressing)

Comment

Used for software documentation can be at end or start of statement.

Uses a semicolon to start the comment.

Examples

label adda 3 ; comment here   
adda label   
clra              ; inherent address   
; this is a full line comment

**Assembler Directives**

|  |  |  |
| --- | --- | --- |
| **Tag** | **Description** | **Example** |
| .org | Where to put code | .org $200 |
| .equ | Define Constant | .equ 100 |
| .setw | Preset memory word | .setw $FFFE,10 |
| .rmb | Reserve Memory | .rmb 16 |
| .byte | Define Variable | .byte 64 |
| Array of bytes | .byte 1,2,3 |
| .word | Define Variable | .word 5000 |
| Array of words | .word 1,2,3 |
| .str | Define string | .str "text" |
| .end | End of Program | .end |

**Number Formats**

|  |  |  |
| --- | --- | --- |
| **Prefix** | **Description** | **Example** |
|  | Decimal | 320 |
| $ | Hex | $240 |
| % | Binary | %0110101 |
| ' | Character | 'm |

**Operand Formats**

|  |  |  |  |
| --- | --- | --- | --- |
| **Format** | **Description** | **Examples** | **Mode** |
|  | Inherent | CLRA | H |
| #<data> | Immediate | LDDA #4 | I |
| <data> | Relative | BRA 10 | R |
| <data> | Direct | LDDA 4 | D |
| <data> | Extended | LDDA label | E |
| <data>,X | Indexed | LDDA 4,X | X |

====================================================

**Instruction Reference**

====================================================

|  |  |
| --- | --- |
| Status Flags   * -  unchanged * \*  updated * 1  set * 0  reset | H I N Z V C | | | | | | | | | | | --- Carry-Borrow | | | | ------ Overflow | | | --------- Zero | | ------------ Negative | --------------- Interrupt Mask ------------------ Half Carry |

|  |  |  |  |
| --- | --- | --- | --- |
| **Mnemonic** | **Operation** | **Mode** | **Status** |
| *6800 Undocmented Instructions* | | | HINZVC |
| HCF | Halt and Catch Fire | H | ------ |
| NBA | AND Accumulators     [A <= A & B] | H | ---\*\*- |
|  | Immediate Versions of STAA STAB STS etc |  | ------ |
| *6811 Specific Instructructions* | | | HINZVC |
| IDIV | Integer Divide 16 x 16 | H | ---\*\*\* |
| FDIV | Factional Divide 16x 16 | H | ---\*0\* |
| MUL | 16 multiply A x B | H | ------ |
| DEY | Decrement index reg Y | H | ---\*-- |
| INY | increment index reg Y | H | ---\*-- |
| STY | Store Index reg. Y | IDXE | --\*\*0- |
| STD | Store composite reg. D | IDXE | --\*\*0- |
| LDY | Load index reg. Y | IDXE | --\*\*0- |
| LDD | Load composite reg. D | IDXE | --\*\*0- |
| CPY | compare  index reg. Y | IDXE | --\*\*\*\* |
| CPD | compare composite reg. D | IDXE | --\*\*\*\* |
| ABX | Add acc.B to index reg.X | H | ------ |
| ABY | Add acc.B to index reg.Y | H | ------ |
| PSHX | push index reg.X onto stack | H | ------ |
| PSHY | push index reg.Y onto stack | H | ------ |
| PULX | pull index reg.X from stack | H | ------ |
| PULY | pull index reg.Y from stack | H | ------ |
| XGDX | exchange composite reg.D with X | H | ------ |
| XGDY | exchange composite reg.D with Y | H | ------ |
| TSY | Transfer SP to index reg.Y | H | ------ |
| TYS | Transfer index reg.Y to SP | H | ------ |
| ADDD | add using composite reg. D | IDXE | --\*\*\*\* |
| LSRD | logical shift right composite reg.D | H | --0\*\*\* |
| ASLD | arith. shift left composite reg.D | H | --\*\*\*\* |
| *Miscellaneous* | | | HINZVC |
| NOP | No Operation | H | ------ |
| RTI | Return from Interrupt  [\**not supported*] | H | \*\*\*\*\*\* |
| RTS | Return from Subroutine | H | ------ |
| SWI | Software Interrupt [\**not supported*] | H | -1---- |
| WAI | Wait for Interrupt | H | -1---- |
| *Status Flags* | | | HINZVC |
| CLC | Clear Carry | H | -----0 |
| CLV | Clear Overflow | H | ----0- |
| CLI | Clear Interrupt Mask | H | -0---- |
| SEC | Set Carry | H | -----1 |
| SEV | Set Overflow | H | ----1- |
| SEI | Set Interrupt Mask | H | -1---- |
| TAP | Set Status Mask [Flags<-AccA] | H | \*\*\*\*\*\* |
| TPA | Load Status Flags [AccA<-Flags] | H | ------ |
| *Branch and Jump* | | | HINZVC |
| BRA | Branch Always | R | ------ |
| BCS | Branch if Carry Set [C=1] | R | ------ |
| BCC | Branch if Carry Clear [C=0] | R | ------ |
| BMI | Branch if Minus [N=1] | R | ------ |
| BPL | Branch if Plus [N=0] | R | ------ |
| BVS | Branch if Overflow Set [V=1] | R | ------ |
| BVC | Branch if Overflow Clear[V=0] | R | ------ |
| BEQ | Branch if Equal [Z=1] | R | ------ |
| BNE | Branch if Not Equal [Z=0] | R | ------ |
| BLT | Branch if < (signed) | R | ------ |
| BLE | Branch if <= (signed) | R | ------ |
| BGE | Branch if >= (signed) | R | ------ |
| BGT | Branch if > (signed) | R | ------ |
| BLS | Branch if Lower or Same (unsgn) | R | ------ |
| BHI | Branch if Higher (unsigned) | R | ------ |
| BSR | Branch to Subroutine | R | ------ |
| JSR | Jump to Subroutine | E,X | ------ |
| JMP | Jump Absolute | E,X | ------ |
| *Stack and Index* | | | HINZVC |
| CPX | Compare X | IDXE | ---\*-- |
| DES | Decrement SP [SP <- SP - 1] | H | ------ |
| DEX | Decrement X [ X <- X - 1] | H | ---\*-- |
| INS | Increment SP [SP <- SP + 1] | H | ------ |
| INX | Increment X [ X <- X + 1] | H | ---\*-- |
| LDS | Load SP | IDXE | --\*\*0- |
| LDX | Load X | IDXE | --\*\*0- |
| STS | Store SP | DXE | --\*\*0- |
| STX | Store X | DXE | --\*\*0- |
| TSX | Transfer SP,X [X <= SP + 1] | H | ------ |
| TXS | Transfer X,SP [SP <= X - 1] | H | ------ |
| *Accumulator Only* ==> NEMz can be either NEMA or NEMB <== | | | HINZVC |
| ASLz | Arithmetic Shift Left Acc | H | --\*\*\*\* |
| ASRz | Arithmetic Shift Right Acc | H | --\*\*\*\* |
| LSRz | Logical Shift Right Acc | H | --0\*\*\* |
| ROLz | Rotate Left Acc | H | --\*\*\*\* |
| RORz | Rotate Right Acc | H | --\*\*\*\* |
| CLRz | Clear Accumulator | H | --0100 |
| COMz | One's Complement | H | --\*\*01 |
| DECz | Decrement Acc [A <- A - 1] | H | --\*\*\*- |
| INCz | Increment Acc [A <- A + 1] | H | --\*\*\*- |
| NEGz | Negate Acc [A <- 0 - A] | H | --\*\*\*\* |
| PSHz | Push Accumulator | H | ------ |
| PULz | Pull/Pop Accumulator | H | ------ |
| TSTz | Test Acc [A <- A - 0] | H | --\*\*00 |
| ABA | Add Accumulators [A <- A + B] | H | \*-\*\*\*\* |
| CBA | Compare Accumulators [A - B] | H | --\*\*\*\* |
| DAA | Decimal Adjust Accumulator [\**not supported*] | H | --\*\*\*\* |
| SBA | Subtract Accumulators [A<-A-B] | H | --\*\*\*\* |
| TAB | Transfer Accumulator [B <- A] | H | --\*\*0- |
| TBA | Transfer Accumulator [A <- B] | H | --\*\*0- |
| *Memory Reference* ==> NEMz can be either NEMA or NEMB <== | | | HINZVC |
| LDAz | Load Accumulator [A <- M] | IDXE | --\*\*0- |
| STAz | Store Accumulator [M <- A] | DXE | --\*\*0- |
| ADCz | Add with Carry [A <- A+M+C] | IDXE | \*-\*\*\*\* |
| ADDz | Add [A <- A+M] | IDXE | \*-\*\*\*\* |
| SBCz | Subtract w. Carry[A <- A-M-C] | IDXE | --\*\*\*\* |
| SUBz | Subtract [A <- A-M] | IDXE | --\*\*\*\* |
| BITz | Bit Test [Z and M] | IDXE | --\*\*0- |
| ANDz | Logical AND [A <- A and M] | IDXE | --\*\*0- |
| EORz | Exclusive OR [A <- A xor M] | IDXE | --\*\*0- |
| ORAz | Inclusive OR [A <- A or M] | IDXE | --\*\*0- |
| CMPz | Compare Memory [A - M] | IDXE | --\*\*\*\* |
| ASL | Arithmetic Shift Left | XE | --\*\*\*\* |
| ASR | Arithmetic Shift Right | XE | --\*\*\*\* |
| LSR | Logical Shift Right | XE | --0\*\*\* |
| ROL | Rotate Left Memory | XE | --\*\*\*\* |
| ROR | Rotate Right Memory | XE | --\*\*\*\* |
| CLR | Clear Memory [M <- 0] | XE | --0100 |
| COM | Complement Memory[M <- !M] | XE | --\*\*01 |
| DEC | Decrement Memory [M <- M - 1] | XE | --\*\*\*- |
| INC | Increment Memory [M <- M + 1] | XE | --\*\*\*- |
| NEG | Negate Memory [M <- 0 - M] | XE | --\*\*\*\* |
| TST | Test Memory [M - 0] | XE | --\*\*00 |

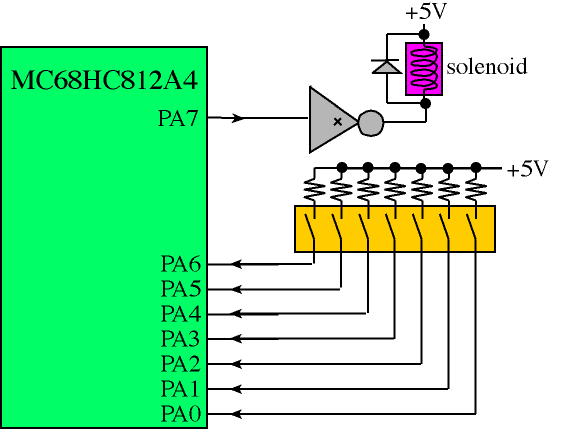
Links

Additional resources on the 6800 can be found:

* [6800 Assembly Language](http://www.ece.utexas.edu/~valvano/assmbly/intro.htm)
* [Original 6800 Reference Manual](http://www.bitsavers.org/pdf/motorola/M6800applMan_Mar75.pdf)

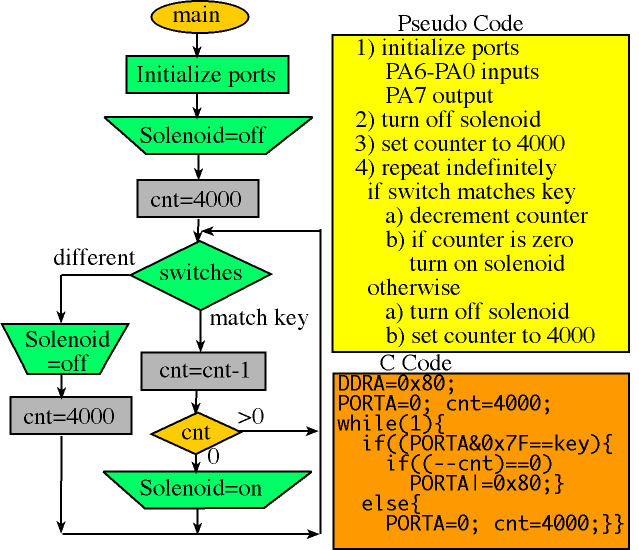
**Developing Software in Assembly Language  
Overview  
By Jonathan W. Valvano**     This article, which discusses assembly language programming, accompanies the book Embedded Microcomputer Systems: Real Time Interfacing published by Brooks-Cole 1999. This document has four overall parts     Overview (this document)     [Syntax](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm) (fields, pseudo ops)     [Local variables](http://users.ece.utexas.edu/~valvano/assmbly/local.htm)     [Examples](http://users.ece.utexas.edu/~valvano/assmbly/exmple.htm)

     Writing assembly language software is similar to other software development processes. For general information on the process of software development see Chapter 2 of Embedded Microcomputer Systems: Real Time Interfacing by Jonathan W. Valvano. Again, this article will focus on assembly language developing using the **TExaS**application. There are two types of commands that exist in our assembly source code. Op codes are regular instructions that translate into machine code to be executed by the computer when the program runs. We use pseudo-op codes in our source code to give instructions to the assembler itself. The following lists the default color settings of the **TExaS**editor.  
     The [labels](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#labels) are shown in purple  
     The [op codes](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#operation) are shown in blue  
     The [pseudo-op codes](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#pseudo) are shown in gray  
     The numbers are shown in dark blue  
     The strings are shown in magneta  
     The [operands](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#operand) are shown in black  
     The [comments](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#comments) are shown in green  
     The [assembly errors](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#errors) are shown in red

--------------------------------------------------------------------------------------  
**Case Study: Microcomputer-Based Lock**  
      To illustrate the software development process, we will implement a simple digital lock. The lock system has 7 toggle switches and a solenoid as shown in the following figure. If the 7-bit binary pattern on Port A bits 6-0 becomes 0100011 for at least 10 ms, then the solenoid will activate. The 10 ms delay will compensate for the switch bounce. For information on switches and solenoids see Chapter 8 of Embedded Microcomputer Systems: Real Time Interfacing by Jonathan W. Valvano. For now what we need to understand is that Port A bits 6-0 are input signals to the computer and Port A bit 7 is an output signal.  


Before we write assembly code, we need to develop a software plan. Software development is an iterative process. Even though we list steps the development process in a 1,2,3... order, in reality we iterative these steps over and over.  
      1) We begin with a list of the inputs and outputs. We specify the range of values and their significance. In this example we will use PORTA. Bits 6-0 will be inputs. The 7 input signals represent an unsigned integer from 0 to 127. Port A bit 7 will be an output. If PA7 is 1 then the solenoid will activate and the door will be unlocked. In assembly language, we use [equ](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm" \l "equ) pseudoops to assign a symbolic names, **PORTA DDRA**, to the corresponding addresses of the ports, **$0000 $0002**.  
PORTA  equ   $0000   ; PA6-PA0 switches, PA7 solenoid lock  
DDRA   equ   $0002   ; specifies input or output  
  
      2) Next, we make a list of the required data structures. Data structures are used to save information. If the data needs to be permanent, then it is allocates in global space. If the software will change its value then it will be allocated in RAM. In this example we need a 16-bit unsigned counter. We use the [org](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#org) pseudoop to place the data structures in RAM, **$0800**. The [rmb](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm" \l "rmb) pseudoop reserves bytes for the structure. These lines also assign the symbolic name **CNT** to the corresponding address of the information **$0800**.  
cnt    rmb   2       ; 16-bit counter  
       org   $F000   ; EEPROM

If data structure can be defined at assembly time and will remain fixed, then it can be allocated in EEPROM. In this example we will define an 8 bit fixed constant to hold the key code, which the operator needs to set to unlock the door. We will place these lines directly after the program so that they will be defined in ROM or EEPROM memory. The [fcb](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm" \l "fcb) pseudoop defines an 8-bit constant. This line also assigns the symbolic name **key** to the corresponding address of the information.  
key    fcb   %00100011    ; key code

It is not real clear at this point exactly where in EEPROM this constant will be, but luckily for us, the assembler will calculate the exact address automatically. After the program is assembled, we can look at the line in the listing file or in the symbol table to see where in memory each structure is allocated.  
      3) Next we develop the software algorithm, which is a sequence of operations we wish to execute. There are many approaches to describing the plan. Experienced programmers can develop the algorithm directly in assembly language. On the other hand, most of us need an abstractive method to document the desired sequence of actions. Flowcharts, pseudo code, and high-level language code are three common descriptive formats. The **TExaS** application is unique in regards that if you draw the flowchart on the computer, you can paste it directly into the program as a comment. There are no formal rules regarding pseudo code, rather it is a shorthand for describing what to do and when to do it. We can place our pseudo code as documentation into the comment fields of our program. The following shows a flowchart on the left and pseudo code and C code on the right for our digital lock example.  


Normally we place the programs in ROM or EEPROM. On the MC68HC812A4 EEPROM begins at $F000, so we put the following [org](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#org) pseudoop before the program.  
      org   $F000   ; EEPROM

Even though this program will not use the stack, we will initialize the stack pointer to the last location of RAM. On the 6812, we execute the following instruction to initialize the [stack](http://users.ece.utexas.edu/~valvano/assmbly/exmple.htm#stackinit).  
main  lds   #$0C00  
Next we write assembly code to implement the algorithm as illustrated in the above flowchart and pseudo code. In step 1), we initialize Port A so that PA7 is an output and PA6-PA0 are inputs. For information on ports and direction registers see Chapter 1 of Embedded Microcomputer Systems: Real Time Interfacing by Jonathan W. Valvano.  
      movb  #$80,DDRA   ; PA6-PA0 input, PA7 output

In step 2), we turn off the solenoid.  
      bclr  PORTA,#$80  ; disable solenoid lock

In step 3), we initialize the counter to 4000, which is the number of loops required to wait 10 ms.  
      movw  #4000,cnt  ; 10,000,000ns/(125\*20)

In step 4) we implement the indefinite loop. We place an assembly label at the program locations to which we wish to branch. The **bra** instruction is an unconditional branch.  
loop  
      bra   loop

Inside the indefinite loop we test to see if the switch pattern matches the key code. In this implementation we branch to **off** if the switches do not match the key code. If they do match we will execute the instruction immediately after the **bne off**.  
loop   ldaa  PORTA   ; [3] input from 7 switches  
       anda  #$7F    ; [1]  
       cmpa  key     ; [3] match key code?  
       bne   off     ; [3]

If the switches match the key code, then the 16-bit counter is decremented.  
       ldx   cnt     ; [3]  
       dex           ; [1]  
       stx   cnt     ; [3]

If the counter becomes zero, then the door is unlocked. The **bne** instruction will go to loop if **cnt** is not equal to zero. The **bset** instruction will set PA7 to 1.  
       bne   loop    ; [2]=20 cycles/loop  
       bset  PORTA,#$80   ; enable solenoid lock  
       bra   loop

If the switches do not match the key code, then the solenoid is turned off and the cnt set back to 4000.  
off    movw  #4000,cnt    ; 10,000,000ns/(125\*20)  
       bclr  PORTA,#$80   ; disable solenoid lock  
       bra   loop

We put the above pieces together to create the source code. The order of the instructions is very important because it determines the sequence of execution. The last two lines will define where the computer will start execution after a reset.  
; \*\*\*\*\*\*lock.rtf\*\*\*\*\*\*\*\*\*\*\*\*  
; activate solenoid (PA7=1) if switches match key code  
; switch bounce is less than 10 ms  
PORTA  equ   $0000   ; PA6-PA0 switches, PA7 solenoid lock  
DDRA   equ   $0002   ; specifies input or output  
       org   $0800   ; RAM  
cnt    rmb   2       ; 16-bit counter  
       org   $F000   ; EEPROM  
main   lds   #$0C00  
       movb  #$80,DDRA   ; PA6-PA0 input, PA7 output  
       bclr  PORTA,#$80  ; disable solenoid lock  
       movw  #4000,cnt   ; 10,000,000ns/(125\*20)  
loop   ldaa  PORTA   ; [3] input from 7 switches  
       anda  #$7F    ; [1]  
       cmpa  key     ; [3] match key code?  
       bne   off     ; [3]  
       ldx   cnt     ; [3]  
       dex           ; [1]  
       stx   cnt     ; [3]  
       bne   loop    ; [2]=20 cycles/loop  
; 7 switches match key code for more than 10 ms  
       bset  PORTA,#$80   ; enable solenoid lock  
       bra   loop     
off    movw  #4000,cnt    ; 10,000,000ns/(125\*20)  
       bclr  PORTA,#$80   ; disable solenoid lock  
       bra   loop     
key    fcb   %00100011    ; key code  
       org   $FFFE  
       fdb   main

   4) The last stage is debugging. You can run this example on the **TExaS** simulator by loading the files **lock.rtf**, **lock.uc** and **lock.io**. There are versions for all four microcomputers, located in the MC6805, MC6808, MC6811 and MC6812 directories. For information on debugging see Chapter 2 of Embedded Microcomputer Systems: Real Time Interfacing by Jonathan W. Valvano.

--------------------------------------------------------------------------------------  
**Assembler Basics**  
     Assemblers are programs that process assembly language source program statements and translate them into executable machine language object files. Cross assemblers (such as **TExaS**) allow source programs written and edited on one computer (the host) to generate executable code for another computer (the target). In our case the target is microcomputer simulator, but since **TExaS**generates [S19](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#s19) records, you could download the S19 file onto a real computer for execution.  
     The symbolic language used to code source programs to be processed by the Assembler is called assembly language. The language is a collection of mnemonic symbols representing: operations (i.e., machine instruction mnemonics or directives to the assembler), symbolic names, operators, and special symbols. The assembly language provides mnemonic operation codes for all machine instructions in the instruction set. The instructions are defined and explained in the Programming Reference Manual (for the 6805, 6808, 6811 or 6812), which can be found in the **pdf** directory of the CD. Printed copies of these Motorola manuals can be obtained from the Motorola Literature Center. A brief overview of each instruction and example usage can be found by searching the **Contents** page of the help engine included with the **TExaS** application. The assembly language also contains mnemonic directives that specify auxiliary actions to be performed by the Assembler. These directives are not always translated into machine language. Many [pseudo-op codes](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#pseudo) are supported by this assembler.

     The **TExaS**assembler is a two-pass assembler. During the first pass, the source program is read to develop the symbol table. During the second pass, the object file is created (assembled) using the symbol table developed in pass one. It is during the second pass that the source program listing is also produced. The symbol table is recreated in the second pass. A [Phasing Error](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#phasing) occurs if the symbol table values calculated in the two passes are different.  
      Errors that occur during the assembly process (e.g., undefined symbol, illegal op code, branch destination too far, etc.) are explained in the listing file (I shortened the comments so each line would fit on the page).  
Copyright 1999-2000 Test EXecute And Simulate - Version 1.00  
                                           ; \*\*\*\*\*\*lock.rtf\*\*\*\*\*\*\*  
                                           ; activate solenoid (PA7=1)  
                                           ; bounce is less than 10 ms  
$0000                                      PORTA equ  $0000  
$0002                                      DDRA  equ  $0002  
$0800                                            org  $0800  
$0800                                      cnt   rmb  2  
$F000                                            org  $F000  
$F000 CF0C00          [ 2](  0){OP        }main  lds  #$0C00  
$F003 180B800002      [ 4](  2){OPwP      }      movb #$80,DDRA  
$F008 4D0080          [ 4](  6){rPOw      }off   bclr PORTA,#$80  
$F00B CC115C          [ 2]( 10){OP        }look  ldd  #4444  
$F00E 7C0800          [ 3]( 12){WOP       }      std  cnt  
$F011 9600            [ 3]( 15){rfP       }loop  ldaa PORTA   
$F013 847F            [ 1]( 18){P         }      anda #$7F  
$F015 B1F028          [ 3]( 19){rOP       }      cmpa key  
$F018 26EE            [ 3]( 22){PPP/P     }      bne  off  
$F01A FE0800          [ 3]( 25){ROP       }      ldx  cnt  
$F01D 09              [ 1]( 28){O         }      dex   
$F01E 7E0800          [ 3]( 29){WOP       }      stx  cnt  
$F021 26EE            [ 3]( 32){PPP/P     }      bne  loop  
                                           ; 7 switches match key code  
$F023 4C0080          [ 4]( 35){rPOw      }      bset PORTA,#$80  
$F026 20E3            [ 3]( 39){PPP       }      bra  look  
$F028 23                                   key   fcb  %00100011  
$FFFE                                            org  $FFFE  
$FFFE F000                                       fdb  main  
                                             
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Symbol Table\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
DDRA         $0002   
PORTA        $0000   
cnt          $0800   
key          $F028   
look         $F00B   
loop         $F011   
main         $F000   
off          $F008   
Assembly successful

     The source code is a file of ASCII characters usually created with an editor. Each source statement is processed completely before the next source statement is read. As each statement is processed, the Assembler examines the label, operation code, and operand fields. The operation code table is scanned for a match with a known opcode. During the processing of a standard operation code mnemonic, the standard machine code is inserted into the object file. If an assembler directive is being processed, the proper action is taken.  
     Any [errors](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#errors) that are detected by the Assembler are displayed after the actual line containing the error is printed. If no source listing is being produced, error messages are still displayed in the TheLog.RTF to indicate that the assembly process did not proceed normally. Object code is the binary values (instructions and data) that, when executed by the computer, perform the intended function. The listing file contains the address, object code, and a copy of the source code. You can optionally include the number of cycles to execute the instruction, a running total of cycles, and explicit details of the cycles types required to execute the instruction. The listing file also provides a symbol table describing where in memory the program and data will be loaded. The symbol table is a list of all the names used in the program along with the values. A symbol is created when you put a label starting in column 1. Examples of this type are DATA, CNT, Start, Rep, Inp, NAME and SIZE. The symbol table value for this type is the absolute memory address where the instruction, variable or constant will reside in memory. The second type of label is created by the equ pseudo-op. The value for this type of symbol is simply the number specified in the operand field. When the assembler processes an instruction with a symbol in it, it simply substitutes the fixed value in place of the symbol. Therefore we will use symbols to clarify (make it easier to understand) our programs. The symbol table for this example is  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Symbol Table\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
      DDRA     $0002  
      PORTA    $0000  
      cnt      $0800  
      key      $F033  
      loop     $F011  
      main     $F000  
      off      $F028

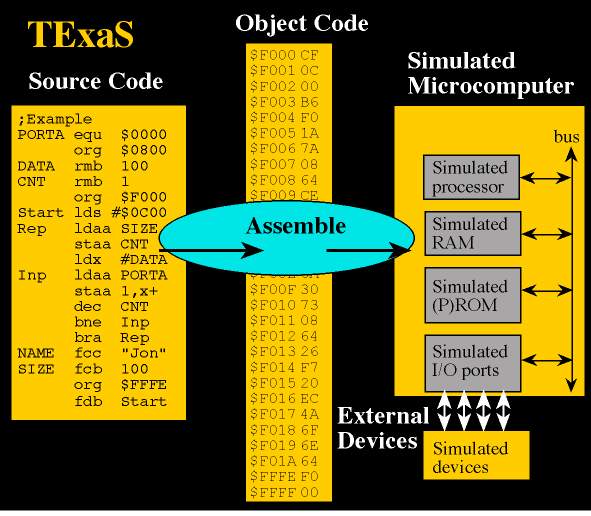
     A compiler converts high level language source code into object code. A cross-compiler also converts source code into object code and creates a listing file except that the object code is created for a target machine that is different from the machine running the cross-compiler. The **TExaS**is a cross-assembler because it runs on an Intel computer but creates 6805/6808/6811/6812 object code. Hiware and ImageCraft's ICC11 and ICC12 include both a cross-assembler and a cross-compilers because they run on the PC and create 6805/6808/6811/6812 object code. For general information on C language software development see Chapter 2 of Embedded Microcomputer Systems: Real Time Interfacing by Jonathan W. Valvano.

     Motorola uses S-records to store object code. The 16 bit values are stored in memory with the most significant byte first. For more information about the specific format of the S-record see [S19 records](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#s19) contain object code. The S-record for the above example is as follows. The actual object code is colored in blue.

S121F000CF0C00180B8000024D008018030FA008009600847FB1F033260EFE0800091F  
S119F01E7E080026EE4C008020E918030FA008004D008020DE23A9  
S105FFFEF0000D  
S903F0000C

--------------------------------------------------------------------------------------  
**Running on an Evaluation Board**When debugging involves controlling real hardware, we must use an actual microcomputer running in real time. Because the programs are stored in ROM or EEPROM on the single chip microcomputer it is difficult to debug. To solve this problem there are evaluation boards that run the microcomputer in expanded mode and include a debugger. With this development system, we also use a cross-assembler or a cross-compiler to convert our source programs into a listing file and object file. The object file is transmitted via the serial port to the evaluation board and loaded into RAM. A simple debugger (like the 6811 EVB using the BUFFALO debugger) allows us to test our software. Typical debugger features include:  
     start execution at a specific address  
     set breakpoint at a specific address  
     read/modify microcomputer registers  
     read/modify microcomputer I/O ports  
     read/modify RAM  
     read/erase/program EEPROM  
     load object code into RAM received from the HOST PC.  
Sophisticated development environments integrate the editor, compiler, assembler, serial port communication and debugger into a single application running on the PC. For more information on software development see Chapter 2 of Embedded Microcomputer Systems: Real Time Interfacing by Jonathan W. Valvano. For information about hardware development platforms, see the Lab Manual that accompanies this book.

--------------------------------------------------------------------------------------  
**Simulation**When developing software using a simulator, we use a cross-assembler or a cross-compiler running on a PC to convert our source programs into a listing file and object file. We then "run" our program on a simulator that emulates microcomputer with its external component. Simulation in this environment is more difficult than other computer application because of the software execution is tightly coupled to (extremely dependent on) the hardware. Simulation of an embedded system is only effective if all the software, computer, external mechanical, and external electrical components are modeled. Another complicating issue is the real time nature of the external mechanical and electrical devices. The following figure outlines the software development process using a simulator.



Additional topics include.  
    [Assembly Language Syntax](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#syntax)  
    [Label Field](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#labels)  
    [Operation Field](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#operation)  
    [Operand Field](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#operand)  
    [Comment Field](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#comments)  
    [Assembly listing](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#listing) gives results of the assembly  
    [Assembly errors](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm" \l "errors) lists the types of assembly errors  
    [Assembly pseudo-op's](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#pseudo) are assembly-time operations  
    [S19 records](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#s19) contain object code

For a detailed explanation of the instructions and their addressing modes, see the help system with the **TExaS** application. Sometimes the best way to learn a new technique is to observe examples. See the **ReadMe.txt** file for a list of examples that can be run on the simulator. There are many assembly language examples included with the **TExaS** application. Some simple 6812 [examples](http://users.ece.utexas.edu/~valvano/assmbly/exmple.htm) are included in this document.

This document has four overall parts     [Overview](http://users.ece.utexas.edu/~valvano/assmbly/intro.htm#top) (this document)     [Syntax](http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm) (fields, pseudo ops)     [Local variables](http://users.ece.utexas.edu/~valvano/assmbly/local.htm)     [Examples](http://users.ece.utexas.edu/~valvano/assmbly/exmple.htm)