The Universal Cross Assembler for the Acorn Archimedes

Read your **LICENCE** conditions
BEFORE OPENING THE SOFTWARE ENVELOPE
You should then make one copy of the entire disc, for backup purposes, **BEFORE** using.

Baildon Electronics
1 Fyfe Crescent
Baildon
Shipley
West Yorkshire
BDI7 6DR
Telephone: 0274 580519
Facsimile: 0274 531626

Note:
The contact details above are no longer valid and are given for historical accuracy.

First Edition
Published June 1991 by Baildon Electronics

This edition published by Retro Hardware April 2020

Retro Hardware can be contacted here:
retrohardware@daveejhitchins.plus.com

CONTENTS

Chapter I

Introduction

Cross-32 is a cost effective tool 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 thirty 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 native assembler, which assembles programs for the same processor it runs on. The assembler built in to ARM BBC BASIC V for the Archimedes family running RISC OS is an example of this. A cross-assembler, however, assembles programs for one (or more) target processors which differ from the host processor. Cross-32 does this and more it can also assemble programs for the host processor, although this is not the primary purpose.

Cross-32 from Universal Cross-Assemblers is a meta-assembler, that is it uses a meta-language to describe the relevant aspects of the desired target processor. The meta-language description is stored in the form of a table which is read in early in the first pass. By using a flexible instruction table structure, the assembler can compile assembly language source code for most microprocessors and microcontrollers with an address word length of 32 bits or less. It is also possible to create microprocessor emulators by developing tables that translate the assembly instructions into the equivalent binary codes for the host processor and thereby allowing direct execution. To further enhance its flexibility, Cross-32 can produce a machine code output file in binary or the Intel and Motorola 8, 16 or 32 bit hexadecimal formats, and also, optionally, a file containing an assembled listing.

About this Guide

This Guide is a reference manual for the Archimedes version of the Universal Cross Assembler's Cross-32 assembler. It is not intended as an introduction to assembly language programming, nor is it a reference manual for any of the processors’ instruction sets that are supported or of programming environments for the machines that use them.

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 new processor instruction table. If the user merely wishes to assemble code for one of the processors with an instruction table already provided, then the chapter describing the instruction table can safely be skipped, but the user should carefully study any notes provided in the example given for target processor of interest. To successfully tailor an existing table or to create a new one, the user will need to study the entire manual, and perhaps several of the supplied tables.

`Courier New` type is used for the text of example program source code and commands. Since all characters are the same width in Courier, this makes it easier to see where there should, or should not, be spaces.

Further Information

For each processor that is supported, there is a table describing the instruction set, and a short example program written using some instructions for that processor. With some of the processors, there are comments about the particular implementation that need to be read before the table can be used properly. The tables can be found in the !Cross-32.Tbl subdirectory (hold the Shit key down when clicking on the !Cross-32 Icon in the directory display to find the Tbl subdirectory.) Manuals and data books can be obtained from the manufacturers (or their agents) of the individual processors, see the list of addresses in Chapter VIII.

If programs are being developed for execution (or emulation) under RISC 0S, then the *RISC OS Programmer's Reference Manual* forms essential further reading.

Installing the Assembler

Read your **LICENCE** conditions
BEFORE OPENING THE SOFTWARE ENVELOPE
You should then make one copy of the entire disc, for backup purposes, **BEFORE** using.

The enclosed disc contains a directory with:

- !Cross32 - the Cross-32 Meta-Assembler; containing numerous processor tables (in the !Cross32.Tbl directory)
- Examples directory with assembly source files corresponding to each table supplied.

and a directory with:-

- !System (containing the current version of the Shared C Library - other applications that need this should work with this version; BUT Cross-32 will not work with versions prior to 3.50)
- !SysMerge use this to update all your copies of !System with more recent version(s) of any module(s) supplied in !System;

!SysMerge is used by first double-clicking on the application icon to launch it, which results in a dialogue box being displayed on the screen.

Now you should find and drag in the icon of the !System folder that is to be updated, i.e. the one on your master applications disc or hard disc root directory. Drop this icon anywhere onto the dialogue box window. The file path name will be displayed in the upper name box. Then repeat this with the !System icon from the Cross-32 master disc. The !SysMerge application will then update your master copy of !System by replacing files for which there is now a more recent version, or by adding new files, but it will not delete any files from within the folder.

Both !SysMerge and !Cross32 will respond to the !Help application (supplied with your computer on Application disc 1) to give brief interactive instructions for use. In addition, both of these applications contain !Help files giving more information about their usage. This can be displayed by choosing App.<name> -> Help from the Filer menu.

Floppy Disc Systems (one or two drives)

Copy !Cross32 and the example file for your target processor to working discs, along with your favourite program editor and a copy of !System. If you have two drives, or if your source code is too big, then put these applications on one applications disc, and keep your source files on one or more programs discs otherwise use one working disc for both, to disc swapping. !Cross32 currently occupies about 1/3rd of an 800k floppy disc, with all the supplied tables. If disc space is at a premium then all the files in !Cross32.TBL EXCEPT the table(s) that are actually needed, can be deleted from the working disc, releasing up to 200K.

Hard Disc Systems

Copy !System into the root directory, if this has not already been done, otherwise use !SysMerge to update the existing copy. Then create a new directory. Copy !Cross32 and Examples into it. As you produce programs, they can also be stored in this directory.

Universal Truths Concerning 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. (i.e. if the value of a label has changed on the second pass).

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

The most significant bit of ASCII characters can be set, but this is only useful within comments or quoted strings. Control codes (i.e. 0 - 8, &0B

- &2F, &7F) are treated as being equivalent to a space character; Tabs are expanded, with fixed tab stops every eight columns.
Only one processor assembly instruction or assembler directive is permitted per line. All label declarations must be terminated with a colon ":"
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.
The assembler makes no distinction between upper- and lower-case alphabetic characters.
Blank lines in the assembly source code are reproduced in the assembly listing but are otherwise ignored.
Undefined labels, and expressions that are out of range or otherwise invalid. 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.
File path names should not exceed 256 characters. Cross-32 currently does not support or include a linker or librarian.

Running the Assembler from The RISC 0S Desktop

First, you need to launch the !Cross-32 application in the usual manner, by double-clicking on the !Cross-32 application icon in a directory display. The icon will then appear on the icon bar.
Using the icon bar menu, you can specify whether or not binary output and/or list output files are to be produced, and if so, then what file path names should be used. You should choose the submenu item (using Select, Adjust or by pressing Return) to enable that type of output file to be generated, or choose the corresponding item in the main menu to deselect that type of output. The two file output options will each be ticked in the main menu if that option has been enabled. If you wish the particular setting to be remembered the next time the application is launched, then choose the Save Settings option in the main menu. The format of the hex file is set in the source file using the HOP directive, explained later in this manual. The processor instruction table used by Cross-32 is also specified in the source file using the CPU directive.
To assemble a file, drag the icon representing a copy of the source text and drop it onto the icon bar icon. The assembler will then be called up to process that file and to produce whatever output has been requested.
The assembler itself does not currently multi-task but will run to completion. When it starts, the screen mode changes (temporarily) to mode 0 (or the nearest equivalent that your monitor can display). Any errors found will be displayed on the screen. Error messages generated by the source text are also included in the listing file, if enabled. When

finished, you will be prompted to press the space bar, after which you are returned to the desktop.

Running the assembler from the command line or from within an obey file

Cross-32 may be run using the following command:

```
<path>.C32 sourcefile [-L <listfile>] [-H <hexfile>]
```

where the square brackets [] indicate optional items, and die 0 indicate a file path name to be supplied by the user. Environment strings can also be used, such as “<Cross32$Dir>” which is automatically replaced with the full pathname of the directory containing the C32 program.
The “-L” instructs Cross-32 to produce an assembly listing file using the immediately following file name. The “-H” tells Cross-32 to produce a binary output file using the immediately following file name. If these are omitted the corresponding files will not be produced. All error codes will be displayed on the screen, regardless of whether a list file is created or not. The format of the hex file is set in the source file using the HOF directive, explained later in this manual. The processor instruction table used by Cross-32 is also specified in the source file using the CPU directive. The order in which the source, listing and hex files are specified in the command line does not matter. Filling system, disc and directory path names may be included in the file names.
For example:

```
<Cross32$Dir>.C32 E8051.ASM  -H hexfile
            source:     @.ASM.E8051 or @.E8051.ASM
            Listing:    NONE
            hex:        @.hexfile
```

Running Cross-32 will produce something similar to the following on the screen:

```
C32 E8051.AsM  -H hexfile

Cross-32 Meta-Assembler RISC OS Version 2 1
Copyright (C) 1991 Universal Cross-Assemblers
Copyright (C) 1991 Baildon Electronics

Starting Pass Number 1
Starting Pass Number 2
Checksum = 4055 &00000FD7

End of Assembly — No Errors
```

Since Cross-32 is a two-pass assembler, the source file is read twice. The processor instruction table is only read once and kept in memory for the second pass. Assembly errors will be displayed on the screen during the second pass, in addition to being placed in the listing if one was specified. The listing and binary files are written during the second pass only. The checksum is a byte by byte addition of the compiled binary code.

File Path Names and Directory Searching

If, for each file specified, whether source, include or output file (etc), a full path name is supplied (including filing system name), then only that name is searched for. Otherwise, Cross-32 will search:- first, with the file path name as supplied (i.e. in the currently selected directory); then, in the directory containing the original source file (supplied in the command line); third, in the parent of that directory (if there is one); finally, in the directory of the !Cross32 Application (this is where the tables are kept) - until the file is found or the possibilities are exhausted. With each, it will attempt to try both the name as supplied and also the name with leaf and extension names swapped (e.g. <Path>.6502.TBL becomes <Path>TBL.6502) if there is an extension name provided this provides compatibility with PC and UNIX conventions.

Stopping the Assembler

Execution of Cross-32 may be aborted at any time by pressing the Escape key.

System Requirements

Cross-32 can operate reasonably well from a single floppy disc drive, but runs much faster from a hard disc (IDEFS, SCSI and ADFS should work equally well). Operation from the RAM disc is faster still but do remember to save your files before turning off!

The !Cross32 desktop support application requires about 32K to operate. The Cross-32 assembler program can be made to execute in about 104K for small programs with small CPU tables, but normally demands a minimum of 192K from the RISC 0S when started, which allows for quite large programs. (This value is set using the Wimpslot command in the file !Cross32.!Obey). The maximum amount of memory available to the assembler is determined by the setting of the ‘Next’ slot of the Task Display, which normally defaults to 640K if there is enough memory available in the system. (See the Archimedes User Guide p63-66 for a description of how to change this setting.)

Chapter II

The Assembly 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. The following sub-sections describe the fundamental building blocks which make up the assembly source file. Examples are given for each section, but these have been assembled and taken from the listing file. Therefore, the user does not provide the first 16 characters of these examples.

Assembly Line Format

Only one assembly instruction or assembler directive is permitted per line. The assembly line is free format, labels need not start in column 1. Each line may contain some or all of the following sequence of identifiers:

```
1ine# label:  operation operand(s)  ;comment
```

Where...

Line # is an optional decimal integer in the range of 0 to 65535. This is created by some editors to represent 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 an "_", “.” or “?” and ending with a colon ":" which is assigned to 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 "?"

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 (or the unsigned binary equivalent) 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 an “_”, “.” or “?” and end with a colon “:” even when used with the EQU assembler directive. Characters within the label must be alphanumeric “A-Z, 0-9” or an “_”, “.” or “?” Except when used with the EQU and SETL directives, a label is optional, and is 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 are taken from an actual Cross-32 listing. They use the equate (EQU) directive described later in the manual. Very simply, the equate (EQU) directive assigns the label on its left, the value of the expression on its right.

```
0200                    ORG   0200H             ;Origin
                        ;Valid examples of labels are:
                        ;
1234 =      STRT1:      EQU   1234H             ;Label on
directive
1234 =      LD_UP:      EQU   1234a             ;Label with "_"
0200        alone:                              ;Stand-alone
label
0200 =      DITO:       EQU   $
0987 =      LONG_LONG_LONG:   EQU          987H      ;Any
length
```

Operands

The following sub-sections describe numeric constants, string constants, and arithmetic operators which in combination with labels may be used to form operands.

Integer Constants

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

1) C programming language (i.e. 0377 u 255 = OXFF)

If the first digit is a zero, the integer is taken to be octal, and then the remaining characters must be in the range “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 die value expected (0255 is not equal to 255, but OQSSD = 255). If the decimal number contains an "8" or "9" an error will be generated, otherwise no warning can be given.

2) Trailing alphabetic (i.e. `0111111118 = 377Q = 255D = OFFH`)

The base of integer constant is indicated by a trailing alphabetic character. 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

WARING: 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.

Trailing	Base		Characters
$	Hexadecimal	Base 16	0-9, A-F

Some examples of numeric constants are:

```
                ;valid examples of numeric constants are:
                ;
00AF   =        BIN1:       EQU   10101111B     ;Binary
00FF   =        OCT1:       EQU   3770          ;Octa1
003C   =        OCT2:       EQU   74Q           ;Octa1
00FF   =        OCT3:       EQU   0377          ;Octal C
CPPPP  =        DEC1:       EQU   -1            ;Decima1
000A   =        DEC2:       EQU   10D           ;Decima1
00FF   =        HEX1:       EQU   0FFH          ;Hexadecima1
00FF   =        HEX2:       EQU   0xff          ;Hexadecima1
C00FF  =        HEX3:       EQU   $FF           ;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, Universal Cross-Assemblers 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 in 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 machine code file, starting at the present memory location shown on the far left of the listing.

```
0000                    ORG  0000H
                        ;Va1id examples of string constants are:
                        ;
0000 5965612C20         DFB  “Yea, yea, yea!”
000E 41426162                DFB  “AB”,“ab”
0012 5468652064         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 an 32 bit signed integer value, a logical operator yields only a true or false, I or 0 respectively. Most of the operators and their precedence are taken from the 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 of y
~ y        Ones complement of y
-y         Unary subtraction or twos complement of y
+y         Unary addition of y
INV y      Reversed (INVerted) byte order of y
BLOG y     Converts y into a 12 bit binary log
           representation, i.e. bits 0-7 containing a
           mantissa and bits 8-13 containing an exponent
           (shift count, in multiples of 2 bits)
x * y      Multiplication of x and y
x / y      Division of x by y
x % y      Remainder after division of x by y (modulus)
x + y      Addition of x and y
x - y      Subtraction of y from x
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
```

Some examples of expressions follow:

```
000100                  ORG   0l00H
000100                  HOP   "INTl6"
                        ;Arithmetic expressions grouped in
                        ;decreasing precedence:
00000100 =              EX0l:              EQU  $     ;Program
Counter
                        ;
00000160 =  EX0Z:       EQU   4*{+ 26}     ;script
                        ;
00000000 =  EX03:       EQU   !15          ;Log1cal negation
FFFFFFFO =  EX04:       EQU   ~15          ;one's complement
FFFFFFF1 =  BX0S:       EQU   -15          ;Two's complement
0000000F =  EX06:       EQU   +15          ;Unary addition
78563412 =  EX07:       EQU   INV 12345678H;Invert
                        ;
00000FEO =  EX08:       EQU   0xfe * 16    ;Mu1tiplication
0000000F =  EX09:       EQU   0xfe / 16    ;Division
0000000E =  EXl0:       EQU   0xfe % 16    ;Remainder
                                               ;(Modulus)
                        ;
0000003C =  EX11:       EQU   40 + 20D     ;Addition
00000014 =  EX12:       EQU   40 - 20D     ;Subtraction
                        ;
00123400 =  EX13:       EQU   1234H << 8   ;Left shift
00000012 =  EX14:       EQU   1234H >> 8   ;Right shift
                        ;
00000001 =  EXI5:       EQU   0 < 2        ;Less than
00000001 =  EXI6:       EQU   0 <= 2       ;Less than
                                               ;or equal to
00000000 =  EX17:       EQU   0 > 2        ;Greater than
00000000 =  EX18:       EQU   0 >= 2       ;Greater than
                                           ;or equal to
                        ;
00000000 =  EXI9:       EQU   0 == 2       ;Equa1 to
00000001 =  EX20:       EQU   0 != 2       ;Not equal
                        ;
00000003 =  EX21:       EQU   "3" & 15     ;Bitwise AND
                        ;
00000001 =  EX22:       EQU   10B ^ 3      ;Bitwise XOR
                        ;
000000FF =  EX23:       EQU   2 | 253      ;Bitwise OR
                        ;
00000000 =  EX24:       EQU   0 && 2       ;Logical AND
00000001 =  EX25:       EQU   0 || 2       ;Logica1 OR
                        ; A little arithmetic
                        ;
0000001F =  EX26:       EQU   8|7^16 & 15<<13-135/~{1 -17}
                                               ;1F
```

Chapter III

Assembler Directives

Cross-32 supports many common assembler directives along with several of its own. These assembler directives should be considered as reserved keywords, and must not be duplicated by the processor mnemonics defined in the processor table. The directives are listed below:

ALGN	ALiGN hexadecimal file (repeating setting)
ALIGN	ALIGN hexadecimal file (once off setting)
CPU	CPU table declaration
DFB	DeFine Byte
DFL	DeFine Long integer (high byte first)
DFS	DeFine data Storage
DLL	Define Long integer (Low byte first)
DWL	Define Word (Low byte first)
DWM	Define Word (High byte first)
END	END of program
EQU	EQUate label to permanent value
HEX	HEXadecimal file on/off
HOF	Hexadecimal Output Format
IF, ELSE, ENDIF	Conditional Assembly
INCL	INCLude source file
LIST	LISTing on/off
MACRO, ENDM	MACRO directives
ORG	ORiGin
PAGE	PAGE control SETL SET Label to dynamic value
PASS	Maximum number of passes
TITL	TITLe on listing
WDLN	set program counter WorD LeNgth

ALGN - Align, repeating setting

The align (ALGN) directive may be used to align the hexadecimal output file to a specified word length. This is used to ensure that code and data always begin on even addresses, for processors (such as the TMS320XX), which are only word-addressable.

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 sixteen bytes, or a default of one is used if not specified. The assembler will insert extra bytes, each with a value of zero, until the number of bytes generated by each line of code is the specified multiple of the word length. These bytes will be seen in both the list and hex files.

```
00050                  ORG 50H
                ;
                ;Va1id examples of the ALGN directive are
                ;
00050                  ALGN 1     ;Default value
00050 414243           DFB "ABC"
00053 44               DPB "D"
00054                  ALGN 2     ;Align to 2 byte word
00054 41424300         DFB "ABC"  ;Notice extra 00
00058 4400             DFB "D"
```

ALIGN - Align, one off setting

The align (ALIGN) directive may be used to align the hexadecimal output file to a specified word length. This is used to ensure that code and data always begin on even addresses, for processors (such as the 68000 and the ARM2), which are byte addressable, but which require instructions and word-length data to be word aligned.

The ALIGN directive has the following syntax:

```
label:     ALIGN [expression] ;comment
```

The word length specified in the expression must be within a range of one to sixteen bytes, or a default of one is used if not specified. The assembler will insert extra bytes, each with a value of zero, until the program counter is a multiple of the value specified of the current word length. These bytes will be seen in both the list and hex files

```
00050                   ORG 501-I
                  ;
                  ;Valid examples of the ALIGN directive are
                  ;
00050 414243            DFB  "ABC"
00053                   ALIGN 1    ;Defau1t value
00053 4445              DFB  "DE"
00055 00                ALIGN 2    ;A1ign to next 2 byte
                                   ;boundary
00056 414243            DFB  "ABC"
00059 000000            ALIGN 4    ;A1ign to next 4 byte
                                   ;boundary
0005C
```

CPU - Table Declaration

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_fi1e_name"          ; comment
```

Only the instruction table file name may be specified after the CPU directive. Filing system, disc and directory path names 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.

```
          ;Bxamp1es of the CPU directive
          ;
0043           CPU  "1802 TEL"       ;CPU TABLE
0043           CPU  "ADFS::Projects.$.TBL.8048"
          ;CPU TABLE
```

DFB - Define Byte

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 5 or 7 bytes of this data will be shown in the listing, it is included in the hex file in its entirety.

```
2000                    ORG 2000H
                ;
                ;Va1id examples of the DFB directive are
                ;
2000 5061746965 NOTE:   DFB "Patient Lead off",13,10
                                        ;ASCII string, CR, LF
2012 000102             DFB 0,1,2       ;Integers
2015 48                 DFB 77Q + 9     ;Expression
2017 3F44               DFB 3FH, 44H    ;Hexadecima1
```

DFL - Define Long Integer (Most Significant Byte 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 5 or 7 bytes of this data will be shown in the listing, however, it is included in the hexfile in its entirety.

The DFL directive has the following syntax;

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

Cross-32 must be able to represent the value of each expression using a 32 bit signed or unsigned integer or an assembly 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 4 characters. The DFL directive stores 32 bit signed integers from the most significant byte, at the lowest location, to the least significant byte at the highest location. By using either the DLL or DFL directives, long words may be placed in memory in a format which corresponds to the format used by the target processor.

```
3000                    ORG 3000H
                ;
                ;Va1id examples of the DFL directive are
                ;
3000 00000002   CONST1: DFL 2           ;Numeric constant
3004 00003000           DFL CONST1      ;Label
3008 00000048           DFL 77Q + 9     ;Expression
300C 12345678           DFL 12345678H   ;Hexadecimal
```

DLL - Define Long Integer (Least Significant Byte 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 5 or 7 bytes of this data will be shown in the listing, however, it is included in the hexfile in its entirety.

The DLL directive has the following syntax;

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

Cross-32 must be able to represent the value of each expression using a 32 bit signed or unsigned integer or an assembly 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 4 characters. The DLL directive stores 32 bit signed integers from the least significant byte, at the lowest location, to the most significant byte at the highest location. By using either the DLL or DFL directives, long words may be placed in memory in a format which corresponds to the format used by the target processor.

```
3000                    ORG 3000l-l
                  ;
                  ;Valid examples of the DLL directive are
                  ;
3000 02000000     CONST1: DLL 2           ;Numeric constant
3004 00300000             DLL CONST1      ;Label
3008 48000000             DLL 77Q + 9     ;Expression
300C 78563412             DLL 1234S678H   ;Hexadecima1
```

DFS - Define Storage

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. To ensure that nothing is written to the hexadecimal file, this directive may be used with the HEX directive to turn the hexadecimal file off.

The DFS directive has the following syntax:

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

The expression can be of any form which 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.

```
5000                    ORG 5000H
                ;
                ;valid example of the DFS directive are:
                ;
5000            STORE_B:   DFS  20 * 1      ;Reserve 20 bytes
5014            STORE_W:   DSF  20 * 2      ;Reserve 20 words
503C            STORE_L:   DFS  20 * 4      ;Reserve 20 long
```

DWL - Define Ward (Least Significant Byte 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 5 or 7 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 of the 16 bit value before the most significant byte. By using either the DWM or DWL directives, words may be placed in memory in a format which corresponds to the format used by the target processor.

```
4000                        ORG 4000H
                  ;
                  ;Va1id examples of DWL directive are:
                  ;
4000 0000010002  CONST2:   DWL 0,1,2     ;Numeric constants
4006 4800                  DWL 77Q+9     ;Expression
400A 3F004400              DWL 3FH,44H   ;Hexadec1mal
400E FFFF                  DWL 0FFFFH    ;up to 16 bits
```

DWM - Define Word (Most Significant Byte 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 5 or 7 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. By using either the DWM or DWL directives, words may be placed in memory in a format that corresponds to the format used by the target processor.

```
3000 ORG 3000H
                ;
                ;Va1id examples of the DWM directive are:
                ;

3000 0000000100 CONSTI:  DWM   0,1,2     ;Numeric constants
3006 3000                DWM   CONST1    ;Labe1
3008 0048                DWM   77Q+9     ;Expression
300C 003F0044            DWM   3FH,44H   ;Hexadecima1
3010 FFFF                DWM   0FFFFH    ;up to 16 bits
```

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 and Motorola hexadecimal machine code file, representing the starting address of the assembly program. If the END directive or expression are not included, this starting address will default to zero.

```
          ;Va1id examples of the END directive are:
          ;
0000                END          ;Simp1e format
0000      THE_END:  END  RESET   ;With starting address
0100                END  0100H   ;Famous starting address
```

EQU - Equate Label

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
```

Neither the label or the expression are 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.

```
                ;Va1id examples of the EQU directive are
                ;
0013 =          CR:     EQU 13H       ;ASCII Carriage Return
OOOA =          LF:     EQU 10D       ;ASCII Line Feed
5012 =          CNTR:   EQU $         ;Program counter
1300 =          EXPR2:  EQU CR << 8   ;Formula
```

HEX - Hexadecimal File Control

The hexadecimal file control directive enables the user to turn the output to the machine code file “ON” or “OFF” A machine code file is not produced at all if a file name is not specified in the command line “-H”.

The HEX directive has the following syntax:

```
label:          HEX “mode”          ;comment
```

where “mode” may 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 which the user does not want included in the machine code file.

```
                ;Valid use of HEX directive
                ;
0000                    ORG   0000H    ;Origin
0000                    HEX   “OFF”    ;TURN HEX FILE OFF
                BYTE1:  DFS   1
                BYTE2:  DFS   1
                BYTE3:  DFS   1
                WORD1:  DFS   2
                WORD2:  DFS   2
                WORD3:  DFS   2
0009                    HEX   “ON”     ;TURN HEX FILE ON
```

HOF - Hexadecimal Output Format

The hexadecimal output format (HOF) directive selects the format of the machine code output file.

The HOF directive has the following syntax:

```
label:          HOP "format"   ;comment
```

A label may be included with the HOP 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 “INTI6” 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 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 and SETL directives, 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 machine code output file may be written in three different formats, binary, Intel and Motorola. Although only the Intel and Motorola formats are actually ASCII hexadecimal, the words hexadecimal and hex are used in this document to refer to all three formats.

```
06500                ORG 6500H
           ;Va1id examples of the HOF directive are:
           ;
6500                 HOF “BIN8”     ;Binary 8 bit
006500               HOF “BIN16”    ;Binary 16 bit
00006500             HOF “BIN32”    ;Binary 32 bit
6500                 HOF “INT8”     ;Inte1 8 bit hex
06500                HOF “INT16”    ;Inte1 16 bit hex
6500                 HOF “MOT8”     ;Motoro1a 8 bit hex
006500               HOF “MOT16”    ;Motoro1a 16 bit hex
00006500             HOF “MOT32”    ;Motorola 32 bit hex
6500                 HOF “INHX8M”   ;8 bit merged
           ;Inte1lec hex format, described by Micro Chip
           ;for use with the PIC16C5O.
```

IF, ELSE, and ENDIF - Conditional Assembly

Cross-32 supports conditional assembly using the IF, ELSE and ENDIF directives to define areas of the source file which are or are not to be assembled. This feature is usually used to reconfigure 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
ENDIF                                     ;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 ENDIF 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 ENDIF 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 ENDIF directive, the ELSE directive being optional. IF blocks may be nested 32 deep before a fatal error occurs.
An example of conditional assembly follows, where a microcontroller writes to a printer, which is connected to either a RS-232C serial port or a Centronics parallel port, but not both.

Note: ‘ENDI’ is also accepted as a synonym for ‘ENDIF’

An example follows:

```
6900                    ORG 6900
                ;
                ;An example of Conditional Assembly is:
                ;
0000 =          FALSE:      EQU   OD
FFFF =          TRUE:       EQU   NOT FALSE


                ;In this example use the RS-232C port

FFFF =          RS232C:     SETL TRUE
0000 =          CENTRONICS: SETL FALSE


                ;Conditional Block starts here:
                ;
FFFF =                      IF RS232C
0034 =          IO_PORT:    EQU 34H             ;RS-232C Port
                            ENDIF
                ;
0000 =                      IF CENTRONICS
                IO_PORT:    EQU 44H             ;Centronics Port
                            ENDIF
```

INCL - Include Source File

The include file (INCL) directive tells Cross-32 to insert a specified source file into the one specified in the command line, or a previous include file.

The INCL directive has the following syntax:

```
label:      INCL "sour:ce_fi1e_name"      ;comment
```

Included source files may only be nested a maximum of 16 deep before a fatal error occurs. A disc drive and/or directory pathname 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 non-existent include file will result in a fatal error.

```
            ;An example of a INCL directive is:
            ;
23A4                    INCL "Projects$eg1>.AsM.IO"
```

LIST - List File Control

The list file control directive enables the user to turn the output to the list file “ON” or “OFF” A list file is not produced at all if a list file name is not specified in the command line (-L TEST.LST).

The LIST directive has the following syntax:

```
label: LIST “mode”  ;comment
```

where “mode” may 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
0000                    ORG 0000H
0000                    LIST “OFF”    ;TURN LIST FILE OFF
000B                    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 which are to be repeated when called.

Macro assembly has the following syntax:

```
Label: MACRO                                    exp(1),
exp(2),...exp(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 are 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.

```
0000            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  DCALL 5    ;BDOS ENTRY
                        ENDM
                ;
0000                    INPUT0C012H, 80
0000 OEOA               MVI   C,10
0002 1112C0             LXI   D,0C012H   ;SET BUFFER ADDRESS
0005 3E50               MVI   A,80       ;SET BUFFER LENGTH
0007 12                 STAX  D
0008 CD0500             CALL  5          ;BDOS ENTRY
000B                    ENDM
```

ORG - Program Counter Origin

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 HOP 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:
                ;
0000            RESET:  ORG 0           ;A common beginning
0100                    ORG 0100H       ;A famous start
```

PAGE - Page Control

The page (PAGE) directive may be used to both eject the current listing page and set the number of lines in each listing page.

In 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.

```
A000                    ORG 0A000H
            ;
            ;Va1id examples of the PAGE directive are:
            ;

A000                    PAGE 56
A000                    PAGE 0              ;No form feeds
A000                    PAGE 60
A000                    PAGE                ;Form feed
                                            ;before next line
```

SETL - Set Label

The set label (SEI'L) 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 maybe redefined using additional SETL directives else where in the assembly source file.

The SETL directive has the following syntax:

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

Neither the label nor the expression are 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 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.

The SETL directive is most commonly used with conditional assembly.

```
             ;Va1id examples of the SETL directive
are:
             ;
0000 =       CPM:    SETL 0          ;CPM is false
FFFF =       MS_DOS:SETL -1          ;MS_DOS is true
```

PASS - Set Maximum Number of Passes

The PASS directive is used to specify the maximum number of passes that Cross-32 will make over that program source. Cross-32 will only ever make more than 2 passes if there are phase errors during the second (or each subsequent) pass. The instruction sets of most processors contain instructions that have different opcode (binary) lengths but have identical syntax. Usually, there are only at most two versions, e.g. with 8 or 16 bit offsets, so three passes is enough to optimise the code to the minimum valid length. However, some processors, such as the Transputer, have more versions so more passes may be needed in order to find the optimal lengths without having range errors.

The PASS directive has the following syntax:

```
label:     PASS      [expression]   ;comment
```

The number of passes specified in the expression must be within a range of one to sixteen, or a default of three is used if not specified. The directive can appear anywhere within the source text, but it is not meaningful to use more than one as only the last one will have any effect. A label may be included with the PASS directive, which will be assigned the current value of the program counter.

```
                ;Example of the PASS directive
                ;
0043                  PASS 7        ;No more than 7 passes
```

TITL - Title of Listing

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.

```
A800                ORG 0A800H
                ;
                ;Va11d examples of the TITL directive are:
                ;
A800                TITL “Cross-32 Test File”
A800                TITL “User’s choice”
```

WDLN - Word Length

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

```
                ;
0000            CPU "THS320.TBL"   ;CPU TABLE
0000            HOF "INT8"         ;HEX OUTPUT FORMAT
0000            WDLN 2             ;2 BYTE WORD LENGTH
                ;
                ;       ORG 0
                ;
0000 7E01       INIT:   LACK 1
0001 7E21               LACK 33
0002 7F8C               CALA
0003 7F8D               RET
                ;
0000                    END
```

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 Intel and Motorola hexcode unlike 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. A hex file splitter, (used to create a 16-bit EPROM from two 8-bit ones), should also work properly. These hexcode standards should be used with extreme caution, when the word length is not set to one (1). The binary format is not affected.

CHAPTER IV

The Listing File

If requested, using the -L directive in the command line, Cross-32 will produce a listing file 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.

Listing Format

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 in chapter 5 of this manual.

Assembly Error Codes

When Cross-32 detects an error during assembly, a second, error message line will be placed immediately following the offending assembly line using the following format:

```
<file> (<row>, <column>): <message>
```

Where . . .

<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 Cross-32's character position on the source file line when the assembly error was detected.

<message> is Cross-32's explanation of the assembly error.

This format has been designed to allow Cross-32 to be used in an integrated development environment. The error code and assembled line are also shown on the video display, so that the user does not have to search the listing (or even generate it) to check for assembly errors. 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. The assembly errors, their meanings, and an example of each, follow:

```
01000                   ORG 1000H
           ;Error 26: Missing operand
01000                   DFB                 ;Missing operand
```

ERROR.ASM(4,32): Error 26 - Missing operand

```
           ;Error 27: - Illegal line number
01000      0G:                              ;Il1egal line number
```

ERROR.ASM(7,3): Error 27 - Illegal line number

```
           ;Error 28: A "Character string" is required
01000                   CPU INT8            ;Character String
```

ERROR.ASM(10,l7): Error 28 - A "Character string" is required

```
           ;Error 29: Missing or illegal label
00000001 =  EQU 1              ;Missing label
```

ERROR.ASM(13,1): Error 29 - Missing or illegal label

```
           ;Error 30: Illegal hexadecimal format
01000                   HOF "TEK8"          ;Illegal hex format
```

ERROR.ASM(16,17): Error 30 - Illegal hexadecimal format

```
           ;Error 31: Unexpected characters at end of line
00000001 = LAB:         EQU 1, 2            ;Unexpected
characters
```

ERROR.ASM(19,20): Error 31 - Unexpected characters at end of line

```
             ;Error 32: Phase error value of label changes
00000001 = LAB1:        EQU 1               ;Phase error
```

ERROR.ASM(22,18): Error 32 - Phase error, value of label changes

```
00000002 = LAB1:        EQU 2               ;Phase error
```

ERROR.ASM(23,l8): Error 32 - Phase error, value of label changes

```
           ;Error 33: Instruction not found
01000 00                MOV A               ;Instruction not
found
```

ERROR.ASM(26,15): Error 33 - Instruction not found

```
            ;Error 34: File control must be ON or OFF
01001                    LIST "YES"          ;File control must
                                             ;be ON or OFF
```

ERROR.ASM(29,17): Error 34 File control must be ON or OFF

```
            ;Error 35: Symbol not found
01001                    xxx                 ;Instruction not
                                             ;found
```

ERROR.ASM(32,33): Error 35 Symbol not found

```
            ;Error 36: operand not in specified range
01001 34                 DFB $1234           ;Too Large
```

ERROR.ASM(35,21): Error 36 operand not in specified range

```
            ;Error 37: Instruction starts with invalid
            ;character
01002                    (MOV                ;Invalid (
```

ERROR.ASM(38,8): Error 37 Instruction starts with invalid character

```
            ;Error 38: Violation of conditional block
            ;(IF-ELSE-ENDI)
                         ELSE                ;No IF
```

ERROR.ASM(41,33): Error 38 Violation of conditional block (IF-ELSE-ENDI)

```
            ;The remaining errors only occur when the
            ;assembler is evaluating an expression


            ;Error 40: Undefined label
101002 00                DFB + LABEL         ;Undefined label
```

ERROR.ASM(50,19): Error 40 Undefined label

```
            ;Error 41: Missing " at end of character string
01003                    LIST "ON            ;File control must
be
                                             ;on or OFF
```

ERROR.ASM(53,52): Error 41 Missing ' at end of character string

```
            ;Error 42: Missing right script bracket }
01003 00                 DFB 4 * {7 3        ;Missing bracket
```

ERROR.ASM(56,27): Error 42 Missing right script bracket }

```
            ;Error 43: Digit is not valid for declared base
01004 3P                 DFB 0779Q                ;I1legal nine
```

ERROR.ASM(59,17): Error 43 Digit is not valid for declared base

```
            ;Error 44: Unexpected second value
01005 00                 DFB 1 2                  ;Two values
```

ERROR.ASM(62,18): Error 44 Unexpected second value

```
            ;Error 45: Undefined operator
01006 00                DFB 1 ’ 2
```
ERROR.ASM(65,18): Error 45 Undefined operator

```
            ;Error 46: Unexpected right script bracket }
01007 00                DEB 4 * {7 - 3}}    ;Extra bracket
```
ERRoR.ASM(68,28): Error 46 Unexpected right script bracket }

```
            ;Error 47: Unexpected end of line
01008 00                DFB 4 *             ;End of line
```
ERROR.ASM(71 20): Error 47 Unexpected end of line

```
            ;Error 48: shift must be less than 32
01009 00                DPB 2 << 33         ;Shift too large
```
ERROR.A8M(74,24): Error 48 Shift must be less than 32

```
            ;Error 49: Unexpected binary operator
0100A 00                DPB 2 ** 33         ;Two operators
```
ERROR.ASM(77 20): Error 49 Unexpected binary operator

```
            ;Error 50: Unexpected unary operator
0100B 00                DFB 33 ~                ;Wrong side
```
ERRoR.ASM(80,19): Error 50 Unexpected unary operator

```
            ;Error 51: string exceeds 4 character
0100C 42434445          DPL “ABCDE”         ;Too long
```
ERROR.ASM(83,24): Error 51 string exceeds 4 character

```
            ;Error 52: Unexpected expression separator
01010 0001              DPB 4 * , 1             ;Unexpected
comma
```
ERROR.ASM(86,20): Error 52 Unexpected expression separator

```
            ;Error 53: Division by zero attempted
01012 00    DFB 4 / 0                       ;Division by zero
```
ERROR.ASM(89,20): Error 53 Division by zero attempted

Chapter V

The Hexadecimal File

The assembler supports the pure binary, and the Intel and Motorola hexadecimal output formats. Although only the Intel and Motorola formats are actually ASCII hexadecimal the words hexadecimal and hex are used in this manual to refer to all three formats. The extended Intel hex format (INTI6) will be produced by default, and any one of eight formats may be selected using the HOP directive. The hexadecimal output file name is controlled using the -H directive in the assembler command line. Examples of each format are provided below for the assembly source file SH0W.ASM.

```
                        ; File SHOW.ASM
                        ;
0000                          HOF "Int8"   ;Hex Output Format
5678                          ORG 12345678h
5678 4865786164 BEGIN:  DFB "Hexadecimal!",10,13
5678                          END BEGIN
```

Hexadecimal File Formats

The binary hexadecimal output 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. 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. If the program counter is reduced with an origin directive, the following warning will appear on the screen and assembly will continue.

```
Warning - Decreasing Program Counter In 'HEX' File
```

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 hexformat. Notice that there is no difference in the hex code produced by the BIN8, BINl6 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)
            48657861646563696D616C210D0A
BIN16 —                 Binary (16 bit format)
            48657861646563696D616C210DOA
BIN32 —                 Binary (32 bit format)
            48657861646563696D616c210DOA
```

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 warming. Intel hex files consist of records of ASCII characters. Each record starts with a colon ":" and ends with an ASCII carriage return (13D) and line-feed (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 de-fault to zero. The remaining Intel features are:

```
NN          2 characters representing the number of data bytes
12345678    characters representing the address of the first
            data byte
00          segment address characters which assembler sets
            to zero
TT          2 characters representing the Record Type, where:
                        00 represents data
                        01 represents end of file
                        02 represents segment address
DD          2 characters representing 1 byte of data
CC          2 characters representing an 8 bit binary checksum
            where:
                        CC= {NN + 12345678 + TT + DD} &0FFH

INT8:-      Intel Hex Format

            :NN5678TTDDDDDDDDDDDDDDDDDDDDDDDDDDCC
            :NN5678TTCC

            :0E5678004865786164656369 6D616C210D0A97
            :0056780131

INT16:-     Extended Intel Hex Format

            :NN000OTT4000CC
            :NN5678TTDDDDDDDDDDDDDDDDDDDDDDDDDDCC
            :NN5678TTCC

            :020000024000F8
            :0E5678004865786164656369 6D616C21D0A97
            :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 a "S" and ends with an ASCII carriage return (13D) and line feed (1OD). 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 2          characters representing the record length,
              including the address, data and checksum fields

12345678      characters representing the address of the
              first data byte

DD            2 characters representing 1 byte of data

CC            2 characters representing an 8 bit binary
              checksum where:

                          CC = ~{NN + 12345678 + DD} & OFFH

MOT8:-        Motorola (8 bit format)
              S1NN5678DDDDDDDDDDDDDDDDDDDDDDDDDDCC
              S9NN5678CC

              S11156784865786164656369 6D616C210DA93
              S90356782E

MOT16:-       Motorola (16 bit format)
              S2NN345678DDDDDDDDDDDDDDDDDDDDDDDDDDCC
              S8NN345678CC

              S212345678486578616465636 96D616C210DOA5B
              S804345678F9

MOT32:-       Motorola (32 bit format)
              S3NN12345678DDDDDDDDDDDDDDDDDDDDDDDDDDCC
              S7NN12345678CC

              S3131234567848657861646563696D616C210D0A4B
              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 Intel and Motorola hexcode unlike 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. A hex file splitter, (used to create a 16-bit EPROM from two 8-bit ones), should also work properly. These hexcode standards should be used with extreme caution, when the word length is not set to one (1). The binary format is not affected.

Chapter VI

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-81.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 normally consists of four sections, to define the processors registers, operands, addressing modes and mnemonics respectively, with an optional fifth section to define mnemonic suffixes. Each of these will be discussed as part of an example for the Intel MCS-48 (8048) microcontroller family. Although this is not the 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 avoid 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 run. 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 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-number,“register0”,“register1”,...“registerN”
.
.
.
register-number,“register0”,“register1”,...“register”
*
```

where:

Register-Number: 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 Universal Cross-Assemblers 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 like 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","Rl"
```

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

```
*
```

Note that it is possible to define aliases for registers, e.g. several processors use an address register as the main stack pointer and allow it to be called "SP" as well as (say) "R7". These can be defined by listing the registers twice in the same line, but with an alternative name (if any) the second time. However, the bit-length for the register field (in the operand definition) should be set for the proper number of actual registers, not the number of register names. See the H8-500 table for an example of this, where 16 names are defined for 8 actual registers, and the associated bit-length is 3 not 4.

An alternative approach to the definition of register names is to use ordinary labels to represent each register. The register names then need to be defined at the beginning of the source text for each program using the EQUate directive. This has the advantage that the associated values do not have to be numbered sequentially from zero, e.g. for the TMS34010 processor, where it was convenient to number registers A0 to A15 from $C0000A00 to $C0000AF0. Also, some manufacturers allow aliases for register names to be defined at will by the programmer

within the source text (e.g. Acorn for the ARM processor), which can only be implemented here by using labels for register names.

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 bitwise ORed with the instruction's opcode during assembly. Each line of the operand definition has the following syntax:

```
operand-number, start-bit, bit-length, expression, low, high
.
.
.
operand-number, start-bit, bit-length, expression, low, high
*
```

where:

Operand-Number: 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 Universal Cross-Assemblers 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.

Expression: 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 register definition section of the table.
3) An opening single quote " ' " represents the length of an instruction's hexcode bytes. This is most often used with relative branch instructions. A tilde "~" was used for this purpose in versions of Cross-32 earlier than 1.5.
4) "&@" immediately followed by a register-number points to a line in the register definition section of the table. This is used where a list of one or more of the registers can be used, with a bit being set in the operand for each that is specified in the program instruction. The bits are allocated from right to left within the field for each

register in the order of the definition line. (The 6809 uses this feature).

5) "&" immediately followed by a register count indicates that a list of registers is expected (i.e. labels used to represent register names), each with a number in the range from 0 to count -1. A bit is set in the operand for each register that is specified in the program instruction. The bits are allocated for each register from right to left within the field using the register value as the bit number. The register count should be the same as the bit-length of the operand field.
6) "&INV" is equivalent to (5) above, but the byte order is reversed to cater for processors such as the ARM where the bytes are ordered within a word from least-significant at the lowest address to most-significant at the highest address.

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 three-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 one-bit value placed in bit seven of the opcode, is defined by:

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

After that, the 8048's immediate operand, a signed eight-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 36 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 must be 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 the assembly language source code. 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 the address mode definition section has the following syntax:

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

where:

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 Universal Cross-Assemblers 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 }
```

where the operand-number is 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.
Note that if you need to use any of the characters “{“, “}”, “\” or “^” as literal constants within the addressing mode, then the following sequences should be substituted: “\{”, “\}”, “\\”, “\^” to avoid confusion with the special uses of these characters. See the ARM table for an example of this.

Hexcode: 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 hexcode, except that the entire line cannot exceed 255 characters. During assembly, the supplied hexcode is converted to a binary value and bitwise 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 hexcodes 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 processors addressing formats can usually be found in the MOVE or LOAD instructions. The inherent addressing mode, which does not have any operands, such as the no operation (NOP) instruction used by many processors, is not defined in this 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 three bit operand at bit five of the one 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

The fourth and final 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^hexcode:
.
.
.
Mnemonic|address_number1-address_number2^hexcode:
*
```

where:

Mnemonic: is an ASCII string representing the instruction. Cross-32 uses what is known as a trie (i.e. special type of binary tree) to search for 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 Universal Cross-Assemblers 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 "I" 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:
```

Hexcode: 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 op-code. There are no limits on the character length of the hex-code, except that the entire line cannot exceed 255 characters. During assembly, the supplied hexcode is converted to a binary value and bitwise 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 hexcodes 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 vertical line is followed by the addressing mode range. 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 at the end of this chapter:

Suffix Definition

The fifth section of the table is defined and used where the processor instruction set has a significant number of optional suffixes that can be applied to a large proportion of the instructions. An example of this is the ARM table, where the 19 varieties of condition codes ("EQ", "NE", "CS" etc) can each be applied to every version of every instruction!

This section has the following syntax:

```
!^hexcode:
Suffix^hexcode:
.
.
.
Suffix^hexcode:
*
```

where:

Suffix: is an ASCII string representing the suffix. This may contain the same characters as the mnemonics listed in the fourth section, except the “1”, which is a special case. This is used to specify the default condition, i.e. where a suffix is allowed but has not been provided.
Hexcode: is the ASCII hexadecimal representation of the part of the instruction’s opcode that is changed by the application of the suffix to the instruction. The format is the same as for the mnemonic definition section.

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

Additional Table Features

There are numerous instruction tables included with Cross-32 which may be studied to gain further insight into the process of creating one. In addition to using the provided tables, Cross-32 provides many other options. The simplest, is that the provided tables may be modified to generate a cross-assembler that is more specific to a particular processor, rather than one that is for a family of processors. This is usually just a matter of deleting or adding several instructions. Therefore, as manufacturers produce new processors based on older processor families, Cross-32 will not become obsolete. Another feature is the ability to alter the manufacturers assembly mnemonics to a format which better suits the user's specific needs or preferences. A single set of common instruction mnemonics could be developed for example, and by only changing the opcodes in the instruction table, these same mnemonics could be assembled into machine language for a number of different processors. Only instructions that are extremely hardware specific would have to be re-examined. Using similar methods, Cross-32 can be used to ease software upgrading to a more recent processor, especially if both processors are from the same manufacturer, and have similar instruction sets.

Checking the Instruction Table

Cross-32 processor tables do not have to be sorted after being created or modified by the user. They should be double checked for accuracy. Any processor table format errors found during assembly will be displayed as a fatal error on the screen.

```
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^AS:
CPL A^37:
CPL C^A7:
CPL F0^95:
CPL F1^B5:
DA A^57:
DEC A^07:
DEC |1^C8:
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^P6:
JF0 |4AB6:
JF1 |4^76:
JMP |5^0400:
JMPP @A^B3:
JNC |4^E6:
JNI |4^86:
JNT0 |4^26:
JNT1 |4^46:
JNZ |4^95:
JT0 |4^36:
JT1 |4^56:
JTF |4^16:
JZ |4^C6:
MOV A,PSW^C7:
MOV A,T^42:
MOV A,|1 2^P0:
MOV A,I3^23:
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:DIS I^15:
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 RB1D5:
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
```

Chapter VII

Non-Assembly Error Messages

During the assembly of a source file, Cross-32 may generate a “Fatal Error” or “Warning” message. Usually these messages will be self-explanatory. Warning messages will not stop assembly, and are a result of a syntax error in the command line or a phase error. Encountering a fatal error will cause Cross-32 to terminate execution. When this occurs, all open files are closed, a fatal error message is sent to the screen and Cross-32 exits to the operating system.
The following example illustrates a warning message when an illegal option “-O”, is placed in the Cross-32 command line. The error count “No Errors” refers to assembly errors only.

```
C32 E8o85.AsM -O E8085 HEX

Cross—32 Meta-Assembler RISC OS Version 2 1
Copyright (C) 1991 Universal Cross-Assemblers
Copyright (C) 1991 Baildon Electronics

Warning — I11ega1 option Ignored
Warning — Extra Source File Ignored

Starting pass number 1

Starting pass number 2

Checksum = 4212
End of Assembly — No Errors
```

The following example illustrates a fatal error when a source file is placed in the Cross-32 command line which does not exist.

```
C32 NOFILE . ASM

Cross-32 Meta-Assembler RISC OS Version 2 1
Copyright (C) 1991 Universal Cross-Assemblers
Copyright (C) 1991 Baildon Electronics

Source file "NOFILE.ASM"
Fatal Error — source file did not open
```

Warning Messages

Warning messages will not stop assembly, and are a result of a syntax error in the command line or a phase error. When such a problem occurs, “Warning —” will appear on the screen, followed by the appropriate message. The message will be one of the following:

```
Illegal Option Ignored
```

Cross-32 only supports the -L and -H command line options. All others will be ignored, as will any file name that follows them.

```
-H Option Ignored — Missing File Name
```

A hexadecimal output file has been requested, but a file name has not been provided. Assembly continues, but a hex file is not produced.

```
-L Option Ignored - Missing File Name
```

A listing has been requested, but a file name has not been provided. Assembly continues, but a listing is not produced.

```
Decreasing Program Counter In ‘HEX’ File
```

An origin (ORG) statement has reduced the value of the program counter in a source program in which either the "BIN8" or "BIN16" hex output format (HOF) has been specified. This can cause a corruption of the hex output file that the user had not intended.

```
Extra Source File Ignored
```

Only one source file can be specified in the command line. Additional files can be included via the INCL directive within the source file. This warning often follows the "Illegal Option Ignored" message.

```
Extra Hex File Ignored
```

Only one hexadecimal output file can be specified after the -H option in the command line. This warning often follows the "Illegal Option Ignored" message.

```
Extra List File Ignored
```

Only one listing can be specified after the -L option in the command line. This warning often follows the “Illegal Option Ignored” message.

`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 syntax but different opcode lengths. Cross-32 assigns the value of the program counter to undefined labels. This can cause Cross-32 to use the instruction with the shortest opcode in the first pass, and an instruction with a longer opcode in the second pass. The 6805 for example has two jump instructions, the first only supports jumps to page zero (00H to 0FFH) while the second supports jumps to the entire address space (00H to 0FFFFH). A jump to an undefined address during the first pass will use the two-byte opcode, while the same jump to a now defined address, which is not on page zero, will use the three-byte opcode during the second pass. Assembly in this manner will produce the shortest, fastest machine code possible. but does require three passes. Users which are not pressed for either space or speed may wish to eliminate this third pass to reduce Cross-32's assembly time. This may be accomplished by defining memory labels at the beginning of the program, not using forward referenced labels and/or removing the shorter opcodes from the instruction table. The final version of your software can always be reassembled with the shorter opcodes installed.

`Fatal Errors`

Encountering a fatal error will cause Cross-32 to terminate execution. When this occurs, all open files are closed, “Fatal Error —” followed by an appropriate message appears on the screen and Cross-32 exits to the operating system. The message will be one of the following:

`No Source File Specified`

A source file name was not included in the command line.

`Source File Did Not Open`

The source file included in the command line did not open. The name of the offending file is displayed on your screen. The file name is incorrect or on a disc drive or directory other than the one specified.

`List File Did Not Open`

The specified list file name is on a non-existent disc drive or directory, or the directory is full.

`Hex File Did Not open`

The specified hex file name is on a non-existent disc drive or directory, or the directory is full.

```
Disc or Directory Full
```

Delete unnecessary files and try again.

```
Too Many Include Files.
```

Include files using the INCL directive may only be nested 16 deep.

```
Too Many Conditional Blocks
```

Conditional blocks using the IF, ELSE and ENDIF directives may only be nested 32 deep.

```
Insufficient memory while loading table
```

This will only occur when using one of the larger tables (i.e. one for a 16 or 32 bit processor), and then only if the Wimpslot value in the file !Cross32.!Obey has been reduced. The solution is to enlarge this value and to ensure that there is enough memory free, perhaps by clearing other loaded applications.

```
Insufficient memory while loading symbol
```

Cross-32 has run out of memory when attempting to store a label in RAM. Reduce the number or the length of the labels in the source file or add more memory. A 1Mb Archimedes should store a minimum of 20,000 labels.

```
Illegal CPU table format
```

Cross-32 has discovered a format error in the CPU table while loading it into RAM. The offending line is displayed on your screen. Correct the problem and run Cross-32 again.

Chapter VIII

Processor Manufacturers

The following is an alphabetical list of the manufacturers (or their agents) from whom further information may be obtained regarding the microprocessors and microcontrollers for which instruction tables are supplied by Universal Cross- Assemblers:

Hitachi Europe Ltd. 21 Upton Road, Walford Herts., WD1 7TB T: 0923 246488 F: 0923 224422	6303 64180 H8/300 H8/500
Intel Corp. Pipers Way, Swindon Wilts., SN3 1RJ T: 0793 696000 F: 0793 641440	8041, 8048 8051, 8096/C196 8085, 8086/88/186/188
Mitsubishi Electric (UK) Ltd. Hertford Place, Denham Way Maple Cross, Rickmansworth Herts., WD3 2BJ T: 0923 770000 F: 0923 775282	37700, 50740/37450
Motorola Semi conductor Products Division Colvilles Road, Kelvin Industrial Estate East Kilbride Glasgow, G75 0TG T: 0355 239101 F: 0355 234582	6800/1/2/3/8, 6805/HC05 6809, 68HC11 68000/8/10/302
National Semiconductor (UK) Ltd. The Maple, Kembray Park Swindon, SN2 6UT T: 0793 614141 F: 0793 22180	COP400 COP800

NEC Semi Conductor (UK) Ltd. — 7500, 78C10
Sunrise Parkway, Linford Wood Business Centre
Linford Wood, Milton Keynes
T: 0908 691133
F: 0908 670290

GE Solid State - RCA Ltd. — 1802/5/6
Beech House, 373 London Road
Camberley Surrey, GU15 3HR
T: 0276 685911

Rockwell International Ltd. — 6502/65C02
Heathrow House, Bath Road
Cranford, Hounslow
Middlesex, TW5 9QW
T: 081 759 2366

Texas Instruments Ltd. — TMS320, TMS340
Manton Lane — TMS370, TMS7000
Bedford, MK41 7PA — TMS9900/95
T: 0234 67466
F: 0234 223459

VLSI Technology Inc — ARM2
(VL86C0l0)
Agent: — ARM3 (VL86C020)
Quarndon Electronics (Semiconductors) Ltd.
Stack Lane, Derby
Derbyshire, DE3 3ED
T: 0332 32651
F: 0332 360922

Zilog (UK) Ltd. — Z8
Zilog House, Moorbridge Road — Z80, Z180, Z280
Maidenhead, SL6 8PL — SUPER8
T: 0628 39200
F: 00628 781227

Note:
The address details above my no longer be valid and are given for historical accuracy

Index

Index

Baildon Electronics

End User

Software Licence Agreement

Licence and Terms

Baildon Electronics and any applicable sublicensors grant to you a non-exclusive, non-transferable licence to use the software programs and related documentation in this package (collectively referred to as the “Software”) on a single processing unit.

THE SOFTWARE AND DOCUMENTATION ARE COPYRIGHTED. YOU MAY MAKE COPIES OF THE SOFTWARE ONLY FOR BACKUP AND ARCHIVAL PURPOSES. UNAUTHORISED COPYING, REVERSE ENGINEERING, DECOMPILING, DISASSEMBLING AND CREATING DERIVATIVE WORKS BASED ON THE SOFTWARE ARE PROHIBITED. TITLE TO THE SOFTWARE IS NOT TRANSFERRED TO YOU BY THIS LICENCE. OWNERSHIP AND TITLE TO THE SOFTWARE AND TO THE ACTUAL CONTENTS OF THIS PACKAGE, INCLUDING THE COPY OF THE SOFTWARE AND THE MEDIA ON WHICH IT IS STORED AND THE ASSOCIATED DOCUMENTATION, ARE RETAINED BY BAILDON ELECTRONICS AND ANY APPLICABLE SUBLICENSORS.

Software Limitations

BAILDON ELECTRONICS DOES NOT WARRANT THAT THE SOFTWARE WILL BE FREE FROM ERROR OR WILL MEET YOUR SPECIFIC REQUIREMENTS. You assume complete responsibility for decisions made or actions taken based on information obtained using the Software. Any statements made concerning the utility of the Software are not to be construed as expressed or implied warranties. BAILDON ELECTRONICS MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE, AND MAKES THE SOFTWARE AVAILABLE SOLELY ON AN “AS IS” BASIS. NEITHER BAILDON ELECTRONICS OR ANY SUBLICENSORS SHALL BE RESPONSIBLE FOR INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES, EVEN IF BAILDON ELECRONICS HAS BEEN INFORMED OF THE LIKELIHOOD OF SUCH DAMAGES OCCURRING.

Limited Warranty on Media and Damages Disclaimer

The media (not the Software) is warranted to the original purchaser against defects in material and workmanship for a period of twelve (12) months from the original purchase. Defective media under warranty will be replaced when it is returned, postage prepaid with a copy of the purchase receipt to Baildon Electronics. IN NO EVENT SHALL BAILDON ELECTRONICS' LIABILITY (WHETHER BASED ON AN ACTION OR CLAIM IN CONTRACT, TORT OR OTHERWISE) TO ANY PARTY EXCEED THE PURCHASE PRICE OF THE PRODUCT. THIS PARAGRAPH EXPRESSES BAILDON ELECTRONICS' SOLE LIABILITY AND YOUR EXCLUSIVE REMEDY.

This warranty does not affect your statutory rights as they may apply in the country or area of purchase.

Tested on the following Acorn RISC OS machines