XA80 Macro Assembler Technical Notes

Duncan Munro

June 2022

Contents

[1 High Level Design 4](#_Toc113612764)

[1.1 Software purpose 4](#_Toc113612765)

[1.2 High level feaures 4](#_Toc113612766)

[1.3 Key components 4](#_Toc113612767)

[2 Assembler 5](#_Toc113612768)

[2.1 General approach 5](#_Toc113612769)

[2.2 First stage parsing 5](#_Toc113612770)

[2.2.1 Labels 5](#_Toc113612771)

[2.2.2 Directives 5](#_Toc113612772)

[2.2.3 Operands 5](#_Toc113612773)

[2.2.4 Comments 6](#_Toc113612774)

[2.3 General source format 6](#_Toc113612775)

[2.4 Files used 7](#_Toc113612776)

[3 Opcode compiler 8](#_Toc113612777)

[3.1 General operation 8](#_Toc113612778)

[3.2 Files and structures 8](#_Toc113612779)

[4 Grammar editor 9](#_Toc113612780)

[4.1 General operation 9](#_Toc113612781)

[4.2 Files and structures used 9](#_Toc113612782)

[5 Detail Design 10](#_Toc113612783)

[5.1 Environment 10](#_Toc113612784)

[5.1.1 Pre-defined Equates 10](#_Toc113612785)

[5.1.2 Grammar Type 10](#_Toc113612786)

[5.1.3 Processor Type 11](#_Toc113612787)

[5.1.4 Command Line Switches 11](#_Toc113612788)

[5.1.5 Environment Variable 12](#_Toc113612789)

[5.1.6 Order of processing 13](#_Toc113612790)

[5.2 Preprocessing / Expansion 13](#_Toc113612791)

[5.3 General processing 13](#_Toc113612792)

[5.4 Source Format 13](#_Toc113612793)

[5.5 Source Processing 14](#_Toc113612794)

[5.5.1 Initial Processing 14](#_Toc113612795)

[5.5.2 Label Processing 14](#_Toc113612796)

[5.5.3 Keyword / Opcode Processing 15](#_Toc113612797)

[5.5.4 Operand Processing – Instructions 15](#_Toc113612798)

[5.5.5 Operand Processing - Directives 16](#_Toc113612799)

[5.6 Macro processing 16](#_Toc113612800)

[5.6.1 Macro recursion 16](#_Toc113612801)

[5.6.2 Replacement of local labels and parameters 17](#_Toc113612802)

[6 Data Types and Structures 18](#_Toc113612803)

[6.1 Preprocessor 18](#_Toc113612804)

[6.1.1 Optimise Comments 18](#_Toc113612805)

[6.1.2 Optimise Keywords 19](#_Toc113612806)

[6.1.3 Optimise Labels 19](#_Toc113612807)

[6.1.4 Optimise Operands 19](#_Toc113612808)

[6.1.5 Optimise Indirection 20](#_Toc113612809)

[6.2 LaCoGen grammar – Operands 20](#_Toc113612810)

[6.3 Opcode compiler 21](#_Toc113612811)

[6.3.1 Opcode compiler operands 21](#_Toc113612812)

[7 File Formats 22](#_Toc113612813)

[7.1 Debug File Format 22](#_Toc113612814)

[7.2 Debug File Line Detail 22](#_Toc113612815)

[8 Appendices 23](#_Toc113612816)

[8.1 List of command line switches 23](#_Toc113612817)

[8.2 List of opcodes by processor type 25](#_Toc113612818)

[8.3 List of grammar sub-options 27](#_Toc113612819)

# High Level Design

This main section details the high level design of the assembler.

## Software purpose

The purpose of the software is to take a source file containing assembly language instructions for the x80[[1]](#footnote-1) series processors and assemble this into an object file which can be executed directly or indirectly in an appropriate processor environment.

## High level feaures

Some of the main functional features of the assembler are:

* Source capabilities
  + Compatibility with original 8080 instructions as well as 8085, Z80, Z180
  + Macro capability with nested macro execution
  + Include file capability
  + Conditional assembly using if / then / else
  + Repeat and while functions
* Advanced features
  + Full expression evaluation
  + Comprehensive error checking
  + Ability to specify different members of the processor family at assembly time
  + Ability to specify different grammatical options
* Output capabilities
  + Object files which can be used with a linker and may include optional debug information
  + Binary images which can be used directly
  + Map files showing all defined labels
  + Hex files
  + Listing files

## Key components

The key components of the software are:

* Assembler, detailed in section 2
* Opcode compiler, detailed in section 3
* Grammar Editor, detailed in section 4

# Assembler

## Overview

The assembler is a command line application that takes source files written in assembly language and converts them into a number of output formats, including binary and listing files.

## General approach

The assembly is conducted in two passes to deal with include files, macro expansions, variable definitions and code generation. The two passes are:

* 1 – Initial pass
* 2 – Final pass

The initial pass will resolve all variable address, preparing for the final pass which carries out the assembly and produces the output files.

Within each pass, the assembly line is parsed by a first stage parser (pre-parser) described in section 2.5, and a second stage parser (expression analyser) described in section 2.6.

### Expansions

There are two types of expansions when processing a file, Macro expansions and file includes. A single stack takes care of each request for a macro expansion or include directive and will contain the following information:

|  |  |  |
| --- | --- | --- |
| Item | Include file | Macro expansion |
| Type of expansion | Include | Macro |
| Name of expansion | Full filename | Macro name |
| Parameters | - | Array of integer |
| Nesting level | Integer | Integer |
| Local Index | Integer | Integer |
| Text | StringList | StringList |
| Current line number | Integer | Integer |

The assembly will start off with the named assembler file being pushed onto this stack and starting at line number 1. Lines will be consumed from the Text at the top of the stack until the information is consumed, at which point the stack top will be popped and the process will repeat for items lower on the stack.

The Local Index is a unique increasing number which specifies the local index. For example, if a macro loop contains the address @LOOP where @ specifies a local label, the label will be appended with the index number, e.g. @LOOP0003. This allows unique labels to be used within each macro or include file.

## General source format

In general, the source code will adhere to the following format:

[label[:]] [directive/instruction [operand[,operand[,…]]]] [comment]

|  |  |  |
| --- | --- | --- |
| Element | Description | Notes |
| Label | Optional label for the line or a mandatory label to be used with an EQU, = or SET directive. Can optionally be suffixed with a colon | Typically starts with a letter although the composition is grammar dependent. If the colon is not used, the label must start in column 1. Grammar overrides may mandate the use of the colon |
| Directive | An assembler directive such as DEFB or INCLUDE. The operands will be dependent on the directive used | The list of directives available will vary with the grammar |
| Instruction | A processor opcode such as MOV or LDIR | Dependent on the processor used |
| Operand | Optional operand to supplement the directive or instruction. For example MOV A,H or DEFB 27,'Hello’ | Dependent on the grammar used in respect of capabilities and text framing |
| Comment | Optional comment which is normally ; followed by any text up to the end of the line | Some grammars may allow the // sequence as in C++, or a \* on the first character of the line |

## Environment

The environment is the set of rules or preferences around the assembler session and consists of:

* Filenames (described in section 1.5)
* Pre-defined equates
* Grammar type
* Processor type
* Other behavioural parameters

The source of the environment, in order of precedence can be:

1. Command line switches
2. Environment variable
3. Defaults

### Pre-defined Equates

Equates can be pre-defined on the command line with the -d / --define switch. Equates can be defined with or without a value, and the value can be an integer or a string of characters.

### Grammar Type

This defines the type of grammar respected by the assembler. Currently only XA80 is recognised, although this can be supplemented to cater for other grammars in the future. Items being considered are:

|  |  |  |  |
| --- | --- | --- | --- |
| Grammar | Author | Product / Source | Status |
| i8080 | Intel | 8080 Programmers Manual | Being considered |
| MACRO80 | Microsoft | MACRO-80 Assembler | Being considered |
| XA80 | duncanamps (open source) | XA80 Cross Assembler | Under implementation |
| Z80RIO | Zilog | Relocating Assembler and Linker | Being considered |

### Processor Type

The processor type can be one of the following:

|  |  |  |
| --- | --- | --- |
| Processor | Covers | Notes |
| 8080 | Intel 8080 | Basic 8080 instruction set using fixed register names. Please see i8080 below for an alternative instruction set |
| 8085 | Intel 8085 | Same as 8080 with addition of RIM and SIM instructions |
| i8080 | Intel 8080 | Same as 8080 but conforms to the documentation in *Intel 8080 Assembly Language Programming Manual, 1975, MCS-482-0275/15K*. Specifically, it defines B=0, C=1, D=2,…, M=6, A=7. Opcode MOV C,A would evaluate as MOV 1,7 which is then parsed as the MOV C,A instruction. This allows the registers to be redefined on the fly, e.g. B SET 2 would make references to the B register encode as C. Only useful with the 8080 grammar defined in section 5.3. as this sets up the initial defines for BCDEHLMA. |
| Z80 | Goldstar Z8400A  NEC uPD780C  SGS Z08400  Sharp LH0080 Zilog Z80 |  |
| Z180 | Zilog Z8018x  Zilog Z8L18x  Zilog Z8S18x |  |

### Command Line Switches

Command line switches will override the environment variable and any defaults. A full list of command line switches is given below:

| Switch | Long form | Parameter | Description | Env? |
| --- | --- | --- | --- | --- |
| -b <bn> | --debug=<bn> | Optional | Set the debug filename to <bn>. If a file extension is not specified, then a default of .dbg80 is used. If <bn> is not specified, the filename is the assembler file name with the filetype changed to .dbg80. If this switch is not present, no debug file is produced | No |
| -g <gt> | --grammar=<gt> | Mandatory | Use the grammar style <gt>. Currently only XA80 is supported. A full list of potential grammars is given in section 2.1.2. The software will first attempt to load an external file in the same place as the main binary called <gt>.grammar which is an XML format file. If this file does not exist, it will fall back to the internally allowed choices, failing this, an error message. This is to allow patching of the grammar rules by provision of a new .grammar file without recompiling the software | Yes |
| -h | --help | Not used | Shows the list of command line options | No |
| -I <id> | --include=<id> | Mandatory | Sets the include directory to <id>. This can be a single directory or a list separated by semicolons | Yes |
| -l <ln> | --listing=<ln> | Optional | Sets the listing filename to <ln>. If a file extension is not specified then a default of .lst is used. If <ln> is not specified, the filename is the assembler file name with the filetype changed to .lst. If this switch is not present, no listing file is produced | No |
| -m <mn> | --map=<mn> | Optional | Sets the map filename to <mn>. If a file extension is not specified then a default of .map is used. If <mn> is not specified, the filename is the assembler file name with the filetype changed to .map. If this switch is not present, no map file is produced | No |
| -o <on> | --object=<on> | Optional | Set the object filename to <on>. If a file extension is not specified then a default of .obj80 is used. If <on> is not specified, the filename is the assembler file name with the filetype changed to .obj80. If this switch is not present, no object file is produced | No |
| -p <pt> | --processor=<pt> | Mandatory | Set the processor type to <pt>. Currently recognised values are 8080, 8085, Z80 and Z180. If no value is specified, a default of Z80 is used. The software will first attempt to load an external file in the same place as the main binary called <pt>.opcode.bin If this file does not exist, it will fall back to the internally allowed choices, failing this, an error message. This is to allow patching of the opcode rules by provision of a new .opcode.bin file without recompiling the software | Yes |
| -s <topic> | --show=<topic> | Mandatory | Show information on a range of topics: • dfae – the DFA table for the expression analyser • dfap – the DFA table for the pre-parser • distribution – the licence rules on distributing the software • grammar – the full list of grammar rules being applied • instructions – the full list of instructions available • nfae – the NFA for the expression analyser  • nfap – the NFA for the pre-parser  • operators – the operators which are used and their priority • reserved – the reserved words used for the chosen processor and grammar • version – the version of the software and other key details • warranty – the warranty available with the software | No |
| -t <n> | --tab=<n> | Mandatory | Sets the tab size to <n>. This value will be used for expanding tabs on source code. If not specified, a default value of 4 is used | Yes |
| -v <n> | --verbose=<n> | Mandatory | Verbose output when assembling. Values are: 0. Normal level (default), 1. Verbose output, 2. War and Peace with lots more output, 3. Debug level. The debug level is only relevant with Debug versions of the software and is not available on production versions of the software | Yes |
| -x <hn> | --hex=<hn> | Optional | Set the hex filename to <hn>. If a file extension is not specified then a default of .hex is used. If <hn> is not specified, the filename is the assembler file name with the filetype changed to .hex. If this switch is not present, no hex file is produced | No |

### Environment Variable

The environment variable XA80, on supported operating systems, can be used to pre-set any of the following switches used by the software, which will override any default values:

-C, --case-sensitive

-d, --define

-g, --grammar

-I, --include

-n, --no-case-sensitive

-p, --processor

-t, --tabsize

-v, --verbose

These are strung together in the same manner as on the command line, e.g.

--define=DEBUG;STYLE=B5 --grammar=XA80 -I source/tables -p 8080 --verbose=1

Note that the include file is additive rather than a replacement. So if the following is true:

Env var = -I c:\temp\includes;”F:\my data\includes”

Command line = --include=c:\temp\oldfiles

Then the include search spec will consist of all three folders in the order of the environment first followed by command line.

### Order of processing

There are three sources of environmental information:

1. Information in Grammar files
2. Environment variable XA80
3. The command line sent to the software

While the environment variable has higher precedence over the grammar files, it is important to decide the grammar file to be used before anything else. The processing for the grammar file takes the following steps:

* Pick a default grammar file (XA80)
* Check if a grammar file is named in the environment variable, if so set the grammar file
* Check if a grammar file is specified on the command line, if so set the grammar file

Now the grammar file is known the rest of the processing can commence with:

* Load the grammar file defined earlier
* Set the environment to the grammar file contents
* Overwrite the environment with any information from the environment variable XA80\*
* Overwrite the environment with any information from the command line\*

\* grammar file type is not overridden in these steps as it has already been selected based on a default / environment variable / commandline.

## Files used

The following files are used and/or generated by the assembler:

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Contents | Ext | Attributes |
| Source | Source code to be assembled | .asm  .inc  .z80  etc. | Input  Human readable |
| Object | Code records produced by the assembler in a binary form which contain assembled code and may include debug information | .obj80 | Permanent  Not readable |
| Hex | Code records produced by the assembler in a text form compatible with the Intel Hex format | .hex | Permanent  Human readable |
| Listing | Listing in human readable form which can be reviewed or printed | .lst | Permanent  Optional  Human readable |
| Log | Details of progress, warnings and errors stemming from the assembly | .log | Permanent  Optional  Human readable |

## First stage parsing

The first stage breaks the input line into four sections described in section 2.2:

1. Label
2. Directive
3. Operand(s)
4. Comment

### Initial Processing

Lines which are empty are not processed any further, other than to include in the listing file if permitted. Active lines have tabs expanded to a default of 4 spaces, however this can be amended on the command line.

### Label Processing

Labels will generally start at character position 1 on the line although this is not mandated. XA80 can figure out what is a label by checking for keywords such as LD, MVI, DEFB etc. and using the context to decide what is a label.

Labels will typically start with an alphabetic character, and maybe some unused control characters such as underscore or #. These are defined in the grammar file. The remainder of the label may also use these characters along with digits. Some examples of valid labels are:

START

\_\_FDECL\_\_

Rec105

@loop15@

There are some exclusions, for example it may not be possible to use $ as this may be set up in the grammar to represents the current program counter. Also, other reserved words such as DB, INCLUDE etc. cannot be used. Labels are not normally case sensitive unless an environmental switch is used to effect this.

### Directive processing

XA80 must decide if an entry in the next position is one of the following options:

* A macro reference, e.g. MOVE\_FPACC
* An ambiguous opcode/directive, specifically SET
* A processor opcode, e.g. LD, XCHG
* A directive, e.g. INCLUDE, DB
* Not one of the above, so an error

Processor opcodes will vary by processor type and are defined in the opcode map file. More details of this are given in section 3.

Directives come from the grammar file, again this is discussed in section 4.

Macro references will always be defined before use so can be looked up from a list of defined macros.

If none of the above lists match, a terminal error condition is displayed.

### Operands

There may be zero or more operands in the assembly line, normally these will be separated by comma characters. Operands, while optional, cannot exist without a directive in place. The content could be:

1. A simple processor operand such as B, (HL), SP
2. A string value such as ‘ABC’
3. A 16 bit numeric value in a variety of formats, such as 123, 0x7AB0, 711Q, 0b10110110
4. An expression combining 2. and 3. with various binary and unary operators
5. A complex processor operand such as [IX+<expr>]

### Operand processing – Instructions

Operand and comment processing for instructions is handled by a mini parser first. Operands can be simple operands such as 12, or [HL], or may be more complex operands such as (IX+rec\_offset-3).

XA80 tries to pull out the simple operands first, failing that it invokes a more advanced parser to resolve the more complex operands.

Note that the label has been removed, also the opcode itself – this is known to be LD by the calling software and does not need to be sent to the parser. All the parser needs to know is there in an opcode and two operands.

Once it returns, XA80 can then determine if the supplied operands are suitable for the opcode by checking against the opcode map.

The operands can be one of the following items:

* A simple operand, e.g A, (HL)
* A complex operand with an expression, e.g. (IX+ZOFFSET)
* An integer expression, e.g. BUFPTR, $-2
* A string expression, e.g. ‘HELLO’

### Operand processing - Directives

Directives will have a set of optional operands which determine how the directive will operate. Simple operands such as number or string will be taken without further processing, however the expression analyser will need to resolve more complex operands.

The expression for directives such as IF, IFDEF, WHILE, REPEAT must be known during phase 1 or it will result in a different set of code being assembled between pass 1 and pass 2.

For example, the following code examples will cause an issue:

IFDEF DEBUG

: : some code

ENDIF

DEBUG EQU 1

BUFFER EQU START+0x0200 ; Calc buffer address

IF BUFFER > 0x0400 ; Big buffer ?

ADD HL, HL ; Yes, double the index

ENDIF

START: MOV A, 0x7A

In the first example above, a reference is made to DEBUG before it has even been defined – the IFDEF will always be false. No warning will be issued as the assembler has no means of knowing if this is deliberate or a mistake.

In the second example, on the first pass the label START will not be known at the time BUFFER is calculated. The assembler will assume that unknown labels are the current program address, and as the constant expression is undefined, a warning will be issued.

### Comments

Comments are the final optional part of the assembler line. The software does not do anything with the comments other than to place them in the listing file. They are effective until the end of the line.

## Second stage parsing

Some of the operands described in section 2.5.3 are resolved by the first stage parser

# Opcode compiler

## Overview

The Opcode Compiler is a command line application which takes a human-readable list of opcodes, operands and code outputs to create a binary file that can be used by the assembler to generate code output.

## Opcode compiler source format

The following fragment from 8080.opcode illustrates what each entry looks like:

CMP M | %10111110

CMP A | %10111111

ANI U8 | %11100110 [1:U8]

XRI U8 | %11101110 [1:U8]

CNZ U16 | %11000100 [1:U16]

The input consists of an opcode, zero or more operands and a binary representation to the right of the ‘|’ character. The comments at the top of each file document all the options.

This input file is then compiled by the oc\_comp utility to turn the 8080.opcode file into 8080.opcode.bin which can be loaded into the assembler.

## Opcode compiler .bin locations

The .opcode and subsequent .bin files created by oc\_comp reside in the program data folder for the operating system. The following command will show the location of the files among other things:

xa80 --show=environment

Once the assembler knows which grammar and processor it will be using, it will search for the .bin file in the program data folder and use it if available.

If the file does not exist, it will try to locate the opcode file within the assembler itself. xa80 has a number of built-on opcode files described in section 3.2.

The reason it attempts to read an external one first, is for those who wish to use the software for assembling only and have no desire to re-compile the software. It allows patching to be carried out if a fault is found in the built-in opcode files.

## Built-in opcode files

The assembler currently has the following opcode files “baked in”:

* 8080
* i8080
* 8085
* z80
* z180

## Opcode compiler operands

The list of operands supported by the opcode compiler is:

|  |  |  |
| --- | --- | --- |
| A  AF  AF'  B  BC  (BC)  C  (C)  D  DE  (DE)  E  H | HL  (HL)  I  IX  (IX+***S8***)  (IX)  IY  (IY+***S8***)  (IY)  L  M  NC  NZ | P  PE  PO  PSW  R  SP  (SP)  ***U16***  (***U16***)  ***U8***  (***U8***)  Z |

Where ***S8*** is a signed 8 bit value (-128..+127), ***U8*** is an unsigned 8 bit value (0..255) and ***U16*** is an unsigned 16 bit value (0..65535).

The operands used in the source .opcode file can only ever come from the list above, it’s not possible to add new registers or addressing modes without changes to the code of the assembler itself.

# Grammar editor

## Overview

The grammar editor is a GUI application which allows the grammar files to be edited. This allows totally new grammars to be created, and existing ones to be amended.

## General operation

## Files and structures used

# Detail Design

## Pre-parser

The pre-parser a lightweight grammar which allows the input to be tokenised into:

* String constants
* Commas
* “Something” else

For example the following input:

HERE: DB 27,’Test’ ; Our definitions

Would be processed into:

|  |  |
| --- | --- |
| HERE: | Something |
| DB | Something |
| 27 | Something |
| , | Comma |
| ‘Test’ | String |
| ; | Something |
| Our | Something |
| definitions | Something |

This then goes through five stages of optimisation to further refine the analysis. These are discussed in sections 3.1.1 to 3.1.5. These will further expand the Something/Comma/String options to:

* Comma
* **Comment**
* **Keyword**
* **Label**
* **Operand**
* Something
* String

At the end of the process there should not be any Something or String entries, they should have been converted into one of the bold categories shown above.

### Optimise comments

The first step in optimisation is to optimise the comments. In our example, the following section:

|  |  |
| --- | --- |
| ; | Something |
| Our | Something |
| definitions | Something |

Is merged into one section marked Comment. Anything with the semicolon or // will become a comment as will anything following it. A \* character in the first position of a line will also force a comment situation.

### Optimise keywords

The next step is to analyse the inputs looking for ‘Something’ entries and matching them against a keyword table. The keyword table consists of directives such as DEFB, INCLUDE, ORG, EQU and opcodes such as LD, MVI, PUSH. The opcodes will vary depending on the processor selected as, for example, Z80 has very different opcodes from 8080.

In our original example:

|  |  |
| --- | --- |
| DB | Something |

The entry DB will be picked up as a keyword and the entry will be marked as such.

### Optimise labels

If a parsed line starts with a ‘Something’ after the keyword optimisation has been carried out, we can safely assume this is a label. It may or may not have a : character on the end. If one exists, it is stripped out.

Much like the keywords routine, the entry will be marked as a label. Checks are applied to ensure the line starts with one of:

* A label on its own or followed by a comment
* A label followed by a keyword
* A keyword

Anything else would represent an error.

### Optimise operands

Optimisation of the operands consists of grouping the items between commas. In our earlier example the ‘Something’ entries already have a one to one relation to operands, however if an input contains:

|  |  |
| --- | --- |
| (IX | Something |
| + | Something |
| ‘E’-‘A’+1) | Something |

All three items will be grouped into one Operand.

### Optimise indirection

One of the issues with the parser is that it’s possible to use brackets to define grouping within a calculation. For example the expression 1+4\*3 would result in 4\*3 being reduced to 12 and the final calculation would be 1+13 = 12.

The operand parser will allow (1+4)\*3 which gives a different result of 15.

It’s the use of brackets that becomes a problem. For example, Z80 code uses brackets to indicate indirection, for example LD A,(HL) or LD A,(0x200).

The parser will try to reduce the (HL) to a define called HL which may not even exist. Likewise, the (0x200) will be reduced to 0x200 which will cause an error as you cannot load a 16 bit value into A. In either instance, the result is not as intended.

The final step for the preprocessor is to intelligently check the operand and replace any outer enclosing brackets with square brackets, for example (HL) 🡪 [HL] and (0x200) 🡪 [0x200].

It’s not just a given that having outer brackets represents indirection, consider the operand (2+2)\*(1+1). While this has outer brackets, it is not an indirection and should stay as it is.

Having gone through five steps of optimisation, our original example will now look like this:

|  |  |
| --- | --- |
| HERE | Label |
| DB | Keyword |
| 27 | Operand |
| , | Comma |
| ‘Test’ | Operand |
| ; Our definitions | Comment |

## Macro processing

Macro processing involves the following actions:

* Recursive processing of macro within a macro
* Replacement of labels with local labels
* Replacement of called parameters

### Macro recursion

A macro stack is used to support recursion.

Checks are in place to ensure a macro does not contain a macro that is already being processed as this would result in an infinite loop.

Each macro expansion instances a new local prefix which is stored on the macro stack.

### Replacement of local labels and parameters

All labels defined in a macro must be stored and used as local labels. For instance the following macro definitions:

COPY\_B MACRO dst,src ; Copy B bytes from [HL] to [DE]

LD HL,src

LD DE,dst

CPYC\_LP LD A,[HL] ; Could use LDIR but this is just

LD [DE],A ; an example

INC DE

INC HL

DJNZ CPYC\_LP

ENDM

COPY\_4 MACRO dst,src ; Copy 4 bytes from [HL] to [DE]

LD B,4

COPY\_B dst,src

ENDM

FPCPY12 MACRO

COPY\_4 FPACC2,FPACC1

ENDM

When expanded with the call:

FPCPY12

Will yield:

>>> FPCPY12

>>> COPY\_4 FPACC2,FPACC1

LD B,4

>>> COPY\_B FPACC2,FPACC1

LD HL,FPACC2

LD DE,FPACC1

@0001@CPYC\_LP LD A,[HL] ; Could use LDIR but this is just

LD [DE],A ; an example

INC DE

INC HL

DJNZ @0001@CPYC\_LP

ENDM

# Data Types and Structures

## Tokens

There are two parsers within the software that tokenise the input material; the following is a list of the token spaces that have been allocated.

There are four token spaces that are recognised:

* Static tokens – these are tokens which will always exist, whatever the weather
* Directive tokens – a fixed set of tokens which represents the directives the assembler uses, such as EQU, DB, INCLUDE
* Opcode tokens – a variable set of tokens which depends on the information in the opcode file
* Operand tokens – a fixed set of tokens based on the operands defined in 3.4

The skeleton table is:

|  |  |  |
| --- | --- | --- |
| ID | Token ID | Purpose |
| IDs 0-9 are used for STATIC TOKENS | | |
| 0 | TOK\_ERR | Error token, could not parse the input |
| 1 | TOK\_EOF | End of input reached |
| 2 | TOK\_COMMENT | A comment |
| 3 | TOK\_UNKNOWN | Unknown data |
| 4 | TOK\_STRING | A quoted string, e.g. “Hello” or ‘xyz’ |
| 5 | TOK\_BINARYVAL | Binary value e.g. 0b1101 |
| 6 | TOK\_OCTALVAL | Octal value e.g. 123Q |
| 7 | TOK\_DECIMALVAL | Decimal value e.g. 7145 |
| 8 | TOK\_HEXVAL | Hex value e.g. |
| 9 | TOK\_LABEL | Label definition |
| IDs 10-34 are used for SIMPLE OPERAND TOKENS | | |
| 10 | TOK\_SOP\_A | A |
| 11 | TOK\_SOP\_AF | AF |
| 12 | TOK\_SOP\_AF\_ | AF’ |
| 13 | TOK\_SOP\_B | B |
| 14 | TOK\_SOP\_BC | BC |
| 15 | TOK\_SOP\_C | C |
| 16 | TOK\_SOP\_D | D |
| 17 | TOK\_SOP\_DE | DE |
| 18 | TOK\_SOP\_E | E |
| 19 | TOK\_SOP\_H | H |
| 20 | TOK\_SOP\_HL | HL |
| 21 | TOK\_SOP\_I | I |
| 22 | TOK\_SOP\_IX | IX |
| 23 | TOK\_SOP\_IY | IY |
| 24 | TOK\_SOP\_L | L |
| 25 | TOK\_SOP\_M | M |
| 26 | TOK\_SOP\_NC | NC |
| 27 | TOK\_SOP\_NZ | NZ |
| 28 | TOK\_SOP\_P | P |
| 29 | TOK\_SOP\_PE | PE |
| 30 | TOK\_SOP\_PO | PO |
| 31 | TOK\_SOP\_PSW | PSW |
| 32 | TOK\_SOP\_R | R |
| 33 | TOK\_SOP\_SP | SP |
| 34 | TOK\_SOP\_Z | Z |
| IDs 35-?? are used for EXPRESSION TOKENS | | |
| 35 | TOK\_FUNC\_LOW | LOW( |
| 36 | TOK\_FUNC\_HIGH | HIGH( |
| 37 | TOK\_EXPR\_LBRACK | ( |
| 38 | TOK\_EXPR\_RBRACK | ) |
| 39 | TOK\_EXPR\_COMMA | , |
| 40 | TOK\_EXPR\_ADD | + |
| 41 | TOK\_EXPR\_DIVIDE | / |
| 42 | TOK\_EXPR\_MODULO | % |
| 43 | TOK\_EXPR\_MULTIPLY | \* |
| 44 | TOK\_EXPR\_SHL | << |
| 45 | TOK\_EXPR\_SHR | >> |
| 46 | TOK\_EXPR\_SUBTRACT | - |
| 47 | TOK\_EXPR\_BITWISEAND | & |
| 48 | TOK\_EXPR\_BITWISEOR | | |
| 49 | TOK\_EXPR\_BITWISEXOR | ^ |
| 50 | TOK\_EXPR\_EQUAL | == |
| 51 | TOK\_EXPR\_GREATER | > |
| 52 | TOK\_EXPR\_GREATEREQUAL | >= |
| 53 | TOK\_EXPR\_LESS | < |
| 54 | TOK\_EXPR\_LESSEQUAL | <= |
| 55 | TOK\_EXPR\_UNEQUAL | != |
| 56 | TOK\_EXPR\_LOGICALAND | && |
| 57 | TOK\_EXPR\_LOGICALOR | || |
| 58 | TOK\_EXPR\_LOGICALXOR | ^^ |
| 59 | TOK\_EXPR\_UNARYMINUS | - |
| 60 | TOK\_EXPR\_UNARYNOT | ! |
| 61 | TOK\_EXPR\_UNARYONESCOMP | ~ |
| 62 | TOK\_EXPR\_UNARYPLUS | + |
| 63 | TOK\_EXPR\_UNARYRESULT | .RES. |

## LaCoGen grammar – Operands

A somewhat heavier LaCoGen grammar is used for the operand processing. This positions the operand as one of the 10 following types:

1. IX on its own
2. IY on its own
3. A simple operand other than IX, IY, for example: HL, A, C, PSW, M
4. [IX + <expression>]
5. [IY + <expression>]
6. [IX]
7. [IY]
8. [ <simple operand> ] e.g. [HL], [C]
9. [ <expression> ] e.g. [0x200]
10. <expression>

A full expression evaluator is built into the grammar. Examples for <expression> could be:

BASE\_ADDR >= 0x1000

(5+2) \* 7 + “A”

(INVAL | BIT\_MASK) & 0xFF

Once the keyword for the opcode and operand type(s) are known, the correct instruction can be selected from the list produced by the Opcode Compiler.

# File Formats

## Debug File Format

The debug file contains source lines and object code which can be used at a later stage by a debugging tool.

The general format is:

Line detail 1

Line detail 2

Line detail 3

: :

This is a binary format.

## Debug File Line Detail

Each debug file line contains the following information in a binary format

16 bit code address, or $FFFF if no code at this address

16 bit code output length indicator

Sequence of bytes output by this line

16 bit string length indicator

String as a sequence of bytes

This continues until the end of the file is reached

# Appendices

## List of command line switches

A full list of command line switches is given below, some may be permitted in the environment variable where indicated.

|  |  |  |  |
| --- | --- | --- | --- |
| Switch | Long form | Description | Env? |
| -b <bn> | --debug=<bn> | Sets the debug info available in the object file. <bn> can be:   * **0** – No debug info (default) * **1** – Line numbers and source names * **2** – Line numbers and all source code | No |
| -g <gt> | --grammar=<gt> | Use the grammar style <gt>. Currently only **XA80** is supported. A full list of potential grammars is given in section 2.1.2. The default, if not specified, is XA80 | **Yes** |
| -h | --help | Shows the list of command line options | No |
| -I <id> | --include=<id> | Sets the include directory to <id>. This can be a single directory or a list separated by semicolons | **Yes** |
| -l <ln> | --listing=<ln> | Sets the listing filename to <ln>. If a file extension is not specified then a default of **.lst** is used. If <ln> is not specified, the filename is the assembler file name with the filetype changed to .lst. If this switch is not present, no listing file is produced | No |
| -m <mn> | --map=<mn> | Sets the map filename to <mn>. If a file extension is not specified then a default of **.map** is used. If <mn> is not specified, the filename is the assembler file name with the filetype changed to .map. If this switch is not present, no map file is produced | No |
| -o <on> | --object=<on> | Set the object filename to <on>. If a file extension is not specified then a default of **.obj80** is used. If <on> is not specified, the filename is the assembler file name with the filetype changed to .obj80. If this switch is not present, no object file is produced | No |
| -p <pt> | --processor=<pt> | Set the processor type to <pt>. Currently recognised values are 8080, 8085, Z80 and Z180. If no value is specified the default from the grammar file is used | **Yes** |
| -s <topic> | --show=<topic> | Show information on a range of topics:   * **distribution** – the licence rules on distributing the software * **environment** – the environment and where it comes from (grammar, environment variable, command line) * **grammar** – the full list of grammar rules being applied * **instructions** – the full list of instructions available * **operators** – the operators which are used and their priority * **reserved** – the reserved words used for the chosen processor and grammar * **version** – the version of the software and other key details * **warranty** – the warranty available with the software | No |
| -t <n> | --tab=<n> | Sets the tab size to <n>. This value will be used for expanding tabs on source code. If not specified, a default value from the grammar or environment variable is used | **Yes** |
| -v <n> | --verbose=<n> | Verbose output when assembling. Values are:   1. Normal level (default) 2. Verbose output 3. War and Peace, lots more output   Debug level\* | **Yes** |
| -x <hn> | --hex=<hn> | 1. Set the hex filename to <hn>. If a file extension is not specified then a default of **.hex** is used. If <hn> is not specified, the filename is the assembler file name with the filetype changed to .hex. If this switch is not present, no hex file is produced | No |

\*-v/--verbose=3 is only effective on debug versions of the software

## List of opcodes by processor type

| 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 |
| 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 |  |  |
| MULT |  |  |  | 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 |
| OTIR |  |  | Y | Y |
| OUT | Y | Y | Y | Y |
| OUT0 |  |  |  | Y |
| OUTD |  |  | Y |  |
| OUTI |  |  | 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 |
| 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 |  |  |

## List of grammar sub-options

Each grammar instance will draw from one or more of the following sub-options:

| Sub-option | Datatype | Description |
| --- | --- | --- |
| CmdDefineBytes | Stringlist | Define Bytes command, e.g. DB;DEFB;DC |
| CmdDefineFloatSingle | Stringlist | Define Float command (IEEE-754 single precision), e.g. DF; DEFF |
| CmdDefineMacro | Stringlist | Define Macro command e.g. MACRO;DEFMACRO |
| CmdDefineString | Stringlist | Define String command, e.g. DM, DEFM |
| CmdDefineStringH | Stringlist | Define String with last character bit 8 set, e.g. DSH |
| CmdDefineStringZ | Stringlist | Define String (zero terminated), .e.g. DSZ |
| CmdDefineWord16 | Stringlist | Define Words command, .e.g. DW;DEFW |
| CmdDefineWord32 | Stringlist | Define Doubleword command, e.g. DD;DEFD |
| CmdEnd | Stringlist | End command, e.g. END |
| CmdEndMacro | Stringlist | End Macro command, e.g. ENDM |
| CmdEndRepeat | Stringlist | End Repeat command, e.g. ENDR |
| CmdEquate | Stringlist | Equate command, e.g. EQU;EQUATE |
| CmdIncludeFile | Stringlist | Include command, e.g. INCLUDE |
| CmdRepeat | Stringlist | Repeat command, e.g. REPEAT;RPT |
| CmdSet | Stringlist | The Set command, e.g. SET;= |
| CommentAnywhere | Stringlist | Comments allowed anywhere other than the start, e.g. ; // |
| CommentStart | Stringlist | Comments allowed at the start of a line. e.g. \* ; // |
| EndBaggage | 1. Disallowed 2. Optional 3. Mandated | Decides if baggage is allowed after the End command, e.g. END MYFILE.ASM |
| EndRule | 1. Disallowed 2. Optional 3. Mandated | Rule for END command |
| EscapeCharacter | Character | Contains escape character if escaping is allowed within quotes or NUL if not allowed, e.g. \ |
| EscapeSet | Set of Character | Set of characters which can be escaped. Typically the escape character itself and some other stuff, e.g. \ “ ‘ n t r. Some special cases where {e} is the escape character:   * {e}a = 07 Alert (bell) * {e}b = 08 Backspace * {e}e = 1B Escape * {e}f = 0C Form feed * {e}n = 0A Newline * {e}r = 0D Carriage return * {e}t = 09 Horizontal tab * {e}v = 0B Vertical tab * {e}{e} = Escape chart * {e}’ = Single quote * {e}” = Double quote * {e}? = Question mark * {e}nnn = Octal character * {e}xhh = Hex character |
| LabelCaseSensitive | Boolean | True if label processing is case sensitive. Can be overridden by the command line and environment variable |
| LabelCharactersStart | Set of Char | Set of characters which can start a label, normally the alphabet and a small handful of others |
| LabelCharactersMid | Set of Char | Characters which can come after the first character of the label |
| LabelColonRule | 1. Disallowed 2. Optional 3. Mandated 4. Mandated on indent | Determines if the colon character : is allowed in labels |
| LabelLocalPrefix | String | Local prefix (if used) for labels |
| LabelLocalSuffix | String | Local suffix (if used) for labels |
| LabelMaximumLimit | Integer | Maximum number of characters allowed in a label. More than this will trigger an assembler error |
| LabelMaximumUsed | Integer | Maximum number of characters recognised in a label. e.g. if this is 6, then FORTUNES and FORTUNE17 will be treated as identical |
| MacroLabelPrefix | String | Prefix, if used, for local labels, e.g. @@ or % |
| MacroLabelRule | 1. No rule always global 2. Local if prefixed 3. Always local | Macro label rule, decides whether labels defined in Macros are local or global |
| MacroParamPrefix | String | Prefix to use on Macro parameters, e.g. @ or # |
| OpcodeSquareRule | 1. Disallowed 2. Optional 3. Mandated | Determines if indirection should be specified by [square] brackets |
| ParserInteropAllowed | Set of Character | In addition to ParserInteropMandated, e.g. {tab} {space} |
| ParserInteropMandated | Set of Character | Characters mandated between operands (only 1 of these characters is required). e.g. , |
| StringTerminator | Set of Character | Characters allowed to be used as string terminators, for example ‘ and “ |

Index

Command line switches

-b, --debug, 8, 25

-g, --grammar, 8, 9, 25

-h, --help, 8, 9, 25, 26

-I, --include, 8, 9, 25

-l, --listing, 8, 25

-m, --map, 8, 25

Comment, 6, 10, 17, 18, 19

Conditional assembly, 4

Directive, 6, 10, 11, 21

Error checking, 4

Expression analyser, 5, 8, 12

Expressions

Evaluation, 4

Grammar

i8080, 7, 14

MACRO80, 7

XA80, 1, 7, 8, 9, 10, 11, 12, 25

Z80RIO, 7

Grammar editor, 2, 16

Hex files, 4

Include file, 4, 5

Instruction, 6

Label, 2, 6, 10, 11, 17, 19, 21

Labels

Local, 3, 5, 19, 20, 31

Listing files, 4

Local label, 3, 5, 19, 20, 31

Macro, 1, 3, 4, 5, 11, 19, 29, 31

Nesting, 4

Map files, 4

Object files, 4

Opcode compiler, 2, 3, 4, 14, 15, 23

Operand, 2, 6, 10, 12, 17, 18, 19, 21

Pre-parser, 17

Processor

Goldstar Z8400A, 7

Intel 8080, 4, 7, 8, 9, 14, 18, 25, 27

Intel 8085, 4, 7, 8, 14, 25, 27

NEC uPD780C, 7

SGS Z08400, 7

Sharp LH0080, 7

Zilog Z180, 4, 7, 8, 25, 27

Zilog Z80, 4, 7, 8, 18, 19, 25, 27

Source code, 6, 8, 25, 26

Source files, 4, 5

Variables, 5

1. x80 collectively refers to the 8080 microprocessor and derivatives such as the 8085, Z80, Z180 [↑](#footnote-ref-1)