Technical Reference Manual

2.20.14

Change Log

This section lists recent changes to the document, with most recent entries first. Each item line should hyperlink to the relevant place in the document where the change has been made.

Changes

`2020.09.09`

- Correction to how the -m option actually works

- Examples for good/bad comment formats added

- Notes about SI unit usage

- Added the -m (Maximum Output File Size) option

- Added the -R (Remove Output File) option

`2020.09.08`

- Added a new Chapter - Source Code, and inside a description of the source code format, including line ending format,and the various commenting styles.

- Clarified the difference in usage of brackets on F8/6502

- Fixed some errors in the Atari 2600 section. Switched to SI standard units for describing powers of 2 sizes. I suspect people may not like that!

`2020.09.07`

- Added the really unusal (and deprecated) definition for labels... `"[ ]...^[ ]...label"`. This is a suppored/valid format, but it is likely to be removed. Do not use.

`2020.09.06`

- Added the RES directive for the F8 processor. Replaces DS on that architecture.

- Merged the interesting Channel F documentation. This has highlighted a few missing things from the main doc (e.g., RES directive)

- Corrected incorrect usage for spaces in filenames.

```
2020.09.04
```

- Documented the range checking of byte and word values.

- Much work on the illegal opcodes section of the 6502 chapter. In particular, all of the mnemonics and opcodes are now checked/cross-referenced with online sources identifying illegal, but functional opcodes. These have been checked against the **dasm** source code to find out which are actually used/available. The tables section includes opcode-to-mnemonic, and also cycle timings for all instructions. The missing opcodes that **dasm** doesn't support have been identified. The next step of all this will be to list the illegal opcodes that are supported and exactly what they do.

```
2020.09.03
```

- Added the information about the broken opcodes in HD6303, retrieved from the file BUGS in the source code. I'm starting to go throught the code itself to bring the manual up to date.

```
2020.09.02
```

- There's a new chapter where all the special-case stuff for specific processors is going. In particular, explicit support of 6502 illegal opcodes is in the process of being documented. Did you even know **dasm** explicitly supports many (but not all) of the illegal opcodes and their addressing modes? I knew about explicit support for "nop 0" and "lax" but few of the others. Note in this section how I have lifted a table from an online page, but have attributed it as I was taught when I was doing my formal research writing, so hopefully this will be OK.

- There's a new chapter with details for each of the machines that are explicitly supported with include files, macros, etc. Of course, I only know the '2600 so that's all there is at this stage. Help needed for the others.

```
2020.08.31
```

- Documented the default **dasm** help message, and in the process spotted two options (-R and -m) which were not in the documentation. But then I found that the latest version of **dasm** doesn't **have** these options; they have recently been removed!

```
2020.08.29
```

- Added extensions equivalent substitutions table

- Tried to simplify the format templates for this document's descriptions of options

- Actual testing shows the default # passes to be 3, so document has been updated to this value

- Modified the 'output' box visuals to black on grey instead of white on black

  `2020.08.24`

- Removed 0x for hexadecimal. Apparently recently removed from the code?

- Upgraded license to GPL v3 to be compatible with .STY files used for generation of this manual

- Fixed table of processor types to explicity list PROCESSOR values

- 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.

1.2 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.

1.3 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

- expressions are computed using 32 bit integers

- no real limitation on label size

- complex pseudo-ops, repeat loops, macros

1.4 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 the unary operators < and > which are used to retrieve the LSB or MSB from a symbol/value.

See Unary Operators.

1.4.1 SI Units

Computer memory sizes are referred to in this document using SI units.

In particular, `kilo` (K) is `1000`, and `kibi` (Ki) is `1024`. A `bit` is `b`, a `byte` is `B`. Thus, `4096 bytes` is `4 kibibytes`, or `4 KiB`.

Historically, disk drive manufacturers were responsible for this change in meaning of the "kilobyte", as they divided capacity by 1000 instead of 1024 when listing drive size. This not only made their drives seem bigger, it created an ambiguity when discussing computer and disk memory size. The adoption of the new SI units for computer memory size removed the ambiguity.

1.4.2 Format Description

The following format templates are used to describe items on the command line.

Format	Result
`item`	one item
`item ...`	one or more items, space-separated
`item,...`	one or more items, comma-separated
`{item|...}`	only one of the items, bar-separated
`[item]`	optional item
`[ ]...`	optional whitespace(s)

Examples

```
A[ {B|C}]       "A" or "A B" or "A C"
DC[{.B|.W|.L}]  "DC" or "DC.B" or "DC.W" or "DC.L"
DC[.{B|W|L}]    the same
```

1.5 Assembler Passes

dasm is most likely to make multiple passes through the source code to resolve all symbols. It is not necessary for anything to be resolved in the first pass. The maximum number of passes (default 3) is controllable.

dasm will return an error if it can't resolve all referenced symbols within the maximum number of passes.

See **-p Number of Passes** and **-P Number of Passes (Fewer Checks)**.

Example

The the following contrived example will resolve in 12 passes:

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

```
 > dasm test.asm -P11
test.asm (8): error: Label mismatch...
 --> addr 000b
test.asm (8): error: Too many passes (12).
```

In the above example, the example does not assemble because the number of passes (set to 11) is insufficient to resolve the value of addr.

```
 > dasm test.asm -P12
 Complete. (0)
```

In the above example, 12 passes is sufficient to assemble the code.

There is generally no harm in setting the number of passes to a sufficiently high value.

Most everything is recursive. You cannot have a macro definition within a macro definition, but you 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.

See **-p Number of Passes**, **-P Number of Passes (Fewer Checks)**, and **SUBROUTINE**.

4

2

Source Code

2.1 Encoding

Source code files are written as plain-text (generally ASCII encoding) files.

Both Windows-style (carriage-return, line-feed) and Unix-style (line-feed) line endings are supported by **dasm**.

2.2 Comments

dasm supports two different comment styles: traditional assembler single-line semicolon-delimited comments, and C-style "`/* ... */`" delimited multiline comments.

Note that comments may contain Unicode characters, as **dasm** effectively ignores the contents of comments, but assembled code cannot contain Unicode.

A comment directive anywhere on a line (`";"` or `"/*"` or `"*/"`) terminates **dasm**'s label/directive/instruction parser for the rest of that line.

2.2.1 Assembler-style Comments: ;...

The presence of a semicolon at any character in a line will flag the rest of that line (semicolon included) as a comment. It is common when writing code to place a semicolon at the beginning of lines to disable them from being assembled.

There may be valid labels, directives, and/or instructions before the comment. These will be correctly assembled.

Examples

```
; this is a comment
      ; and so is this
label ; this is a label followed by a comment
;label   this is a comment, NOT a label
  lda #2 ; <-- this instruction IS assembled
  ;lda #2   <-- this instruction is NOT assembled
```

```
  MAC TEST ;{1}=value  <-- a comment showing params
  ENDM
```

2.2.2   C-style Comments: /* ... */

C-style comments mark all text between and including `/*` and `*/` as a comment.

C-style comments make it easy to disable the assembly of blocks of code spanning multiple lines.

Correct Examples

```
/* This single line comment is valid */
 lda #1 /* And this comment is valid too */

/* And so is this
Multiline comment */
```

It can sometimes be useful to enable/disable multi-line comments with a semicolon...

```
;/*
 lda #2  ;<-- comment disabled. This IS assembled
;*/


/*
 lda #2  ;<-- comment enabled. This is NOT assembled
 */
```

Malformed Examples

```
/* the opcode to the right will be ignored */ lda #1

/* and so is the opcode
that follows this block
*/ lda #1

*/ a closing comment directive without a starting one first
   is flagged by dasm as an error

/* This comment is missing the closing end-of-comment.
   dasm will flag an error
```

2.2.3   Commenting Out Large Blocks

There are several ways to disable ("comment-out") large parts of the source
code.

- Place a semicolon at the start of every line of the code to disable
  (Assembler-style)

- Bracket the part to disable with /* and */ (C-style)

- Surround the block with the conditionals "IF 0" at the start, followed
  by "ENDIF" at the end. To re-enable the code, change the 0 to 1. Note
  that this method is may fail if the enclosed code has conditionals.

3

Command-Line

3.1 Usage

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

One option you are most likely to need is `-o Output File` to specify the binary file for output.

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

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

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

```
> dasm
boo@Andrews-MacBook-Pro dasm % bin/dasm
DASM 2.20.14-SNAPSHOT
Copyright (c) 1988-2020 by the DASM team.
License GPLv2+: GNU GPL version 2 or later (see file LICENSE).
DASM is free software: you are free to change and redistribute it.
There is ABSOLUTELY NO WARRANTY, to the extent permitted by law.

Usage: dasm sourcefile [options]

-f#      output format 1-3 (default 1)
-oname   output file name (else a.out)
-lname   list file name (else none generated)
-Lname   list file, containing all passes
-sname   symbol dump file name (else none generated)
-v#      verboseness 0-4 (default 0)
-d       debug mode (for developers)
-Dsymbol             define symbol, set to 0
-Dsymbol=expression  define symbol, set to expression
-Msymbol=expression  define symbol using EQM (same as -D)
-Idir    search directory for INCLUDE and INCBIN
-p#      maximum number of passes
-P#      maximum number of passes, with fewer checks
-T#      symbol table sorting
         (default 0 = alphabetical, 1 = address/value)
-E#      error format (default 0 = MS, 1 = Dillon, 2 = GNU)
-S       strict syntax checking
-R       remove binary output-file in case of errors
-m#      max. allowed file-size in kB

Report bugs on https://github.com/dasm-assembler/dasm please!

Fatal assembly error: Check command-line format.
```

3.2 Format

3.2.1 Spaces in Filenames

If a filename (source, output, listing file) contains spaces, the whole filename
may be surrounded with quotes, or the spaces may be escaped with a
backslash character - depending on your OS support for these.

9

```
dasm "source file.asm"
dasm source\ file.asm
```

3.2.2   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 their own 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 "`/`" prefix, 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 -oout.bin -llist.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 `out.bin` and a listing file written to `list.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`.

3.3 Options

3.3.1   -d Debug

```
-d
```

This option is for **dasm**'s developers, and is essentially inactive in release versions.

3.3.2   `-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 `-F Define Symbol`, `-M Define Symbol`.

Example

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

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

```
; Use the symbol to assemble different code
  IF SPEED < 30
    jsr FastDraw    ; not assembled
  ELSE
    jsr SlowDraw    ; is assembled
  ENDIF
```

3.3.3   `-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

3.3.4 `-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.

3.3.5 `-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. This can be useful for controlling the conditional assembly of parts of code.

See related options `-D Define Symbol` and `-M Define Symbol`.

Example

```
> dasm source.asm -FTEST
```

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

3.3.6 `-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.

3.3.7 `-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 alternate: `-L Listing Filename (all Passes)`.

3.3.8 `-L Listing Filename (all Passes)`

```
-Lfilename
```

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 alternate: `-l Listing Filename`.

3.3.9 `-M Define Symbol`

```
-Msymbol=exp
```

Deprecated.

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

See similar: `-D Define Symbol`, `-F Define Symbol`.

3.3.10 `-m Maximum Output File Size`

```
-msize
```

The `-m` switch is not intended to limit the size of the created binary. Instead, it provides a safety-halting mechanism to prevent erroneous/recursive code from creating huge binaries, in situations that may not otherwise be detected.

14

The `size` parameter is the halting condition. **dasm** will halt with an error if the size of the generated binary exceeds this value. The default maximum size is `640 KiB`.

This switch is generally useful for architectures with more than `10` memory banks. It can be used in conjunction with **-R Remove Output File** to detect assembly anomalies and clean-up afterwards.

Example

```
> dasm source.asm -orom.bin -m128
```

This example will assemble the file `source.asm` and write the file `rom.bin`, halting with an error if the output binary exceeds `128 KiB` (an indication of a potential recursion error in the code generation).

See aslo **-R Remove Output File**.

3.3.11   -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`". See **-f Output Format** for the format of the output file. If you want a pure binary output file without headers, you **must** add option `-f3`.

Example

```
> dasm source.asm -orom.bin -f3
```

This example will assemble the file `source.asm` and write the file `rom.bin` with the binary results of the assembly, without header information.

3.3.12 `-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 3.

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

3.3.13 `-P Number of Passes (Fewer Checks)`

```
-pvalue
```

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

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

And these are...?

See `-p Number of Passes`.

3.3.14 `-R Remove Output File`

```
-R
```

Removes the binary output file if **dasm** encounters an error during assembly.

Can be used in conjunction with `-m Maximum Output File Size` to remove erroneous binaries, for example if a ROM size limit is exceeded.

See aslo `-m Maximum Output File Size`.

16

3.3.15   `-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

```
> dasm source.asm -ssource.sym
```

After the execution of the above, 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.
```

3.3.16 `-S Strict Syntax Checking`

```
-S
```

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

Duplicate macro definitions are flagged as errors.

TODO complete list of strict checks

3.3.17 `-T Sort Symbol Table`

```
-Tvalue
```

Controls the sorting of lines in the symbol table.

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

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

3.3.18 `-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 segment
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."

<div align="right">Seamus Heaney</div>

4

Numbers, Expressions and Operators

4.1 Constants

4.1.1 Magnitude

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

Numbers are range-checked at point of usage. Byte values should be between `$80` (`-128`) and `$FF` (`255`) inclusive, as these values are the extremes of what can be represented with signed and unsigned 8-bit values. Word values should be between `$8000` (`-32768`) and `$FFFF` (`65535`) as these are the extremes of signed and unigned 16-bit values.

Note that the assembler cannot tell the difference between the representation (in 8 and 16 bits) of signed/unsigned representation of negative and positive numbers, as they share the same bit patterns.

```
lda #%11111111   ; is this 255 or -1?
```

The answer to the above question is that it depends on what the programmer does with the value, as in signed 8-bit arithmetic, `%11111111` is `-1`, and in unsigned 8-bit arithmetic it is `255`. This is not as confusing as it sounds, as the assembler works in signed 32-bit arithmetic and the signedness of these values (particularly when taking the low byte/word) is unambiguous.

It is common for programmers who want all bits set to simply use `-1`.

4.1.2 Base Representation

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	0n	0-7
Decimal	n	0-9, first digit non-0
Hexadecimal	$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, as that describes an octal number, the octal 0 is equivalent. In other words, $0_{10} = 0_8$.

Negative signs are placed before the number prefix (e.g., -$123).

21

Examples

```
lda #%101          ; binary = 5 decimal
lda #%10101010     ; binary = 170 decimal
lda #015           ; octal = 13 decimal
lda #$FF           ; hexadecimal = 255 decimal (unsigned)
                   ;              = -1 decimal (signed)
lda #'A            ; ASCII character = 65 decimal
lda #'A'           ; ERROR - no trailing quote allowed!
```

```
VAL = -129
lda #VAL           ; ERROR - outside byte range
```

```
; 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
```

4.2 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.

4.2.1 Brackets

The precedence of operators is the same as for the C-language in almost all respects.

Either square brackets [] or round brackets () may be used to group

expressions and to clarify/adjust precedence, depending on which **PROCESSOR** is in effect for the assembly. Differences are related to the use of round brackets in assembler instructions.

F8 and Other Processors

Either bracket type may be used in all situatinos.

6502 Processor

Use square brackets `[]` when you are unsure. The reason round brackets `()` cannot be used to group expressions is due to a conflict with 6502 assembly language 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
```

4.3 Operators

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

4.3.1 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	20
		(0 if exp is non-zero, 1 if exp is zero)	
<exp	exp&$FF	take LSB of the low word	20
>exp	[exp>>8]&$FF	take MSB of the low word	20

Table 4.1: 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.

However, 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.

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

Examples

```
; Special case handling of 8-bit negatives
  lda #-1           ; OK
  lda #$FF          ; same as -1
  lda #-129         ; ERROR - outside 8-bit size
```

```
; Extracting low and high byte of value
  lda #<addr        ; low byte of symbol address/value
  lda #>$12345678   ; = $56, the high byte of the low word
```

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 4.2: Binary Operators

25

4.4 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 MACROs 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. Symbol definitions may end with a colon, but reference must omit the colon.
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 MACROs, like '.name'. Note that '0$' and 00$ are distinct, as are 8$ and 010$ (mainly for compatibility with other assemblers).

Table 4.3: Symbols

4.5 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:

26

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)

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.

5

Symbols and Labels

5.1 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

```
; Usage of colon in label names
loop:        jmp loop     ; OK
             jmp loop:    ; error: Illegal character ':'
```

```
; Examples of label/symbol definitions
Label1               ; standard
Label2:              ; optional colon
lab3 = %101010       ; life, the universe, and everything
LAB4 SETSTR "Hello"  ; allocated as string
.lab6                ; local
lab,{1}              ; dynamic, inside macro
lab@                 ; some symbols are valid too
 labWhoops           ; invalid - not in 1st column
lab SET -INFINITY    ; SET: initial definition
lab SET 0            ; SET: and re-use!
```

5.2 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.

5.3 Dynamic Labels

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 in arguments must 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.

Examples

```
; define and use a dynamic label
CON,"cat"                ; define label
        jmp CONcat       ; use the generated label
```

```
; Use a dynamic label inside a macro

N SET 0           ; instance number

  MAC dynm        ; {1}=base name
{1},"_",N         ; define label using {1} and instance #
N SET N+1
  ENDM

  dynm fna
  jmp fna_0       ; OK

  dynm fna
  jmp fna_1       ; OK

  dynm fnb
  jmp fnb_2       ; OK
  jmp fnb_0       ; ERROR - does not exist
```

5.4 Deprecated Form

dasm currently supports labels defined as per the following...

```
[]...^[]...label
```

That might look a bit weird because, basically, it is. Essentially, whitespace carat whitespace and then the label name. This is a **deprecated** format that may not be supported by future versions of **dasm**. Do not use.

```
  ^ weirdLabel   ; this is a weird way to define a label
 normalLabel     ; this is a normal way
```

6

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. Directives are not case-sensitive, but in this document they are shown in uppercase.

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.

6.1 Includes

6.1.1 INCBIN

```
INCBIN "filename"
```

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

6.1.2 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.

6.1.3 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"
```

6.2 Assigments

6.2.1 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.

6.2.2 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.

6.2.3 SET

```
symbol SET exp
```

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

Example

```
; Using SET to do calcualtions
N SET 1
SUM SET 0
  REPEAT 10
SUM SET SUM+N
N SET N+1
  REPEND
  ECHO "Sum of 1 to 10 is", [SUM]d
```

```
Sum of 1 to 10 is 55
```

6.2.4 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


    CALL HelloWorld        ; test it...
```

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

6.3 Data

6.3.1 DC

```
    DC[{.B|.W|.L}] 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,...
    WORD exp,...
    LONG 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
```

6.3.2 DS

Not available for the F8 processor - use **RES**

```
DS[{.B|.W|.L}] exp[,fillvalue]
```

Declare space and fill with a fillvalue (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
```

6.3.3 DV

```
DV[{.B|.W|.L}] eqmlabel 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 EQM.

6.3.4 HEX

```
HEX {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
```

6.3.5 RES

Not available for 6502 - use DS

Since DS is an F8 instruction (decrement scratchpad register), the DS directive isn't available anymore if dasm assembles F8 code, and this RES directive is provided as an alternative.

```
RES[{.B|.W|.L}] exp[,fillvalue]
```

Declare space and fill with a fillvalue (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

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

6.4 Conditionals

Conditionals allow selected selections of code to be assembled.

6.4.1 `IFCONST`

```
IFCONST exp
```

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

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

```
> dasm source.asm -DPI=3
Are you sure?
```

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
```

6.4.2 IFNCONST

```
IFNCONST exp
```

Example

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

6.4.3 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.

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`.

6.4.4 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`.

6.4.5 ENDIF, EIF

```
ENDIF
EIF
```

Terminate a conditional block.

`ENDIF`, `EIF` are equivalent.

Paired with `IF`, `IFCONST`, `IFNCONST`.

6.5 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`, `MACRO`s, which are described in their own chapter.

See `MAC`, `MACRO`.

6.5.1 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**.

6.5.2 **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**.

6.6 Structure

6.6.1 **ORG**

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

44

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.

6.6.2 **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.

6.6.3 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.

6.6.4 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 zero-page variables (starting at location `$80`) `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
```

6.6.5 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`. The **ALIGN** default value should not be confused with the **ORG** directive's default fill value.

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
```

6.7 Control

6.7.1 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	Identity	Endian	Byte Order
6502	MOS Technology 6502	little-endian	LSB, MSB
68HC11	Motorola 68HC11	big-endian	MSB, LSB
68705	Motorola 68705	big-endian	MSB, LSB
6803	Motorola 6803	big-endian	MSB, LSB
HD6303	Hitachi HD6303	big-endian	MSB, LSB
F8	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 endianness (byte ordering of word values, being little-endian or big-endian). The processor's endiannessdoes 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, andlong values.

6.7.2 ECHO

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

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

To output values in decimal use the format `[exp]d`

Example

```
answer = 42
  ECHO "Hex=",answer,"Decimal=",[answer]d
```

```
Hex= $2a Decimal= 42
```

6.7.3 SUBROUTINE

```
SUBROUTINE [name]
```

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

Local labels 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.

Example

```
  MAC DO
  ; body
  ENDM

.lab
  jmp .lab      ; OK

  DO  ; instantiate macro

  jmp .lab      ; ERROR
```

See `MAC`, `MACRO`.

6.7.4 ERR

```
  ERR
```

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

Example

```
  MAC CALL ; function name
    IFNCONST {1}
FNAME SETSTR {1}
    ECHO FNAME," does not exist!"
    ERR
  ELSE
    jsr {1}
  ENDIF
  ENDM

test

  CALL test         ; OK
  CALL test2        ; "test2 does not exist!" then halts
```

```
test2 does not exist!
source.asm (37): error: ERR pseudo-op encountered
```

6.7.5 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.

6.7.6 .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)

First character equivalent substitutions:

Orig	Alt	Alt	Meaning
b	z	d	byte, zeropage, direct
w	e	a	word, extended, absolute

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

Henry Mintzberg

7

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.

7.1 Usage

7.1.1 `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
```

Example

```
; Inserts a page break if the object would overlap a page

  MAC OPTIONAL_PAGEBREAK ; { labelString , size }
    IF (>( * + {2} -1 )) > ( >* )
.EARLY_LOCATION  SET *
      ALIGN 256
        ECHO "Page break for", {1}
        ECHO "wasted", [* - .EARLY_LOCATION]d, "bytes"
    ENDIF
  ENDM
```

Paired with `ENDM`.

7.1.2 ENDM

```
  ENDM
```

End of macro definition.

No label is allowed to the left of the directive.

Paired with `MAC`, `MACRO`.

7.1.3 MEXIT

```
  MEXIT
```

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

See Conditionals.

8

6502

Special-case information, and tips and tricks for the processors supported by **dasm** are described here.

8.1 Illegal Opcodes

The effects of the "unused" opcodes on the 6502 are by now relatively well documented. These are variously described as illegal, undocumented, invalid, and unspecified. Modern programs use some of these, as they provide additional capabilities (particularly speed improvements) over the use of the standard opcodes. **dasm** explicitly supports some of the commonly used 'stable' illegal opcodes.

http://www.oxyron.de/html/opcodes02.html was used as a reference for most of the data shown in the following tables. They have been cross-referenced with the **dasm** source code to determine what instructions and opcodes are supported.

8.1.1 Abbreviations and Colours used in Tables

Addressing Modes

Abbreviation	Mode	Example
abs	Absolute	`LDA $F000`
abx	Absolute indexed by X	`LDA $F000,x`
aby	Absolute indexed by Y	`LDA $F000,y`
idx	Indexed indirect X	`LDA ($23,x)`
idy	Indirect Y	`LDA ($23),y`
imm	Immediate	`LDA #1`
imp	Implied. Operates on register or flag	`LSR`
ind	Indirect absolute	`JMP ($F000)`
rel	Relative to PC	`BCS addr`
zp	Zero-page	`LDA 1`
zpx	Zero-page indexed by X	`LDA $23,x`
zpy	Zero-page indexed by Y	`LAX 0,y`

8.1.2 Mnemonics, and Opcodes for Addressing Modes

Opcode Colour Key

Stable. Supported by `dasm`.
Stable. Not supported.
Unstable in some situations. Not supported.
Highly unstable. Not supported.

	imp	imm	zp	zpx	abs	abx	aby	idx	idy	zpy	rel	ind
ADC	✗	$69	$65	$75	$6D	$7D	$79	$61	$71	✗	✗	✗
AHX	✗	✗	✗	✗	✗	✗	✗	✗	$93	✗	✗	✗
ANC	✗	$0B	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
		$2B										
AND	✗	$29	$25	$35	$2D	$3D	$39	$21	$31	✗	✗	✗
ANE	✗	$8B	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ARR	✗	$6B	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ASL	$0A	✗	$06	$16	$0E	$1E	✗	✗	✗	✗	✗	✗
ASR	✗	$4B	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
AXS	✗	$CB	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
BCC	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	$90	✗
BCS	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	$B0	✗
BEQ	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	$F0	✗
BIT	✗	✗	$24	✗	$2C	✗	✗	✗	✗	✗	✗	✗
BMI	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	$30	✗
BNE	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	$D0	✗
BPL	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	$10	✗
BRK	$00	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
BVC	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	$50	✗
BVS	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	$70	✗
CLC	$18	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CLD	$D8	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CLI	$58	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CLV	$B8	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CMP	✗	$C9	$C5	$D5	$CD	$DD	$D9	$C1	$D1	✗	✗	✗
CPX	✗	$E0	$E4	✗	$EC	✗	✗	✗	✗	✗	✗	✗
CPY	✗	$C0	$C4	✗	$CC	✗	✗	✗	✗	✗	✗	✗
DCP	✗	✗	$C7	$D7	$CF	$DF	$DB	$C3	$D3	✗	✗	✗
DEC	✗	✗	$C6	$D6	$CE	$DE	✗	✗	✗	✗	✗	✗
DEX	$CA	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
DEY	$88	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
EOR	✗	$49	$45	$55	$4D	$5D	$59	$41	$51	✗	✗	✗
INC	✗	✗	$E6	$F6	$EE	$FE	✗	✗	✗	✗	✗	✗
INX	$E8	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
INY	$C8	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ISB	✗	✗	$E7	$F7	$EF	$FF	$FB	$E3	$F3	✗	✗	✗

	imp	imm	zp	zpx	abs	abx	aby	idx	idy	zpy	rel	ind
JMP	✗	✗	✗	✗	$4C	✗	✗	✗	✗	✗	✗	$6C
JSR	✗	✗	✗	✗	$20	✗	✗	✗	✗	✗	✗	✗
KIL	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
LAS	✗	✗	✗	✗	✗	✗	$BB	✗	✗	✗	✗	✗
LAX	✗	✗	$A7	✗	$AF	✗	$BF	$A3	$B3	$B7	✗	✗
LDA	✗	$A9	$A5	$B5	$AD	$BD	$B9	$A1	$B1	✗	✗	✗
LDX	✗	$A2	$A6	✗	$AE	✗	$BE	✗	✗	$B6	✗	✗
LDY	✗	$A0	$A4	$B4	$AC	$BC	✗	✗	✗	✗	✗	✗
LSR	$4A	✗	$46	$56	$4E	$5E	✗	✗	✗	✗	✗	✗
LXA	✗	$AB	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
NOP	$EA	$80	$04	$14	$0C	$1C	✗	✗	✗	✗	✗	✗
	$1A		$44	$34		$3C						
	$3A		$64	$54		$5C						
	$5A			$74		$7C						
	$7A			$D4		$DC						
	$82			$F4		$FC						
	$89											
	$C2											
	$DA											
	$E2											
	$FA											
ORA	✗	$09	$05	$15	$0D	$1D	$19	$01	$11	✗	✗	✗
PHA	$48	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
PHP	$08	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
PLA	$68	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
PLP	$28	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
RLA	✗	✗	$27	$37	$2F	$3F	$3B	$23	$33	✗	✗	✗
ROL	$2A	✗	$26	$36	$2E	$3E	✗	✗	✗	✗	✗	✗
ROR	$6A	✗	$66	$76	$6E	$7E	✗	✗	✗	✗	✗	✗
RRA	✗	✗	$67	$77	$6F	$7F	$7B	$63	$73	✗	✗	✗
RTI	$40	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
RTS	$60	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
SAX	✗	✗	$87	✗	$8F	✗	✗	$83	✗	$97	✗	✗
SBC	✗	$E9	$E5	$F5	$ED	$FD	$F9	$E1	$F1	✗	✗	✗
		$EB										

	imp	imm	zp	zpx	abs	abx	aby	idx	idy	zpy	rel	ind
SBX	✗	$CB	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
SEC	$38	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
SED	$F8	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
SEI	$78	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
SHA	✗	✗	✗	✗	✗	✗	$9F	✗	$93	✗	✗	✗
SHS	✗	✗	✗	✗	✗	✗	$9B	✗	✗	✗	✗	✗
SHX	✗	✗	✗	✗	✗	✗	$9E	✗	✗	✗	✗	✗
SHY	✗	✗	✗	✗	✗	$9C	✗	✗	✗	✗	✗	✗
SLO	✗	✗	$07	$17	$0F	$1F	$1B	$03	$13	✗	✗	✗
SRE	✗	✗	$47	$57	$4F	$5F	$5B	$43	$53	✗	✗	✗
STA	✗	✗	$85	$95	$8D	$9D	$99	$81	$91	✗	✗	✗
STX	✗	✗	$86	✗	$8E	✗	✗	✗	✗	$96	✗	✗
STY	✗	✗	$84	$94	$8C	✗	✗	✗	✗	✗	✗	✗
TAX	$AA	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
TAY	$A8	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
TSX	$BA	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
TXA	$8A	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
TXS	$9A	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
TYA	$98	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

8.1.3 Opcode Mnemonics

	$x0	$x1	$x2	$x3	$x4	$x5	$x6	$x7	$x8	$x9	$xA	$xB	$xC	$xD	$xE	$xF
$0x	BRK	ORA	KIL	SLO	NOP	ORA	ASL	SLO	PHP	ORA	ASL	ANC	NOP	ORA	ASL	SLO
$1x	BPL	ORA	KIL	SLO	NOP	ORA	ASL	SLO	CLC	ORA	NOP	SLO	NOP	ORA	ASL	SLO
$2x	JSR	AND	KIL	RLA	BIT	AND	ROL	RLA	PLP	AND	ROL	ANC	BIT	AND	ROL	RLA
$3x	BMI	AND	KIL	RLA	NOP	AND	ROL	RLA	SEC	AND	NOP	RLA	NOP	AND	ROL	RLA
$4x	RTI	EOR	KIL	SRE	NOP	EOR	LSR	SRE	PHA	EOR	LSR	ALR	JMP	EOR	LSR	SRE
$5x	BVC	EOR	KIL	SRE	NOP	EOR	LSR	SRE	CLI	EOR	NOP	SRE	NOP	EOR	LSR	SRE
$6x	RTS	ADC	KIL	RRA	NOP	ADC	ROR	RRA	PLA	ADC	ROR	ARR	JMP	ADC	ROR	RRA
$7x	BVS	ADC	KIL	RRA	NOP	ADC	ROR	RRA	SEI	ADC	NOP	RRA	NOP	ADC	ROR	RRA
$8x	NOP	STA	NOP	SAX	STY	STA	STX	SAX	DEY	NOP	TXA	XAA	STY	STA	STX	SAX
$9x	BCC	STA	KIL	AHX	STY	STA	STX	SAX	TYA	STA	TXS	TAS	SHY	STA	SHX	AHX
$Ax	LDY	LDA	LDX	LAX	LDY	LDA	LDX	LAX	TAY	LDA	TAX	LAX	LDY	LDA	LDX	LAX
$Bx	BCS	LDA	KIL	LAX	LDY	LDA	LDX	LAX	CLV	LDA	TSX	LAS	LDY	LDA	LDX	LAX
$Cx	CPY	CMP	NOP	DCP	CPY	CMP	DEC	DCP	INY	CMP	DEX	AXS	CPY	CMP	DEC	DCP
$Dx	BNE	CMP	KIL	DCP	NOP	CMP	DEC	DCP	CLD	CMP	NOP	DCP	NOP	CMP	DEC	DCP
$Ex	CPX	SBC	NOP	ISC	CPX	SBC	INC	ISC	INX	SBC	NOP	SBC	CPX	SBC	INC	ISC
$Fx	BEQ	SBC	KIL	ISC	NOP	SBC	INC	ISC	SED	SBC	NOP	ISC	NOP	SBC	INC	ISC

Stable. Supported by `dasm`.
Stable. Not supported.
Unstable in some situations. Not supported.
Highly unstable. Not supported.

8.1.4 Instruction Cycle Counts

	$×0	$×1	$×2	$×3	$×4	$×5	$×6	$×7	$×8	$×9	$×A	$×B	$×C	$×D	$×E	$×F
$0×	7	6	∞	8	3	3	5	5	3	2	2	2	4	4	6	6
$1×	2+	5+	∞	8	4	4	6	6	2	4+	2	7	4+	4+	7	7
$2×	6	6	∞	8	3	3	5	5	4	2	2	2	4	4	6	6
$3×	2+	5+	∞	8	4	4	6	6	2	4+	2	7	4+	4+	7	7
$4×	6	6	∞	8	3	3	5	5	3	2	2	2	3	4	6	6
$5×	2+	5+	∞	8	4	4	6	6	2	4+	2	7	4+	4+	7	7
$6×	6	6	∞	8	3	3	5	5	4	3	3	3	5	4	6	6
$7×	2+	5+	∞	8	4	4	6	6	2	4+	2	7	4+	4+	7	7
$8×	2	6	2	6	3	3	3	3	2	2	2	2	4	4	4	4
$9×	2+	6	∞	6	4	4	4	4	2	5	2	5	5	5	5	5
$A×	2	6	2	6	3	3	3	3	2	2	2	2	4	4	4	4
$B×	2+	5+	∞	5+	4	4	4	4	2	4+	2	4+	4+	4+	4+	4+
$C×	2	6	2	8	3	3	5	5	2	2	2	2	4	4	6	6
$D×	2+	5+	∞	8	4	4	6	6	2	4+	2	7	4+	4+	7	7
$E×	2	6	2	8	3	3	5	5	2	2	2	2	4	4	6	6
$F×	2+	5+	∞	8	4	4	6	6	2	4+	2	7	4+	4+	7	7

(+) add 1 cycle if branch instruction performed
(+) add 1 cycle if page boundary is crossed

∞ = Instruction never completes

9

The Machines

dasm supports specific machines and processors through the provision of companion source-code files that can assist with programming each platform. These files are located in the `machines` subdirectory. Support is provided for...

- Atari 2600
- Atari 7800
- Channel F
- 68hc11
- 68hc908

9.1 Atari 2600

The Atari 2600 is a game console from 1977 that uses a 6507 processor. This processor is similar to the 6502 processor (supported by **dasm**). The difference in the processors is the number of hardware address lines on the chips; these being 16 on the 6502, and 13 on the 6507. Thus, the 6502 can directly address 64 KiB of memory and the 6507 only 8 KiB of memory. From the point of view of **dasm**, the machines are identical, as the 6502 and 6507 share a common instruction set.

For programming the Atari 2600, use `PROCESSOR 6502` at the start of your program.

9.1.1 Support Files

The Atari 2600 is explicitly supported with two files generally included in most programs for that machine.

vcs.h

Contains the standardised register definitions for the RIOT and TIA chips, defined with uninitialised segments. The implementation allows relocation of the TIA base address to a shadow register address.

macro.h

Contains some useful macros.

9.2 HD6303

9.2.1 Broken Opcodes Bug

The AIM, OIM, EIM, and TIM opcodes are broken. These instructions are **three** bytes long according to the data sheets, yet in **dasm** they are treated as **two** byte instructions.

These instructions are supposed to work as follows:

```
        tim     #$10,$C2
        tim     #$80,$00,x
```

So there's an immediate value **and** a zero-page address for these instructions! **dasm**, however, only accepts these:

```
        tim     $10
        tim     $10,x
```

These opcodes simply don't fit into the "regular pattern" of 8-bit CPUs we deal with in **dasm**. Fixing this will require changes to **dasm** beyond just fixing the instruction table, and the parser code is not even remotely ready for this.

The "workaround" for now is to use macros instead of the actual instructions, see ../test/broken6303hack.asm for details.

9.3 Channel F

9.3.1 Processor selection

With DASM, the target CPU is selected with the PROCESSOR directive inside the source file that should be assembled. The F8 CPU is selected like this:

```
        processor f8
        processor F8    ; case insensitive
```

9.3.2 Expressions with parentheses

Some of DASM's backends, for instance the one for the 6502, don't allow parentheses in expressions that are part of a mnemonic's operand, because parentheses are used in the 6502's assembly language to denote indirect addressing. Instead, you have to use brackets.

This is not the case with the F8 backend. Both parentheses and brackets can be used everywhere, so the following lines are parsed and assembled correctly:

```
        as      (2+2)*2 ; Assembles to $c8
        as      [2+2]*2 ; Assembles to $c8
```

9.3.3 Data definition directives

Since DS is an F8 instruction (decrement scratchpad register), the DS directive isn't available anymore if **dasm** assembles F8 code. Instead, use the RES directive, which works just like the DS directive:

```
        ds.b    4,$33   ; Would assemble to $33 $33 $33 $33,
                        ; but isn't available in F8 mode
        res.b   4,$33   ; Assembles to $33 $33 $33 $33
```

Of course RES.W and RES.L do exist as well.

For source code compatibility with **f8tool** (another F8 assembler), some additional data definition directives are available : DB, DW and DD. These work just like DC.B, DC.W and DC.L:

```
        dc.b    $f8     ; Assembles to $f8
        db      $f8     ; Assembles to $f8
```

9.3.4 Special register names

For some of the special registers, multiple names are accepted:

Register	Accepted Names
DC0	DC,DC0
PC0	P0,PC0
PC1	P, PC1
J	J, Any expression that evaluates to 9
	(This may seem strange, but J is really
	just an alias for scratchpad register 9)

The names `DC`, `P0`, `P` and `J` are standard syntax, the other forms have been introduced for compatibility with other assemblers.

Thus, the following lines assemble all correctly:

```
        lr      h,dc    ; Assembles to $11
        lr      h,dc0   ; Assembles to $11
        lr      p0,q    ; Assembles to $0d
        lr      pc0,q   ; Assembles to $0d
        lr      p,q     ; Assembles to $09
        lr      pc1,q   ; Assembles to $09
        lr      w,j     ; Assembles to $1d
        lr      w,3*3   ; Assembles to $1d
```

9.3.5 Scratchpad register access

There are several ways to access scratchpad registers:

Access Mode	Accepted Syntax
Direct access to registers `0..11`	Any expression that evaluates to `0..11`
Access via `ISAR`	`S`, `(IS)`, any expression that evaluates to 12
Access via `ISAR`, `ISAR` incremented	`I`, `(IS)+`, any expression that evaluates to 13
Access via `ISAR`, `ISAR` decremented	`D`, `(IS)-`, any expression that evaluates to 14

The `(IS)`, `(IS)+` and `(IS)-` forms are not standard syntax and have been mainly introduced for compatibility with `f8tool`.

For some of the directly accessible scratchpad registers aliases exist:

Register	Alias Name
9	J
10	HU
11	HL

Originally, `J` was only used with the `LR` instruction when accessing the flags, but since `J` is just an alias for register `9`, `J` can also be used in normal scratchpad register operations.

The following lines assemble all correctly

```
        xs      2+2             ; Assembles to $e4
        xs      s               ; Assembles to $ec
        xs      (is)            ; Assembles to $ec
        xs      12              ; Assembles to $ec
        xs      i               ; Assembles to $ed
        xs      (is)+           ; Assembles to $ed
        xs      13              ; Assembles to $ed
        xs      d               ; Assembles to $ee
        xs      (is)-           ; Assembles to $ee
        xs      14              ; Assembles to $ee
        xs      9               ; Assembles to $e9
        xs      j               ; Assembles to $e9
        xs      hu              ; Assembles to $ea
        xs      hl              ; Assembles to $eb
```

9.3.6 No instruction optimizations are done

The assembler doesn't optimize instructions where a smaller instruction could be used. It won't optimize between `OUT/OUTS`, `IN/INS` and `LI/LIS`.

For instance, the following line assembles to `$20 $00`, even though the `LIS` instruction could be used, which would need only one byte.

```
        li      0           ; Assembles to $20 $00
```

Index