Cross-32 Meta-Assembler

User's Guide

Version 4.0 for Windows

Data Sync Engineering
P.O. Box 539
Footbridge Lane, Building 3
Blairstown, NJ 07825
USA
Tel: (908) 362-6299
Fax: (908) 362-5889
E-mail: sales@datasynceng.com
http://www.datasynceng.com

Copyright

License

Use of this product by any person other than the purchaser or on more than one keyboard of any computer at any specific time is strictly prohibited.

Warranty

Data Sync Engineering will replace the assembler disk and/or reference manual once for a period of 90 days from the date of purchase if the original disk and/or user's guide are unreadable upon there return.

If the product does not meet the specifications published by Data Sync Engineering, return it within 30 days of our invoice date, with a written explanation of the problem, for a full refund.

Disclaimer

Data Sync Engineering disclaims all liabilities to any damages or losses that may occur through the use of its products or the breach of its warranty, other than the single replacement of the original product.

Table of Contents

Welcome

Congratulations! You have selected one of the most cost effective tools for developing assembly language programs for 4, 8, 16 and 32 bit microprocessors and microcontrollers. As supplied, Cross-32 will compile assembly language programs for over 50 different processors, and it may be expanded by the user to handle many more! But first a few words on assembly language programming in general.

An assembly language program is a text source file where the manufacturer's assembly instruction mnemonics or "memory aids" are used to directly represent the binary machine codes or opcodes that a computer actually executes. An assembler is a computer program that converts these mnemonics into their corresponding binary codes. The simple word "assembler" usually refers to a resident or self-assembler, which assembles programs for the same processor it runs on. The Microsoft Macro Assembler for the 80x86 family running MS-DOS is an example of this. A cross-assembler, assembles programs for a single target processor which differs from the host processor. Examples of these, supplied by other software houses, may be found in the more technical computer and electronic magazines.

Cross-32 from Data Sync Engineering is a meta-assembler, in that it assembles programs for numerous different target processors. It reads an assembly language source file and a corresponding assembler instruction table from disk, and writes an assembled listing and a hexadecimal or binary machine code output file. By using a flexible instruction table structure, the assembler is designed to compile assembly language source code for most microprocessors and microcontrollers with an address word of 32 or fewer bits. To further enhance its flexibility, Cross-32 will produce a machine code output file in the binary, Intel and Motorola hexadecimal formats.

As one proceeds through this manual, one will pass from the broad generalities needed to use the assembler, to the detailed specifics necessary to write a processor instruction table. If the user simply wishes to assemble code for one of the processors with a provided instruction table, then the user need not read about creating an instruction table, but should carefully study any notes provided in the example given for their target processor. To successfully tailor a provided table or create a new one, the user must study the entire manual, and perhaps several of the provided tables.

Universal Truths About This Assembler

The following are broad generalities for those already familiar with compilers and assemblers. All of the following points will be explained in greater detail in the appropriate sections of this manual.

Cross-32 is a two pass cross-assembler with an optional third pass if a phase error is detected.

All input files, output files and processor tables contain ASCII characters with each line terminated by an ASCII carriage return and line feed.

Only one processor assembly instruction or assembler directive is permitted per line.

All label declarations must be terminated with a colon or start in column 1.

All labels, expressions and operands are internally stored and manipulated using 32-bit signed integers.

Expression operators are similar in both format and precedence to the ANSI C programming language.

Blank lines in the assembly source code are reproduced in the assembly listing but otherwise ignored.

Overflow errors, undefined labels, and invalid expressions are assigned the current value of the program counter.

There are no restrictions on the character length of labels or processor instructions and all characters are significant. However, Cross-32 input lines must not exceed 255 characters in length.

Cross-32 does not support, need, or include a linker or librarian.

Installing the Assembler

The Cross-32 Meta-Assembler for Windows Version 4.0 is supplied on CDROM, containing a 16-bit version for Windows 3.1 and later, and a 32-bit version for Windows 95 and later. To install either version, insert the CDROM, the root directory contains the Windows 95 version, the subdirectory C32W31 contains the Windows 3.1 version. From Windows run:

```
D:\SETUP
```

where **D:** is the name of the CDROM drive. After that, simply follow the instructions

To run the program, double click on the Cross-32 icon.

The assembler includes the Cross-32 Windows program (C32W4Wxx.EXE), a DOS command line version called (C32D4CL.EXE), and numerous processor tables (cpu.TBL) with corresponding example source files (Ecpu.ASM). The installer will copy all of these files to your disk, but many users only require a couple of files. For example, if you only wish to assemble 8051 code with the Windows 95 assembler, you only need C32W4W95.EXE, C32W4.HLP, C32W4.INI, 8051.TBL and E8051.ASM in a directory on disk. The other files may be deleted. The C32W4.HLP file contains the help information. The C32W4.INI contains the names and positions of the open files on the Cross-32 desktop, and will restore itself if deleted.

Please read the Ecpu.ASM file for your target processor before writing your own programs.

Running the Assembler

To run the assembler, double-click the new group icon called "Universal Cross-Assemblers" to open it. Then double-click on the program-item icon named "Cross-32 W3.1". (The icon looks like a 3.5" diskette on top of a PLCC chip.)

Users the Windows 95 may run Cross-32 via the following menu path:

```
Start
Programs
Cross-32 W95
```

The assembler's integrated development environment will fill the center of the screen.

Running the MS-DOS command line version Cross-32 is detailed later in this manual.

Uninstalling the Assembler

When Cross-32 is installed, all files are placed in the directory specified by the user, example C:\WINC32_4. The installer does not change any operating systems files or settings. Therefore simply deleting the Cross-32 directory and group will completely remove the application from your system.

Users of the Windows 95 version should uninstall Cross-32 automatically via the following menu path:

```
Start
Settings
Control Panel
Add/Remove Programs
```

The uninstaller may complain that it was unable to remove some files. These are the user's files, and may be removed manually.

Integrated Environment Assembler

The assembler provides an integrated development environment (IDE) with everything needed to write and assemble your programs. There are four visible components of the IDE, the menu bar at the top providing access to menu commands, the toolbar just below it with icon buttons to access menu commands, the desktop in the middle containing windows, dialog boxes and empty space, and a status bar at the bottom providing relevant information. Data Sync Engineering assumes that the user is familiar with common Windows procedures and will only describe them briefly in the following text.

The menu bar is the primary access to all the assembler commands, which drop down when a menu is selected. The menu bar is a series of menu titles at the top of the screen that look like:

File Edit Search Assemble Options Window Help

To select a menu command from the keyboard, press the <F10> or <Alt> key to activate the menu bar. Then choose a menu title by pressing the underlined letter in that menu, or using the left and right arrow keys and pressing <Enter>. Once the menu commands have dropped down, choose a command by pressing an underlined letter, or using the up and down arrow keys and pressing <Enter>. The drop down menus also contain a number of shortcut keys, such as Ctrl+P for print. They can be used to activate frequently used commands without using the menus. The layout of both the menus and shortcuts was inspired by a number of Windows editors, such as Microsoft Word.

To select a menu command with the mouse, click the desired menu title and click the desired command when the menu drops down. Or, drag straight down from the menu title to the desired menu command, and release the mouse button. A menu command that ends in "..." indicates that a dialog box will appear before the command executes.

File Menu

The File menu provides commands for creating new files, opening existing files, saving files, printing files, and exiting the application.

New	Create a new, untitled, file window.
Open	Open an existing file.
Close	Close the current file window.
Save	Save the current file if it has changed.
Save as	Save the current file under a new name.
Save all	Save all open modified files
Print	Print the current file.
Print preview	View a sample printout of the current file.
Print setup	Set printer characteristics.
1, 2, ..., 8 file	Most recently used files list.
Exit	Exit Cross-32.

File | New Ctrl+N Command

Opens and activates a new untitled file window, the contents of which are stored in a temporary buffer.

The assembler will prompt you to name and save the file when closing the window or exiting the assembler.

File | Open Ctrl+O Command

Displays the Open a File dialog box to select a file to load into a new file window. One can also create a new file by naming a file that doesn't currently exist.

File | Close Command

Closes the currently active window.

The assembler will prompt you to save the file if it has been modified.

File | Save Ctrl+S Command

The File | Save command saves the file in the active window to disk. If the file is unnamed, the Save File As dialog box is displayed so you can name the file, and choose where it is to be saved.

File | Save as Command

The File | Save as command allows you to save a file under a new name, or in a new location on disk. The command displays the Save File As dialog box. You can enter the new file name, including the drive and directory. All windows containing this file are updated with the new name. If you choose an existing file name, you are asked if you want to overwrite the existing file.

File | Save all Command

The File | Save all command saves all open, modified files to disk. If the file is unnamed, the Save File As dialog box is displayed so you can name the file, and choose where it is to be saved.

File | Print Ctrl+P Command

The File | Print command opens the Windows Print dialog box, to print the file in the active window. Use File | Print preview to see how the file will be laid out on printer pages. Use File|Page setup to set page margins, select a printer, and set printer options.

File | Print Preview Command

File | Print Preview opens a special window that shows how the active file will appear when printed. The preview window shows one or two pages of the active file, as they would be laid out on printer pages. Controls on the window allow you to page through the pages of the file.

File | Page Setup Command

The File | Page Setup command displays the Windows Page Setup dialog box which lets the user format the printed page and select and configure the printer to be used to print files in the application.

File | 1 2...8 Most Recently Used Files

The File | 1 2...8 command allows the user to quickly open the most recently used files. After the file menu pops up, select the desired file with the mouse or by typing the corresponding number.

File | Exit Alt+F4 Command

The File | Exit command allows the user to quit the assembler. The assembler will prompt the user to save any modified files before quitting.

Edit Menu

The Edit menu provides commands to undo edits, access the clipboard, and to delete text.

Undo	Undo the previous editing operation.
Cut	Move selected text to the clipboard.
Copy	Copy selected text to the clipboard.
Paste	Place text from the clipboard at the cursor.
Delete	Delete selected text in the current.
Select all	Highlights all the text in the current window.

Edit | Undo Ctrl+Z Command

The Edit | Undo command restores the file in the current window to the way it was before your most recent edit operation. Undo inserts any characters you deleted, deletes any characters you inserted, replaces any characters you overwrote, and moves your cursor back to its previous position.

The Undo command buffer saves all changes since the most recent file save. Remember that all files are saved each time the Assemble command is called.

Edit | Cut Ctrl+X Command

The Edit | Cut command removes the selected text from the current file window and places it in the clipboard. Choose Edit | Paste to place the cut text at the current cursor location. The text remains in the clipboard until another copy or cut command is executed, and can be pasted at the cursor any number of times.

Edit | Copy Ctrl+C Command

The Edit | Copy command leaves the selected text intact and places an exact copy of it in the clipboard. To paste the copied text into another file window, choose Edit | Paste. The text remains in the clipboard until another copy or cut command is executed, and can be pasted at the cursor any number of times.

Edit | Paste Ctrl+V Command

The Edit | Paste command inserts the text currently selected in the clipboard into the current window at the cursor position. If text is selected in the current window, it is replaced by the text in the clipboard. The text remains in the clipboard until a copy or cut command is executed, and can be pasted at the cursor any number of times.

Edit | Delete Del Command

The Edit | Delete command deletes the currently selected text, or the character to the right of the cursor if no text is selected. The text is not placed in the clipboard, and cannot be retrieved with the Edit | Paste command. Use the Edit | Undo command to restore deleted text.

Edit | Select all Ctrl+A Command

The Edit | Select all command highlights the entire contents of the file in the active window. The cursor is moved to the end of the file. This command is often used to paste an entire file into another, with the Copy and Paste commands.

Search Menu

The Search menu provides commands to find and replace text, and go to assembly errors.

Find	Find a pattern of text.
Replace	Replace found pattern of text.
Next	Find the next occurrence of a text pattern.
Previous error	Find the previous assembly error
Next error	Find the next assembly error

Search | Find Ctrl+F Command

The Search | Find command searches the current document for a text pattern. The command displays the Find dialog, which controls the search process. Options in the dialog determine whether only whole words are to be matched, whether the case of characters is significant, and whether the search should be conducted forwards or backwards through the document. As each match is found, it is highlighted in the document.

Search | Replace Ctrl+H Command

The Search | Replace command searches the current document for a text pattern, and replaces occurrences of the pattern with new text. The command displays the Replace dialog box, which controls the search/replace process. Options in the dialog determine whether only whole words are to be matched, and whether the case of characters is significant. The dialog is also used to specify the pattern to search for, and the text to replace occurrences with.

Search | Next F3 Command

The Search | Next command repeats the last Find or Replace operation.

Search | Previous Error Alt+F7 Command

The Previous error command moves the cursor to the location of the previous assembly error. The command only functions if assembly errors have been found with the Assemble command.

The assembler will open the file containing the error if it is not already open.

Search | Next Error Alt+F8 Command

The Next error command moves the cursor to the location of the next assembly error. The command only functions if assembly errors have been found with the Assemble command.

The assembler will open the file containing the error if it is not already open.

Assemble Menu

The Assemble menu has the following commands:

Assemble	Begin assembly with the Main file
Main	Choose the Main file

Assemble | Assemble Command

Assembles your source code starting with the Main file. The Main file is chosen with the Main command. If a Main file has not been selected, the assembler will ask you to choose one.

If any errors occur, the cursor will be placed at the first one. Other errors may be found using the Previous error and Next error commands.

Assemble | Main Command

Activates the file open dialog box so you may choose the Main File.
The Main file is where the assembler starts its compilation.

Options Menu

The Options menu has the following commands:

Font...	Set the font used by the editor and printer
Tabs...	Set the tab length used by the editor and printer
Save desktop	Save the desktop to disk
Restore desktop	Restore the desktop from disk

Options | Font Command

Opens the windows Font dialog box. A font may be selected that is used for all file windows and printing.

Assembly code is often written in a fixed pitch font, such as Courier or Fixedsys. If your code appears staggered on the screen, try one of these. The assembler will display a size 10 Courier font when it is first installed.

The tab size may be set with the Tab Size command. The font and tab settings are saved when the assembler is exited.

Options | Tabs Command

Opens a dialog box that allows the user to set the tab length from 2 to 16 characters. The editor uses the average character width when spacing variable pitch fonts. Actual characters may be wider or narrower than this distance. Fixed pitch fonts such as Courier or Fixedsys often work best for assembly language programming. The font, style and size may be set with the Font command.

The assembler is set to a tab length of 8 characters when it is installed.

The font and tab settings are saved when the assembler is exited.

Options | Save Desktop Command

Save the filenames of open windows, main file name and other assembler settings to disk.

These setting may be restored using the Restore desktop command.

The assembler automatically saves and restores desktop settings in file C32W4.INI when the assembler is run and exited.

Options | Restore Desktop Command

Restores the open file windows, main file name and other assembler settings from disk.

These setting are usually saved using the Save desktop command.

The assembler automatically saves and restores the desktop in file C32W4.INI when the assembler is run and exited.

Window Menu

The Window menu provides commands to control the position and layout of application's windows.

Cascade Resize and position all windows in a diagonal overlapping pattern.
Tile horizontal Resize and position all windows in a horizontal non-overlapping pattern.
Tile vertical Resize and position all windows in a vertical non-overlapping pattern.

Arrange icons Align all iconized windows along a grid.
Close all Close all file windows.

1, 2... Activate one of the currently open file windows.

Window | Cascade Shift+F5

The Window | Cascade command piles all file windows from the top-left of the application's main window in an overlapping pattern so that the all title bars are visible. The currently active file window is fully visible on the top of the pile.

Window | Tile Horizontal Shift+F4

The Window | Tile horizontal command arranges all file windows from top to bottom in a non-overlapping pattern. The currently active file window is at the top of the screen.

Window | Tile Vertical Command

The Window | Tile vertical command arranges all file windows from side to side in a non-overlapping pattern. The currently active file window is at the left of the screen

Window | Arrange Icons Command

The Window | Arrange Icons command arranges all iconized windows into rows along the bottom of the application's main window.

Window | Close All Command

The Window | Close All command closes all document windows open in the application.

Window | 1, 2...8 Select Window Command

The Window | 1, 2...8 Select Window command chooses the active window from a list of open file windows. The currently active file window has a check mark in front of its number. The window filenames are listed in the order that they were opened.

Help Menu

The Help menu provides access to the help system and the about dialog box.

Contents Cross-32 Help's table of contents.
Context Contextual help system.
Index Cross-32 Help's index.
Using Using Help's table of contents.

About Information on the assembler.

Help | Contents F1 Command

The Help | Contents command displays the Table of Contents for the Cross-32 Help System. Click on any underlined item to proceed further.

Help | Context Ctrl+F1 Command

The Help | Context command displays the help cursor, or goes directly to the help system if an assembly error has been found. The help cursor may be clicked on any toolbar button or pull down menu to get further information. When obtaining information on a menu item, the mouse button should not be released until the pointer is on the desired command.

Help | Index Command

The Help | Index command displays the index for the Cross-32 Help System. Enter a subject to proceed further.

Help | Using Command

The Help | Using command displays the contents section of the using and customizing help Windows help system.

Help | About Command

Display the about dialog box with some information on the application. Press OK to close the box and continue.

The Toolbar

The Toolbar is a row of buttons at the top of the main window, which represent application commands. This is a dockable toolbar, and may be dragged and sized to any location with the mouse. Clicking one of the buttons is a quick alternative to choosing a command from the menu. Buttons on the toolbar activate and deactivate according to the state of the application.

Command Line Assembler

A DOS command line version of the Cross-32 Meta-Assembler is provided for those who desire it. Not having, an integrated development environment, it may be used by those who wish to use an editor other than the one provided, or who may be running out of memory when assembling very large programs.

To use the command line version, type:

```
C32D4CL source [-L list] [-H hex] [-E error]
```

where the square brackets [] indicate optional items.

The "-L" instructs Cross-32 to produce an assembly list file using the following file name. The "-H" tells Cross-32 to produce a hex file using the following file name. The "-E" tells Cross-32 to produce an error output file using the following file name. If these are omitted the corresponding files will not be produced. The format of the hex file is set using the HOF directive. Errors will be displayed on the screen if an error file is not requested. The order in which the source, listing and hex files are specified in the command line does not matter. Disk and directory names may be included in the file names.

Some examples are:

```
C32D4CL E8051.ASM

C32D4CL E8051.ASM  -H E8051.HEX  -E E8051.ERR

C32D4CL E8086.ASM  -H \BIN\E8086.COM
```

Execution of the command line version of Cross-32 may be aborted any time by pressing the <Ctrl+C> or <Ctrl+Break> keys.

Assembler Source File

The source file is the ASCII assembler source code to be assembled by Cross-32. From the source file, the processor instruction table is selected using the CPU directive, and the format of the hexadecimal file is set using the HOF directive.

Line Format

The source file is the ASCII assembler source code to be assembled by Cross-32. From the source file, the processor instruction table is selected using the CPU directive, and the format of the hexadecimal file is set using the HOF directive.

Only one assembly instruction or assembler directive is permitted per line. The assembly line is free format, meaning that following fields do not have to appear at specific columns. Each line may contain some or all of the following sequence of identifiers:

```
line#    label:    operation          operand(s)
;comment
```

Where...

Line# is an optional decimal integer in the range of 0 to 65535 created by some editors representing the source code line number. If present, Cross-32 will ignore this field except to reproduce it in the listing.

Label is a phrase starting with an alphabetic ASCII character "A-Z" or a "_", "." or "?", which is assigned the present value of the program counter or other user specified value. Characters within the label must be alphanumeric "A-Z", "0-9" or an "_", "." or "?". The first character of a label must be located in column one of the file, or the label must end with a colon.

Operation is a Cross-32 assembler directive, or processor assembly instruction defined in the instruction table. All operations must start with an alphabetic character "A-Z".

Operands are registers, labels, constants or expressions representing integer values in the range of -2,147,483,648 to 2,147,483,647 and/or character strings. They may be embodied within an assembly language instruction.

Comment is a statement following a semicolon ";" usually used to describe the assembly language program. The ";" may be placed anywhere on the assembly line and the assembler ignores all characters following it on that line.

Labels

A label is an alphanumeric series of characters representing an integer in the range -2,147,483,648 to 2,147,483,647. The label field must start with an alphabetic ASCII character "A-Z" or a "_", "." or "?", even when used with the EQU assembler directive. Characters within the label must be alphanumeric "A-Z, 0-9" or an "_", "." or "?". The first character of a label must be located in column one of the file, or the label must end with a colon. Except for the EQU and SETL directives, a label is optional, and assigned the current value of the program counter. Labels may be of any character length, except that a Cross-32 input line cannot exceed 255 characters, and all characters are significant. Cross-32 makes no distinction between upper and lower case characters. A label may also stand alone on a line, in which case it will be assigned the current value of the program counter. A dollar sign "$" may be used as an operand representing the current value of the program counter.

The following examples use the equate (EQU) directive described later in the manual. Very simply, the equate (EQU) directive assigns the label on it's left, the value of the expression on its right.

```
;Valid examples of labels are:
;
STRT1:   EQU       1234H               ;Label on
directive
LD_UP:  EQU     1234H           ;Label with "_"
ABCD:   EQU     _STRT1          ;
alone:                              ;Stand alone
label
```

Integer Constants

An integer constant is a series of ASCII digits representing a 32 bit signed integer in one of several number bases. Cross-32 supports the following three numeric constant formats:

1) C programming language (i.e. 0377 = 255 = 0xff)

If the first digit is a zero, the integer is taken to be octal, and the remaining characters must be "0-7". If the first digit is a zero, immediately followed by an "x", the integer is taken to be hexadecimal, and the following digits must be "0-9" or "A-F". Otherwise, the constant is taken to be decimal, and all the digits must be in the range of "0-9". The valid bases with their corresponding C language identifiers and character ranges are as follows:

```
Leading                                         Base
Characters
-----------------------------------------------
----
  0                 Octal                 Base   8
0-7
  1-9                Decimal              Base  10
0-9
  0x           Hexadecimal      Base 16       0-9,
A-F
```

WARNING: Do not place unnecessary leading zeros at the beginning of decimal numbers, the assembler will interpret them to be C octal numbers and may not generate the value expected (0255 is not equal to 255, but 0255D = 255). If the decimal number contains an "8" or "9" an error will be generated, otherwise no warning will be given.

2) Trailing alphabetic (i.e. 011111111B = 377Q = 255D = 0FFH)

A trailing alphabetic character indicates the base of integer constant. All constants must start with a numeric digit "0-9". The default base is base 10. Both "O" and "Q" may be used to specify octal numbers to avoid confusion between "0" and "O". The valid bases with their corresponding trailing alphabetic characters and character ranges are as follows:

```
Trailing                                    Base
Characters
-----------------------------------------------
----
  B                 Binary               Base   2
0-1
  O                 Octal                Base   8
0-7
  Q                 Octal                Base   8
0-7
  D                 Decimal              Base  10
0-9
 H            Hexadecimal      Base 16       0-9,
A-F
```

WARNING: Hexadecimal integers must start with a numeric (0-9) constant! "FFH" is not a valid integer, and will probably be flagged as an undefined label. "0FFH" is a valid integer.

3) Leading dollar sign (i.e. 255 = $FF)

If the first digit is a dollar sign "$", the integer is taken to be hexadecimal, and the following digits must be "0-9" or "A-F". The dollar sign by itself represents the current value of the program counter.

```
Leading                                     Base
Characters
-----------------------------------------------
----
 $            Hexadecimal        Base 16    0-9,
A-F
```

```
;Valid examples of numeric constants are:
;
BIN1:   EQU     10101111B       ;Binary
OCT1:   EQU     377O            ;Octal
OCT2:   EQU     74Q             ;Octal
DEC1:   EQU     -1              ;Decimal
DEC2:   EQU     10D             ;Decimal
HEX1:   EQU     0FFH            ;Hexadecimal
HEX2:   EQU     0x0B            ;Hexadecimal
HEX3:   EQU     0x0f            ;Hexadecimal
HEX4:   EQU     0x0D            ;Hexadecimal
```

String Constants

String constants "string" consist of a series of ASCII characters between two quotation marks ("). Cross-32 will convert these constants to a hexadecimal representation of their ASCII values in the listing. Lower case characters within string constants will be represented as such. A quotation mark (") cannot appear within a character string, for it will be interpreted as being the end of that string. If a quotation mark is needed, Data Sync Engineering recommends that an apostrophe be used ('). The user may also terminate the string, insert the binary value of a quotation mark (22H), and then start another string. A string constant may also be used as an operand where applicable. In a DFB statement, a string constant may be of any length, bearing in mind that an assembly source line must not exceed 255 characters in length. When used as an operand, or in the DWM, DWL, DFL or EQU directives, an error will be flagged if the string constant exceeds the length of the operand specified by the assembler directive. A string constant cannot be extended beyond its present line without terminating the string with a quotation mark. Although only the first five or seven bytes of the string constant are shown in the listing, it is placed the machine code file in its entirety.

The following examples use the define byte (DFB) directive as described later in this manual. Very simply, the define byte (DFB) directive places the byte by byte value of the expressions on its right into the hex file, starting at the present memory location shown on the far left of the listing.

```
;Valid examples of string constants are:
;
        DFB "Yea, yea, yea!"
        DFB "AB","ab"
        DFB "The dog Said 'Woof Woof!'"
```

Arithmetic and Logical Expressions

The assembler will accept arithmetic and logical expressions made up of labels, integer constants, script brackets {} and operators. An arithmetic operator results in a 32 bit signed integer value, a logical operator yields only a true or false, 1 or 0 respectively. Most of the operators and their precedence are taken from the ANSI C programming language. A list of operators follows, grouped in decreasing precedence, where x and y represent integer values:

```
    $           Present value of program counter
   { }          script brackets

    !  y        Logical negation
    ~  y        Ones complement
    -   y            Unary subtraction or twos
complement
    +  y        Unary addition
  INV  y        Reversed (INVerted) byte order

 x  *  y        multiplication
 x  /  y        division
  x   %   y            remainder after division
(modulus)

 x  +  y        addition
 x  -  y        subtraction

 x  << y         Left shift of x by y bits (y <
32)
 x  >> y        Right shift of x by y bits (y <
32)

 x  <  y        Less than
 x  <= y        Less than or equal to
 x  >  y        Greater than
 x  >= y        Greater than or equal to

 x  == y        Equal to
 x  != y        Not equal

 x  &  y        Bitwise AND of x and y

 x  ^  y        Bitwise XOR of x and y

 x  |  y        Bitwise OR of x and y

 x  && y        Logical AND of x and y

 x  || y        Logical OR of x and y


;Arithmetic expressions grouped in decreasing
;precedence:
EX01:   EQU     $               ;Program Counter
;
EX02:   EQU     4*{"A"+26}      ;Script Brackets
;
EX03:    EQU       !15                  ;Logical
negation
EX04:    EQU       ~15                   ;One's
complement
EX05:    EQU       -15                   ;Two's
complement
EX06:   EQU     +15             ;Unary addition
EX07:   EQU     INV 12345678H   ;Invert
;
EX08:   EQU     0xfe * 16       ;Multiplication
EX09:   EQU     0xfe / 16       ;Division
EX10:    EQU       0xfe  %  16          ;Remainder
(Modulus)
;
EX11:   EQU     40 + 20D        ;Addition
EX12:   EQU     40 - 20D        ;Subtraction
;
EX13:   EQU     1234H << 8      ;Left Shift
EX14:   EQU     1234H >> 8      ;Right Shift
;
EX15:   EQU     0 <  2          ;Less than
EX16:   EQU     0 <= 2          ;Less than or =
EX17:   EQU     0 >  2          ;Greater than
EX18:   EQU     0 >= 2          ;Greater than or
=
```

```
;
EX19:   EQU     0 == 2          ;Equal to
EX20:   EQU     0 != 2          ;Not equal
;
EX21:   EQU     "3" & 15        ;Bitwise AND
;
EX22:   EQU     10B ^ 3         ;Bitwise XOR
;
EX23:   EQU     2 | 253         ;Bitwise OR
;
EX24:   EQU     0 && 2          ;Logical AND
;
EX25:   EQU     0 || 2          ;Logical OR

; A little arithmetic
EX26:   EQU     8 | 7 ^ 16 & 15 << 13 - 135/~{1-
17}
```

Assembler Directives

Assembler directives or pseudo-operations are not usually translated into machine language instructions the way mnemonics are. These instructions are directives to the assembler itself. They assign the program to certain areas in memory, designate areas of RAM for variable storage, define labels and constants, and perform other housekeeping functions. Cross-32's assembler directives are:

ALGN - Align Directive

The align (ALGN) directive may be used to align the hexadecimal output file to a specified word length. This directive is useful to insure that code and data begins on even addresses in processors such as the 68000 and TMS320xx. The ALGN directive has the following syntax:

```
label:  ALGN    expression      ;comment
```

The word length specified in the expression must be within a range of one to ten bytes. The assembler will insert bytes with a value zero until the code is aligned. These bytes will be seen in both the list and hex files.

```
;Valid examples of the ALGN directive are:
;
        ALGN    1               ;Default value
        DFB     "ABC"
        ALGN    2               ;Align to 2 byte
word
        DFB     "ABC"           ;Notice extra FF
;
          ALGN     1                  ;Return to
Default
```

ALIAS - Alias Directive

The alias (ALIAS) directive may be used to rename any assembler directive, including the ALIAS directive. This will help to make Cross-32 more compatible with other assemblers. It is most often used when porting existing code to Cross-32. The ALIAS directive has the following syntax:

```
label:  ALIAS   "new-directive" ;comment
```

The **label** is the old directive, and the **new-directive** must be surrounded with double quotation marks ("). The new directive must follows the rules for a label. It must start with alphabetic ASCII character "A-Z" or a "_", "." or "?". Characters within the label must be alphanumeric "A-Z", "0-9" or an "_", "." or "?". The colon is not included as part of the new directive.

```
; Make Cross-32 more like Intel assemblers
;
DB      ALIAS  "DFB"
        DB      12h
;
dw      alias   "dwl"           ;
        dw      1234h
```

CASE - Case Directive

The CASE directive allows the user to make the assembler case sensitive; meaning that it differentiates between upper and lower case characters ("the" is not the same as "THE"). The directive must be followed with the words "ON" or "OFF", in double quotes. "ON" makes the assembler case sensitive. This includes all mnemonics, directives and labels. Directives must be uppercase, unless they are changed with an ALIAS or RPTXT directive. The default "OFF", and previous versions of Cross-32, are case insensitive. A label may be placed before the CPU directive, which will be assigned the current value of the program counter.

The CASE directive has the following syntax:

```
Label:  CASE    "switch"        ;comment
```

If used, the CASE directive is normally placed once near the beginning of the first assembly source file. The ST9 processor

is one chip that requires this directive. In this instance, it must be placed before the CPU directive, to assemble properly.

```
;Examples of the CASE directive
;
        CASE    "OFF"           ;Default
        CASE    "ON"            ;Case sensitive
```

CPU - Table Declaration Directive

The CPU directive tells Cross-32 which processor instruction table is to be loaded during assembly. The CPU directive has the following syntax:

```
Label:  CPU     "cpu_file_name" ;comment
```

Only the instruction table file name may be specified after the CPU directive. A disk drive and/or directory may be included in the file specification. A label may be placed before the CPU directive, which will be assigned the current value of the program counter. The instruction table is only read once by Cross-32 during assembly, and all subsequent CPU directives will be ignored. An invalid CPU file name will result in a fatal error.

```
;Examples of the CPU directive
;
        CPU     "1802.TBL"      ;CPU TABLE
        CPU     "B:8048.TBL"    ;CPU TABLE
```

DFB - Define Byte Directive

The define byte (DFB) directive allows the user to define the value of storage areas on a byte by byte basis. The DFB directive has the following syntax:

```
label:     DFB           expr1,expr2,...,expr(n)
;comment
```

Except for a string constant, the result of each expression must represent an 8-bit value (-128 to 255) or an error will be flagged. An expression may consist of a numeric constant, a string constant, a label, or a formula. There is no limit on the number of bytes that may be defined using a single DFB directive, except that the length of the source line must not exceed 255 characters. Although only the first few bytes of this data will be shown in the listing, it is included in the hex file in its entirety.

```
;Valid examples of the DFB directive are:
;
CONST1: DFB     12H             ;Hexadecimal
        DFB     -1,2,3          ;Integers
        DFB     CONST1 / 8      ;Label
        DFB     77Q + 9         ;Expressions
NOTE:   DFB     "Hello",0       ;ASCII string
        dfb     3,,5            ;Null
```

DFDF- Define Double Float

The define double floating point number (DFDF) directive allows the user to define the value of storage areas with 64-bit IEEE real format (double precision) floating point numbers. These numbers must be numeric constants, expressions are not supported. There is no limit on the number of floating point numbers that may be defined using a single DFDF directive, except that the length of the source line must not exceed 255 characters. Although only the first few bytes of this data will be shown in the listing, it is included in the hex file in its entirety.

```
label:      DFDF      float1,float2,...,float(n)
;comment
```

Each floating-point number must be made up of the following:

An optional string of tabs and spaces

An optional sign

A string of digits and an optional decimal point (the digits can be on both sides of the decimal point)

An optional e or E followed by an optional signed integer

The characters must match this generic format:

```
[whitespace]    [sign]    [ddd]    [.]    [ddd]
[e|E[sign]ddd]
```

The double precision (64-bit) numbers are stored in the following format:

```
  SEEEEEEE EEEEMMMM MMMMMMMM ... MMMMMMMM

    Sign - Most significant bit
Exponent - Next 11 bits, biased by 03FF
Mantissa -  remaining 52 bits, most significant
bit
           is implied (not shown)
```

Examples of the DFDF directive are:

```
        DFDF     1.7E-308       ;Min
        DFDF     1.7e+308       ;Max
        DFDF     0.0
        DFDF     3.141592654
```

```
DFDF     1.0, 2.0, -4.0
```

DFF - Define Floating Point Number

The define floating point number (DFF) directive allows the user to define the value of storage areas with 32-bit IEEE real format (single precision) floating point numbers. These numbers must be numeric constants, expressions are not supported. There is no limit on the number of floating point numbers that may be defined using a single DFF directive, except that the length of the source line must not exceed 255 characters. Although only the first few bytes of this data will be shown in the listing, it is included in the hex file in its entirety.

```
label:      DFF       float1,float2,...,float(n)
;comment
```

Each floating-point number must be made up of the following:

An optional string of tabs and spaces

An optional sign

A string of digits and an optional decimal point (the digits can be on both sides of the decimal point)

An optional e or E followed by an optional signed integer

The characters must match this generic format:

```
[whitespace]    [sign]    [ddd]    [.]    [ddd]
[e|E[sign]ddd]
```

The single precision (32-bit) number is stored in the following format:

```
 SEEEEEEE EMMMMMMM MMMMMMMM MMMMMMMM
    Sign - Most significant bit
Exponent - Next 8 bits, biased by 07FH
Mantissa - remaining 23 bits, most significant
bit
           is implied (not shown)
```

Examples of the DFF directive are:

```
        DFF      3.4E-38         ;Min
        DFF      3.4e+38         ;Max
        DFF      0.0
        DFF      3.141592654
        DFF      1.0, 2.0, -4.0
```

DFL - Define Long Integer (MSB First)

The define long integer (DFL) directive allows the user to define the value of storage areas on a long integer or long word basis (a long integer is 32 bits or four bytes). There is no limit on the number of long integers that may be defined using a single DFL directive, except that the length of the source line must not exceed 255 characters. Although only the first few bytes of this data will be shown in the listing, it is included in the hex file in its entirety.

```
label:     DFL            expr1,expr2,...,expr(n)
;comment
```

An expression may consist of a numeric constant, a string constant, a label, or a formula. Although ASCII string constants may be used, an error will be flagged if they exceed 4 characters. The DFL directive stores 32 bit signed integers from the most significant byte (MSB) to the least significant byte.

```
;Valid examples of the DFL directive are:
;
CONST2: DFL     12345678H       ;Hexadecimal
        DFL     -1,2,3          ;Integers
        DFL     CONST2          ;Label
        DFL     77Q + 9         ;Expressions
        DFL     "ABCD"          ;ASCII string
        DFL     3,,5            ;Null
```

DFLDF - Define Long Double Float

The define long double floating point number (DFLDF) directive allows the user to define the value of storage areas with 80-bit IEEE 754 extended real format (long double precision) floating point numbers. These numbers must be numeric constants, expressions are not supported. There is no limit on the number of floating point numbers that may be defined using a single DFLDF directive, except that the length of the source line must not exceed 255 characters. Although only the first few bytes of this data will be shown in the listing, it is included in the hex file in its entirety.

```
label:      DFLDF     float1,float2,...,float(n)
;comment
```

Each floating-point number must be made up of the following:

An optional string of tabs and spaces

An optional sign

A string of digits and an optional decimal point (the digits can be on both sides of the decimal point)

An optional e or E followed by an optional signed integer

The characters must match this generic format:

```
[whitespace]    [sign]    [ddd]    [.]    [ddd]
[e|E[sign]ddd]
```

The long double precision (80-bit) numbers are stored in the following format:

```
  SEEEEEEE   EEEEEEEE   EMMMMMMM   MMMMMMMM   ...
MMMMMMMM

    Sign - Most significant bit
Exponent - Next 16 bits, biased by 07FFFH
Mantissa - remaining 63 bits, most significant
bit
           is implied (not shown)
```

Examples of the DFLDF directive are:

```
        DFLDF   3.4E-4932       ;Min
        DFLDF   1.1e+4932       ;Max
        DFLDF   0.0
        DFLDF   3.141592654
        DFLDF   1.0, 2.0, -4.0
```

DLL - Define Long Integer (LSB First)

The define long integer (DLL) directive allows the user to define the value of storage areas on a long integer or long word basis (a long integer is 32 bits or four bytes). There is no limit on the number of long integers that may be defined using a single DLL directive, except that the length of the source line must not exceed 255 characters. Although only the first few bytes of this data will be shown in the listing, it is included in the hex file in its entirety.

```
label: DLL    expr1,expr2,...,expr(n)  ;comment
```

An expression may consist of a numeric constant, a string constant, a label, or a formula. Although ASCII string constants may be used, an error will be flagged if they exceed 4 characters. The DLL directive stores 32 bit signed integers from the least significant byte (LSB) to the most significant byte.

```
;Valid examples of the DLL directive are:
;
CONST2: DLL     12345678H       ;Hexadecimal
        DLL     -1,2,3          ;Integers
        DLL     CONST2          ;Label
        DLL     77Q + 9         ;Expressions
        DLL     "ABCD"          ;ASCII string
        DLL     3,,5            ;Null
```

DFS - Define Storage Directive

The define storage (DFS) directive may be used to reserve a section of memory with unspecified contents during assembly. This directive is often used to reserve an area of RAM or volatile memory for the target system. The DFS directive has the following syntax:

```
label:  DFS     expression      ;comment
```

The expression can be of any form that represents a 32-bit positive integer value, but only one expression is allowed. The value of the expression is added to the program counter and assembly continues. Except for the changed value of the program counter, no values are written to the hex file.

```
;Valid examples of the DFS directive are:
;
STOR1:   DFS      1                 ;Reserve for
byte
STOR2:   DFS      2                 ;Reserve for
word
         DFS      4                 ;Reserve for
long
        DFS     8 * {4}         ;Expression
        DFS     12H             ;Hexadecimal
          DFS      "A"                ;character
string
```

DWL - Define Word (LSB First)

The define word (DWL) directive allows the user to define the value of storage areas on a word by word basis (one word is two bytes). There is no limit on the number of words that may be defined using a single DWL directive, except that the length of the source line must not exceed 255 characters. Although only the first few bytes of this data will be shown in the listing, it is included in the hex file in its entirety. The DWL directive has the following syntax:

```
label: DWL      expr1,expr2,...,expr(n) ;comment
```

The result of each expression must represent a 16 bit integer value (- 32768 to 65535) or an error will be flagged. An expression may consist of a numeric constant, a string constant, a label, or a formula. Although ASCII string constants may be used, an error will be flagged if they exceed 2 characters in length. The DWL directive will store the least significant byte (LSB) of the 16-bit value before the most significant byte.

```
;Valid examples of the DWL directive are:
;
CONST3: DWL     1234H           ;Hexadecimal
        DWL     -1,2,3          ;Integers
        DWL     CONST3          ;Label
        DWL     77Q + 9         ;Expressions
        DWL     "AB"            ;ASCII string
        DWL     3,,5            ;Null
```

DWM - Define Word (MSB First)

The define word (DWM) directive allows the user to define the value of storage areas on a word by word basis (one word is two bytes). There is no limit on the number of words that may be defined using a single DWM directive, except that the length of the source line must not exceed 255 characters. Although only the first few bytes of data will be shown in the listing, it is included in the hex file in its entirety. The DWM directive has the following syntax:

```
label:     DWM            expr1,expr2,...,expr(n)
;comment
```

The result of each expression must represent a 16 bit integer value (- 32768 to 65535) or an error will be flagged. An expression may consist of a numeric constant, a string constant, a label, or a formula. Although ASCII string constants may be used, an error will be flagged if they exceed 2 characters in length. The DWM directive will store the most significant byte of the 16-bit value before the least significant byte.

```
;Valid examples of the DWM directive are:
;
CONST4: DWM     1234H           ;Hexadecimal
        DWM     -1,2,3          ;Integers
        DWM     CONST4          ;Label
        DWM     77Q + 9         ;Expressions
        DWM     "AB"            ;ASCII string
        DWM     3,,5            ;Null
```

END - End of Source Program

The end of assembly (END) directive is optional, but when used it will be the last line of the assembly source file assembled (the remainder being ignored). This directive has the following syntax:

```
label:  END     expression      ;comment
```

The expression is optional, but will represent any positive 16 or 24 bit integer value, depending on the hexadecimal output format in use. When an expression is given, its value will be included as the address in the final line of the Intel or Motorola hex file, representing the starting address of the assembly program. If the END directive or expression is not included, this starting address will default to zero.

```
;Valid examples of the END directive are:
;
          END           ;Simple format
THE_END:  END  RESET    ;With starting address
          END   0100H         ;Famous  starting
address
```

EQU - Equate Label Directive

The equate (EQU) directive may be used to assign an integer value to a label. The EQU directive has the following syntax:

```
label:  EQU     expression      ;comment
```

The label and the expression are not optional in this directive. The expression may consist of any numeric constant, character string or formula whose value can be represented in 32 bits. Cross-32 will place the value of expression in the label field followed by an equal sign "=" to show that this value is not the current value of the program counter. Defining a label more than once, or placing multiple expressions after the EQU directive, will result in an assembly error.

```
;Valid examples of the EQU directive are:
;
CR:       EQU  13H       ;ASCII Carriage Return
LF:       EQU  10D       ;ASCII Line Feed
CNTR:     EQU  $         ;Program counter
EXPR2:    EQU  CR << 8   ;Formula
```

FILL - Fill the Binary File Directive

The binary hexadecimal output files are pure binary or machine code, not an ASCII representation with corresponding address information. The binary hex code will start at the address of the first origin directive (ORG), unless it is preceded by an instruction, in which case the machine code starts at address location zero. Positive jumps in the program counter using the origin directive will be filled with the binary value 0FFH (unless another value is specified with the FILL directive). The FILL directive has the following syntax:

```
label:  FILL    BYTE            ;comment
```

BYTE must be an integer between -128 and 255. A label may be included with the directive, which will be assigned the current value of the program counter. The FILL directive is optional, it does not affect Intel and Motorola hex files, and forward program counter jumps in binary hex files will be filled with 0FFH if it is not present.

```
;Valid example of the FILL directive:
;
        CPU     "6805.TBL"      ;PROCESSOR TABLE
        HOF     "BIN8"          ;HEX FORMAT
        FILL    9DH             ;EPROM FILL WITH
NOP
```

HEX - Hexadecimal File Control Directive

The HEX directive allows the user to turn the output to the hex file "ON" or "OFF". The program counter continues to increase when the machine code output is turned off. A machine code file is not produced if a file name is not specified in the command line "-H". The HEX directive has the following syntax:

```
label:  HEX     "mode"          ;comment
```

"Mode" must be "ON" or "OFF". A label may be included with the HEX directive, which will be assigned the current value of the program counter. The HEX directive is optional, with the default mode at the beginning of each pass being "ON". The HEX directive may appear anywhere in the assembly source file. It is usually used when defining locations in RAM memory that the user does not want included in the machine code (ROM) file.

```
;Valid example of the HEX directive:
;
        ORG     $A000

        HEX     "OFF"           ;RAM AREA
BYTE1:  DFB     0
BYTE2:  DFB     0
SUM:    DWM     0
MEAN:   DFL     0
REMAIN: DWM     0
        HEX     "ON"            ;ROM AREA
        ORG     $F000
;  ETC.
```

HOF - Hexadecimal Output Format

The hexadecimal output format (HOF) directive selects the format of the hex file. The HOF directive has the following syntax:

```
label:  HOF     "format"        ;comment
```

A label may be included with the HOF directive, which will be assigned the current value of the program counter. The HOF directive is optional, and if not included Cross-32 will default to the "INT16" format. The HOF directive may appear anywhere in the assembly source file. If it appears more than once with different hexadecimal formats specified, the format of the hexadecimal file will change without an error code being generated.

The HOF directive partially controls the format of the assembled listing. If a HOF directive in not used, or one of the 16 bit hexadecimal formats is declared, Cross-32 will produce a listing with a 32 bit value preceding the EQU and SETL directives, 24 bit addresses, and up to 11 bytes of code on each line. If one of the 8 bit hexadecimal formats is declared, Cross-32 will produce a listing with a 16 bit value preceding the EQU and SETL directives, 16 bit addresses, and up to 9 bytes of code on each line. This allows the listing format to correspond to the address word of the target processor.

The hex file may be written in three different formats, binary, Intel and Motorola. Although only the Intel and Motorola formats are actually ASCII hexadecimal, the expression "hex file" is used throughout this document to refer to all three formats.

```
;Valid examples of the HOF directive are:
;
        HOF     "BIN8"          ;Binary 8 bit
        HOF     "BIN16"         ;Binary 16 bit
        HOF     "BIN32"         ;Binary 32 bit
        HOF     "INT8"          ;Intel 8 bit hex
         HOF      "INT16"          ;Intel 16 bit
hex
        HOF     "MOT8"           ;Motorola 8 bit
hex
        HOF     "MOT16"         ;Motorola 16 bit
hex
        HOF     "MOT32"         ;Motorola 32 bit
hex
```

IF, ELSE, and ENDI - Conditional Assembly Directive

Cross-32 supports conditional assembly using the IF, ELSE and ENDI directives to define areas of the source file that are or are not to be assembled. This feature is usually used to configure a single assembly language program for different hardware environments. Conditional assembly has the following syntax:

```
 IF    expression    ;comment

       line 1
       line 2
         .
         .
         .
       line n

 ELSE                ;comment

       line 1
       line 2
         .
         .
         .
       line n

 ENDI                ;comment
```

Upon encountering an IF statement Cross-32 evaluates the single expression following it. All labels used in this expression must be defined previous to the IF. If the

expression evaluates to zero, the statements between the IF and either an ELSE or an ENDI are not assembled, but are shown in the listing. If the expression results in a non-zero value, the statements between the IF and either an ELSE or an ENDI are assembled. The ELSE is an optional directive, allowing only one of the two sections of the source file within the IF block to be assembled. All conditional blocks must have an IF directive and an ENDI directive, the ELSE directive being optional. IF-ELSE-END blocks may be nested 16 deep before an error occurs.

An example of conditional assembly follows, where a microcontroller writes to a printer, which is connected to either an RS-232C serial or a Centronics parallel port, but not both.

```
;An example of Conditional Assembly is:
;
FALSE:       EQU  0D
TRUE:        EQU  NOT FALSE

;In this example use the RS-232C port
;
RS232C:      SETL  TRUE
CENTRONICS:  SETL  FALSE

;Conditional Block Starts Here
;
             IF   RS232C

IO_PORT:     EQU  34H        ;RS-232C Port

             ENDI

             IF   CENTRONICS

IO_PORT:     EQU  44H        ;Centronics Port

             ENDI
```

INCL - Include Source File Directive

The include file (INCL) directive inserts the specified source file into the present file. Cross-32 will assemble the entire included file, then return and continue with the existing file. The INCL directive has the following syntax:

```
label   INCL    "source_file_name"      ;comment
```

Included source files may only be nested a maximum of 4 deep before an error occurs. A disk drive and/or directory may be included in the file specification. A label may be placed before the INCL directive, which will be assigned the current value of the program counter. All included files are read once each pass, and are included in their entirety in the listing. Should an END directive be encountered in an included file, assembly of all source files will cease at that point. A nonexistent include file will result in an error.

```
;An example of a INCL directive is:
;
             INCL    "A:\ASM\IO.ASM"
```

LIST - List File Control Directive

The list file control directive enables the user to turn the output to the list file "ON" or "OFF". The LIST directive has the following syntax:

```
label:  LIST    "mode"          ;comment
```

"Mode" must be "ON" or "OFF". A label may be included with the LIST directive, which will be assigned the current value of the program counter. The LIST directive is optional, with the default mode at the beginning of each pass being "ON". The LIST directive may appear anywhere in the assembly source file. It is usually used when debugging a specific section of source code, and the entire listing is not desired.

```
; Valid use of LIST directive
;
        LIST    "OFF"           ;TURN LIST FILE
OFF
        LIST    "ON"            ;TURN LIST FILE
ON
```

MACRO and ENDM - Macro Assembly

Cross-32 supports macro assembly using the MACRO and ENDM directives to define areas of the source file that are to be repeated when called. Macro assembly has the following syntax:

```
Label: MACRO  expr(1),        expr(2),...expr(n)
;comment

       line 1
       line 2
         .
         .
         .
       line n

       ENDM
```

Upon encountering a MACRO directive, Cross-32 stores the source code between it and the next ENDM directive, assigning it to the mandatory label on the MACRO line. Although the code within the macro definition is checked for syntax errors, the resulting machine code is not written to either the list or hexadecimal files. When the macro's label is found as a macro call later in the assembly source code, the entire MACRO is expanded at this location. Any expressions appearing after the macro definition is replaced by those appearing after the macro call in the expanded code. These are character by character replacements, so ensure that the expressions in the macro definition are truly unique. The number of expressions in the macro definition must equal the number of expressions in the macro call. Nested macros are not permitted. An example of macro assembly follows, originally written for the CP/M- 80 operating system.

```
        CPU  "8085.TBL"

; . . . . . .
. .
;       INPUT MACRO
;       INPUT CHARACTER STRING FROM CONSOLE
;
;       INPUT   ADDR,BUFLEN
;
;               ADDR    START OF TEXT BUFFER
;               BUFLEN  LENGTH OF BUFFER
;
INPUT:  MACRO   ADDR,BUFLEN
        MVI     C,10
          LXI       D,ADDR            ;SET BUFFER
ADDRESS
          MVI       A,BUFLEN          ;SET BUFFER
LENGTH
        STAX    D
        CALL    5               ;BDOS ENTRY
        ENDM

        INPUT   0C012H, 80
        MVI     C,10
          LXI       D,0C012H          ;SET BUFFER
ADDRESS
          MVI       A,80              ;SET BUFFER
LENGTH
        STAX    D
        CALL    5               ;BDOS ENTRY
        ENDM
```

ORG - Program Counter Origin Directive

The origin (ORG) directive allows the user to specify the value of the program counter during assembly. The ORG directive has the following syntax:

```
label:  ORG     expression      ;comment
```

The ORG directive may be used as often as desired, but Cross-32 will not flag areas that may be defined more than once in a single source file. The expression is not optional in this directive and may consist of any numeric constant, character string or formula whose value can be represented in a 16 or 24 bit positive integer, depending on which hexadecimal format has been declared using the HOF directive. Cross-32 will place the new value of the program counter in the address field. A missing expression or multiple expressions after the ORG directive will result in an assembly error being flagged.

```
;Valid examples of the ORG directive are:
;
RESET:   ORG       0                    ;A common
beginning
        ORG     0100H           ;A famous start
```

PAGE - Page Control Directive

The PAGE directive may be used to both eject the current listing page and set the number of lines in each listing page. This directive has the following format:

```
label:  PAGE                    ;comment
```

Cross-32 will insert an ASCII form feed (0CH) into the listing before the next line of the listing. This causes the printer to continue the listing on a new page. However, the next format:

```
label:  PAGE    expression      ;comment
```

will set the page length to the value of the expression. Each time the page length is reached, Cross-32 inserts an ASCII form feed into the listing. All positive integer values except 1 and 2 are legal. If the expression has a value of zero, or a page size is not specified, form feeds will not be inserted into the listing.

```
;Valid examples of the PAGE directive are:
;
        PAGE    56
        PAGE    60
        PAGE    0
```

RPTXT - Replace Text Directive

The replace text (RPTXT) directive may be used to exchange any text source string with any other text string. It is usually used to give processor registers and bits more descriptive names, or makes Cross-32 more compatible with other assemblers. A good example of using the RPTXT directive is at the end of the ST9 table, file ST9.TBL. It is also used when porting existing code to Cross-32. A label may be included with the HEX directive, which will be assigned the current value of the program counter. The RPTXT directive has the following syntax:

```
label:      RPTXT      "old-string",”new-string”
;comment
```

The “old-string” is how the text appears in the assembly source file. The “new-string” is what the old-string is changed to during assembly, and how it appears in the listing.

This directive is extremely powerful, and should be used with caution. It can easily change source code in unexpected ways. Both the old-string and new string should be fairly unique.

The RPTXT directives are applied to the source code in the reverse order that they are declared, and after any macro has been expanded. The text being replaced must be surrounded by white space, or an ASCII characters other than "A-Z", "a-z", “0-9”, "_", "." or "?".

```
; From ST9.TBL
;
        CASE    “ON”

        RPTXT   "FCW","RR230"   ;
        RPTXT   "fcw","rr6"     ;
```

SETL - Set Label Directive

The set label (SETL) directive may be used to assign an integer value to a label. It is similar to the EQU directive except that the value of the label may be redefined using additional SETL directives elsewhere in the assembly source file. The SETL directive is most commonly used with conditional assembly. The SETL directive has the following syntax:

```
label: SETL   expression          ;comment
```

The label and the expression are not optional in this directive. The expression may consist of any numeric constant, character string or formula whose value can be represented in 32 bits. The assembler will place the value of the expression in the label field followed by an equal sign "=" to show that it is not the current value of the program counter. Missing or multiple expressions after the SETL directive will result in an assembly error.

```
;Valid examples of the SETL directive are:
;
        MINUS1: SETL    1
        MINUS1: SETL    -MINUS1
```

TITL - Title of Listing Directive

The title (TITL) directive places the "character string", time, date and page number, at the top of each page of the listing. The TITL directive has the following syntax:

```
label:  TITL    "character string"      ;comment
```

Both the label and the comment are optional for this directive. The "character string" is not, and must at the very minimum be a null string "". The length of the character string is unlimited, except that the source line cannot exceed 255 characters. If a page length is not specified using the PAGE directive, the title will only appear at the beginning of the listing. A TITL directive is not required to produce a correctly formatted listing.

```
;Valid examples of the TITL directive are:
;
AAA:    titl
        TITL    "Test File"
        page


Test File    PAGE 2    Wed Aug 18 21:05:42 1996
```

WDLN - Word Length Directive

The word length (WDLN) directive may be used to change the program counter word length from its default value of one byte, to any positive integer from one to ten inclusive. It is used by the TMS320 digital signal processor family, which has a 2 byte, or 16 bit, program word. The WDLN directive has the following syntax:

```
label:  WDLN    expression      ;comment
```

The label is optional with the WDLN directive, and will be assigned the current value of the program counter. The expression may be any numeric constant with a value between one and eight inclusive. Missing or multiple expressions after the WDLN directive will result in an assembly error being flagged.

```
;Using the WDLN directive with the TMS320 family
;
        CPU     "TMS320.TBL"    ;CPU TABLE
          HOF       "INT8"            ;HEX OUTPUT
FORMAT
         WDLN     2                 ;2 BYTE WORD
LENGTH
;
;       ORG     0
;
INIT:   LACK    1
        LACK    33
        CALA
        RET
```

WARNING: When the WDLN assembler directive is used to set the program word length to a value other than 1 byte, Cross-32 may produce an Intel or Motorola hex file different from what the user expects. In particular, the program counter is multiplied by the word length specified by WDLN, so the program counter and the number of bytes in each record of the hex file correspond on a one to one basis. Therefore, an eight bit EPROM can be programmed normally. Programs that split the hex file, (used to create a 16-bit EPROM from two 8-bit ones), should also work properly. The Intel and Motorola hex files should be used with caution when the word length is not set to 1. The binary hex file format is not affected.

Assembler Output Files

List Files (.LST)

Cross-32 automatically produces a listing (.LST) during the second and third passes of the assembly source file. The HOF directive partially controls the format of the assembled listing. If a HOF directive in not used, or one of the 16 bit hexadecimal formats is declared, Cross-32 will produce a listing with a 32 bit value preceding the EQU directive, 24 bit addresses, and up to 7 bytes of code on each line. If one of the 8 bit hexadecimal formats is declared, Cross-32 will produce a listing with a 16 bit value preceding the EQU directive, 16 bit addresses, and up to 5 bytes of code on each line. This allows the listing format to correspond to the address word of the target processor.

The listing is the original assembly source file, with 16 or 24 additional ASCII characters inserted at the beginning of each line. The next four or six characters represent the hexadecimal value of the program counter. Following the program counter, is another blank, followed by the hexadecimal value of the assembly instruction or assembler directive. This value will not be displayed after the fifth or seventh byte, but will be placed in the hexadecimal file in its entirety. The EQU assembly directive is an exception to this rule, and further information on it may be found elsewhere in this manual.

Hexadecimal Files (.HEX)

A hex file, with the name of the main file ending in HEX (MAINFILE.HEX), is automatically generated during assembly. The assembler supports the binary, Intel and Motorola hex file formats. Although only the Intel and Motorola file formats are actually ASCII hexadecimal files, the expression "hex file" is used in this manual to describe all three formats. The extended Intel hex format (INT16) will be produced by default, and any one of eight formats may be selected using the HOF directive. Examples of each format are provided below for the assembly source file SHOW.ASM.

```
; File SHOW.ASM
;
        HOF     "int8"          ;Hex Format
        ORG     12345678h
BEGIN:  DFB     "Hexadecimal!",10,13
        END     BEGIN
```

The **binary** hex files are pure binary or machine code, not the ASCII representation shown below. Although this example is a character string, the binary format should not normally be edited or written to the screen or printer. Binary code starts at the address of the first ORG directive, unless it is preceded by an instruction, in which case the binary code starts at address location 0000H. Positive jumps in the program counter using the origin directive will be filled with the binary value 0FFH (unless another value is specified with the FILL directive). If the program counter is reduced with the ORG directive, an error message is generated.

Reducing the program counter using a binary hex code format can corrupt the hex file, and the user should either rearrange the code or use another hex format. Notice that there is no difference in the hex code produced by the BIN8, BIN16 and BIN32 hex types, only the format of the listing is changed. "BIN8" should be used unless the program counter exceeds 0FFFFH.

```
BIN8   --   Binary (8 bit format)

      486578616465636 96D616C210D0A

BIN16  --   Binary (16 bit format)

      486578616465636 96D616C210D0A

BIN32  --   Binary (32 bit format)

      486578616465636 96D616C210D0A
```

There are two **Intel** hex formats, regular and extended. The regular format supports addressing to $FFFF, while the extended format uses a segment address record for addressing to $FFFFF. The assembler truncates larger addresses (such as 12345678H in the example) without issuing a warning. Intel hex files consist of records of ASCII characters. Each record starts with a colon ":" and ends with an ASCII carriage return (13D) and linefeed (10D). If an expression follows the END directive its value will be included as the execution starting address in the end of file record. If the END directive and/or expression are not included, the end of file address will default to $0000. The remaining Intel features are:

```
NN            number of data bytes
12345678      address of the first data byte
00            segment address (set to zero)
TT            the Record Type, where:
                    00 represents data
                    01 represents end of file
                    02    represents    segment
address
DD            byte of data
CC            8 bit binary checksum where:
              CC = -{NN+12345678+TT+DD} & 0FFH
```

```
 INT8   --   Intel Hex Format

       :NN5678TTDDDDDDDDDDDDDDDDDDDDDDDDDDCC
       :NN5678TTCC

       :0E567800486578616465636 96D616C210D0A97
       :0056780131
```

```
 INT16  --   Extended Intel Hex Format

       :NN0000TT4000CC
       :NN5678TTDDDDDDDDDDDDDDDDDDDDDDDDDDCC
       :NN5678TTCC

       :020000024000F8
       :0E567800486578616465636 96D616C210D0A97
       :0056780131
```

There are three **Motorola** hex formats, S19, S28 and S37. The S19 format supports addressing to $FFFF, the S28 format supports addressing to $FFFFFF, and the S37 format supports addressing to $FFFFFFFF. The assembler truncates larger addresses (such as 12345678H in the example) without issuing a warning. Motorola hex files consist of records of ASCII characters. Each record starts with an "S" and ends with an ASCII carriage return (13D) and linefeed (10D). If an expression follows the END directive, its value will be included as the execution starting address in the end of file record (S9, S8 or S7). If the END directive and/or expression are not included, the end of file address will default to zero.

```
The remaining Motorola features are:

NN            record length, including the
              address, data and checksum fields
12345678      address of the first data byte
DD            1 byte of data
CC            1 byte checksum where:
              CC = ~{NN + 12345678 + DD} & 0FFH
```

```
 MOT8   --   Motorola (8 bit format)

       S1NN5678DDDDDDDDDDDDDDDDDDDDDDDDDDCC
       S9NN5678CC

       S1115678486578616465636 96D616C210D0A93
       S90356782E
```

```
 MOT16  --   Motorola (16 bit format)

       S2NN345678DDDDDDDDDDDDDDDDDDDDDDDDDDCC
       S8NN345678CC

       S212345678486578616465636 96D616C210D0A5E
       S804345678F9
```

```
 MOT32  --   Motorola (32 bit format)

       S3NN12345678DDDDDDDDDDDDDDDDDDDDDDDDDDC
C
       S7NN12345678CC

       S3131234567848657861646563696D616C210D0A4
B
       S70512345678E6
```

WARNING: When the WDLN assembler directive is used to set the program word length to a value other than 1 byte, Cross-32 may produce an Intel or Motorola hex file different from what the user expects. In particular, the program counter is multiplied by the word length specified by WDLN, so the program counter and the number of bytes in each record of the hex file correspond on a one to one basis. Therefore, an eight bit EPROM can be programmed normally. Programs that split the hex file, (used to create a 16-bit EPROM from two 8-bit ones), should also work properly. The Intel and Motorola hex files should be used with caution when the word length is not set to 1. The binary hex file format is not affected.

Error Files and Codes (.ERR)

Cross-32, with its integrated development environment, will automatically display error messages in the status line and highlight the offending text in a file window. Full error messages are placed in both the list (.LST) and error (.ERR) files. The error file is an ASCII text file with one line of text for each assembly error. The identical format is used by the MS-DOS command line version of Cross-32 when it displays errors on the screen.

Error messages are of the following format:

```
File( Row, Column): Message
```

File is the name of the source file in which the assembly error was detected.
Row is the line number of the source file in which the assembly error was detected.
Column is character position in the source file line when the assembly error was detected.
Message is Cross-32's explanation of the assembly error.

Many programming editors, in addition to the Cross-32 IDE, can use this format directly, allowing the user to create an integrated development environment with another editor.

As Cross-32 searches the instruction table, it will display the first error code generated for the given assembly line, even though other errors may occur. Therefore, certain combinations of instructions and syntax errors can produce a misleading error code. If the code does not seem to be relevant, look for other possible errors.

Assembly Errors

Cross-32 may generate the following error messages.

Error 01 - Source file did not open

File specified by CPU or INCL directive did not open.
Assembly stopped immediately.
Insure the file name is correct and is in the specified directory.

Error 02 - Disk or Directory Full

Unable to write or close list, hex or error file.

Delete unnecessary files and try again.

Error 03 - Too many include files

Include files using the INCL directive may only be nested 4 deep.

A common approach is to create a main file that is just a series of INCL statements, keeping the includes to just 1 level.

Error 04 - Too many conditional blocks

Conditional blocks using the IF, ELSE and ENDI directives may only be nested 16 deep.

Error 05 - No source file specified

A main or source file must be declared.

Error 06 - Illegal CPU table format

The assembler has discovered a format error in the CPU table while loading it into RAM. It is most likely missing '^' or ':' character.

Correct the problem and try again.

Error 07 - List file did not open

The specified list file name is on a nonexistent disk drive or directory, or the directory is full.

Error 08 - Hex file did not open

The specified hex file name is on a nonexistent disk drive or directory, or the directory is full.

Error 09 - Insufficient memory while loading table

The assembler has run out of memory while loading the instruction table.
This should only occur using a large CPU table on an MS-DOS computer with less than 640kB of RAM.
Close open files, add more RAM, exit other tasks, or use the command line version of the assembler (C32D4CL.EXE).

Error 10 - Insufficient memory while loading symbol

The assembler has run out of memory when attempting to store a symbol (label) in RAM.

Reduce the number or the length of the labels in the source file.

Close open files, add more RAM, remove any TSR (Terminate and Stay Resident) utilities, or use the command line version of the assembler.

The command line version on an MS-DOS computer with 640kB of available RAM should store 19,000 8-character labels.

Error 11 - Insufficient memory while loading macro

The assembler has run out of memory when attempting to store a macro in RAM.
Insure that all macro declarations are terminated with an ENDM directive.
Reduce the number or the size of the macros in the source file.
Close open files, close other applications, add more RAM, remove any TSR (Terminate and Stay Resident) utilities, or use the command line version of the assembler.

Error 12 - Interrupted by user

The assembler stopped because the user hit the cancel button.
Press F9 to start assembler.

Error 26 - Missing operand

The assembler has found a valid instruction or directive, but cannot find the necessary operand(s) after it.

Add the correct operand and try again.

```
;Error  26:     Missing operand
        DFB                     ;Missing Operand
```

Error 27 - Illegal line number

The assembler has found a line number containing an illegal character.
Line numbers are only produced by some editors.
The error can also be caused by a label or instruction that starts will a numeric character.

```
;Error  27:     Illegal line number
0G:                                 ;Illegal line
number
```

Error 28 - A "Character string" is required

A Cross-32 directive, such as CPU, HEX, INCL or LIST that requires a character string does not have one.
Add the correct string, surrounded by double quotes ("), and try again.

```
;Error  28:     A "Character string" is required
          CPU       INT8              ;Character
String
```

Error 29 - Missing or illegal label

A Cross-32 directive that requires a label, such as EQU, MACRO or SETL, does not have one.
A valid label must start in column 1 and/or end with a colon (:).
Add a label and try again.

```
;Error  29:     Missing or illegal label
        EQU     1               ;Missing label
```

Error 30 - Illegal hexadecimal format

The hex file format in the HOF directive must be one of:

"BIN8"Binary 8-bit format

"BIN16" Binary 16-bit format

"INT8"Intel 8-bit format

"INT16" Intel extend 16-bit format

"MOT8"Motorola 8-bit S19 format

"MOT16" Motorola 16-bit S28 format

"MOT32" Motorola 32-bit S37 format

Add the correct format and try again.

```
;Error  30:     Illegal hexadecimal format
          HOF       "TEK8"            ;Illegal hex
format
```

Error 31 - Unexpected characters at end of line

The assembler has processed a complete instruction or directive and then found additional characters at the end of the line.
Correct the syntax of the line and try again.

```
;Error  31:     Unexpected characters at end of
line
           LAB:     EQU       1,2     ;Unexpected
characters
```

Error 32 - Phase error

One or more labels have been assigned a value in the second pass that was not equal to the assigned value in the first pass. Cross-32 will perform a third pass to correct this problem.

Some processors have instructions with identical syntaxes but different opcode lengths. Cross-32 assigns undefined labels the value of the program counter. This can cause Cross-32 to use the instruction with the shortest opcode (relative) in the first pass, and an instruction with a longer opcode (absolute) in the second pass. One code shift, will generate a phase error at all remaining labels, so don't worry about the number of phase errors.

To reduce phase errors:

1) Define memory labels at the beginning of the program.

2) Define subroutines at the beginning of the program, thereby avoiding forward referenced labels.

3) Where available, specify the size of branch instructions. Example: Use < and > with the Motorola micro-controllers.

4) Add an ORG directive half way through your code.

To find the instruction causing a phase error:

1) Add labels to all instructions between the last label that does not generate a phase error, and the first label that does generate a phase error.

2) Assemble the code again.

3) The phase error is being caused by the instruction immediately preceding the first phase error.

```
;Error   32:       Phase error, value of label
changes
LAB1:   EQU     1               ;Phase error
LAB1:   EQU     2               ;Phase error
```

Error 33 - Instruction not found

The assembler is unable to identify the first word of the instruction or directive. Correct its syntax and try again.

If a large number of these errors occur, insure that your code contains the correct CPU directive.

```
;Error  33:     Instruction not found,
        MOV      A              ;Instruction not
found
```

Error 34 - File must be ON or OFF

A HEX or LIST directive is not followed by a "ON" or "OFF". Correct the above and try again.

```
;Error  34:     File control must be ON or OFF
        LIST    "YES"           ;Must be ON/OFF
```

Error 35 - Symbol not found

Nothing on this line can be identified.
Correct its syntax and try again.

If your program has a lot of these errors, there is probably a missing or incorrect CPU directive.

Insure that the assembler, source file and processor table is all operating out of the same directory.

```
;Error  35:     Symbol not found,
         XXX                    ;Instruction not
found
```

Error 36 - Operand not in specified range

The highlighted operand or expression is greater than or less than its legal range.

Correct its value and try again.

In a branch instruction, the label may be too far away.

```
;Error  36:     Operand not in specified range
        DFB     $1234           ;Too Large
```

Error 37 - Instruction starts with invalid character

Assembly instructions and directives must start with an alphabetic character (A-Z). Correct the syntax and try again.

```
;Error  37:     Starts with invalid character
        (MOV                    ;Invalid (
```

Error 38 - Violation of conditional block

An IF, ELSE or ENDI directive has appeared where the assembler was not expecting it. Insure that conditional blocks start with and IF and end with an ENDI. Double check nested conditional blocks to insure that there are enough ENDI directives.

```
;Error  38:     Violation of conditional block
        ELSE                    ;No IF
```

Error 39 - Decreasing Program Counter

An ORG directive has reduced the value of the program counter when the assembler is writing the hex file in a binary format.
The hex code generated is probably invalid.

Insure that RAM definitions use EQU or DFS directives, or surround them with HEX "OFF" and HEX "ON" statements.

This error will not occur with the Intel and Motorola hex file formats.

Error 40 - Undefined label

An alphanumeric expression used in an operand is undefined. Check the syntax and try again.

NOTE!!!

Hexadecimal constants with a trailing 'H' radix must start with a number (0-9). FFH is not a hexadecimal number, it should be 0FFH.

```
;Error  40:     Undefined label
        DFB      + LABEL        ;Undefined label
```

Error 41 - Missing " at end of string

A character string that begins with a double quote (") does not end with one before the end of the line.

A string should be written: "String"

Correct the syntax and try again.

```
;Error   41:      Missing " at end of character
string
        LIST    "ON             ;Missing "
```

Error 42 - Missing right script bracket }

An expression has at least one more left script bracket than it does right script brackets.

Match the brackets and try again.

```
;Error  42:     Missing right script bracket }
        DFB     4 * {7 - 3      ;Missing bracket
```

Error 43 - Digit is not valid for declared base

Digits must be in the range permitted by their declared base.

Base 2: Range 0-1

Base 8: Range 0-7

Base 10: Range 0-9

Base 16: Range 0-9, A-F

```
;Error  43:      Digit is not valid for declared
base
        DFB     0779Q           ;Illegal nine
```

Error 44 - Unexpected second value

A second operand is found where it was not expected.

Correct the syntax and try again.

```
;Error  44:     Unexpected second value
        DFB     1 2             ;Two values
```

Error 45 - Undefined operator

An illegal operator was found in an operand or expression. Examples of valid operators are +, -, % and >=

Correct the syntax and try again.

```
;Error  45:     Undefined operator
          DFB      1 ' 2              ;Undefined
operator
```

Error 46 - Unexpected right script bracket }

The assembler found a right script bracket before it found a left script bracket.

Match the brackets and try again.

```
;Error  46:     Unexpected right script bracket
}
        DFB     4 * {7 - 3}}    ;Extra bracket
```

Error 47 - Unexpected end of line

The assembler has found the end of the line while looking for an expression or parts of an expression.

Correct the syntax and try again.

```
;Error  47:     Unexpected end of line
        DFB     4 *             ;End of line
```

Error 48 - Shift must be less than 32

The bits of an integer must not be rotated left << or right >> by more than 31 bits. What would be the point?

Correct the syntax and try again.

```
;Error  48:     Shift must be less than 32
        DFB     2 << 33         ;Shift too large
```

Error 49 - Unexpected binary operator

A binary operator was found where it was not expected. Often caused by two binary operators together (+-).

Correct the syntax and try again.

```
;Error  49:     Unexpected binary operator
        DFB     2 ** 33         ;Two operators
```

Error 50 - Unexpected unary operator

A unary operator was found where it was not expected. Often caused by two operators together (~!).

Correct the syntax and try again.

```
;Error  50:     Unexpected unary operator
        DFB     33 ~            ;Wrong side
```

Error 51 - String exceeds 4 characters

A string that is used in an integer expression must not be longer than 4 characters. Correct example: "ABCD"

Correct the syntax and try again.

```
;Error  51:     String exceeds 4 character
        DFL     "ABCDE"         ;Too long
```

Error 52 - Unexpected expression separator

An expression separator (usually a comma) was found were it was not expected. The assembler was looking for the right hand side of an expression.

Correct the syntax and try again.

```
;Error  52:     Unexpected expression separator
         DFB      4 * , 1           ;Unexpected
comma
```

Error 53 - Division by zero attempted

The denominator (right hand side of the / operator) evaluates to zero.

This would cause an overflow.

Correct the syntax and try again.

```
;Error  53:     Division by zero attempted
         DFB      4 / 0             ;Division by
zero
```

Assembler Processor Files (.TBL)

The Instruction Table

The instruction table defines the mnemonics, operands and opcodes of individual microprocessors and microcontrollers. To simply use one of the tables provided, this section of the manual may be skipped, but the example assembly file for the target processor should be studied carefully. Cross-8 1.X, Cross-16 1.X, and Cross-32 users prior to version 1.5 are advised that the table format is not compatible with these earlier products. Early Cross-32 tables may be converted to the Cross-32 V1.5 and later format, simply by changing the arithmetic operators to those used by the ANSI C programming language.

The instruction table consists of five sections, to define the processor's registers, operands, addressing modes, mnemonics, and assembler respectively. Each of these will be discussed as part of an example for the Intel MCS-48 (8048) microcontroller family. Although this is not the most world's most sophisticated processor, its instruction set has some unique attributes, which will exhibit most of the features and flexibility of the table format.

The Cross-32 processor tables are ASCII files, which may be edited using any text editor. Word processors should be avoided, or used in a non-document mode, to elude the hidden formatting commands that are added to files by many of these products. The only limit on the size of the processor table, is that it must reside entirely in the memory of your computer when Cross-32 is assembling. Any single line in the table must not exceed the maximum Cross-32 source line length of 255 characters. Cross-32 expects each line to end with an ASCII carriage return and line feed. Blank lines and comments may be entered at any point in the table using a semicolon ";", and are not stored in memory by Cross-32 during assembly.

Register Definition

The first section of the processor table defines the syntax of any registers or similar labels to which a constant value may be assigned. Cross-32 will use this section to identify operands as an exact character string. Each line of the register section has the following syntax:

```
register-
#,"register0","register1",..."registerN"
       .
       .
       .
register-
#,"register0","register1",..."registerN"
*
```

Register-#: is a unique integer, used by the operand section of the table to identify a register line. The line numbers need not be actual file line numbers, need not appear in numerical order, and may be assigned any positive integer value less than 32768. Tables supplied by Data Sync Engineering will start with register line number one and increase in steps of one. Each register-number must be followed by a comma ",".

Register: are character strings representing register names, defined as any other character string used with Cross-32, preceded and terminated by quotation marks ("), with each character string being separated by a comma ",". Cross-32 assigns the first character string the value of zero, and increases the value of each successive string by one.

* Marks the end of the register section of the table.

The 8048 uses eight registers, which it labels R0 through R7 and assigns an address value of zero through seven respectively. The 8048 supports direct addressing of all eight registers, and indirect addressing registers R0 and R1. All eight directly addressed registers are defined by:

```
1,"R0","R1","R2","R3","R4","R5","R6","R7"
```

The two registers used for indirect addressing are defined by:

```
2,"R0","R1"
```

The end of the register section of the table is marked by:

```
*
```

Operand Definition

The second section of the table defines the size, format and position of operands in the instruction's opcode. The binary value of these operand definitions will be ORed with the instruction's opcode during assembly. Each line of the operand definition has the following syntax:

```
operand-#, start-bit, bit-length, express, low,
high
        .
        .
        .
operand-#, start-bit, bit-length, express, low,
high
*
```

Operand-#: is a unique integer, used by the addressing mode section of the table to identify this line. The operand-numbers need not be actual file line numbers, need not appear in numerical order, and may be assigned any positive integer value less than 32,768. Tables supplied by Data Sync Engineering usually start with line number one and increase in steps of one. Each operand-number must be followed by a comma ",".

Start-bit: is the position of the first bit of the operand within the instruction's opcode. This value must be a positive integer followed by a comma ",". The bit positions are numbered from left to right (most significant bit first) starting at 0 and preceding to a maximum of 255.

Bit-length: is the number of bits of the operand to be placed in the opcode, beginning with the least significant bit. The operand bit-length must be a positive integer ranging from 1 to a maximum of 32 bits.

Express: is an integer arithmetic formula, whose value represents the operand to be incorporated into the opcode. This expression is not unlike any other used by Cross-32, with two exceptions:

1) A number sign "#" represents the original value of the operand found in the source file.

2) An at sign "@", immediately followed by a register-number, points to a line in the register definition section of the table. The value generated by this expression will start at 0 and increase by 1 for each register string (0,1,2,...). This format is used by most register instructions.

3) At and ampersand signs "@&", immediately followed by a register-number, points to a line in the register definition section of the table. The value generated by this expression will start at 1 and be rotated left 1 bit for each register string (1,2,4,8...). This format is used by a few push and pull register instructions (see 6809.TBL).

4) An opening single quote (`) represents the length of an instruction's opcode in bytes. This is most often used with relative branch instructions. A tilde "~" was used for this purpose in Cross-32 version 1.0.

Low: is the smallest allowable value of the original operand found in the source file. If the operand value is less than the specified low, Cross-32 will look for another matching mnemonic located later in the table.

High: is the largest allowable value of the original operand found in the source file. If the operand value is greater than the specified high, Cross-32 will look for another matching mnemonic located later in the table.

* Marks the end of the operand definition section of the table.

The 8048 instruction set uses five different operand types. The first two of these are part of the previously defined direct and indirect register addressing modes. In the direct register addressing mode, a 3-bit expression representing the register address is placed in the opcode starting at bit five. The character strings representing the registers were defined in register-number 1 of the register definition section. The direct register operand is defined by:

```
1,      5,      3,      @1,     0,      7
```

Similarly, the indirect register operand, with its 1-bit value placed in bit 7 of the opcode, is defined by:

```
2,      7,      1,      @2,     0,      1
```

After that, the 8048's immediate operand, a signed 8-bit value starting at bit 8, is defined by:

```
3,      8,      8,      #,      -128,   255
```

Next, the 8048 has an eight-bit branch-addressing mode. The operand in this mode is eight bits long and must be on the same 256 byte page as the next instruction {$+1}. This operand can be defined and checked using the expression:

```
4,      8,      8,      # & 255,       {$+1} & 3840,
        {$+1} | 255
```

Finally, the 8048 has an eleven-bit branch-addressing mode. The operand in this mode is eleven bits long and is in one of the two memory banks. The most significant three bits of the operand are also shifted left five bits, to bits 0 through 2. However, even this operand can be defined and range checked using the following expression:

```
5,      0,      16,     {{# & 1792}*32}|{# & 255},
        0,      4095
```

Addressing Mode Definition

The third section of the table defines the addressing modes of the processor and the position of the operands in each instruction. In conjunction with the mnemonic definition in the next section, the addressing mode also defines the value and length of the opcode. Each line of this section of the table has the following syntax:

```
address-number, addressing-mode^opcode:
        .
        .
        .
address-number, addressing-mode^opcode:
*
```

Address-Number: is a unique integer, used by the mnemonic section of the table to identify an addressing mode. The address-numbers need not be actual file line numbers, need not appear in numerical order, and may be assigned any unique positive integer less than 32,768. Tables supplied by Data Sync Engineering start with address-number 1 and increase in increments of one. Each address-number must be followed by a comma ",".

Addressing-mode: is an ASCII string showing the relative positions of the characters and operands that make up the different addressing modes of a processor. The constant characters are simply listed. The variable operands are inserted using script brackets {} enclosing an operand-number as shown below:

```
{operand-number}
```

The operand-number must be defined in the operand section of the table. There are no limits on the character length of the addressing-mode-definition, except that it may not exceed 255 characters.

Opcode: is the ASCII hexadecimal representation of the instruction's opcode, that is placed between a carat "^" and a colon ":". From this and a similar field in the mnemonic section of the table, Cross-32 determines both the value and length of the instruction's opcode. There are no limits on the character length of the opcode, except that the entire line cannot exceed 255 characters. During assembly, the supplied opcode is converted to a binary value and ORed with a similar field in the mnemonic section of the table and any operands defined within script brackets "{}". The opcodes binary length is determined by the longer of the addressing and mnemonic opcodes defined in the last two sections of the table.

* Marks the end of the addressing section of the table.

Most processor's instruction sets actually have more different instruction formats than manufacturer defined addressing modes. Each different instruction format that contains variable operands must be defined in this section of the table. When writing a table from scratch, examples of most of the processor's addressing formats can usually be found in the MOVE or LOAD instructions. The inherent addressing mode, such as a NOP instruction, which does not have any operands, is defined only in the next section of the table. The 8048 has eleven different addressing modes or formats including the inherent mode.

The first is the register-addressing mode, where one of the processors eight registers is the operand, indicated by the "{1}". The previous two sections of the table place the 3-bit operand at bit 5 of the 1-byte opcode, and state that it must be R0 through R7.

```
 1, {1}^08:
```

The second is the register indirect addressing mode, where either R0 or R1 is the operand, indicated by the {2}. The previous two sections of the table place the operand in bit seven of the one byte opcode and state that it must be R0 or R1.

```
 2, @{2}^00:
```

The third is an immediate addressing mode, where an eight bit signed number is the operand, indicated by the {3}. The previous two sections of the table place the operand in the second byte of the two-byte opcode and state that it must be in the range of -128 to 255.

```
 3, #{3}^0000:
```

The remaining addressing formats are defined in a similar manner:

```
 4, {4}^0000:
 5, {5}^0000:
 6, @{2},A^00:
 7, {1},{4}^0000:
 8, {1},A^08:
 9, {1},#{3}^1800:
10, @{2},#{3}^1000:
*
```

Mnemonic Definition

This section of the table defines the actual assembly mnemonics specified by the processor manufacturer, and the addressing modes used by each. In conjunction with the opcode definition in the previous section, the mnemonic section also defines the value and length of the instructions opcode. Each line of the mnemonic definition has the following syntax:

```
Mnemonic|address_number1-address_number2^opcode:
        .
        .
        .
Mnemonic|address_number1-address_number2^opcode:
```

Mnemonic: is an ASCII string representing the instruction. Cross-32 uses a hash search routine to find the correct mnemonic for each line of the assembly language program. Therefore, the first (non-label) word of each assembly language source line must exactly match the first word defined in this section of the table, in both length and content. A word is a character string separated by white space (blank, tab or carriage return). After the first word, Cross-32 compares the source line with the mnemonic on a character by character, operand by operand basis. White space between the mnemonic and the first vertical line "|" is ignored. Each mnemonic must start with an alphabetic character "A-Z", and may contain alphabetic characters "A-Z", ".", "_", ":", and "?". Mnemonics do not have to be placed in any particular order. The mnemonics in tables supplied by Data Sync Engineering are usually in alphabetical order for ease of reference. There are no limits on the character length of the mnemonic, except that the entire line cannot exceed 255 characters.

Address_number: is the number of one of the addressing modes defined in the previous section of this table. Each address number is preceded by a vertical line "|" character. A range of consecutive address numbers may be defined by placing a hyphen "-" between two address numbers, i.e.

```
|1-4
```

Single address numbers and address number ranges may be mixed, as shown by the MOV line of the 8048 table:

```
MOV |6|8-10^A0:
```

Opcode: is the ASCII hexadecimal opcode for the instruction that is placed between a carat "^" and a colon ":". From this field and a similar one in the previous section of the table, Cross-32 determines both the value and length of the instruction's opcode. There are no limits on the character length of the opcode, except that the entire line cannot exceed 255 characters. During assembly, the supplied opcode is converted to a binary value and ORed with a similar field in the previous section of the table and any operands defined within the script brackets "{}". The total opcode length is determined by the longer of the addressing and mnemonic opcodes defined in the last two sections of the table.

* Marks the end of the mnemonic section of the table.

There are three different addressing formats for the 8048's add to accumulator without carry instruction. The first word of this mnemonic is "ADD". This word starts the mnemonic line of the table, followed by a space. The second part of the mnemonic is an "A," which is followed by a vertical line "|". The addressing mode range follows the vertical line. The "ADD A," mnemonic must be listed twice in the table, because the opcode for the immediate addressing mode bears no resemblance to that of the register and indirect modes.

```
ADD A,|1-2^60:
ADD A,|3^03:
```

When compiling an "ADD A" instruction, Cross-32 will search the table in the order that the addressing modes have been listed, i.e. 1, 2 and 3. If none of the addressing modes match, an assembly error will be flagged.

The remaining MCS-48 mnemonics are defined as follows:

```
ADDC A,|1-2^70:
ADDC A,|3^13:
ANL A,|1-2^50:
ANL A,|3^53:
ANL BUS,|3^98:
ANL P1,|3^99:
ANL P2,|3^9A:
ANLD P4,A^9C:
ANLD P5,A^9D:
ANLD P6,A^9E:
ANLD P7,A^9F:
CALL |5^1400:
CLR A^27:
CLR C^97:
CLR F0^85:
CLR F1^A5:
CPL A^37:
CPL C^A7:
CPL F0^95:
CPL F1^B5:
DA A^57:
DEC A^07:DEC |1^C8:
DIS I^15:
DIS TCNTI^35:
DJNZ |7^E800:
EN I^05:
EN TCNTI^25:
ENT0 CLK^75:
IDL^01:
IN A,P1^09:
IN A,P2^0A:
INC A^17:
INC |1-2^10:
INS A,BUS^08:
JB0 |4^12:
```

```
JB1 |4^32:
JB2 |4^52:
JB3 |4^72:
JB4 |4^92:
JB5 |4^B2:
JB6 |4^D2:
JB7 |4^F2:
JC |4^F6:
JF0 |4^B6:
JF1 |4^76:
JMP |5^0400:
JMPP @A^B3:
JNC |4^E6:
JNI |4^86:
JNT0 |4^26:
JNT1 |4^46:
JNZ |4^96:
JT0 |4^36:
JT1 |4^56:
JTF |4^16:
JZ |4^C6:
MOV A,PSW^C7:
MOV A,T^42:
MOV A,|1-2^F0:
MOV A,|3^23:
MOV PSW,A^D7:
MOV T,A^62:
MOV |6|8-10^A0:
MOVD A,P4^0C:
MOVD A,P5^0D:
MOVD A,P6^0E:
MOVD A,P7^0F:
MOVD P4,A^3C:
MOVD P5,A^3D:
MOVD P6,A^3E:
MOVD P7,A^3F:
MOVP A,@A^A3:
MOVP3 A,@A^E3:
MOVX A,|2^80:
MOVX |6^90:
NOP^00:
ORL A,|1-2^40:
ORL A,|3^43:
ORL BUS,|3^88:
ORL P1,|3^89:
ORL P2,|3^8A:
ORLD P4,A^8C:
ORLD P5,A^8D:
ORLD P6,A^8E:
ORLD P7,A^8F:
OUTL BUS,A^02:
OUTL P1,A^39:
OUTL P2,A^3A:
RETR^93:
RET^83:
RL A^E7:
RLC A^F7:
RR A^77:
RRC A^67:
SEL MB0^E5:
SEL MB1^F5:
SEL RB0^C5:
SEL RB1^D5:
STOP TCNT^65:
STRT CNT^45:
STRT T^55:
SWAP A^47:
XCH A,|1-2^20:
XCHD A,|2^30:
XRL A,|1-2^D0:
XRL A,|3^D3:
```

Assembler Definition

This final and optional section of the assembler table allows the user to place any assembler directive or mnemonic in the table. An asterisk "*" must be placed between this and the previous section of the table. The assembler section may be used to define special function registers, or assembler attributes with the HOF, WDLN, EQU, ALGN, ALIAS, RPTXT or PAGE directives. The user is able to hide all Cross-32 configuration directives, except for CPU, in the table. Since the CPU table is only read on the first pass, directives that actually write hexcode, such as DFB, should not be placed in this section of the table.

For the 8048, one might wish to continue to use the register designation for other areas of on board RAM. For backward compatibility with earlier versions, this is not done in the 8048 table supplied with Cross-32. A good example of assembler definition is in the ST9 table, file ST9.TBL.

```
*
        HOF     "INT8"
R24     EQU     24
R25     EQU     25
R26     EQU     26
; etc
```

This page intentionally left blank.

This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.

Index

—L—

—M—

—N—

—O—

—P—

—Q—

—R—

—S—

—T—

—U—

—W—

—X—

Registration

All users who purchased the Cross-32 Meta-Assembler directly from Data Sync Engineering are automatically registered. Others, who purchased the product through a distributor, or the purchasing department of a large institution, should return this page in the envelope provided to Data Sync Engineering. Should you not have the original manual page and envelope, please include a copy of your receipt or invoice. A **FREE** CDROM update disk will be air mailed to you, anywhere in the world, containing the latest release of Cross-32 Version 4.0.

This will insure that you personally receive updates, discounts on future products, or other information published by Data Sync Engineering.

Name:

_____

Title:

_____

Company:

_____

Address:

_____

_____

_____

Country:

_____

Telephone:

_____

Facsimile:

_____

E-mail:

_____

CPUs used:

_____

Comments:

_____

_____

_____

This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.

This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.
This page intentionally left blank.