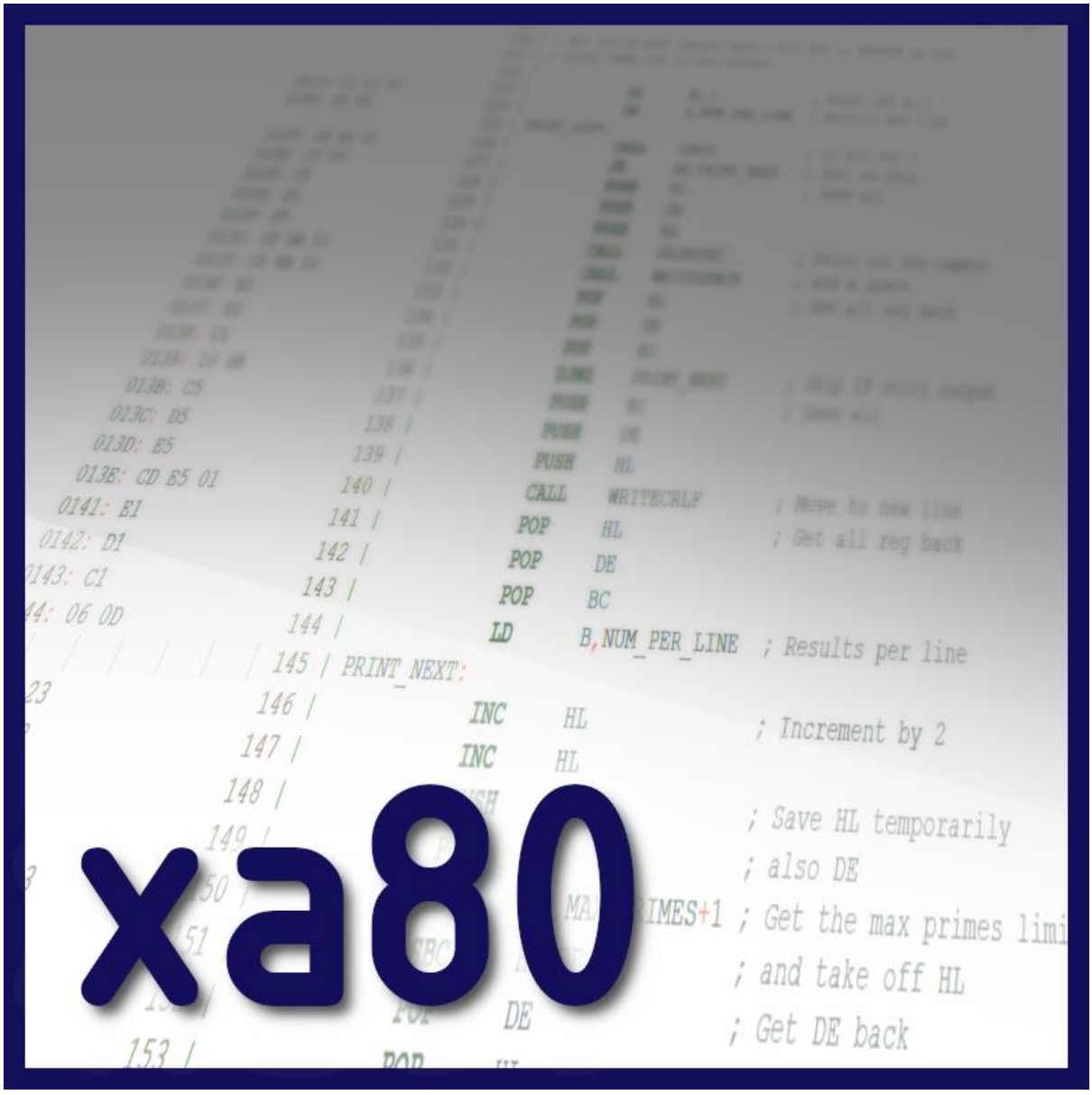
xa80 Cross Assembler
User Manual

V1.0 (DEV)



Contents

Document Release Status

Version	Date	Changes
0.3 R1	08-Jul-2023	Initial issue of V0.3 documentation

Contact

For contact about the content of this document, please contact Duncan Munro:

duncan@duncanamps.com

The software, and this document, can be obtained from:

https://github.com/duncanamps/xa80

1. Introduction

1.1. Disclaimer

xa80 is experimental open source software and is not guaranteed to work correctly in all conditions.

All trademarks are acknowledged as belonging to their respective owners.

1.2. Document purpose

This document is the User Manual for xa80, **x** (Cross) **a**ssembler for x**80** processors. Its purpose is to provide a reference on how the application should be used, with examples where appropriate.

1.3. Application scope

xa80 is a multi-platform cross assembler available for Windows, Linux and MacOS. It is intended to be used with the following 8/16 bit processors:

- 8080
- 8085
- Z80
- Z180

Being open source, the software naturally lends itself to being extensible should other processor types or families be required or other target operating systems be required.

1.4. Application features

Here are some of the key features of xa80:

- Open source
- Two pass assembler
- Supports mnemonics from different processors (8080, 8085, Z80, Z180)
- Opcode compiler so you can add your own secret/hidden instructions and extend to other processor variants in the "family"
- Macro capability with nested expansion of macros allowed
- Conditional assembly with IF / IFDEF / IFNDEF
- Full expression evaluator with many functions and string handling capability
- Rich set of command line switches
- xa80 Environment variable for commonly used parameters
- Runs on any hardware supported by Lazarus/FPC (Windows, macOS, Linux, etc. etc.)

- Fast - will assemble the CP/M BDOS22.ASM (3,289 lines) and CCP22.ASM files (1,325 lines) with map file and listing outputs (total 105 pages) in approx 0.15 seconds using a Core i7 laptop, Acer Aspire 5 A515-56

1.5. Related documents

Opcode compiler user manual

2. xa80 General Structure

2.1. File structure

The general structure of an assembly program is to have one or more source files which can be linked together as part of a larger piece of work.

Alternatively, a single source file could generate a **.COM** file or **.HEX** image which can be executed or transferred respectively.

2.2. Include files

Optionally, include files can be referenced in source files using the **INCLUDE** directive and this can allow common definitions to be shared among a number of assembly source files.

2.3. Segmentation

Starting with xa80 version 1.0, segments can be defined. This allows common groups of code or data to be combined later on as part of a linking process.

2.4. Object files

Also starting with xa80 version 1.0, object files can be created. These describe the segments and labels used within an assembly, along with any external references. Debug information can also be included.

The purpose of object files is that they can be combined into a library and/or combined with other object files to create a larger piece of work.

3. xa80 Line Structure

The assembly is divided into a number of text lines, each of which is processed independently. The main elements of each line are:

- Labels
- Directives
- Instructions
- Macro references
- Operands
- Comments

3.1. Labels

The purpose of labels is to either identify a program location, or to define a variable value.

Labels are predominately alphabetic but can contain a number of allowed special characters. After the first character of the label, digits are also allowed.

The special characters are: **. ? @ _**

Finally, the label may be suffixed by a colon : character, although this is optional. A popular convention is for program locations to use a colon and variable definitions to exclude this, however this convention is not enforced.

3.1.1. Program location labels

Program location labels are used to access embedded data or to facilitate branch/jump/call instructions. They must always start on the first character of the line. A typical example would be:

```
START:                                  ; Program starts here
            LD      B,4                 ; Loop 4 times
            LD      HL,TABLE            ; Point to data
LOOP:       AND     A,(HL)              ; Mask bits
            JR      Z,EXIT              ; Leave if done
            INC     HL                  ; Bump pointer
            DJNZ    LOOP                ; Back round again
EXIT:
            :
            :
            :
TABLE:      DB      0x1F,0x23,0x7A,0xB7
```

3.1.2. Variable definition labels

Labels to define variables can be formed with either of the equate directives; these are **EQU** and **=**.

EQU would normally be used for a variable which is defined once and is unlikely to change during the assembly. If the value is re-assigned, assembly will continue, however a warning will be issued.

= has the same functionality as **EQU**, however no warning is issued when the value changes. This makes it well matched to looping structures such as **WHILE** and **REPEAT** which are covered in sections 5.20 and 5.16 respectively.

Labels defined as program locations can never be redefined and an error will be issued if this is attempted.

Some examples of variable definition labels:

```
MEMSIZE     EQU   8192                ; Set program memory size
FAILMASK    EQU   %10110110           ; Mask for fail bits
MAX_RECS    EQU   128                 ; Max records allowed
REC_SIZE    EQU   14                  ; Size of each record
MEM_USE     EQU   MAX_RECS*REC_SIZE   ; Calculated definition
            I = 0                     ; Reset loop var
            I = I + 1                 ; Increment loop var
```

Label case sensitivity

Labels are, by default, not case sensitive. This can be set by the environment variable and further overridden by the command line option **-k** or **--case-sensitive**. See sections 6.1.2 for more information on the environment and command line variables.

3.1.3. Using commands as labels

Some legacy code may use labels such as **WORD** which is an xa80 reserved word, a synonym for **DW**. In this instance, xa80 will allow the label to be used. A non-terminal warning will be issued.

However, the **WORD** command cannot be used subsequently in the assembly and this will generate a fatal error message if attempted.

Similarly, using any directive will prevent it being used as a label further on in the assembly.

Here are some examples:

```
WORD        EQU   2                   ; Will generate a warning
LABEL       WORD  3                   ; Error, WORD has been used as label

CRLF        DEFB  10,13               ; Use the DEFB directive
DEFB        LD    A,'('               ; Error, DEFB has been used as cmd
```

3.2. Directives

Directives are the commands that drive the assembler behaviour which do not form part of the code itself. Directives may be followed by zero or more operands; this will depend on the nature of the directive itself.

More information on each of the directives and their use can be found in section 5.

Directives can be used “as is” or with a period preceding the directive, for example **EQU** and **.EQU** are interchangeable.

Some examples of directives are:

```
            .ORG  0x0200              ; Set program addr
            WARNOFF                   ; Turn warnings off
            TITLE "Disk controller V2"; Title of reports
            .INCLUDE "macrodef.inc"   ; Include file
            DB    "Hello",13,10,"$"   ; Define some text
```

As directives can conditionally alter the assembly, it’s not good practice to put a label on some directive lines as there is no guarantee the label will be handled correctly. Please review the following example:

```
DELAY       MACRO
            JR    NC, EXIT{#}
            LD    B, 18
D_LOOP{#}:  NOP
            DJNZ  D_LOOP{#}
EXIT{#}:    ENDM
```

The macro, when expanded, will not include the first line **DELAY MACRO** or the last line **EXIT{#}: ENDM** lines so the **EXIT{#}** label will never be created.

But for some directives, a label is desirable or even essential, for example:

```
TABLE:      DW    0x12FA, 0x1410, 0x2AB8…
```

For this reason, the following behaviour is used:

Directive example	Label	Examples
Macro expansion	Will cause Error	ENDM
Not data defining	Will cause Warning	ORG, IF, WHILE, WARNOFF
Data or macro defining	Accepted	DB, DW
Macro definition	Mandatory	MACRO

3.3. Instructions

Instructions are the backbone of the code generation. The general format is:

[label[:]] opcode [operand1 [, operand2] [comment]

The label will always be at the start of the line and may, optionally, be suffixed with a colon. The colon is not stored in the label list or referenced later on.

The opcode is one of the mnemonics defined for the chose processor type. Please note that the mnemonics for the Z80/Z180 are very different to the 8080/8085 although the functionality may be the same.

Whilst the opcodes for the referenced processors are set in stone, it's possible to use the opcode compiler to create extra instructions that are not present in the legacy processors. A separate document details the operation of the opcode compiler[1].

A full list of opcodes available is detailed in appendix section 6.2.

3.4. Macro references

Once a macro has been defined, it can be referenced simply by using the name of the macro. Optional operands may follow the macro name. Some examples of macro references are:

```
            DELAY 18                ; Delay for 18 loops
            MVMEM BUF,TABLE,32      ; Move table to buffer
            WRITE_STR               ; Write out a string to console
```

[1] Amending the opcodes for existing processors will require that the assembler is recompiled to create a new binary. Adding new processors will also require amendments to the assembler source code

3.5. Operands

Operands are optionally used by directives, instructions and macro references. They can be registers, numeric or string, and a rich set of operators and functions allows calculations to be performed.

Expressions involving calculations are covered in more detail in section 4.

3.5.1. Register operands

Register operands are short and fixed definitions which typically refer to processor registers or flag conditions, the list is:

A	H	NC
AF	HL	NZ
AF'	(HL)	P
B	I	PE
BC	IX	PO
(BC)	(IX)	PSW
C	(IX+disp)	R
(C)	IY	SP
D	(IY)	(SP)
DE	(IY+disp)	Z
(DE)	L	
E	M	

Not all operands are available on all processor types, for example PSW is available on 8080/8085, (IX) is available on Z80/Z180.

Any items from the table above, if used on a specific processor, effectively become reserved words. For example, **`PSW`** cannot be used as a variable in a 8085 assembler file as it's a register but it *could* be used in a Z80 assembler file.

Examples of register operands are:

```
        LD    A,B
        LD    HL,BC
        LD    A,(HL)
        LD    (IX+2),A
```

Some of the processor's use **`(IX+n)`** or **`(IY+n)`** where the intended address is a signed number added to the contents of the index register.

It is possible for the assembly file can contain these references *without* the displacement, for example:

```
            OR    A,(IX+0)            ; The correct case, zero displacement
            OR    A,(IX)              ; No displacement used
```

The first is the documented case, although the second example can alternatively be used. The second example will be assembled as if it were **`OR A,(IX+0)`** as there is no underlying Z80/Z180 code for **`OR A,(IX)`**.

3.5.2. Numeric operands

Numeric operands are 8 or 16 bit values which can be numeric constants, addresses, variables, or calculated values.

Numeric constants can be in binary, octal, decimal or hexadecimal forms. Examples are:

```
            LD    A,%10110100         ; Binary
            LD    B,0b10110100        ; Binary
            LD    C,10110100B         ; Binary
            LD    D,123O              ; Letter O, (not zero) for octal
            LD    E,123Q              ; Letter Q also used for octal
            LD    H,95                ; Decimal
            LD    BC, #AB20           ; Hex
            LD    DE,$AB20            ; Hex
            LD    HL,0xAB20           ; Hex
            LD    IX,0AB20H           ; Hex (must start with numeric!)
            LD    IY,2AB8H            ; Hex
```

The use of letters in the constants is not case sensitive, **`q`** and **`x`** are treated the same as **`Q`** and **`X`**.

Addresses can be:

- Numeric operands
- Addresses which have been defined as program locations already
- Addresses which have yet to be defined

The following are examples of the above:

```
DEST:       DS    128                 ; Buffer is 128 bytes
START:      LD    BC,0x0400           ; Number of bytes to copy
            LD    HL,TABLE            ; Get table address, defined later
            LD    DE,DEST             ; Get destination, already defined
TABLE:      DB    1,2,3,5,8,13,…
```

3.5.3. String operands

String operands can be enclosed within single or double quotes. If a numeric value is expected, the ordinal (ASCII) value of the first character of the string is taken. If the string is empty, then 0 is used.

Some examples are:

```
TITLE:      DB    "Hello World"
MESSAGE:    DB    'Error on line '
            LD    A,'0'               ; Offset to convert digit
```

Escape characters are permitted; the following are defined:

Sequence	Result	Notes
\a	07h BEL	Bell character
\t	09h TAB	Tab character
\n	0Ah LF	Line Feed character
\r	0Dh CR	Carriage Return character
\e	1Bh ESC	Escape character
\"	22h "	Double quote
\'	27h '	Single quote
\\	5Ch \	Backslash

3.5.4. Indirection

Some operands use indirection, for example:

```
            LD    A,0x12              ; Load A with the value 0x12
            LD    A,(0x12)            ; Load A with value stored in
                                      ; memory location 0x0012
```

With parenthesis **( )**, the instruction is taken to mean load from the address pointed to by the number. Without parenthesis, as in the first line, the instruction simply loads the literal value into the register.

The assembler also uses parenthesis to alter the precedence of results, however a set of rules allows precedence to be differentiated from indirection. The following examples demonstrate how this works in practice:

```
            LD    A,(1+2*3+4)         ; Indirect
            LD    A,(1+2)*(3+4)       ; Not indirect
            LD    A,(IX+4)            ; Indirect
            LD    A,(1+2)*3           ; Not indirect
```

Please note that this feature is switched off for Intel (8080 / 8085) processors as the opcodes for these do not require or support indirection in the operand field.

3.6. Comments

Comments allow descriptive text to be added without influencing the operation of the assembler. There are three different types of comments available:

Style	Format	Description
1a	optional_text ; comment	Any text from a ; onwards will be treated as a comment. Text prior to the ; will be treated as valid information and will be processed by the assembler
1b	optional text // comment	Any text from the // onwards will be treated as a comment
2	* comment	A * at the start of a line will process all following characters as a comment

The following code example shows how comments can be used:

```
************************************************************************
*                                                                      *
*  ASSEMBLY FILE TEST                                                  *
*                                                                      *
************************************************************************

BIT_MASK    EQU   01101001B           ; Use this to get correct flags
FACTOR      EQU   (10 + 3) * 2        // Calculation used

// Code starts here

START:
            XOR  A,A                  ; Zero A
            :    :
```

4. Expressions

Expressions can be integer or string in nature and are formed from literal values, symbols, operators and functions. Examples are:

```
A > B
1 << bit_5
2 + 3 * 4
LOW(address)
15 * (1 + 2)
Pos("-",title)
IIF(i>5,1,0)
build()
Left(title,3)
IIF(p==0,"Zero","Non-zero")
```

4.1. Literal values

Literal values can be:

- Binary numbers, prefixed by %, 0b or suffixed with B. For example %01101001, 0b11011 or 110B
- Octal numbers, suffixed with letter O or Q. For example 123O or 777q
- Decimal numbers – for example 123, 123D or 0
- Hexadecimal numbers, which can be prefixed by #, $, 0x or suffixed by H[2]. For example #33A, $ff78 or 33AH[3]
- String values enclosed in single or double quotes, for example “MyString”
- ASCII values of characters in single or double quotes, for example ‘A’ returns the hex value 65

4.2. Symbols

Symbols are constant values or variables used within the assembly. They can be associated with:

- A null value
- An integer value
- A string value

A null value is produced when a symbol is declared but has no specific value associated with it. It is of most use with the **`DEFINED()`** function.

[2] For hex literals, and B/H suffixes these are not case sensitive

[3] Hex literals using the H suffix must start with a digit. This is to avoid confusion with labels as FABH could be a hex literal or a label. In this instance, use 0FABH to make it clear to the assembler that this is a literal value

4.3. Operators

The following table of operators has been defined in the assembler. Please see section 4.4 for details of the precedence used when calculating.

Group	Operator	Purpose	Notes
Math	*	Multiply	
	/	Divide	
	+	Add	
	-	Subtract	
	% mod	Modulo	Both % and mod are synonymous
	( )	Grouping	Use to group lower precedence items, e.g. 15*(2+3)
String	+	Concatenate	Adds two strings together
Bitwise	~	Unary NOT	Operates in a 16 bit space, so ~0x1fff yields 0xe000
	<< shl	Shift left	Both << and shl are synonyms
	>> shr	Shift right	Both >> and shr are synonyms
	&	Bitwise AND	
	|	Bitwise OR	
	^	Bitwise XOR	
Logical	!	Logical NOT	
	&&	Logical AND	
	||	Logical OR	
	==	Compare equal	
	!= <>	Compare not equal	!= and <> are synonyms
	<	Compare less than	
	<=	Compare less than or equal	
	>	Compare greater than	
	>=	Compare greater than or equal	

Comparison operators such as <= can be used on numeric values as well as strings. The only proviso is that you don’t try and compare strings with numeric values as this

will trigger an error. String comparisons are case sensitive and this is not affected by the case sensitivity option for labels.

To compare strings in a case insensitive way, use **`UPPER(a) <= UPPER(b)`** for example.

Logical operators take inputs of 0 = False, anything else = True. They will return 0 for False or 1 for True. For example:

```
            DB    123 || 7            ; Yields 1
            DB    (1 > 2) && 17       ; Yields 0
            DB    !123                ; Yields 0
```

4.4. Operator and Expression Precedence

Expressions are evaluated using the following precedence, lowest precedence number is evaluated first:

Precedence	Type	Element
1	Top of food chain	(bracketed expression) Functions
2	Unary expression	+ - ! (logical not) ~ (bitwise not)
3	Multiplicative expression	* / % mod
4	Additive expression	+ -
5	Shift expression	<< shl >> shr
6	Compare expression	< > <= >=
7	Equivalence expression	== != <>
8	Binary AND expression	&
9	Binary XOR expression	^
10	Binary OR expression	|
11	Logical AND expression	&&
12	Logical OR expression	||

4.5. Integer Functions

These are functions returning an integer value. They may be dealing with strings.

Function	Description
ASC(string)	Takes the ASCII ordinal value of the first character of the string. If the string is empty, a value of zero is returned
DEFINED(variable)	Returns 1 if a variable has been defined or 0 if not
HIGH(expression)	Returns the high byte of an expression (bits 8 to 15)
IIF(expression,true_exp,false_exp)	If the expression is non-zero, true_exp is returned otherwise false_exp is returned
LENGTH(string)	Returns the length of a string in characters
LOW(expression)	Returns the low byte of an expression (bits 0 to 7)
ORG() $	Returns the current program counter. $ is a synonym for ORG()
POS(substr,string)	Returns the position of a substring within a string. If the substring is not found, zero is returned
VALUE(string)	Converts a string to a numeric value

4.6. String Functions

A number of string functions are available within xa80:

Function	Description
BUILD()	Provides the build number of the software as a string
CHR(expression)	Converts a numeric expression into an ASCII character. For example CHR(65) gives “A”
DATE()	Return the date as a string in the form YYYY-MM-DD
HEX(expression) HEX(expression, digits)	Returns a string of hex digits which represents the number. If digits is present, it is used to specify the minimum size of the result
IIF(expression,true_exp,false_exp)	If integer expression is non-zero, the string expression true_exp is returned otherwise the

	string expression false_exp is returned
LEFT(string,count)	Take the leftmost count characters from a string
LOWER(string)	Take the lower case value of string
MID(string,start,count)	Take the middle of a string from start for count characters
PROCESSOR()	Returns the processor selected for the assembly, e.g. “Z80”
STRING(number)	Convert a number to a string value
RIGHT(string,count)	Take the rightmost count characters from a string
TIME()	Return the time as a string in the form HH:MM:SS
UPPER(string)	Return the upper case value of a string
VERSION()	Version string for the assembler

5. Directives

This section discusses the directives in more detail.

5.1. CPU

Ignored directive. It has been included for compatibility with earlier source code and raises a warning when used. xa80 is unable to support this directive as the processor type needs to be known at the start to create parsers, tokenisers, etc.

5.2. DB / DEFB – Define Bytes

The **DB** directive allows bytes of data to be defined in memory. These can come from 8 bit signed or unsigned values, or from string values. **DEFB** and **TEXT** are synonyms for **DB**.

Examples are:

```
TABLE:      DB    1,1,2,3,5,8,13,21…
HELLO:      DB    "Hello World"
COUNTL:     DB    LOW(BUFSIZE << 3)
COUNTH:     DB    HIGH(BUFSIZE<<3)
```

5.3. DC / DEFC – Define Characters

The **DC** directive operates identically to the **DB** directive, however strings are always encoded with bit 7 set of the last character in the string. **DEFC** is a synonym for **DC**.

For example:

```
KEYWRD:     DB    "END", "FOR", "NEXT", …
```

Would be encoded as:

```
E  N  D  F  O  R  N  E  X  T
45 4E C4 46 4F D2 4E 45 58 D4 …
```

Instead of:

```
45 4E 44 46 4F 52 4E 45 58 54 …
```

Expressions yielding integers can be added in and these are not affected.

5.4. DS / DEFS – Define Storage

The **DS** directive takes one of two forms. Either with one operand to allocate so many bytes of memory, or two operands to fill memory with a certain byte value. **DEFS** is a synonym for **DS**.

Examples are:

```
BUF:        DS    128                ; Reserve 128 bytes
FPVAL:      DS    4,0                ; Empty floating point value
```

5.5. DW / DEFW – Define Word

The **DW** directive allows bytes of data to be defined in memory. These can come from 16 bit signed or unsigned values. **DEFW** is a synonym for **DW**. Data is stored in little-endian form, the low 8 bits is stored first.

Examples are:

```
TABLE:      DW    1,1,2,3,5,8,13,21…
TABSIZE:    DW    $ - TABLE
COUNT:      DB    BUFSIZE << 3
```

5.6. EXTERN

Not implemented at this time.

5.7. END

Defines the end of assembly. Any operands after the **END** directive will raise a warning. This directive is not actually needed to signal the end of the input.

5.8. EQU / = / SET - Equate

EQUate allows a label to equate to a value. The value can be either a 16 bit integer or a string value, xa80 allows both.

EQU and **=** are similar, the only difference being that if a label is redefined with **EQU**, a warning is displayed. If the label is redefined with = then no warning is issued.

SET is like **=** however it is only available on Intel processors (8080 / 8085) as **SET** is already used as an opcode for Zilog processors.

Some examples:

```
START:                                  ; Program label no warning
LABEL       EQU   10                    ; No warning
J           = 7                         ; No warning
USER        EQU   "Fred"                ; No warning
NAME        =     "Bill"                ; No warning

START:                                  ; Fatal error – redefining prog label
LABEL       EQU   12                    ; Warning – redefining EQU label
J           = J + 1                     ; No warning, = allows this
USER        EQU   "John"                ; Warning – redefining EQU label
NAME        =     "Mike"                ; No warning, = allows this
```

5.9. GLOBAL

Not implemented at this time.

5.10. IF / ELSE / ENDIF

Allows conditional assembly. The format is:

```
            IF    <expression>
            <program block>
              :
            ENDIF
```

or:

```
            IF    <expression>
            <program block>
              :
            ELSE
            <program block>
              :
            ENDIF
```

<expression> is any expression producing a logical value (zero or non-zero). Should the expression evaluate to 0 and the block is not assembled, it will still be necessary to part-assemble each line so that **`ELSE`** and **`ENDIF`** directives can be captured.

An example is:

```
            IF    $ < 0xF800
            DB    "Big long description of the assembler "
            DB    "can be put in here",13,10,"$"
            ELSE
            DB    "Short description",13,10,"$"
            ENDIF
```

When the input is assembled, the resulting listing file will put | at the sidebar to show code which has been included, and : to show code that has been excluded. This is helpful to follow the logic of **IF** / **ELSE** / **ENDIF**.

An example of the listing output is:

```
                            25 |   IF GOODVAL
                            26 |         MSGINFO "Producing some code"
0000: 00                    27 |         NOP
0001: 00                    28 |         NOP
0002: 00                    29 |         NOP
                            30 |     IF 1 > 2
                            31 :         MSGERROR "We shouldn't be here"
                            32 :     ELSE
0003: 3E 7B                 33 |         LD      A,123
                            34 |     ENDIF
                            35 |   ELSE
                            36 :     MSGERROR "We shouldn't be here"
                            37 :     LD      A, LOW(NEWVAL + 5678)
                            38 :   ENDIF
                            39 |
```

5.11. IFDEF / IFNDEF

These operate much like the IF command where the operand is a label name. IFDEF succeeds if the label is defined and IFNDEF if the label is not. The following lines are equivalent:

```
    IFDEF table_base
    IF DEFINED(table_base)
```

5.12. LISTOFF / LISTON

These directives take no operands and decide whether a listing output is produced or not. The directives can be interspersed at any point in the code.

5.13. MACRO / ENDM

Macros allow repetitive coding tasks to be represented as templates which can be deployed multiple times.

5.13.1. General format

The general format is:

```
<label>     MACRO <opt-param-names>
            <macro-definition>
            :
            ENDM
```

5.13.2. Optional parameter names

If parameters are going to be passed to the macro, the names can be listed during the **MACRO** line. These should follow the rules for naming labels, for example not starting off with a digit.

The parameter names are later expanded, using the **{ }** characters.

5.13.3. Macro definition

The macro definition is like any other code block, however substitutions can be used either with a unique expansion serial number using **{#}** or the parameter with **{param-name}**.

5.13.4. ENDM

The ENDM directive must be present as it is the only way of terminating the macro definition.

5.13.5. Macro expansion

Macro expansion takes place when the label used to define the macro is later used as a directive. The following code gives an example of a simple macro definition and it being used.

```
ZERO4       MACRO address
            XOR   A,A                 ; Zero A
            LD    [{address}],A
            LD    [{address}+1],A
            LD    [{address}+2],A
            LD    [{address}+3],A
            ENDM

            ZERO4 fpval1
```

5.13.6. Macro program labels

Program labels defined within macros are always global. Consider the following macro:

```
DELAY       MACRO
            LD    B, 18
D_LOOP:     NOP
            DJNZ  D_LOOP
            ENDM
```

Upon expanding the macro, one program label will be produced which is **D_LOOP**. This is all OK until you reference the macro a second time, it will create another **D_LOOP** variable which will cause an assembler error.

This is resolved by using the macro expansion serial number **{#}**. Each macro expansion will generate a sequentially increasing serial number which is guaranteed to be unique to that expansion. This can be embedded in the label, or indeed anywhere in the macro definition, to be substituted when the macro is expanded.

A rewrite of the previous example could be:

```
DELAY       MACRO
            LD    B, 18
D_LOOP{#}:  NOP
            DJNZ  D_LOOP{#}
            ENDM
```

Using the macro reference DELAY twice in succession, will cause the following code to be assembled:

```
            LD    B, 18
D_LOOP0:    NOP
            DJNZ  D_LOOP0
            :
            LD    B, 18
D_LOOP1:    NOP
            DJNZ  D_LOOP1
```

Note that the labels are now unique to each macro expansion, so no error message will be generated.

5.13.7. Macro parameter expansion

Any parameter names supplied with the macro definition can be expanded. The passed parameter can be an integer value, a string, or even a register name.

An example of the parameters is:

```
DELAY       MACRO cycles
            LD    B,{cycles}          ; cycles contains the parameter
DLY2{#}:    NOP                       ; Short delay
            DJNZ DLY2{#}              ; Loop back if more to do

            DELAY 18                  ; Invoke the macro
```

Another example, this time using registers:

```
SWAP8       MACRO reg1, reg2          ; Swap 8 bit reg (but not A !)
            PUSH  AF                  ; Save A for now
            LD    A,{reg1}
            LD    {reg1},{reg2}
            LD    {reg2},A
            POP   AF

            SWAP8 H,L
            SWAP8 D,B
```

Macro parameters will match the case sensitivity of the assembler labels.

If case sensitivity is off (default or **`--case-sensitive=0`**) then a macro with named parameters of **`Reg1`** and **`Reg2`** can be expanded with **`{REG1}`**, **`{reg1}`** etc.

If case sensitivity is on (**`--case-sensitive=1`**) then a macro with named parameter of **`Reg1`** can only be expanded with **`{Reg1}`**. Using **`{REG1}`** or **`{reg1}`** will fail to expand.

5.13.8. Grouped macro parameters

Macro parameters can be grouped using the **<** and **>** characters to allow parameters to contain commas. The following gives an example of the use of grouped macro parameters:

```
mymacro     MACRO bcode, count
            DW    {count}
            DB    {bcode}
            ENDM

main        mymacro                   <1,2,3,5,8,13,21>, 7
```

The above would expand to:

```
main        mymacro                   <1,2,3,5,8,13,21>, 7
            DW    7
            DB    1,2,3,5,8,13,21
```

5.14. MSGINFO / MSGWARNING / MSGERROR

These provide messages on the console, or error log as the assembly takes place. They are only produced in pass 1 of the assembly.

The format is:

```
            MSGINFO                   "Will show on console as info"
            MSGWARNING                "Will show on console as warning"
            MSGERROR                  "Will show on console as error"
```

Use of MSGERROR will terminate the assembly process.

5.15. ORG

Sets the code origin for the assembler. The assembler starts assembling from address 0 and will increment as code and data bytes are output, unless this instruction is encountered.

An example is:

```
                                      ; Code starts at 0
            ORG   0x0200              ; Set origin to 0200
```

5.16. REPEAT / ENDR

REPEAT allows a block of code to be repeated a pre-defined number of times. For example, the macro example shown in section 5.13.5 could employ from the **REPEAT** directive:

```
ZERO4       MACRO address
            XOR   A,A                  ; Zero A
            I = 0
            REPEAT 4
            LD    [address+I],A
            I = I + 1
            ENDR
            ENDM
```

5.17. SEGMENT

SEGMENT declares either a code or data segment which can be used later as part of an object library to group with similar segments generated from other source files. The general format is:

```
            SEGMENT mandatory-name [, opt_modifier [, opt_modifier [,…]]]
```

If code is generated with no preceding segment definition whatsoever, a default segment named **CSEG** will be created automatically.

5.17.1. Segment modifiers

The optional modifiers describe the segment and influence how it will be used later on at the linking stage.

These are:

Feature on	Feature off	Notes
FIXED	RELOCATABLE	Decides whether a piece of code is fixed (the lowest generated address of code will be taken as the start address), or relocatable in which case the generated code will start from 0000 and end up at a different address once linked
READONLY	READWRITE	Decides whether a piece of memory can be overwritten or not. *Not currently implemented*
UNINITIALISED	INITIALISED	Uninitialised means memory is reserved, however no bytes are written to that area. The linker will simply put 00 bytes in that area; useful for variables. Initialised means that the memory contains defined bytes which could be

		zero or something else
UNINITIALIZED	INITIALIZED	US spelling synonyms for UNINITIALISED and INITIALISED, they do the same thing

'Feature off' is the default so:

```
SEGMENT CSEG         ; is exactly the same as…
SEGMENT CSEG, RELOCATABLE, READWRITE, INITIALISED
```

5.17.2. Segment syntactic checks

The assembler will check for lines which are syntactically incorrect such as:

```
SEGMENT CSEG, RELOCATABLE, FIXED     ; Clash of modifiers
SEGMENT CSEG, FIXED, FIXED           ; doesn't make sense
SEGMENT CSEG, FUBAR                  ; non-existent modifier
```

5.17.3. Segment semantic checks

Some combinations have limited practicality, for example READONLY and UNINITIALISED so a warning or error will be issued. The full list of combinations is:

Movable	Writable	Initialised	Priority	Action
RELOCATABLE	READWRITE	UNINITIALISED	4	
RELOCATABLE	READWRITE	INITIALISED	3	
RELOCATABLE	READONLY	UNINITIALISED		Error – invalid
RELOCATABLE	READONLY	INITIALISED	2	
FIXED	READWRITE	UNINITIALISED		Error – invalid
FIXED	READWRITE	INITIALISED	1[+]	
FIXED	READONLY	UNINITIALISED		Error – invalid
FIXED	READONLY	INITIALISED	1[+]	

[+]There can only be one fixed segment so all of these would be priority 1.
It is unlikely that a fixed segment would be uninitialized or empty, hence the error message.
Priority 1 would be loaded into memory first with priority 4 last.

5.17.4. Segment reintroduction

It may be necessary to bring a segment back into play, in which case just the name should be used. Any modifiers will be ignored on second and subsequent definitions, and a warning message will be issued if they are present.

Example of CSEG being reintroduced:

```
            SEGMENT CSEG, FIXED, READWRITE

            ORG   100H

START:
            LD    HL,(vec_addr)
            LD    A,(HL)
            JP    FINAL

            SEGMENT DSEG, RELOCATABLE

vec_addr:   DW    0                   ; Vector address goes here

            SEGMENT CSEG              ; Reintroduce CSEG NO MODIFIERS!

FINAL:
            RET   ; Back to caller
```

5.18. TITLE

Sets the title of the listing files, an example is:

```
            TITLE "Disk controller V2"
```

5.19. WARNOFF / WARNON

These directives take no operands and decide whether warnings are produced or not. The directives can be interspersed at any point in the code.

They are overridden by the command line and environment in that if the environment/CL turns warnings off, the code cannot turn them on again.

5.20. WHILE / ENDW

The **WHILE** directive is very similar to the **REPEAT** directive. Unlike **REPEAT** where the expression is evaluated once at the start, the **WHILE** directive is evaluated every time it goes round the loop.

For this reason, it is possible for an infinite loop to be created.

An example of the **WHILE** directive is the following piece of code which creates a table of prime numbers:

```
;
; TEST_WHILE.Z80
;
; Test the WHILE and ENDW statements
;
; Duncan Munro
; 05/06/2023
;

;
; Generate a table of prime numbers up to 100
;

MAXPRIME    EQU   100

PRIMETABLE:
            J = 1
            WHILE J <= MAXPRIME
              MSGINFO "J=" + STRING(J)
              PRIME = 1
              K = 3
              WHILE PRIME && (K <= J / 2)
                MSGINFO "  K=" + STRING(K)
                IF (J MOD K) == 0
                  PRIME = 0
                ELSE
                  K = K + 2
                ENDIF
              ENDW
              IF PRIME
                DW J
              ENDIF
              J = J + 2
            ENDW

            PRIMECOUNT = ($ - PRIMETABLE) / 2

            END
```

6. Appendices

6.1. Appendix - Environment

6.1.1. Defaults and precedence

The program environment is controlled by a series of defaults, the xa80 environment variable and the command line options.

In terms of precedence, the command line overrides the environment variable, and the environment variable overrides the defaults.

For example if the default for case sensitive is No, it can be overridden in the environment variable with

```
SET xa80=--case-sensitive=1;option;option;…
```

This will enable the assembler to respect case sensitivity on each use. However, the command line can override this.

```
xa80 source_files\*.asm --case-sensitive=0
```

The above will turn case sensitivity back off again.

6.1.2. Environment list

The following table lists the variables for the assembler and whether they can be amended in the environment or command line:

Short	Long	Values	Default
-b	--debug	*NOT IMPLEMENTED*	0
-c	--com	Specifies .com file	None
-d	--define	Defines one or more symbols	Empty
-e	--errorlog	Specifies the error log file	None
-h	--help	Displays help on the program use	N/A
-i	--include	Sets the include folders to use	Empty
-k	--case-sensitive	0 = Not case sensitive 1 = Case sensitive	0
-l	--listing	Specifies the listing file	None
-m	--map	Specifies the map file	None
-o	--object	Specifies .obj80 file	None

Short	Long	Values	Default
-p	--processor	8080 = Intel 8080 8085 = Intel 8085 Z80 = Zilog Z80 Z180 = Zilog Z180	Z80
-s	--show	Show the specified topic	N/A
-t	--tab	Specifies the tab indent to use, typically 4 or 8	4
-v	--verbose	0 = Silent, only fatal errors 1 = Show only warnings and errors 2 = Normal informational level 3 = Verbose, show more info 4 = WarAndPeace, show lots of info 5 = Debug info, only used while developing	2
-w	--warnings	0 = Warnings off 1 = Warnings on	1
-x	--hex	Specifies the file to use for Intel hex listing	

6.1.3. Specifying files

Filenames can be filenames, folders or the wildcard. They default to empty, so for example running the assembler with the following command line will not produce any output of any kind:

```
xa80 myfile.asm --processor=8080
```

6.1.3.1. Filenames

A set filename can be used on the command line, although it's of limited use for the environment variable.

```
xa80 myfile.asm --hex=output
xa80 test.z80 --hex=myfile.hx2
```

The first line above will create the output file **`output.hex`** in the same folder as **`myfile.asm`** is located. Note that the **`.hex`** extension has automatically been added. If an extension is specified, as in the second line, it will not be overridden.

6.1.3.2. Folders

If the option obviously looks like a folder, it will be used and the output name will be made from the source assembly name with an appropriate extension. For example:

```
xa80 myfile.asm -c c:\temp\comfiles
```

If the folder **`c:\temp\comfiles`** exists, this will assemble **`myfile.asm`** into **`c:\temp\comfiles\myfile.com`** otherwise it will assemble it into **`c:\temp\comfiles.com`**.

To ensure the the parameter is recognised as a folder, put a suitable trailing delimiter for the operating system. For Linux or MacOS the following could be used:

```
xa80 myfile.asm -c /tmp/outputfiles/
```

If the folder **`/tmp/outfiles/`** does not exist, it will be created. The result file will be **`/tmp/outputfiles/myfile.com`**.

6.1.3.3. Wildcard output

It may be desirable to create an output file with the same base name as the input, in which case the following will carry this out:

```
xa80 myfile.asm --com --map
```

The above command will create the output files **`myfile.com`** and **`myfile.map`**.

You may also specify this by **`*`**.

```
xa80 myfile.asm --com=* --map=*
```

The example above has exactly the same outcome as the previous example. Finally, there is an option to use a wildcard with a different file extension. This is useful if you want to assemble a whole directory of **`.asm`** files but want, for example, a **`.bin`** file extension for the output:

```
xa80 *.asm --com=*.bin
```

6.1.3.4. Wildcard environment variable

If it's necessary to produce output files by default, these can be put in the environment variable by using just the * character on its own:

```
SET xa80=--com=*;--map=*
```

The above will always create a suitably named .com and .map file as output.

6.1.4. Symbol defines

Symbols can be predefined either from the command line or environment variable. Definition options are:

- Null value – symbol is defined but doesn't contain anything, useful with the **`DEFINED()`** function as it is set as defined with the value zero
- Numeric value, 16 bit signed or unsigned
- String value

Different fields are separated with the semicolon ; character, for example:

```
xa80 myfile.asm --define=DEBUG;BUFSIZE=128;TITLE="New prog"
```

There cannot be spaces between fields, however there can be spaces within strings.

6.1.5. Include folders

Much like symbol defines, a list of include folders can be submitted with folders separated by ; characters. Folders containing spaces must be enclosed by double quotes. For example:

```
xa80 myfile.asm –include=C:\temp;"c:\users\Duncan Munro"
```

6.1.6. Procesor types

The list of available processor types can be increased by amending the software source code and compiling new opcode lists. This is an activity which is beyond the scope of this document.

6.1.7. Show topic

There are a number of “show” topics available from the assembler, some give useful information to the user, others are of most used when debugging the software. The full list of topics which can be displayed are:

Topic	Description
Distribution	Display the distribution terms for this software
Environment	Shows the environment for the assembler and whether each environment option is a default, set by the xa80 environment variable, or overridden by the command line
Instructions	Shows the list of available instructions / mnemonics for the chosen processor. Best used with the -p / --processor option
Operators	Display a list of the operators and functions provided by the software
Processors	Display a list of the processors supported by the assembler
Reserved	Display a complete list of reserved words that cannot be used for variables (this will vary depending on processor so use -p if needed)
Version	Displays the version and build numbers of the software
Warranty	Displays the warranty provided by the software

The topics are not case sensitive. For example

```
xa80 --show=Version
xa80 --show=RESERVED --processor=8080
```

6.2. Appendix – Opcodes

The following opcodes are defined by the application:

Opcode	8080	8085	Z80	Z180
ACI	Y	Y		
ADC	Y	Y	Y	Y
ADD	Y	Y	Y	Y
ADI	Y	Y		
ANA	Y	Y		
AND			Y	Y
ANI	Y	Y		
BIT			Y	Y
CALL	Y	Y	Y	Y
CC	Y	Y		
CCF			Y	Y
CM	Y	Y		
CMA	Y	Y		
CMC	Y	Y		
CMP	Y	Y		
CNC	Y	Y		
CNZ	Y	Y		
CP	Y	Y	Y	Y
CPD			Y	Y
CPDR			Y	Y
CPE	Y	Y		
CPI	Y	Y	Y	Y
CPIR			Y	Y
CPL			Y	Y
CPO	Y	Y		
CZ	Y	Y		
DAA	Y	Y	Y	Y
DAD	Y	Y		
DCR	Y	Y		
DCX	Y	Y		
DEC			Y	Y
DI	Y	Y	Y	Y
DJNZ			Y	Y
EI	Y	Y	Y	Y
EX			Y	Y
EXX			Y	Y
HALT			Y	Y
HLT	Y	Y		
IM			Y	Y
IN	Y	Y	Y	Y
IN0				Y

Opcode	8080	8085	Z80	Z180
INC			Y	Y
IND			Y	Y
INDR			Y	Y
INI			Y	Y
INIR			Y	Y
INR	Y	Y		
INX	Y	Y		
JC	Y	Y		
JM	Y	Y		
JMP	Y	Y		
JNC	Y	Y		
JNZ	Y	Y		
JP	Y	Y	Y	Y
JPE	Y	Y		
JPO	Y	Y		
JR			Y	Y
JZ	Y	Y		
LD			Y	Y
LDA	Y	Y		
LDAX	Y	Y		
LDD			Y	Y
LDDR			Y	Y
LDI			Y	Y
LDIR			Y	Y
LHLD	Y	Y		
LXI	Y	Y		
MOV	Y	Y		
MLT				Y
MVI	Y	Y		
NEG			Y	Y
NOP	Y	Y	Y	Y
OR			Y	Y
ORA	Y	Y		
ORI	Y	Y		
OTD				Y
OTDM				Y
OTDMR				Y
OTDR			Y	Y
OTI				Y
OTIM				Y
OTIMR				Y

Opcode	8080	8085	Z80	Z180
OTIR			Y	Y
OUT	Y	Y	Y	Y
OUT0				Y
OUTD			Y	Y
OUTI			Y	Y
PCHL	Y	Y		
POP	Y	Y	Y	Y
PUSH	Y	Y	Y	Y
RAL	Y	Y		
RAR	Y	Y		
RC	Y	Y		
RES			Y	Y
RET	Y	Y	Y	Y
RETI			Y	Y
RETN			Y	Y
RIM		Y		
RL			Y	Y
RLA			Y	Y
RLC	Y	Y	Y	Y
RLCA			Y	Y
RLD			Y	Y
RM	Y	Y		
RNC	Y	Y		
RNZ	Y	Y		
RP	Y	Y		
RPE	Y	Y		
RPO	Y	Y		
RR			Y	Y
RRA			Y	Y
RRC	Y	Y	Y	Y

Opcode	8080	8085	Z80	Z180
RRCA			Y	Y
RRD			Y	Y
RST	Y	Y	Y	Y
RZ	Y	Y		
SBB	Y	Y		
SBC			Y	Y
SBI	Y	Y		
SCF			Y	Y
SET			Y	Y
SHLD	Y	Y		
SIM		Y		
SLA			Y	Y
SLP				Y
SPHL	Y	Y		
SRA			Y	Y
SRL			Y	Y
STA	Y	Y		
STAX	Y	Y		
STC	Y	Y		
SUB	Y	Y	Y	Y
SUI	Y	Y		
TST				Y
XCHG	Y	Y		
XOR			Y	Y
XRA	Y	Y		
XRI	Y	Y		
XTHL	Y	Y		

6.3. Appendix – Error Codes

6.3.1. Informational codes

Code	Text	Notes
I0000	<user message>	Generated in response to the MSGINFO directive described in section 5.14
I0001	Assembly started	
I0002	Assembly completed	
I0003	Assembling file <filename>	
I0004	Filename for <output type> is <filename>	Used to describe the hex output file name, com file name, etc.
I0005	Processor is <proctype>	Information and describes the processor type for which the assembly is being made
I0006	Searching for include file <name> at <location>	
I0007	Processing include file <name>	
I9999	<debug message>	Issued to the console or log when debug messages are enabled. Refer to section 6.1.2 for setting verbosity levels. Note that debug messages cannot be produced by production releases of the software

6.3.2. Warning codes

Code	Text	Notes
W0000	<user warning>	Generated in response to the MSGWARN directive described in section 5.14
W0001	Code wrapped round back to zero	Occurs if address FFFFh has been filled in a segment causing the origin counter to go back to zero
W0002	Directive <name> ignored	
W0003	Symbol <identifier> has been redefined	
W0004	Operands after END directive ignored	Some assemblers allow operands after an END directive, e.g. END MYFILE however xa80 ignores them
W0005	Symbol <identifier> is undefined	
W0006	Symbol <identifier> replaces command of the same name	It’s possible to repurpose commands as symbols in certain circumstances, however this will result in a warning
W0007	Macro parameter count	Caused by calling a macro with the

	mismatch - expected <number> parameters, received <number>	wrong number of parameters
W0008	Outputting code with no segment definition, default CSEG created	V1.0 onwards requires at least one segment to operate correctly. If a code generating line is encountered with no segment, a default will be created. Use the SEGMENT command to explicitly set up a segment and avoid this warning
W0009	Segment modifiers ignored, segment <segname> has already been defined	On the first definition of a segment, you may put specifiers such as Fixed, ReadOnly, etc. On subsequent declarations of the same segment name, it will not be possible to use specifiers and they will be ignored
W0010	Segment modifier <modifier> clashes with a preceding modifier	Appears when conflicting or opposing modifers are present, e.g. SEGMENT Fixed,Relocatable
W0011	.COM file is empty	A .COM file has been specified on the command line, however no code has been generated and the resulting .COM file will have zero bytes
W0012	.HEX file is empty	A .HEX file has been specified on the command line, however no code has been generated and the resulting .HEX file will not declare any bytes
W0013	ORG command is forcing relocatable segment <segname> to become Fixed	If a segment has been declared as relocatable, any use of an ORG command will force the segment to have a fixed attribute
W0014	Unresolvable value	The expression parser has encountered operations on an external reference or relocatable internal address. For example if an external address PRINT has been declared, a statement such as DW PRINT<<3 will come up with an unresolvable value

6.3.3. Error codes

Code	Text	Notes
E0000	<user error>	Generated in response to the MSGERROR directive described in section 5.14
E0001	Illegal escape character <char>, valid are <valid_list>	
E0002	Unterminated string <text>	
E0003	Unrecognised content <text>	

E0004	Expected number <text>	
E0005	Integer overflow	
E0006	Binary literal <text> is too short	
E0007	Octal literal <text> is too short	
E0008	Hex literal <text> is too short	
E0009	Divide by zero	
E0010	Expected string <text>	
E0011	Expected positive number <text>	
E0012	String <text> failed to convert	
E0013	Parser error <text>	
E0014	Unable to parse input <text>	
E0015	Code symbol <text> has already been defined	
E0016	Mandatory colon not present in code label <text>	Only of use if mandatory colons for labels are specified. This is likely to be removed in later versions of the software
E0017	Unexpected operands	A directive had operands when operands were not expected
E0018	Operand no. <number> is of indeterminate data type	
E0019	Expected integer	
E0020	Unexpected label <text>, ignored	
E0021	Instruction not available for <text>	
E0022	Operands expected	
E0023	Byte must be in range -127..255	
E0024	Code buffer limit of <number> bytes exceeded	
E0025	Empty string is not allowed	
E0026	Integer must be in the range <number> to <number>	
E0027	Illegal distance of <number> for relative branch, should be -128..+127	
E0028	Bit number for SET/RES must be in the range 0..7	
E0029	Operand for IM instruction must be in the range 0..2	
E0030	Using reserved word for label <text>	
E0031	File not found <text>	
E0032	Unexpected character <text> in input	
E0033	Undefined parser table action for state <number> and token <text>	

E0034	Unexpected token <text> in input	
E0035	PARSER_STACK_SIZE_MAX (<number>) exceeded	
E0036	Invalid character in label	
E0037	Attempt to perform activities after END directive	
E0038	Invalid option <text> for -s/--show command line parameter	
E0039	Mandatory value missing after switch <text>	
E0040	Equals expected but not found in command or environment	
E0041	Premature end of string in command or environment <text>	
E0042	Invalid command line switch <text>	
E0043	<text> is not a valid command directive or processor instruction	
E0044	Include file <text> not found	
E0045	Maximum number of includes (<number>) exceeded	
E0046	Expected label, found <text>	
E0047	Unexpected end: <text>	
E0048	Unexpected ENDIF	
E0049	Unexpected ELSE	
E0050	More than one ELSE statement between IF and ENDIF	
E0051	Unexpected ENDW	
E0052	ENDW in different file to WHILE statement (<text>)	
E0053	Unexpected ENDR	
E0054	ENDR in different file to REPEAT statement (<text>)	
E0055	Unexpected ENDM	Typically occurs if ENDM is encountered when not in a macro definition
E0056	Cannot define a macro within another macro	Nesting of macros at the definition stage is not allowed, however a macro may contain another (predefined) macro
E0057	Macro <text> not found	
E0058	Cannot place a label on an ENDM directive	
E0059	Command line define error, could not process <text>	
E0060	Unexpected directive <text>,	

	have already processed directive for <text>	
E0061	Could not load processor details from <text>	
E0062	Command <text> has already been used as a label	
E0063	Command <text> cannot now be used as a label	
E0064	Failed macro expansion	
E0065	Illegal segment modifier <text>	
E0066	Cannot create .COM file as only 1 fixed segment must be present	
E0067	Cannot create .HEX file as only 1 fixed segment must be present	
E0068	Cannot change Relocatable segment to Fixed after code has been generated	
E0069	Global symbol <text> not found in this module	
E0070	External symbol <text> has already been defined locally	

6.3.4. Internal Error codes

These occur when an error exists within the software; specifically, something has happened internally which should not have happened.

These are not typically caused by a user mistake.

Codes are numbered X3001 to X3018 and specific text will be displayed to allow the finding to be reported. There is also a catch all X3999 which is an unhandled exception error.

7. Index