ETec374 – Off Campus
Student Lab Manual

Off-Campus 68HC912EVB

Todd Morton

Western Washington University
Electronics Engineering Technology

Todd Morton
Electronic Engineering Technology
Western Washington University
Bellingham, Washington 98225
(206) 650-2918
email: toddm@etec.wwu.edu

Contents

1 912EVB SOFTWARE DEVELOPMENT GUIDE

1.1 System Overview

This is a guide to the software development system, hardware and software, used in Western's Electronic Engineering Technology off-campus program. The software development system is used to efficiently write and assemble (or compile) a program, download the resulting machine code to the processor memory for testing.

Note: This manual supports Motorola's MC68HC912B32 EVB Board.

The software development process used is typical of a low-cost process used in industry for designing embedded systems and consists of the following steps:

1) A text editor is used to create/change your program source code.

2) An assembler is used to convert the source code to machine code in s-record format.

3) The machine code is downloaded to the 912EVB RAM.

4) The 912EVB monitor program, DBug-12, is used to execute and debug the program.

5) Steps 1 through 4 are repeated until the program is working as required.

The software development system used to carry out these steps is made up of the following hardware:

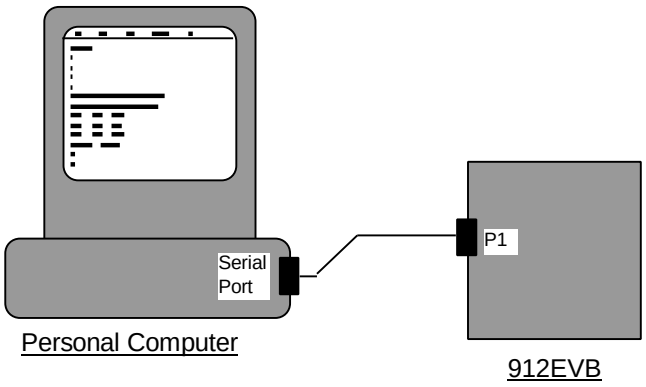
- The M68HC912EVB board
- A terminal or a microcomputer running a terminal emulator
- An optional host system (typically UNIX)
- Interconnect cables

And, the following software:

- A text editor on the microcomputer or host
- An assembler or compiler on microcomputer or host.
- The D-Bug12 monitor program on the 912EVB.

There are two possible system configurations when using the 912EVB for software development as shown in Figure 1-1 and Figure 1-2.

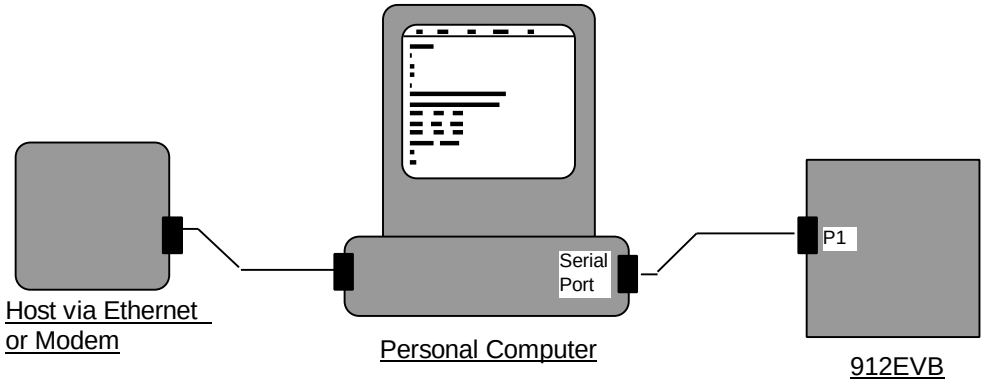
Figure 1-1 Microcomputer only configuration.



The system shown in Figure 1-1 is made up of a personal computer and a 912EVB. The assembler and editor are microcomputer applications. The microcomputer must also have a terminal emulator and a serial port for communication with the 912EVB. The serial port communications is used for access to the 912EVB monitor, DBug-12, and downloading object code, in S-record format, to the 912EVB memory.

When using this configuration, the assembler, IASM12, from Motorola is used for assembling 68HC12 programs. Other cross-assemblers or high-level language cross-compilers can be purchased or found on the Internet for PC and Macintosh systems. This is the typical configuration for students working at home **without** modem access.

Figure 1-2 Terminal/Host configuration.



The second configuration shown in Figure 1-2 makes use of an Ethernet or high-speed modem to communicate between the personal computer and a host. It will be the primary configuration used in ETec374 on- and off-campus. The workstation requires one port for host communications (Ethernet or modem) and a serial port for the 912EVB.

The host used for editing and assembling the source code is the EET program's UNIX workstation, *hoh*. Students are expected to work on *hoh* for all of the ETec374 assignments. The workstations in the NSCC lab are connected to *hoh* via Ethernet. To access *hoh* from home a high-speed modem can be used to connect to hoh through an Internet service provider. To work on *hoh* from the workstations in the lab or over a modem, you use a telnet program as described below. A terminal emulator such as HyperTerminal can be used to communicate to the 912EVB board.

The assembly code is written and assembled on the host, downloaded to a file on the workstation, then the file is downloaded to the 912EVB. The intermediate step of creating a file on the workstation is typically skipped for small programs by using a copy and paste between the host and the 912EVB terminal windows. This is adequate for all programs written for ETec374.

hoh contains a software development system from Introl Corporation that is made up of a C cross compiler, a macro assembler, simulator/debugger and, a linker. In addition, a shell script, *asm11* has been written to simplify the macro assembler. *asm12* is currently required for all submitted ETec374 laboratories.

To establish communications with the 912EVB, open a terminal window to the appropriate COM port and press the RESET button on the 912EVB board. You should get an introduction prompt indicating the version of the monitor software and the DBug-12 prompt, >. For more information on using the 912EVB board and the DBug-12 monitor, refer to Motorola's 912EVB users manual and the course textbook.

1.2 Getting Started with UNIX

Like DOS and VMS, UNIX is an operating system. Many of the commands are the same as DOS and VMS so it should not take long to learn the basics required for 68HC12 software development. These basics include: file/directory creation and organization, editing (using vi, pico, or emacs), email for sending source code to the instructor and, assembling source code using the Introl assembler.

The Electronic Engineering Technology's UNIX system for student use is called *hoh*. To learn more about UNIX there are several good introductory books on UNIX. There is also an on-line

manual, which has more information than you would ever want to know. To use the on-line manual type:

```
man command
```

where command is a valid UNIX command.

```
man -k keyword
```

where 'keyword' is some subject that you are looking for. If you can't seem to find information on a certain topic try:

```
man csh
```

csh is the C-shell under which you are working. There are also many books available on the UNIX operating system including:

'Learning the UNIX Operating System', O'Reilly & Associates.

'UNIX in a Nutshell', O'Reilly & Associates.

O'Reilly & Associates also has many other books on UNIX systems and utilities including 'Learning the VI Editor' which is a good guide to using the vi editor.

1.3 Editing

A text editor is used for writing your 68HC12 assembly language programs or, source code. In ETec374, three editors will be supported, the *vi* editor for the UNIX system, *bbedit* for Macs and, *Notepad* for the PCs. Other editors or word processors can be used as long as a 'text only' output format can be selected.

It can be frustrating to learn a new editor but once the basic techniques are learned, it will be a simple process. It is very important to learn these techniques as soon as possible and; '*The best way to learn an editor is to use it!*'

This document will cover *vi* only. For documentation on other editors, refer to user manuals or on-line help.

The VI Editor

The VI editor is normally supplied on all UNIX-based machines. It is a bit awkward but, because of its wide availability, it is worth learning. A VI command list is included here to provide a quick and easy working reference. Also, as mentioned above, O'Reilly & Associates has a good book on VI.

Most UNIX systems also have other editors available such as emacs and pico. They are used for some computer science courses. If you are already familiar with emacs or pico, you are welcome to use them.

When creating your source file using any editor on *hoh*, the correct file extension conventions must be followed. The required source code filename extension for the Introl assembler is *.s12*. So, to start editing a source file, type:

```
vi filename.s12
```

where 'filename' is the name you give your program.

If your terminal type is correct, the screen will contain the current contents of the file along with tildes in column 1 after the end of the file.

There are two modes in VI: 'command mode' and 'text entry mode'. The command mode is used for moving the cursor or scrolling up or down, etc, while the text entry mode is used for entering new text. To enter the text entry mode, type *a, i, o, O* etc. You return to the command mode by typing the *esc* key.

The advantage of using VI instead of emacs is that by knowing only a few commands you can perform all of the basic editing tasks required for simple software development. Table 1-1 is a short list of commands that I find useful. Once you are finished editing, exit VI by typing:

:x to save changes.

:w to save changes but keep window open.

:quit! to exit without saving changes.

Table 1-1 Basic VI Commands

A	Append to text after cursor.
i	Insert text at cursor.
x	Delete character.
r	Replace character.
cw	Change word
dd	Delete line.
dw	Delete word
yy	Yank line(copy line)
G	Go to end of file
u	Undoes last change.
o	Opens new line below cursor.
O	Opens new line above cursor.
J	Joins lines.
p	Puts last deleted or copied text at cursor (paste).
w	Go to next word
^d	Move 1/2 screen down
^u	Move 1/2 screen up
/pat	Search and go to the next occurrence of 'pat'.
.	Repeat last command.

1.4 D-Bug12 Utility Subroutines

Contained in the *D-Bug12* monitor program is a collection of useful utility subroutines. These utilities can be used to avoid rewriting software for common I/O tasks. These routines are C-

functions but can be interfaced when programming in assembly by following the parameter passing rules noted.

Documentation Notes:

Subroutine Address. The absolute memory location used to access the subroutine. In this case, the address shown is the location of a pointer to the subroutine in the D-Bug12 jump table. The addresses shown are for D-Bug12 version 2.0.0 and later. To access these routines an indirect jump must be made.

Example: Here are some examples for jumping to *main()*. First the most straight forward:

```
ldx MAIN
jsr ,x
```

You could also use:

```
jsr [MAIN-*-4,pc]
```

but, it's a bit cryptic. The Introl assembler supports the 'pcr', PC relative syntax, so you can use:

```
jsr [MAIN,pcr]
```

Entry requirements. These are the requirements for passing parameters to the subroutine and the stack space required. For all of these routines, the arguments are passed on the stack and in register ACCD. The first argument in the list is passed in ACCD. The remaining arguments are pushed onto the stack from right-to-left.

Exit characteristics. These describe how the parameters are passed back to the calling routine. All results and return values are passed through ACCD. If the parameter is a single byte, it will be in ACCB.

Register Preservation. Assume no registers are preserved so they must be preserved before calling the routine.

Routines

MAIN - Starts D-Bug12 without the startup hardware initialization.

C prototype:	*void main(void)*
Subroutine Address:	$f680
Entry Requirements:	Stack pointer must be initialized.
Exit Characteristics:	Does not return.
Stack Requirements:	None

GETCHAR - Get a character from the serial port. Blocks until a character is received.

C prototype:	*int getchar(void)*
Subroutine Address:	$f682
Entry Requirements:	None
Exit Characteristics:	ACCB contains the character. Does not return until a character is received.
Stack Requirements:	2 bytes

PUTCHAR - Send a character out the serial port. Blocks until the transmit register is empty.

C prototype:	*int putchar(int)*
Subroutine Address:	$f684
Entry Requirements:	ACCB contains the ASCII character to be sent.
Exit Characteristics:	ACCB contains the ASCII character sent. No error reporting.
Stack Requirements:	4 bytes

PRINTF - Formatted Output - Translates binary values to characters.

C prototype:	*int printf(char *format,...)*
Subroutine Address:	$f686
Entry Requirements:	ACCD points to the null terminated format string. The argument list is pushed onto the stack from right-to-left. Single bytes are placed in ACCB and pushed as ACCD.
Exit Characteristics:	ACCD contains the number of characters sent.
Stack Requirements:	64+ bytes

GETCMDLINE - Obtain a line of input from the user. Blocks until a carriage return is received.

C prototype:	*int GetCmdLine(char *CmdLineStr, int CmdLineLen)*
Subroutine Address:	$f688
Entry Requirements:	ACCD contains a pointer to a line buffer. The top two bytes of the stack contain the maximum length of the line +1.
Exit Characteristics:	Stores received characters in the buffer, echoes the received characters until a carriage return ($0d) is received. The characters are converted to uppercase and the carriage return is replaced by a NULL in the buffer.
Stack Requirements:	11 bytes

SSCANHEX - Convert an ASCII hexadecimal string to a binary integer in the range $0000 to $ffff.

C prototype:	*char * sscanhex(char *HexStr, unsigned int *BinNum)*
Subroutine Address:	$f68a
Entry Requirements:	ACCD contains a pointer to the null terminated ASCII string to be converted. A pointer to the resulting binary number must be on top of the stack.
Exit Characteristics:	Pointer to the terminating character or $0000 if an error occurred.
Stack Requirements:	6 bytes

ISXDIGIT - Checks for membership in the set [0..9,a..f,A..F].

C prototype:	*int isxdigit(int c)*
Subroutine Address:	$f68c
Entry Requirements:	ACCB contains the character to be checked.
Exit Characteristics:	ACCD = 0 if not a member, 1 if it is a member.
Stack Requirements:	4 bytes

TOUPPER - Converts lower case characters to upper case. If the character is not a lowercase alpha character toupper() returns the original character.

C prototype:	*int toupper(int c)*
Subroutine Address:	$f68e
Entry Requirements:	ACCB contains the character to be converted.
Exit Characteristics:	ACCB contains the converted character.
Stack Requirements:	4 bytes

ISALPHA - Checks for membership in the set [a..z,A..Z].

C prototype:	*int isalpha(int c)*
Subroutine Address:	$f690
Entry Requirements:	ACCB contains the character to be checked.
Exit Characteristics:	ACCD = 0 if not a member, 1 if it is a member.
Stack Requirements:	4 bytes

STRLEN - Returns the length of a null terminated string.

C prototype:	*Unsigned int strlen(const char *cs*
Subroutine Address:	$f692
Entry Requirements:	ACCD contains a pointer the the null terminated string.
Exit Characteristics:	ACCD contains the number of characters in the string.
Stack Requirements:	4 bytes

STRCPY - Copies a null terminated string.

C prototype:	*char * strcpy(char *s1, char *s2)*
Subroutine Address:	$f694
Entry Requirements:	ACCD contains a pointer to the string to be copied into. The top of the stack contains a pointer to the string to be copied.
Exit Characteristics:	ACCD contains a pointer to the string to be copied into.
Stack Requirements:	8 bytes

OUT2HEX - Displays 8-bit number as two ASCII hex characters.

C prototype:	*void out2hex(unsigned int num)*
Subroutine Address:	$f696
Entry Requirements:	ACCB contains the byte to be displayed.
Exit Characteristics:	None
Stack Requirements:	70 bytes

OUT4HEX - Displays 16-bit number as four ASCII hex characters.

C prototype:	*void out4hex(unsigned int num)*
Subroutine Address:	$f698
Entry Requirements:	ACCD contains the 16-bit word to be displayed.
Exit Characteristics:	None
Stack Requirements:	70 bytes

SETUSERVEC - Configure D-Bug12 to jump to a user interrupt service routine.

C prototype: *int SetUserVector(int VectNum, int UserAddress)*

Subroutine Address: $f69a

Entry Requirements: The address of the interrupt service routine must be in the top two bytes of the stack. ACCD contains the D-Bug12 vector number. (Refer to the list below)

Exit Characteristics: Normally a zero is returned in ACCD. If an illegal vector number is passed to SetUserVector() a -1 will be returned. If a -1 is passed to SetUserVector() the starting address of the vector RAM table is returned.

Stack Requirements: 8 bytes

Table 1-2 D-Bug12 Vector Numbers (version 2.0.0 and later)

Interrupt	Number
PortH Wake-up	7
PortJ Wake-up	8
AtoD	9
SCI #1	10
SCI #0	11
SPI #0	12
PAcc Edge	13
PAcc Overflow	14
Timer Channel 0	15
Timer Channel 1	16
Timer Channel 2	17
Timer Channel 3	18
Timer Channel 4	19
Timer Channel 5	20
Timer Channel 6	21
Timer Channel 7	22
Timer Channel 8	23
RTI	24
IRQ	25
XIRQ	26
SWI	27
Illop Trap	28
JumpTable Address	-1

SetUserVector can be used in two ways illustrated by the following examples.

Example 1. Use *SetUserVec()* to configure D-Bug12 to jump to the user interrupt service routine, *rti_isr*, when an RTI interrupt occurs.

```
*********************************************************
* Equates
*********************************************************
UserRTI     equ 24      ;D-Bug12 RTI Vector number
SETUSERVEC  equ $f69a   ;SetUserVector() pointer location
*********************************************************
* Program
               org $0800
main
               ldd #rti_isr            ;Put ISR Address on stack
               pshd
               ldd #UserRTI            ;Load ACCD with Vector number
               ldx #0
               jsr [SETUSERVEC,x]      ;Call SetUserVector()
               puld                    ;Reallocate stack space
               •
               •
               •
rti_isr        •
               •
               •
               rti
```

SetUserVector() is called during program initialization. This means D-Bug12 will be configured during 'runtime'. This is required because the D-Bug12 interrupt jump table is reinitialized when the board is reset. If the jump table was loaded during s-record, download the user ISR address would be lost after the board is reset. Notice a couple things about this program. The stack must be balanced after calling *SetUserVector()* and the interrupt service routine must end with an rti. This method should be used when setting only a few vectors.

Example 2. The RTI and IRQ interrupt vectors are set to user interrupt service routines using a pointer into D-Bug12's interrupt jump table.

```
*********************************************************
* Equates
*********************************************************
UserRTI     equ 24              ;D-Bug12 RTI Vector number
UserIRQ     equ 25              ;D-Bug12 IRQ Vector number
SETUSERVEC  equ $f69a   ;SetUserVector() pointer location
*********************************************************
* Program
               org $0800
main
               ldd #-1                 ; -1 to get table address
               ldx #0
               jsr [SETUSERVEC,x]      ;Call SetUserVector()
               tfr d,x                 ;IX points to jump table
               ldd #UserRTI*2          ;RTI number x 2 for offset
               movw #rti_isr,d,x       ;Put address in table
               ldd #UserIRQ*2          ;IRQ number x 2 for offset
               movw #irq_isr,d,x       ;Put address in table
               •
               •
               •
```

In this example *SetUserVector()* was used to get the base address of D-Bug12's interrupt jump table then the addresses were stored into the table directly. It is safe to access the D-Bug12 interrupt jump table directly as long as the base address is received by using *SetUserVector()*. This

allows programs to work with future versions of D-Bug12. This method should be used when setting many vectors.

WRITEEEBYTE - Write a data byte to on-chip EEPROM.

C prototype:	*Unsigned char WriteEEByte(int EEAddress, Byte EEData)*
Subroutine Address:	$f69c
Entry Requirements:	ACCD contains the EE address to be written to. The top of the stack contains the byte to be written.
Exit Characteristics:	If the written byte is not verified a zero(False) will be returned in ACCD
Stack Requirements:	12 bytes

EraseEE - Bulk erase on-chip EEPROM.

C prototype:	int EraseEE(void)
Subroutine Address:	$f69e
Entry Requirements:	Entry Requirements -EEBase and EESize are used to determine the EE block location and size.
Exit Characteristics:	Returns a non-zero value in ACCD if any byte in the EE block is not $FF.
Stack Requirements:	4 bytes

1.5 Extension Utility Subroutines

These subroutines are extensions to the D-Bug12 routines provided by Motorola. They can be used in addition to the D-Bug12 routines if they are loaded into the byte erasable EEPROM. The space required for these routines is the last 256 bytes in the EEPROM, $0f00-$0fff. This leaves 512 bytes available for user code. They use D-Bug12's *PUTCHAR()* so the monitor is required. The stack requirements listed do not include the two bytes for the 'jsr'. To call these routines use an extended jsr to the address indicated. For example:

```
PUTSTRG         equ $0fe3

                ldd #message
                jsr PUTSTRG

message         fcc 'Hi'
                fcb 0
```

INPUT - Reads the serial port one time. Returns ASCII character in ACCB if no character was received returns zero in ACCB. Non-blocking alternative to GETCHAR().

C prototype:	*char input(void)*
Subroutine Address:	$0fe0
Entry Requirements:	none.
Exit Characteristics:	If a character was received it is returned in ACCB. Otherwise ACCB is zero and the Z flag is set. All registers are preserved except CCR and ACCB.
Stack Requirements:	0 bytes

PUTSTRG - Outputs a NULL terminated ASCII string.

C prototype:	*char * putstrg(char *)*
Subroutine Address:	$0fe3
Entry Requirements:	ACCD contains a pointer to the NULL terminated string.
Exit Characteristics:	Returns with ACCD pointing to the NULL. All other registers are preserved except CCR.
Stack Requirements:	12 bytes

OUTBYTE - Outputs one hex byte. Alternative to OUT2HEX().

C prototype:	*char * outbyte(char *)*
Subroutine Address:	$0fe6
Entry Requirements:	ACCD contains a pointer to the byte to be displayed.
Exit Characteristics:	Returns with ACCD pointing to the next byte in memory. All other registers are preserved except CCR.
Stack Requirements:	12 bytes

OUT2BYTE - Outputs two hex bytes. Alternative to OUT4HEX().

C prototype:	*int * out2byte(int *)*
Subroutine Address:	$0fe9
Entry Requirements:	ACCD contains a pointer to the first of two bytes to be

displayed

Exit Characteristics: Returns with ACCD pointing to the next byte in memory. All other registers are preserved except CCR.

Stack Requirements: 14 bytes

OUTCRLF - Outputs a carriage return, $0d, and a line feed, $0a.

C prototype: *void outcrlf(void)*

Subroutine Address: $0fec

Entry Requirements: none.

Exit Characteristics: All registers are preserved except CCR.

Stack Requirements: 12 bytes

2 ASSEMBLERS

This document describes the Introl-CODE cross-assembler based shell script, *asm12*, for the UNIX system, *hoh.*

2.1 UNIX Assembler, asm12

This assembler is a shell script that makes the more powerful macro-assembler, as12, emulate a simple assembler that deals only with absolute program locations. By using absolute locations, a linking file is not required. When using asm11 the filename extensions are as follows:

filename.s12 source file from the editor

filename.lst assembled listing from asm12

filename.0 s-record output from asm12

Assemble your source file by typing: (Note: suffix is optional)

asm12 filename

2.2 Introl Assembler Documentation

The following is a subset of the documentation that describes the Introl assembler. This subset describes the required components of the assembler for ETec374. For the full documentation, see the on-line documents on hoh. The on-line documentation is in html format and can be read using the command *introlman* on any text terminal window.

Typographical Conventions

bold italic	Meta symbols used to represent variable quantities
italic	File Names, notes.
`sans serif`	Assembly keywords, symbols and commands.
bold	Choices in command line options
[brackets]	Optional text.
{curly braces}	Repeatable text

Expressions and Operators

All results of expressions at assembly time are 32-bit truncated integers

-	unary negate (2's complement)
~	not (1's complement)
*	multiplication
/	division
%	modulus (remainder)
+	addition
-	subtraction
<<	shift left
>>	shift right
&	bitwise AND
^	bitwise exclusive OR
|	bitwise OR
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal to
!=	not equal to

Precedence, from higher to lower:

-(unary) ~
* / %
+-
>> <<
<> <= >=
== !=
&
^
|

Constants

A constant is preceded by a radix indicator as follows

[***radix***]***n***

where radix is:

$	hexadecimal
@	octal
%	binary
& or nothing	decimal

and n is 0...9, A...F, or a...f and must be a valid digit for the given radix.

The assembler also recognizes a special constant that represents the current address in the section:

$ or *. When $ or * is used in an expression, the value taken is the current address in the section at the beginning of the line containing the $ or *.

Symbols

Symbols are made up of letters (a...z,A...Z), digits (0...9), ?, $, _, and periods (.). They must begin with a letter, a period, or a question mark, and can be any length (only the first 16 characters are printed in the listing file). Symbols are case sensitive so ABc is not the same as ABC.

When using the Introl assembler, symbols starting with a period should be reserved for section names.

Format of Input Files

The assembler expects its input to be an ASCII file, which contains assembler text. Each source line is either a line of the following format:

[***symbol***][:] [***operation*** [***operand***{,***operand***}]] [;comment]

or, a line beginning with an asterisk or semicolon, indicating a comment line. Comments may be placed at the end of a source line; they may be separated from the operation or operand field by either a space or a semicolon. A source line containing only a symbol but no operation or operand field can only be commented by separating the comment with a semicolon; otherwise, the assembler will interpret the first word of the comment as an opcode.

Any whitespace (spaces or tabs) can separate the operation field from the symbol field, the operand field from the operation field or the comment field from the operand field. Since the assembler uses spaces as its separator character, expressions used in the operand field may not contain spaces.

An optional colon can be placed after the symbol name in the first field, the label. References to the label do not contain the colon. (Note: when using multiple modules the colon exports the label. This may not be desired)

It is recommended to use semicolons to separate the comment field from the rest of the source line even if it is only optional.

The operation field of a statement consists of an assembler directive, an instruction, or a macro call. Operations may be in either upper or lower case.

Directives

The following describes the metanotation used in the directive definitions.

expr any arbitrary expression

abs_expr an absolute expression

resricted_expr may contain absolute expressions and relocatable expressions consisting only of constants and symbols previously defined within a section in the current input file.

symbol a symbol name as described above.

size one of b, w, l indicating an integer operand size of 8-, 16-, or 32-bits.

flag directive-specific modifier

string an arbitrary series of characters, sometimes delimited with quote characters.

delimited_string an arbitrary series of characters delimited with character or characters not contained in the series.

Directives for ETec374

The following is a list of directives needed for ETec374. There are more directives defined in the Introl assembler manual in the labs.

dc - Define Constant

```
[symbol]        dc[.size]       expr{,expr}
```

Generate one or more constants in the current section. The size of the constant is defined by the ***size*** modifier. If ***expr*** is a string, a constant is generated for each character in the string of that character's ASCII value. Not supported by Motorola's AS11, use fcb, fcc or fdb for ETec374 programs.

ds - Define Storage

```
[symbol]        ds{.size}       abs_expr
```

Allocate ***abs_expr*** of ***size*** in the current section at the current location. Not supported by some assemblers, use rmb for ETec374 programs.

equ - Equate

```
symbol  equ     restricted_expr
```

Assigns the value of ***restricted_expr*** to the ***symbol***. In this context, ***restricted_expr*** may contain absolute expressions and relocatable expressions consisting only of constants and symbols previously defined within the current input file.

fcb - Form Constant Byte

```
[symbol]        fcb     expr{,expr}
```

Generate one or more 8-bit constants in the current section. If ***expr*** is a string, a constant is generated for each character in the string of that character's ASCII value. Same as dc.b

fcc - Form Constant Character

```
[symbol]        fcc     delimited_string
```

The ***delimited_string*** is stored as ASCII bytes in the current section. Any delimiter can be used as long as it is not contained in the string. The first character after the directive is used as the delimiter. For ETec374 you should use either: / or ' as delimiters.

fdb - Form Double Byte

```
[symbol]        fdb     expr{,expr}
```

Generate one or more 16-bit constants in the current section. If ***expr*** is a string, a constant is generated for each character in the string of that character's ASCII value. Same as dc and dc.w

org - Orgin

```
                        org     restricted_expr
```

In this context, ***restricted_expr*** may contain absolute expressions consisting only of constants and symbols previously defined within the current input file. The assembler creates an anonymous absolute section at the address specified by ***restricted_expr***. If there was a previous org directive with the same address, the assembler overwrites the section at that address.

rmb - Reserve Memory Byte

```
[symbol]        rmb     abs_expr
```

Allocate ***abs_expr*** bytes in the current section at the current location. Same as ds.