XA80

Grammar Editor User Manual

Contents

[1. Introduction 5](#_Toc112763950)

[1.1. Document purpose 5](#_Toc112763951)

[1.2. The Editor 5](#_Toc112763952)

[1.3. Grammar Files 5](#_Toc112763953)

[1.4. Stored Items 5](#_Toc112763954)

[2. Element / Setting pairs 6](#_Toc112763955)

[2.1. Boolean Data Type 8](#_Toc112763956)

[2.2. CharSet Data Type 8](#_Toc112763957)

[2.3. N/O/M Data Type 9](#_Toc112763958)

[2.4. N/P/L Data Type 9](#_Toc112763959)

[2.5. String Data Type 9](#_Toc112763960)

[2.6. StringList Data Type 10](#_Toc112763961)

[2.7. StringListOp Data Type 10](#_Toc112763962)

[2.8. Text Data Type 10](#_Toc112763963)

[2.9. Word Data Type 11](#_Toc112763964)

[Index 12](#_Toc112763965)

Document Release Status

|  |  |  |
| --- | --- | --- |
| Version | Date | Changes |
| 0.1 |  | Initial document release in draft form |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Contact

For contact about the content of this document, please contact Duncan Munro [duncan@duncanamps.com](mailto:duncan@duncanamps.com)

# Introduction

## Document purpose

This document is the User Manual for the Grammar Editor that supports XA80, **X** (Cross) **A**ssembler for x**80** processors. Its purpose is to provide a reference on how the editor should be used, with examples where appropriate.

## The Editor

XA80 is a highly configurable assembler for x80 8/16 bit processor architectures. It can accommodate multiple grammars and multiple processor types.

The purpose of the editor is to create a new grammar, or to amend an existing one.

## Grammar Files

Grammar files are stored alongside the assembler binary image; the location of this will vary depending on the operating system. The purpose of the grammar file is to record a set of behaviours that can be applied to the assembler.

## Stored Items

These behaviours are stored as Element / Setting pairs, for example:

CmdDefineBytes DEFB

LabelCharactersStart [?0-9A-Za-z]

A full list of the Element / Setting pairs are described in detail in section 2 below.

# Element / Setting pairs

The full list of Elements/Settings is:

| Sub-option | Default | Data Type |
| --- | --- | --- |
| Author | Duncan Munro | Text |
| CmdCPU |  | StringList |
| CmdDefineBytes | DB DEFB | StringList |
| CmdDefineStorage | DS DEFS | StringList |
| CmdDefineString | DM DEFM | StringList |
| CmdDefineStringH | DC DEFC | StringList |
| CmdDefineStringZ | DZ DEFZ | StringList |
| CmdDefineWord16 | DW DEFW | StringList |
| CmdDefineWord32 | DD DEFD | StringList |
| CmdElse | ELSE | StringList |
| CmdEnd | END | StringList |
| CmdEndIf | ENDIF | StringList |
| CmdEndMacro | ENDM | StringList |
| CmdEndRepeat | ENDR | StringList |
| CmdEndWhile | ENDW | StringList |
| CmdEquate | EQU | StringList |
| CmdError | ERROR | StringList |
| CmdExtern | EXTERN EXTERNAL | StringList |
| CmdGlobal | GLOBAL | StringList |
| CmdIf | IF | StringList |
| CmdIfdef | IFDEF | StringList |
| CmdIfndef | IFNDEF | StringList |
| CmdIncludeFile | INCLUDE | StringList |
| CmdMacroDefine | MACRO | StringList |
| CmdMessage | MESSAGE | StringList |
| CmdOrg | ORG ORIGIN | StringList |
| CmdRepeat | RPT REPEAT | StringList |
| CmdSEGC | CSEG CODE | StringList |
| CmdSEGD | DSEG DATA | StringList |
| CmdSEGU | USEG UDATA | StringList |
| CmdSet | SET = | StringList |
| CmdTitle | TITLE | StringList |
| CmdWarning | WARNING | StringList |
| CmdWhile | WHILE | StringList |
| CommentAnywhere | ; // | StringList |
| CommentStart | \* ; // | StringList |
| DefaultOrg | 0 | Word |
| DefaultProcessor | Z80 | Text |
| EndBaggage | Optional | N/O/M |
| EndRule | Optional | N/O/M |
| EscapeCharacter | \ | String |
| EscapeNumbers | octal Xhex xhex | StringList |
| EscapeSet | [?\"'abefnrtv] | CharSet |
| ExprOrg | $ | StringList |
| FuncHigh | HIGH( [2] | StringListOp |
| FuncLow | LOW( [2] | StringListOp |
| LabelCaseSensitive | False | Boolean |
| LabelCharactersMid | [0-9A-Za-z\_] | CharSet |
| LabelCharactersStart | [A-Za-z\_] | CharSet |
| LabelColonRuleEqu | Never | N/O/M |
| LabelColonRuleNormal | Optional | N/O/M |
| LabelLocalPrefix | @ | String |
| LabelLocalSuffix |  | String |
| LabelMaximumLimit | 128 | Word |
| LabelMaximumUsed | 128 | Word |
| LabelPredefineReg | False | Boolean |
| LiteralASCIIquote | ["'] | CharSet |
| LiteralBinaryFormat | 0bnnnn 0Bnnnn nnnnb nnnnB | StringList |
| LiteralDecimalFormat | nnnn nnnnd nnnnD | StringList |
| LiteralHexFormat | 0xnnnn 0Xnnnn #nnnn nnnnh nnnnH | StringList |
| LiteralOctalFormat | nnnnO nnnno nnnnQ nnnnq 0onnnn 0Onnnn | StringList |
| MacroLabelPrefixG | %G% | String |
| MacroLabelPrefixL |  | String |
| MacroLabelRule | Always Local | N/P/L |
| MacroParamPrefix |  | String |
| MacroParamUse |  | String |
| OpBinaryAdd | + [5] | StringListOp |
| OpBinaryDivide | / DIV [3] | StringListOp |
| OpBinaryMod | MOD [3] | StringListOp |
| OpBinaryMultiply | \* [3] | StringListOp |
| OpBinaryShl | << [4] | StringListOp |
| OpBinaryShr | >> [4] | StringListOp |
| OpBinarySubtract | - [5] | StringListOp |
| OpBitwiseAnd | & [3] | StringListOp |
| OpBitwiseOr | | [5] | StringListOp |
| OpBitwiseXor | ^ [3] | StringListOp |
| OpBracketClose | ) [1] | StringListOp |
| OpBracketOpen | ( [1] | StringListOp |
| OpcodeSquareRule | Optional | N/O/M |
| OpCompEqual | == [6] | StringListOp |
| OpCompGreater | > [6] | StringListOp |
| OpCompGreaterEqual | >= [6] | StringListOp |
| OpCompLess | < [6] | StringListOp |
| OpCompLessEqual | <= [6] | StringListOp |
| OpCompUnequal | <> != [6] | StringListOp |
| OpLogicalAnd | && AND [7] | StringListOp |
| OpLogicalOr | || OR [8] | StringListOp |
| OpLogicalXor | ^^ XOR [7] | StringListOp |
| OpUnaryMinus | - [2] | StringListOp |
| OpUnaryNot | ! NOT [9] | StringListOp |
| OpUnaryPlus | + [2] | StringListOp |
| OpUnaryResult |  | StringListOp |
| ParserInterfldAllowed | [] | CharSet |
| ParserInterfldMandated | [/t ] | CharSet |
| ParserInteropAllowed | [/t ] | CharSet |
| ParserInteropMandated | [,] | CharSet |
| StringTerminator | ["'] | CharSet |
| Title | XA80 Grammar | Text |
| TokeniserTabSize | 4 | Word |

Some symbols such as “-“ may have both a unary operator and a binary operator role. In these situations, the input will initially be marked as “unary or binary” and during the parsing phase, the context will decide whether the operator ends up as unary or binary.

In the following example, the first line the - sign is unary, whereas in the second line it is binary. If the operator appears where an expression is expected then it becomes unary:

8 + - 5 (unary -)

8 – 5 (binary -)

Please refer to section 2.7 for more details on how operators and precedences are dealt with

Other ambiguities are not allowed, for example the keyword AND as a logical and bitwise operator at the same time. In the C language, this is resolved with & as the bitwise and, && as the logical and. In these situations where the keyword AND must be used, declare only the bitwise option as this will allow it to be used in a bitwise and logical context.

## Boolean Data Type

The Boolean data type can have a value of True or False.

## CharSet Data Type

The CharSet data type describes a selection of characters which can apply to the element. Each CharSet has a structure of:

“[“ atom\* “]”

Each atom can be one of:

chardef

chardef-chardef

Single character definitions or sequential ranges of characters can be represented by the above. Each chardef can be one of:

char

“\” escapechar

“\0” hexdigits

Note the use of the backslash character as an escape prefix. Please note also that this escape character is fixed and is not altered by the EscapeCharacter element – the latter applies only to the operation of the assembler itself.

A list of escape characters is:

* \t ASCII tab (character 9)
* \n ASCII newline (character 10)
* \r ASCII carriage return (character 13)
* \- ASCII - character (required as the hyphen is used for ranges)
* \\ ASCII \ character (required as the \ is the escape character)
* \] ASCII ] character (required as the ] is the character set terminator)
* \0nn Where nn is two, and always two hex digits, e.g.

Examples character sets are:

* [0-9A-Za-z] All digits and letters
* [\01B\-?^] Some symbols and an example hex input

## N/O/M Data Type

The N/O/M data type can have one of the following values:

* Never
* Optional
* Mandatory

## N/P/L Data Type

The N/P/L data type can have one of the following values:

* **N**ever Local
* Local if **P**refixed
* Always **L**ocal

## String Data Type

The String data type is a simple text field, however some encoding and decoding takes place to make the setting unambiguous.

The encoding is as follows:

|  |  |
| --- | --- |
| ASCII Character | Encoded |
| 44 , (comma) | {comma} |
| 123 { (opening brace) | {openbrace} |
| 124 | (vertical rule) | {rule} |
| 125 } (closing brace) | {closebrace} |

## StringList Data Type

The StringList data type holds a list of 0 or more String data type elements. The same encoding/decoding rules apply as with the String data type described in section 2.5.

Each string list is visually represented as one line separated by vertical rule characters. For example:

{rule} | DEFB | {comma}

## StringList Data Type (EscapeNumbers)

This is a subset of the StringList data type and allows for a selection of escape number formats to be used. The values available are:

|  |  |
| --- | --- |
| Element | Example |
| octal | \033 |
| xhex | \x1b |
| Xhex | \X1b |

### StringList Data Type (LiteralBinaryFormat)

This is a subset of the StringList data type and allows for a number of literal binary formats to be selected. The values available are:

|  |  |
| --- | --- |
| Element | Example |
| %nnnn | %11010110 |
| 0bnnnn | 0b11010110 |
| 0Bnnnn | 0B11010110 |
| 0nnnnb | 011010110b |
| 0nnnnB | 011010110B |

Note that if the % character is used elsewhere, for example as a modulo binary operator, it cannot be selected as one of the options for the binary format. To do so would create an ambiguity that could not be resolved by the assembler.

### StringList Data Type (LiteralDecimalFormat)

This is a subset of the StringList data type and allows for a number of literal decimal formats to be selected. The values available are:

|  |  |
| --- | --- |
| Element | Example |
| nnnn | 1234 |
| nnnnd | 1234d |
| nnnnD | 1234D |

### StringList Data Type (LiteralHexFormat)

This is a subset of the StringList data type and allows for a number of literal hexadecimal formats to be selected. The values available are:

|  |  |
| --- | --- |
| Element | Example |
| $nnnn | $79AB |
| 0xnnnn | 0x79ab |
| 0Xnnnn | 0X79ab |
| #nnnn | #79AB |
| nnnnh | 79ABh |
| nnnnH | 0FAD7H |

The case of the hexadecimal digits A-F is not significant so 79ABh is equivalent to 79abh. For the nnnnh/nnnnH options, the hex value must start with a digit or it will be interpreted as a label. For example, FAD7H would be interpreted as a 5 character label but 0FAD7H would be represented as the hexadecimal value FAD7.

Note that if the $ or # characters are used elsewhere, for example as the origin symbol, they cannot be selected as one of the options for the hexadecimal format. To do so would create an ambiguity that could not be resolved by the assembler.

### StringList Data Type (LiteralOctalFormat)

This is a subset of the StringList data type and allows for a number of literal octal formats to be selected. The values available are:

|  |  |
| --- | --- |
| Element | Example |
| nnnno | 317o |
| nnnnO | 317O |
| nnnnq | 317q |
| nnnnQ | 317Q |
| 0onnnn | 0o317 |
| 0Onnnn | 0O317 |

## StringListOp Data Type

A variant on the StringList data type, the StringListOp data type will have a minimum of two elements, the last of which will be a precedence number in square brackets, e.g. [4].

The precedence number will control the order of evaluation of expressions in the assembler. For example, it is generally recognised that the following expression will equate to 7 rather than 9:

1 + 2 \* 3

The reason being that the precedence of the multiply operator is generally set higher (a lower number) than the addition operator. The default grammar uses precedence [5] for addition and precedence [3] for multiplication.

Precedence values can be between [0] and [99]. Please note that evaluation will be in the following order irrespective of the precedence number assigned:

1. Bracketed expressions including function arguments
2. Functions
3. Unary operators
4. Binary operators

Binary operators need expressions to work with, for example LOW(X)+HIGH(Y) the two functions will have to resolve to values before the + operator can make any sense of them.

## Text Data Type

The Text data type is a simple textual string – there are no encoding/decoding or other transformations applied to the content. Therefore, this setting should contain something which is human readable.

## Word Data Type

The Word data type is a 16 bit unsigned value which can be in the range 0-65536.

# Index