Technical Reference Manual

DASM 2.20.14

August 24, 2020

Change Log

This section lists recent changes to the document, with most recent entries first. Each item line hyperlink to the relevant place in the document.

Changes

`2020.08.24`

- Documented dasm bug in constants signed range checking for 8-bit operands
- Constants and Numbers duplicate sections merged.
- Corrected error in table - || is logical-OR and && is logical-AND when used in expressions. They will return 0 or 1 results.
- Clarified the ORG/RORG usage in the `-f Output Format` table, in regards to the requirement that data must be in order in initialised segments
- Revamped the comment about error types
- Mentioned default number of passes
- Clarified the -d option
- Clarified commmand-line format
- Added note about special-casing of negative operands
- Added alternates to unary operator table
- Added section describing Constants

`2020.08.23`

- Added some usage notes about colon in label names and why it can be advantageous. I still hate it.
- Added this change log section.

Contents

"Do you program in Assembly?" she asked.
"NOP", he said.

1 Introduction

1.1 About

This is the Technical Documentation and User Guide for the **dasm** macro-assembler. It explains how to use **dasm** and the supported assembler directives.

1.1.1 Home

Since release 2.20.12, **dasm** has lived at

https://dasm-assembler.github.io/

On that page you can download prebuilt binaries for MacOS, Linux, and Windows operating systems. You can also download the full source code and build the program binary yourself.

1.1.2 Features

dasm is packed with features...

- fast assembly
- supports several common 8 bit processor models
- takes as many passes as needed
- automatic checksum generation, special symbol '...'
- several binary output formats available.

- allows reverse indexed origins.
- multiple segments, BSS segments (no generation), relocatable origin.
- expressions, as in C but [] is used instead of () for parenthesis. (all expressions are computed with 32 bit integers)
- no real limitation on label size, label values are 32 bits.
- complex pseudo-ops, repeat loops, macros
- etc...

1.1.3 Conventions in this Document

This document uses standardised terminology to describe usage and function.

Should the name be "`dasm`*", "*`DASM`*" or "*`Dasm`*"?*

Yes. In this document we shall refer to it as **`dasm`**.

Usage of directives and command-line options are shown in a box like this...

```
dasm source.asm -f3 -v5 -otest.bin
```

Items/examples that appear in source code are shown like this...

```
MAC END_BANK
  IF _CURRENT_BANK_TYPE = _TYPE_RAM
    CHECK_RAM_BANK_SIZE
  ELSE
    CHECK_BANK_SIZE
  ENDIF
ENDM
```

In 8-bit microprocessors, 16-bit values are represented by pairs of bytes, either in low/high or high/low ordering. The ordering, called the "endianness", differs between processors. In this document, `LSB` refers to the least-significant byte, and `MSB` refers to the most-significant byte, independent of the endianness of the processor. See Unary Operators for the unary operators `<` and `>` which are used to retrieve the `LSB` or `MSB` from a symbol/value.

`[item]`	Optional item
`[item...]`	As many optional items as needed, separated by commas
`item[,item...]`	At least one item followed by comma-separated items

1.2 Assembler Passes

Almost nothing need be resolved in pass 1. **dasm** is most likely to make several passes through the source code to resolve all symbols. The maximum number of passes (default 10) is controllable by **-p Number of Passes** and **-P Number of Passes (Fewer Checks)**. **dasm** will return an error if it can't resolve all referenced symbols within the maximum number of passes.

The the following contrived example will resolve in 12 passes:

```
   ORG 1
   REPEAT [[addr < 11] ? [addr-11]] + 11
     DC.b addr
   REPEND
addr:
```

Most everything is recursive. You cannot have a macro definition within a macro definition, but can nest macro calls, repeat loops, and include files.

The other major feature in this assembler is the **SUBROUTINE** directive , which logically separates Local Labels (starting with a dot). This allows you to reuse label names (for example, `.1`, `.fail`) rather than think up crazy combinations of the current subroutine to keep it all unique.

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

Martin Golding

2 Command-Line

2.1 Usage

dasm is a command-line tool. It parses the command-line for the input source file, which must be present, and optional and assemble-time options, assembles the source file, and produces a binary output as well as other outputs as specified in the options. The source file must be ASCII-encoded, but comments may contain Unicode characters.

The assembler will return 0 on successful compilation, 1 otherwise.

```
dasm [sourcefile[ option]...]
```

If no `sourcefile` is given, then **dasm** will output a short help message, and exit. Otherise, the `sourcefile` becomes the file that **dasm** will assemble, and further parameters are parsed as options.

2.2 Options

Options are specified on the command-line, after the source file. There may be zero or more options each separated by whitespace. Some options require parameters. Default values (where an option is not explicitly defined) are described with each option below.

An option is prefixed by a dash "`-`" or a slash "`/`" symbol, and followed by the option letter and then the parameter (if there is one). There must be no

whitespace between the prefix, option letter or the parameter.

Example

```
dasm source.asm -f2 -ooutput.bin -llisting.txt -v3 -DVER=4
```

This example will assemble the file "source.asm", using output format 2 (random access segments). The resultant binary will be saved to the file "output.bin" and a listing file written to "listing.txt". During the assembly, verbosity of the output is set to 3 (unresolved and unreferenced symbols displayed every pass). The value of the symbol (which will be available in the source code) "VER" is set to 4.

2.2.1 -d Debug

```
-d
```

This option is for developers of the **dasm** assembler, and is essentially inactive in release versions.

2.2.2 -D Define Symbol

```
-Dsymbol=exp
```

Defines a symbol and sets it to the expression `exp`.

Can be used to set values for symbols used inside the code.

See also **-F Define Symbol**, **-M Define Symbol**.

Example

```
dasm source.asm -DSPEED=40
```

```
  lda #SPEED    ; will load 40
  sta velocity
```

2.2.3 -E Error Format

```
-Eformat
```

Sets the format of the output of error information. Many programmers' editing environments (IDEs) are able to monitor the output from tools such as **dasm** and parse it for information about errors and warnings. If an IDE is able to resolve file names and line numbers for these errors, then the IDE can provide quick-links to the user to allow ease of editing.

This option switches the format of error/warning output of **dasm** between several "standard" formats.

format	Result
0	Microsoft (default)
1	Dillon
2	GNU

2.2.4 -f Output Format

```
-fvalue
```

Defines the format used in the binary output file generated by **dasm**.

value	Function
1	`default`. The output file contains a two byte origin in little-endian order, then data until the end of the file. Any instructions which generate output (within an initialised segment) must do so with an ascending `ORG` address (this address being the offset in the binary/ROM where the output is placed, as opposed to the `RORG` which is the address to which the code is assembled). Initialised segments must occur in ascending order.
2	`RAS` (Random Access Segment). The output file contains one or more hunks. Each hunk consists of a 2 byte origin (little-endian), 2 byte length (little-endian), and that number of data bytes. The hunks occur in the same order as initialized segments in the assembly. There are no restrictions to segment ordering (i.e. reverse indexed ORG statements are allowed). The next hunk begins after the previous hunk's data, until the end of the file.
3	`RAW`. The output file contains data only (format #1 without the 2 byte header). Restrictions are the same as for format #1. **No** header origin is generated. You get nothing but data.

It is a common problem to forget the format option on the command line, especially if you are expecting a pure binary ROM without a header. Use `-f3` if you are assembling ROMs.

2.2.5 `-F` Define Symbol

```
-Fsymbol
```

Define a symbol and set its value to 0.

This symbol is then usable in the source code as if it were a part of the code itself, defined via `EQM`. This can be useful for controlling the conditional assembly of parts of code via the `IFCONST` and `IFNCONST` directives.

See also `-D` Define Symbol, `-M` Define Symbol.

Example

In this example after our command line `dasm source.asm -FTEST`, or `dasm source.asm` we can control which code is assembled, with constructs as shown below.

```
IFCONST TEST
  ; code here only assembled when TEST is defined
ENDIF
```

2.2.6 -I Include Directory

```
-Idirectory
```

This adds the `directory` to the search path **dasm** uses when looking for files when it encounters **INCLUDE** and **INCBIN** directives. Use of this option on the command line is equivalent to an **INCDIR** directive placed at the beginning of the source file.

See **INCDIR** for the format of the directory name.

2.2.7 -l Listing Filename

```
-lfilename
```

dasm is able to produce a comprehensive and extremely useful listing of the assembled source code. This file includes symbol values, code locations, and the source code itself. To enable generation of a listing file, use the `-l` option.

See also **-L Listing Filename (all Passes)**.

2.2.8 -L Listing Filename (all Passes)

This option behaves the same as **-l Listing Filename**, but lists the code on every pass. Warning: this can lead to some very big listings if **dasm** needs to perform many passes on your code!

See also **-l Listing Filename**.

2.2.9 -M Define Symbol

```
-Msymbol=exp
```

Deprecated.

Defines a symbol and sets it to the expression `exp]`.

See also **-D Define Symbol**, **-F Define Symbol**.

2.2.10 -o Output File

```
-ofilename
```

Set the name of the filename for the output binary result of the assembly. If no name is specified, **dasm** will write to the file "`a.out`".

Example

```
dasm source.asm -orom.bin
```

This example will assemble the file `source.asm` and write the file `rom.bin` with the binary results of the assembly. The format of the binary file is controlled by **-f Output Format**. If you want a pure binary output file without headers, you **must** also use option `-f3`.

2.2.11 -p Number of Passes

```
-pvalue
```

Sets the maximum number of passes performed by **dasm** to `value`.

dasm will keep performing passes until all references are resolved, or until the maximum number of passes is reached (in which case it will exit with an unresolved symbol error). Typically on machines these days, **dasm** is so fast that a high number of passes is acceptable.

The default number of pases is 10.

See also -P Number of Passes (Fewer Checks).

2.2.12 -P Number of Passes (Fewer Checks)

This is the same as -p Number of Passes, but instructs dasm to perform fewer checks.

And these are...?

See also -p Number of Passes.

2.2.13 -s Symbol Table File

```
-sfilename
```

At the conclusion of assembly, the -s option directs dasm to save a symbol table to the specified file. A symbol table is a table listing all the symbols encountered during an assembly, and their values if known. By default, no symbol table file is generated.

The symbol table may be sorted alphabetically or numerically with the -T Sort Symbol Table option.

Example

After the execution of the command "dasm source.asm -ssource.sym", the file source.sym would contain the symbol table in the format as shown in the example below. Each line gives a symbol name, its final resolved address/value, and a flag field. In the flag field, (R) indicates the symbol has been used/referenced and not just defined.

```
--- Symbol List (sorted by symbol)
AUDC0                    0015
StartAddress             1000              (R )
TIA_BASE_ADDRESS         0000              (R )
TIM1T                    0294
TIM64T                   0296
TIM8T                    0295
TIMINT                   0285
var1                     0080
var2                     0081
varn                     008b
VBLANK                   0001
VERSION_VCS              0069
WSYNC                    0002
--- End of Symbol List.
```

2.2.14 -S Strict Syntax Checking

This option switches on more stringent checking of source code issues.

Duplicate macro definitions are flagged as errors.

TODO complete list of strict checks

2.2.15 -T Sort Symbol Table

```
-Tvalue
```

Controls the sorting of lines in the symbol table.

See also **-s Symbol Table File** to enable symbol table output.

value	Sort By
0	Symbol Alphabetically (default)
1	Address/Value

2.2.16 -v Verbosity Level

```
-vvalue
```

The -v option controls the amount of information output by **dasm** while it is assembling your code. This information includes warnings, errors, a segement table, a symbol table, unresolved and unreferenced symbols, and reasons for performing extra passes. Use of the -v option can assist with diagnosing anomalous behaviour.

value	Result
0	Only warnings and errors (default)
1	Segment table information generated after each pass Include file names are displayed Item statistics on why the assembler is going to make another pass R1,R2 reason code: R3 where R1 is the number of times the assembler encountered something requiring another pass to resolve. R2 is the number of references to unknown symbols which occured in the pass (but only R1 determines the need for another pass). R3 is a BITMASK of the reasons why another pass is required.
2	Mismatches between program labels and symbols are displayed on every pass (usually none occur in the first pass unless you have re-declared a symbol name). Displayed information for symbols: ???? = unknown value str = symbol is a string eqm = symbol is an eqm macro (R) = symbol has been referenced (s) = symbol created with SET or EQM directive
3	Unresolved and unreferenced symbols are displayed every pass
4	Symbol table displayed to STDOUT every pass

"If you have the words, there's always a chance that you'll find the way."

Seamus Heaney

3

Numbers, Expressions and Operators

3.1 Constants

All numbers and addresses in **dasm** are represented internally as signed 32-bit values.

Values in **dasm** can be specified in base 2 (binary), 8 (octal), 10 (decimal), 16 (hexadecimal), or as ASCII characters. It doesn't matter to **dasm** which format you use, so use what makes the most sense in your code. The interpretation of the value is determined by the prefix and digits used, as shown in the following table.

Type	Format	Content
Binary	**%**n	0-1
Octal	**0**n	0-7
Decimal	n	0-9, first digit non-0
Hexadecimal	**$**n or **0x**n	case insensitive 0-9,A-F
Character	**'**c	ASCII character
String	**"**cc..**"**	zero-terminated ASCII character string **not** zero-terminated when used in DC/DS/DV.
	[exp**]d**	The constant expressions is evaluated and its decimal result converted into an ASCII string. Useful in conjunction with ECHO diagnostic output.

Even though decimal numbers can't start with `0`, the octal `0` is equivalent. In other words, `0` is `0`.

Negative signs are placed before the number prefix (e.g., `-0x123`).

Examples

```
 lda #%101           ; binary = 5 decimal
 lda #%10101010      ; binary = 170 decimal
 lda #%100000001     ; ERROR - only 8 bits in binary bytes
 lda #015            ; octal = 13 decimal
 lda #$FF            ; hexadecimal = 255 decimal (unsigned)
                     ;             = -1 decimal (signed)
 lda #'A             ; ASCII character = 65 decimal
```

```
; A great approximation for Pi is 355/113
PIDIGITS = 1000000 * 355/113
  ECHO "PI DIGITS: ", PIDIGITS    ; obscure
  ECHO "PI DIGITS: ",[PIDIGITS]d  ; aha!
```

```
PI DIGITS:  $2fefd8
PI DIGITS:  3141592
```

3.2 Expressions

Expressions are calculations involving symbols and numbers. These calculations are performed by **dasm** during the assembly process. Often, symbols will have unkonwn values during an assembly pass, and thus an expression cannot be evaluated. **dasm** will, in these cases, perform another assembly pass - often, unknown values are resolved later in the assembly. A successful assembly is one where no errors have been detected, and all referenced symbols have been resolved.

Square brackets `[]` may be used to group expressions. The precedence of operators is the same as for the C-language in almost all respects. Use brackets `[ ]` when you are unsure. The reason round brackets `()` cannot be used to group expressions is due to a conflict with 6502 and other assembly

languages which use them to specifiy indirect memory access (for example, "`lda (zp),y`").

It is possible to use round brackets `()` instead of square brackets `[]` in expressions following directives, but not following mnemonics.

So this works:

```
    IF target & (pet3001 | pet4001) ; OK
```

but this doesn't:

```
    lda #target & (pet3001 | pet4001) ; fails
```

3.3 Operators

Some operators, such as `||` (logical-OR), can return a resolved value even if one of the expressions is not resolved.

3.3.1 Operator Precedence

Operators in any expression are evaluated in order of precedence. The following tables list the various operators, their function, and precedence (PR). Operators are handled in precedence order high to low.

Unary Operators

Operator	Alternate	Function	PR
~exp	exp^-1	one's complement	20
-exp	[exp^-1]+1	mathematical negation	20
!exp	exp==0	logical NOT (0 if exp is non-zero, 1 if exp is zero)	20
<exp	exp&$FF	take LSB of the low word	20
>exp	[exp>>8]&$FF	take MSB of the low word	20

Table 3.1: Unary Operators

Special Case

Some operations will result in non-byte values when a byte value was wanted. For example: ~1 is not $FF, but instead $FFFFFFFF. Preceding it with a < (take LSB of) will solve the problem.

There is a special-case for negative numbers used in operands. Although internally 32-bit, numbers in the range -1 to -128 are treated as two's complement 8-bit numbers in this situation. Another way of thinking of this - it is not necessary to take the LSB of the number if it is in the range -128 to 255, as **dasm** will recognise this as a valid signed/unsigned 8-bit number and do this automatically.

```
lda #-1       ; OK
lda #$FF      ; same as -1
lda #-129     ; ERROR - outside 8-bit size
```

Bug: Currently **dasm** allows values in the range -$FF to +$FF. This is incorrect. The correct range is -$80 to +$FF

Binary Operators

Operator	Function	PR
*	Multiplication	19
/	Division	19
%	Modulus	19
+	Addition	18
-	Subtraction	18
>>	Aritmetic shift right	17
<<	Arithmetic shift left	17
>	Greater than	16
>=	Greater than or equal to	16
<	Less than	16
<=	Less than or equal to	16
==	Logical equal to.	15
=	Logical equal to. Deprecated! (use '==')	15
!=	Not equal to	15
&	Arithmetic AND	14
^	Arithmetic exclusive-OR	13
|	Arithmetic OR	12
&&	Logical AND. Evaluates as 0 or 1	11
||	Logical OR. Evaluates as 0 or 1	10
?	If the left expression is TRUE, result is the right expression, else result is 0. [10?20] returns 20. The function of the C conditional operator a?b:c can be achieved by using [a?b-c]+c.	9
[ ]	Group expressions	8
,	Separates expressions in list (also used in addressing mode resolution, so be careful!	7

Table 3.2: Binary Operators

3.4 Symbols

Symbol	Meaning
`...`	Checksum so far (of actually-generated data)
`..`	Evaluated value in `DV` directive
`.`	Current program counter
`*`	Synonym for '`.`' when not confused as an operator.
`.name`	Represents a local label name. Local labels may be reused inside `MACRO`s and between `SUBROUTINE` directives, but may not be referenced across `MACRO` or `SUBROUTINE` scope. (as of the beginning of the instruction)
`name`	Represents a global symbol name. Beginning with an alpha character and containing letters, numbers, or underscores.
`nnn$`	Local label, much like '`.name`', except that defining a non-local label has the effect that `SUBROUTINE` has on'`.name`'. They are unique within `MACRO`s, like '`.name`'. Note that '`0$`' and `00$` are distinct, as are `8$` and `010$` (mainly for compatibility with other assemblers).

Table 3.3: Symbols

3.5 "Why" Codes

`dasm` can display the reason (via `-v Verbosity Level`) it needs to do another pass. Internally, these reasons are stored in the "why" word.

The list of available reasons include:

Bit	Usage
0	expression in mnemonic not resolved
1	-
2	expression in a DC not resolved
3	expression in a DV not resolved (probably in DV's EQM symbol)
4	expression in a DV not resolved (could be in DV's EQM symbol)
5	expression in a DS not resolved
6	expression in an ALIGN not resolved
7	ALIGN: Relocatable origin not known (if in RORG at the time)
8	ALIGN: Normal origin not known (if in ORG at the time)
9	EQU: expression not resolved
10	EQU: value mismatch from previous pass (phase error)
11	IF: expression not resolved
12	REPEAT: expression not resolved
13	a program label has been defined after it has been referenced (forward reference) and thus we need another pass
14	a program label's value is different from that of the previous pass (phase error)

Table 3.4: WHY Codes

There are three types of error; those that cause the assembly to abort immediately, those that complete the current pass and then abort assembly, and those that allow another assembly pass in the hope that the error will self-correct.

"No symbols where none intended."

Samuel Beckett

4

Symbols and Labels

4.1 Labels

The terms symbols and labels are synonymous. However, common usage is to use "label" for a symbol referring to a memory address, and that convention is generally used in this document. Otherwise, it is referred to as a symbol.

Labels are and symbols assigned addresses or values by **dasm**. These values are calculated during the assembly process by resolving the location or value of expressions defining the label. Often this may take multiple assembly passes to resolve.

Label definitions start at the beginning of a line and are encoded in ASCII; they must start with a letter or `@` or `_`, and can include letters, numbers, and some symbols.

Colon Usage

Label definitions can end with a colon, but the usage of the label must not include the colon. This can be helpful when you are editing your code if you want to search for your label definition `label:` which will return just one result (unless it's a local label, which may be duplicated), or `label` which will return all instances.

Examples

```
loop:       jmp loop    ; OK
            jmp loop:   ; error: Illegal character ':'
```

```
Label1
Label2:
```

4.2 Local Labels

Local labels begin with a dot ".". They are local to the scope of the current SUBROUTINE directive boundary, and may be re-used in other subroutine scopes. Note that the usage of the term subroutine can be misleading; local labels are local to blocks defined by usage of the directive SUBROUTINE, not to code-subroutines.

Usually local labels are used in macros and within actual code subroutines. This is handy where simple names such as '.loop' can be re-used many times. It is particularly useful in macros, where global labels are problematic due to the inability to declare a global label more than once.

Example

```
    ; Define macro

    MAC DO
        ; Implicit SUBROUTINE inserted here!
.mac    jmp .mac        ; OK - local macro label
    ENDM


    ; elsewhere in the code...

.local  jmp .local      ; OK - local label
global  jmp global      ; OK - global label

        DO              ; use macro

    ; implicit new scope has happened
    ; after macro instantiated

        jmp global      ; OK - global scope
        jmp .local      ; error - outside scope
        jmp .mac        ; error - outside scope
```

The example above shows the result of the use of local and global labels, and the effects of implicit `SUBROUTINE` as a result of a macro instance.

4.3 Dynamic Labels

Does this work?

When used in a symbol name, the "," operator indicates one or more arguments that follow should be evaluated, and the resulting values should be concatenated to the label, to create a dynamic symbol name.

```
symbol, arg1[, argn...]
```

String literals can be specified with quotes around the string text. Expression operators can also be used, but due to label parsing constraints, they should not contain spacing.

The concat-eval "," operator also works on the expression side of EQU/SET directives, so dynamic labels can be used with opcodes.

Example

```
CON,cat
```

"Success is often the result of taking a misstep in the right direction."

Al Bernstein

5 Directives

Also known as pseudo-ops, directives appear in the source code. They instruct **dasm** what to do during assembly. These are distinct from the mnemonics in the source code, which contains the human-readable instructions for the microprocessor itself. Directives include macros, segment definitions, setting the origin/location of code, etc. They are not case-sensitive.

There must be whitespace before a directive. Thus, directives must not appear in the first column of any line.

Some directives cannot have labels on the same line - for example, those where there is no possibility of evaluating a label's value because no origin/segment has yet been defined. For directives where a label is illegal, or does not make sense, this is explicitly stated.

If a label is present, then its value will be set to the current **ORG**/**RORG** either before or after a directive is processed. Most of the time, the label to the left of a directive is set to the current **ORG**/**RORG**. The following directives' labels are given their value **after** execution of the directive: **SEG**, **ORG**, **RORG**, **REND**, **ALIGN**.

All directives (and incidentally also the mnemonics) can be prefixed with a dot "`.`" or a crosshatch "`#`" for compatibility with other assemblers. So, "`.IF`" is the same as "`IF`" and "`#IF`". In the case of the dot, this works only because unattached, lone `.FORCE` extensions are meaningless.

5.1 Includes

5.1.1 INCBIN

```
INCBIN "file.name"
```

Include the binary contents of another file literally in the output.

5.1.2 INCDIR

```
INCDIR "directory"
```

Add the given directory name to the list of places where INCLUDE and INCBIN search their files. Multiple directories can be added through multiple INCDIR commands. When the other includes directives look for files, first the names are tried relative to the current directory, if that fails and the name is not an absolute pathname, the directory list is tried. You can optionally end the directory name with a "/".

AmigaDOS filename conventions imply that two slashes at the end of a directory indicates the parent directory, and so this does an INCLUDE "/directory"

The command-line option -Idirectory is equivalent to an INCDIR "directory" directive placed at the beginning of the source file.

The directory list is not cleared between passes, but each exact directory name is added to the list only once.

5.1.3 INCLUDE

```
INCLUDE "file name"
```

Effectively inserts the contents of another file at the point of the INCLUDE and continues assembling the original as if it were one merged file.

Example

```
; Typical first few lines in an Atari 2600 program...
   processor 6502
   include "vcs.h"
   include "macro.h"
```

5.2 Assigments

5.2.1 EQU, =

```
symbol EQU exp
symbol = exp
```

The expression is evaluated and the result assigned to `symbol`.

`EQU`, `=` are equivalent.

You can use the common idiom of "`.=.+3`" - in other words, you can assign to "." or "*" directly, instead of using an `ORG` or `RORG` directive.

More formally, a directive of the form "`. EQU exp`" is interpreted as if it were written "`ORG exp`" or "`RORG exp`". The `RORG` is used if a relocatable origin is already in effect, otherwise `ORG` is used. Note that the first example is **not** equivalent to "`DS 3`" when the `RORG` is in effect.

A symbol can also be defined through the command-line options `-D Define Symbol`, `-F Define Symbol` and `-M Define Symbol`.

5.2.2 EQM

```
symbol EQM exp
```

The string representing the expression is assigned to the symbol. Occurrences of the label in later expressions causes the string to be evaluated for each occurrence. Also used in conjuction with the `DV` psuedo-op.

5.2.3 SET

```
symbol SET exp
```

Same as EQU, =, but the symbol may be reassigned later.

5.2.4 SETSTR

```
symbol SETSTR exp
```

The expression is converted to a string, and assigned to the symbol. Typical use-case is within a macro, to allow the macro to echo or otherwise use the name of an argument.

Example

```
; Use SETSTR to output a parameter as a string
    MAC CALL
    ; {1} = function name
.FNAME  SETSTR {1}
        ECHO "This is the function name: ", .FNAME
    ENDM


; test it...

    CALL HelloWorld
```

```
This is the function name: HelloWorld
```

5.3 Data

5.3.1 DC

```
DC[.B|.W|.L] exp[,exp...]
```

Declare data in the current segment. No output is generated if within a uninitialised .U segment. The byte ordering (the endian order) for the selected processor is used for each entry.

The default size extension (.B, .W, .L) is .B (byte).

Alternates

```
BYTE exp[,exp...]
WORD exp[,exp...]
LONG exp[,exp...]
```

Examples

```
; various ways of defining data...
  DC 0,1,2,3
  BYTE -1,1,2,3, <Value
  .WORD 100,1000,10000, VectorTable
  LONG 100000, 50*50*50
  dc 'a'  ; ERROR - should be 'a
```

```
; generate the bytes 0 to 9 inclusive
VAL SET 0
    REPEAT 10
        .byte VAL
VAL SET VAL + 1
    REPEND
```

5.3.2 DS

```
DS[.B|.W|.L] exp[,fill]
```

Declare space and fill with value (if specificed, otherwise default is 0). The optional size extender (.B, .W, .L) defines the data size (1, 2 or 4) bytes. Data is not generated if within an uninitialized segment, but the origin still changes accordingly (this is very useful for defining variables). The number of bytes generated is `exp` × data size (1,2, or 4)

The default size extension is a byte.

The fill value is not related to the fill value used by **ORG**.

Examples

```
ds 2      ; 2 bytes of default value 0
ds 2,10   ; 2 bytes of value 10
ds 10,2   ; 10 bytes of value 2
ds.w 2    ; 4 bytes (2 words) of default value 0
ds.l 0    ; define no space at all
```

```
; Declare some zero page variables
; in an uninitialised segment
    SEG.U variables
    ORG $80
var1          ds 2      ; 2 bytes             @ $80-$81
var2          ds.w 10   ; 20 bytes (10 words) @ $82-$8B
varn          ds.w 2    ; 4 bytes (2 longs)   @ $8C-$8F
```

5.3.3 DV

```
DV[.B|.W|.L] eqmlabel exp[,exp...]
```

This is equivalent to **DC**, but each `exp` in the list is passed through the symbolic expression specified by the `eqmlabel`. The expression is held in a special symbol dotdot '`..`' on each call to the `eqmlabel`.

See also: EQM.

5.3.4 HEX

```
HEX hh[hh...]
```

This sets down raw hexadecimal data. Whitespace is optional between each hh byte. No expressions are allowed. Note that you do NOT place a "$" in front of the hexadecimal digits. This is a short form for creating tables compactly. Data is always layed down on a byte-by-byte basis.

Example

```
HEX 1A45 45 13254F 3E12
```

produces the following sequence of decimal values in the binary...

```
26 69 69 19 37 79 62 18
```

5.4 Conditionals

Conditionals allow selected selections of code to be assembled.

5.4.1 IFCONST

A useful method is to use IFCONST or IFNCONST to check for the definition of a symbol and then conditionally assemble code based on the result. This can be especially useful with symbols defined via the command-line.

Examples

```
dasm source.asm -DPI=3
```

```
  IFCONST PI
    IF PI=3
      ECHO "Are you sure?"
    ENDIF
  ENDIF
```

Is TRUE if the expression result is defined, FALSE otherwise and no error is generated if the expression is undefined.

Example

```
symbol  ; defined!
    IFCONST symbol
        ECHO "Defined!"  ; we'll see this!
    ENDIF
```

5.4.2 IFNCONST

```
  IFNCONST exp
```

Example

```
IFNCONST symbol
    ECHO "Not defined!"  ; we'll see this!
ENDIF
```

5.4.3 IF

```
  IF exp
    ; block TRUE
  [ELSE
    ; block FALSE
  ]
  ENDIF
```

Evaluates exp and if TRUE (exp is defined and non-zero) will insert the following block of code.

Neither IF nor ELSE will be executed if the expression result is undefined. In that case, another assembly pass is performed and phase errors (in the next pass only) will not be reported unless the verbosity is set to 1 or more.

Example

A useful method is to use IFCONST or IFNCONST to check for the definition of a symbol and then conditionally assemble code based on the result. This can be especially useful with symbols defined via the command-line.

Examples

IF is a handy way to comment out large sections of code or text. There is a caveat to this method - the code is still parsed by dasm while looking for the ENDIF, EIF, so this can have some unexpected side-effects if further conditionals are encountered.

```
IF 0
  ; disabled block that won't assemble
ENDIF
```

Paired with ENDIF, EIF, ELSE.

5.4.4 ELSE

```
ELSE
```

Begin an ELSE block for the current conditional.

If the current conditional is IF and exp is undefined, the ELSE will not be executed.

Paired with IF, IFCONST, IFNCONST.

5.4.5 ENDIF, EIF

```
ENDIF
EIF
```

Terminate a conditional block.

ENDIF, EIF are equivalent.

Paired with IF, IFCONST, IFNCONST.

5.5 Code Generation

There are two sets of directives that provide ways to insert meta-blocks of code and/or data. These are the REPEAT/REPEND pair, and MAC, MACROs, which are described in their own chapter.

See MAC, MACRO.

5.5.1 REPEAT

```
REPEAT exp
  ;body...
REPEND
```

exp copies of the body are inserted at the current location, and assembled.

This looks like a loop, but it isn't. It's a text-insert of exp blocks of code, so beware of code bloat when using this construct. REPEAT/REPEND can be very useful for data table generation.

If exp==0, the body is ignored.

If exp<0, a warning "REPEAT parameter < 0 (ignored)" is output and the body is ignored.

Example

```
YV  SET 2
    REPEAT 2
XV  SET 2
      REPEAT 4
        .byte XV, YV, XV*YV
XV  SET XV+1
      REPEND
YV  SET YV+1
    REPEND
```

The above example generates the following code, which is then assembled:

```
 .byte 2, 2, 4
 .byte 3, 2, 6
 .byte 4, 2, 8
 .byte 5, 2, 10
 .byte 2, 3, 6
 .byte 3, 3, 9
 .byte 4, 3, 12
 .byte 5, 3, 15
```

Labels within a **REPEAT** block should be local labels, preceeded by a **SUBROUTINE** directive to keep them unique.

Example

```
; Use SUBROUTINE to delineate local label usage
VAL SET 0
  REPEAT 4
  SUBROUTINE
    cmp #VAL
    bne .reused     ; reused local label
    ; do something here
    jmp .exit
.reused
VAL SET VAL+1
  REPEND
.exit
```

The above example generates 4 blocks of code, each comparing with a specific immediate value and branching to a re-used local label which is made distinct by the use of the SUBROUTINE directive.

Paired with REPEND.

5.5.2 REPEND

```
REPEND
```

Bottom or a REPEAT/REPEND block. They must be in matched pairs.

Any label to the left of a REPEND is assigned **after** the complete text insert for the REPEAT/REPEND block has finished.

Paired with REPEAT.

5.6 Structure

5.6.1 ORG

```
ORG exp[,fill]
```

This directive sets the current origin. You can also set the global default fill character (a byte value) with this directive. No filler data are generated until the first data-generating opcode/directive is encountered after this one.

Sequences like:

```
    org   0,255
    org   100,0
    org   200
    dc    23
```

... will result in 200 zeroes and a 23. This allows you to specify some ORG, then change your mind and specify some other (lower address) ORG without causing an error (assuming nothing is generated in-between).

Normally, DS and ALIGN are used to generate specific filler values.

Any label on the ORG line will be allocated its value after the directive is processed.

5.6.2 RORG

```
RORG exp
```

This activates the relocatable origin. All generated addresses, including '.', although physically placed at the true origin, will use values from the relocatable origin. While in effect both the physical origin and relocatable origin are updated.

The relocatable origin can skip around (no limitations). The relocatable origin is a function of the segment. That is, you can still SEG to another segment that does not have a relocatable origin activated, do other (independant) stuff there, and then switch back to the current segment and continue where you left off.

Any label on the RORG line will be allocated its value after the directive is processed.

5.6.3 REND

```
REND
```

Deactivate the relocatable origin for the current segment. Generation uses the real origin for reference.

Any label on the REND line will be allocated its value after the directive is processed.

5.6.4 SEG

```
SEG[.U] [name]
```

This switches to a new segment, creating it if neccessary. If the optional .U extension is present, the segment is an **uninitialised** segment. Segments

may be defined in parts; the `.U` is not needed when going back to an already created uninitialized segment, though it makes the code more readable.

Unitialised segments are particularly useful for declaring variable locations without writing data to the binary output. They have no origin restrictions. This is useful for determining the size of a certain assembly sequence without generating code, and for assigning RAM addresses to labels.

An uninitialised segment with a `name` will result in the generation of a warning for a `reference to an unkonwn symbol`. This is harmless, but a good reason not to name uninitialised segments.

For segments which are not uninitialised, the segment name is used when producing the diagnostic output at the end of each pass to indicate the memory usage of the named segments. For uninitialised segments, use of a segment name will generate a "reference to undefined symbol" warning that can be ignored.

Any label on the `SEG` line will be allocated its value after the directive is processed.

The following should be considered when generating ROMs:

- The default fill character when using `ORG` (and `-F Define Symbol -f1` or `-f3`) to skip forward in segments is 0. This is a **global** default and affects all segments.
- The fill value for `DS` has nothing to do with segment space padding, so don't confuse them!

Example

```
; Declaration of zero page variables
  SEG.U variables
  ORG $80
foo1          ds 1
bar2          ds 10
varn          ds 2
```

In the example shown above, the `variables` segment is unitialised. The variables `foo1`, `bar2`, and `varn` are declared using `DS` directive to "reserve/allocate" appropriate amounts of memory. Their addresses are automatically calculated by `dasm`. The relevant part of the symbol table is shown below, to make clear that although the segment is unitialised, the labels/variables

have correct values.

```
foo1                    0080
bar2                    0081
varn                    008b
```

5.6.5 ALIGN

```
  ALIGN n[,fill]
```

Align the current program counter to an n-byte boundry. If the `fill` option is present, then that value will be used to fill the space generated. The default fill value is 0.

This should not be confused with the default fill value used by the ORG directive.

Any label on the ALIGN line will be associated its value after the directive is processed.

Example

```
; using ALIGN to move to 256-byte page boundary
  ORG $1000
  DS 10
  ; origin now $100A
  ALIGN 256
  ; origin now $1100
```

5.7 Control

5.7.1 PROCESSOR

```
  PROCESSOR type
```

dasm needs to know the target microprocessor for which it is assembling the code.

This is indicated via the PROCESSOR directive, which should be the first line (other than whitespace and comments) in your source code file. Only one PROCESSOR directive may be declared in the entire assembly.

The PROCESSOR directive appears in the source code before the declaration of code origin, and thus any label present on the same line will remain unresolved at the end of assembly, causing an error.

Thus, do not place a label on the PROCESSOR line.

Supported Microprocessors

Type	Endian	Byte Order
MOS Technology 6502	little-endian	LSB, MSB
Motorola 68HC11	big-endian	MSB, LSB
Motorola 68705	big-endian	MSB, LSB
Motorola 6803	big-endian	MSB, LSB
Hitachi HD6303	big-endian	MSB, LSB
Fairchild F8	big-endian	MSB, LSB

Example

```
PROCESSOR 6502
```

For the 6507 microprocessor (as used in the Atari 2600 machine), use "PROCESSOR 6502" as these two microprocessors are identical except for their addressing range.

Different processor models use different byte orderings (little-endian, big-endian) formats. The processor's endian format does not affect the header in the output files (-f1 and -f2), which are always little-endian (LSB, MSB). The processor byte ordering affects all address, word, and long values.

5.7.2 ECHO

```
ECHO exp[,exp...]
```

The expressions (which may also be strings), are echoed on the screen and into the list file.

5.7.3 SUBROUTINE

```
SUBROUTINE [name]
```

This isn't really a subroutine, but a boundry that resets the scope of Local Labels. Those which are defined before the SUBROUTINE directive are not visible after it.

Local labels are must be unique within the scope of the subroutine in which they are defined, and cannot be accessed outside of that scope. Local label names do not need to be unique, provided that they are not duplicated within a single scope. In other words, names can be re-used.

Macros implicitly define a new subroutine scope both at their beginning, and end. Local labels defined inside a macro are not avalable outside it, and local labels defined before a macro usage instance are also no longer visible after the instantiation. Automatic new local label scope boundries occur for each macro level.

Example

```
Fn10
            SUBROUTINE
.loop       dex             ; 1st definition of .loop
            bne .loop       ; branches to 1st .loop
.exit       rts

Fn20        SUBROUTINE

  ; new scope here because of the SUBROUTINE directive
  ; previous local labels are no longer reachable

.loop       dex             ; 2nd definition of .loop
            bne .loop       ; branches to 2nd .loop

            jmp .exit       ; ERROR - out of scope
```

The above example defines two functions (`Fn10`, `Fn20`) which both use the local label `.loop`. The correct label for each is used by the branch, by way of the `SUBROUTINE` directive setting local scope. If the second `SUBROUTINE` directive was removed, the assembler would generate an error because of the duplicate label.

Note that the function name label can be on the same line as the directive, if desired.

An implicit `SUBROUTINE` scope is in effect when Macros are instantiated, so local labels cannot be accessed spanning a macro instantiation.

5.7.4 ERR

```
  ERR
```

Abort assembly. Useful in conjunction with Conditionals to end an essembly if required.

Example

```
; Failsafe call of function in another bank
  MAC CALL ; function name
    IF SLOT_{1} == _BANK_SLOT
FNAME SETSTR {1}
      ECHO "ERROR: Incompatible slot for ", FNAME
      ERR
    ENDIF
    lda #BANK_{1}
    sta SET_BANK
    jsr {1}
ENDM
```

5.7.5 LIST

```
  LIST ON|OFF
```

Globally turns listing on or off, starting with the current line.

When you use LIST the effect is local to the current macro or included file. For a line to be listed both the global and local list switches must be on.

5.7.6 .FORCE

```
  mnemonic[.force]
```

FORCE extensions (placed after a mnemonic) are used to force an addressing mode. In some cases, you can optimize the assembly to take fewer passes by telling it the addressing mode. Force extensions are also used with DS,DC, and DV to determine the element size.

Not all extensions are available for all processor types.

Extension	Function
.0	Implied
.0x	Implied indexing (0,x)
.0y	Implied indexing (0,y)
.a	Absolute (equivalent to .e, .w)
.b	byte (equivalent to .d, .z)
.bx	byte address indexed x
.by	byte address indexed y
.d	Direct (equivalent to .b, .z)
.e	Extended (equivalent to .a, .w)
.i	Implied
.ind	Indirect word
.l	long word (4 bytes) (DS/DC/DV)
.r	Relative
.u	Uninitialized (SEG)
.w	word address (equivalent to .a, .e)
.wx	word address indexed x
.wy	word address indexed y
.z	Zero page (equivalent to .b, .d)

"Everyone is against micro managing but macro managing means you're working at the big picture but don't know the details."

Henry Mintzberg

6 Macros

Macros are user-defined Directives, and when used well they can provide extremely powerful code constructs and simplify programming.

A macro is effectively a text-substitution template. Wherever the name of a macro is used, the body of the macro is inserted. During the insertion, parameters passed to the macro may be substituted inside the body as specified by the macro definition.

Macros automatically generate an implicit `SUBROUTINE` when instantiated, which guarantees distinct local labels for that macro instance.

This can sometimes be inconvenient, as it can "hide" local labels in code using the macro, but there is currently no way known to prevent this.

6.1 Usage

6.1.1 MAC, MACRO

```
; Declaration
; parameters available as {1}, {2}, etc.
; {0} = full instantiation line
  MAC name
    ; body line 1
    ; ...
    ; body line n
  ENDM
```

```
; Instantiation
  name param1, param2, ...
```

MAC, MACRO are equivalent.

Source code lines between MAC, MACRO and ENDM are the macro's body. You cannot recursively declare a macro. You can, however, recursively use a macro (reference a macro in a macro).

No label is allowed on the macro declaration line.

The macro name is not case-sensitive, either in declaration or use.

Macros can be redefined, so beware of potential issues related to unexpected usage.

You should always use Local Labels (e.g., `.loop`) inside macros which you use more than once.

Macros are instantiated by using the macro's name (case-insensitive), followed by an optional list of arguments. The body of the macro definition can refer to arguments passed with the format "`{#}`", where `#` is replaced by the argument number. The first argument passed to a macro is therefore `{1}`. `{0}` represents an exact substitution of the entire instantiation line.

Examples

```
; Generate low/high tables pointing to functions

; Uses a macro to contain the list of functions,
; and the parameter to declare low byte or high byte

  MAC VECTORS
  ; usage: {1} is < or >
    .byte {1}Routine1
    .byte {1}Routine2
    .byte {1}Routine3
  ENDM

LoTable VECTORS <
HiTable VECTORS >
```

In the above example, a list of pointers to functions is generated in two tables (one containing the low addresses of the functions, and the other the high addresses). These two tables are always in-synch (no extra or missing entries) through the single-point definitino in the macro itself.

The two calls to the macro generate the low bytes and the high bytes into two separate tables. This will result in the following code being generated, and then inserted into the source code in place of the macro calls...

```
LoTable
        .byte <Routine1
        .byte <Routine2
        .byte <Routine3
HiTable
        .byte >Routine1
        .byte >Routine2
        .byte >Routine3
```

6.1.2 ENDM

```
  ENDM
```

End of macro definition.

No label is allowed to the left of the directive.

6.1.3 MEXIT

```
MEXIT
```

Used in conjuction with conditionals. Exits the current macro level.

See Conditionals.

7 Legal

7.1 Authorship

7.1.1 dasm

The `dasm` macro assembler is...

7.1.2 Manual

This manual is authored by Andrew Davie and...

7.2 License

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License (https://opensource.org/licenses/gpl-2.0.php) for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.