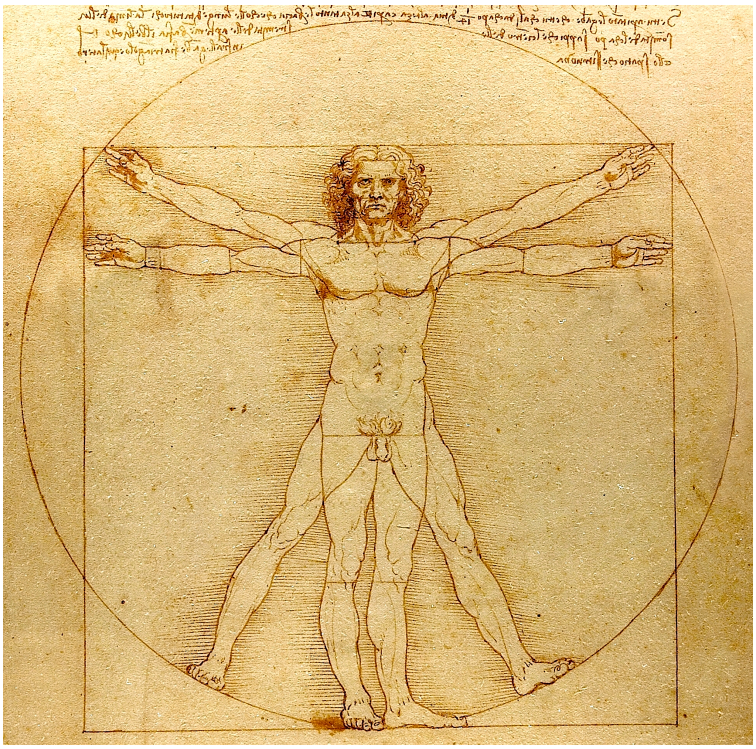
Da Vinci Assembler



Framework

Paolo Roberti

Contents

Chapter 1

Overview

The Da Vinci Assembler or DVASM™ is a set of Java® modules that implements a multi-architecture assembler framework.

DVASM™ has been written specifically to be used by programmers to be able to efficiently write large systems, such as a whole OS kernel, and easily maintain them. Great attention has been given to directives, support of a powerful macro environment, and a very detailed and informative output listing.

Any computer architecture can be supported by the DVASM™ by writing a compatible module that complements the framework module to support the target architecture.

This is a reference manual that describe the DVASM™ part that is architecture independent. For each architecture supported, now or in the future, a separate manual will be provided.

Chapter 2

Introduction

2.1 DVASM™ functionality

DVASM™ framework has been designed to support any number of computer architectures. Any computer architecture that satisfies the following conditions can be supported by the DVASM™ framework by coding an architecture Java® module:

- Storage addressing is by octets or eight bit bytes;
- Machine instructions are on an octet boundary and their length a multiple of octets;

Both Von Neuman and Harvard architectures are supported, as well as architectures with or without registers such as stack machines.

2.2 File structure

DVASM™ is delivered as a single Java® JAR file. This file contains the framework module, and any module of currently supported computer architectures, with their associated macros. If the user needs any architecture or architectural extension not currently supported by the DVASM™ JAR file, a module can be written in Java® and used as an external module. The file structure of any external module, and its associated macros, must follow precise rules in the layout of the file structure, so that it can correctly interact with the framework module.

The file structure inside any DVASM™ JAR file is always the same and is as follows:

`/framework` Directory containing all `Java®` classes for the framework module (main DVASM™ JAR file only).

`/framework/macros` Directory containing macros that are architecture independent (main DVASM™ JAR file only).

`/arch/`*`ARCH_NAME`* Directory containing all `Java®` classes for the *ARCH_NAME* architecture module. Any architecture name **must** be upper case.

`/arch/`*`ARCH_NAME`*`/macros`
Directory containing all macros for the *ARCH_NAME* architecture module.

`/arch/`*`ARCH_NAME`*`/ext/`*`EXT_NAME`*
Directory containing all `Java®` classes for the the extension *EXT_NAME* module for architecture *ARCH_NAME*. Any extension name **must** be upper case.

`/arch/`*`ARCH_NAME`*`/ext/`*`EXT_NAME/macros`*
Directory containing all macros for the the extension *EXT_NAME* module for architecture *ARCH_NAME*.

If `Java®` classes and macros that support a new architecture or extension are organized in a separate JAR file, the JAR file structure is the same as the framework DVASM™ file structure without the framework directory and its sub-directories.

If Java® classes and macros that support a new architecture or extension are organized as a set of directories instead of a JAR file, the directories must be organized exactly the same way, under a single main directory in the OS file system as follows:

`.../main_dir/arch/`*`ARCH_NAME`*
: Directory containing all Java® classes for the *ARCH_NAME* architecture module. Any architecture name **must** be upper case.

`.../main_dir/arch/`*`ARCH_NAME`*`/macros`
: Directory containing all macros for the *ARCH_NAME* architecture module.

`.../main_dir/arch/`*`ARCH_NAME`*`/ext/`*`EXT_NAME`*
: Directory containing all Java® classes for the the extension *EXT_NAME* module for architecture *ARCH_NAME*. Any extension name **must** be upper case.

`.../main_dir/arch/`*`ARCH_NAME`*`/ext/`*`EXT_NAME/macros`*
: Directory containing all macros for the the extension *EXT_NAME* module for architecture *ARCH_NAME*.

2.3 Installation

DVASM™ has been entirely coded in Java® and requires the installation of GraalVM® community edition. It can be used on any OS that is supported by GraalVM® . You can download GraalVM® from https://github.com/graalvm/graalvm-ce-builds/releases/tag/vm-21.0.0 Make sure that you select Java® 11, or higher, based version. After downloading, unzip/untar the downloaded file and follow the installation instructions.

After GrallVM has been installed, change your PATH so that Java® installed with GraalVM® becomes the default Java® used in the system.

Next download DVASM™ JAR file from https://www.dvasm.com/downloads and copy the JAR file to any suitable location on your system.

2.4 Executing DVASM™

DVASM™ is invoked with the following command:

```
java -jar   jar_file_name                                          \
            [ -i input_dir_name ]                                  \
            [ -o output_dir_name ]                                 \
            [ -l listing_dir_name ]                                \
            [ -m macro_dir_name ... -m macro_dir_name ]            \
            [ -a arch_dir_or_jar_file_name ]                       \
            [ -x arch_ext_dir_or_jar_file_name ]                   \
            input_file_name .... input_file_name
```

Command options must follow `jar_file_name` and precede `input_file_name(s)` and they can be in any order.

`jar_file_name` — Name of the main DVASM™ JAR file that contains the framework module.

`input_dir_name` — Name of the directory containing the input file(s). It is optional and when not specified it defaults to the current directory. It applies only when an `input_file_name` is not a full path name.

`output_dir_name` — Name of the directory where DVASM™ writes the output binary file(s) in ELF format. It is optional and it defaults to the `input_dir_name`.

`listing_dir_name` — Name of the directory where DVASM™ writes the assembly listing file(s) in HTML format. It is optional and it defaults to the `output_dir_name`

`macro_dir_name` — Name of one or more directories where DVASM™ searches for macros. They are optional and when not specified search is defaulted to the `input_dir_name`. DVASM™ always searches for a macro in the `macro_dir_names` directories, in the order they are specified, before searching in extension, architecture, and framework macro directories in this specific order.

`arch_dir_or_jar_file_name` — Name of the main directory or JAR file containing the architecture module. It is optional and it defaults to `jar_file_name`.

arch_ext_dir_or_jar_file_name
: Name of the main directory or JAR file containing the architecture extension module. It is optional and it defaults to `arch_dir_or_jar_file_name`.

One or more input files are specified, all with file extension `.asm`. Output ELF files are named after the corresponding input file names with file extension `.o`. Listing HTML files are named after the corresponding input file names with file extension `.html`.

Chapter 3

The Input File

An assembler programmer writes his/her code into an input file which is processed by DVASM™ to produce an object code in ELF format and an output listing in HTML format. In this chapter we will describe the syntax of DVASM™ input file.

3.1 Input File Syntax

An assembler program is composed of statements. Each statement is composed by:

Label	A label represent a symbol that is used to 'label' the statements so that it can be referenced by other statements or itself. It is optional.
Opcode	The opcode represents either a directive to the assembler, a machine instruction or a macro. The opcode is always required.
Parameters	Zero or more parameters for the opcode.
Comments	Comments are optional and can be specified at the end of each line.

Here is an example of a single statement:

```
Lbl001      MV      R4,R7      // Copy content of register 7 to register 4
```

where `Lbl001` is the label, `MV` is the opcode, `R4,R7` are two opcode parameters and `// Copy content of register 7 to register 4` is a comment.

3.2 Label

Labels must start on the first character of a line. If the first character is a space DVASM™ assumes that there is no label. A label can be coded as one or more words separated by dots (i.e. `'.'`). Each word must start with a alphabetic character or the underscore character (i.e. `'_'`) followed by zero or more alphanumeric characters or the underscore characters. The regular expression for labels is:

```
[a-z,A-Z,_][a-z,A-Z,_,0-9]*([\.][a-z,A-Z,_][a-z,A-Z,_,0-9]*)*
```

Labels are **case insensitive**.

3.3 Opcode

Opcodes share the same lexical rules used for labels, and in fact share the same regular expression. Opcodes are always **case insensitive**.

3.4 Parameters

Parameters syntax rules are different for macro opcodes and non-macro opcodes. Here we give the general rules on how to code **non-macro** parameters.

There are two type of parameters: positional and key-word parameters. All parameters are comma separated.

Positional parameters are composed of a single expression followed by zero or more sub-parameters. Each sub-parameter is also composed by a single expression. The list of positional sub-parameters is enclosed by square brackets and is comma separated. For example a positional parameter with two sub-parameters is coded as follows:

```
expression [ expression, expression ]
```

A positional parameter without sub-parameters is coded as follows:

```
expression
```

Opcode ADDI with two positional parameters, the first with no sub-parameters and the second with one subparameter is coded as follows:

```
Lblb01      ADDI        R1, 256-64 [ R2 ]
```

Expressions can be mathematical or string expressions, whose result can be an integer, with different attributes, a floating point number or a string. Expressions are described at length on page 19.

Key-word parameters are composed by a key-word followed by the equal character '=' followed by an expression. There are no sub-parameters for key-word parameters. Key word parameters can be positioned anywhere in the parameter list and do not affect the position number of positional parameters. For example key-word parameter Cond= is coded as follows:

```
Cond= expression
```

Opcode LD, with two positional parameters, the second with one sub-parameter, and one keyword parameter can be coded in the following different ways with the same end result:

```
Aaa001      LD      R2, 128-32[R8],     Cond= yes
//
Aaa002      LD      Cond= yes,          R2, 128-32[R8]
//
Aaa003      LD      R2,     Cond= yes,  128-32[R8]
```

Key-word parameters are intended to be used when a parameter is expected to be set, most of the time, to a default value, and only occasionally to a non-default value. In these cases, the use of key-word parameters will save typing to the coder and provide better readability of the source code.

3.5 Comments

There are two types of comments: single line or last line comments, and continuation comments. Single line comments are started with the '//' sequence as in C and C++. In multiple lines statements this type of comment can only be used in the last line of the statement. In all other lines comments are started with '/>' sequence and they also

indicate that the statement continues in the next line. As such, continuation comments are required when multiple line statements are used, even when an actual comment is not needed. Here is how a statement can be written on one line or multiple lines:

```
Lbl001      MV      R4,R7       // Copy content of register 7 to register 4
//
Lbl002      MV                  /> Label and opcode on this line only
                    R4,R7       // Copy content of register 7 to register 4
//
Lbl003                          /> Label only on this line
            MV                  /> opcode only on this line
                    R4,R7       // Copy content of register 7 to register 4
```

It is important to remember that the first character of any line that is not the first line of a statement **must** be a blank space.

In addition to statements DVASM™ input file can contain blank lines and lines containing only comments.

Chapter 4

Sections and Symbols

When DVASM™ reads the input file, it generates binary data consisting of machine instructions encoded in binary form or data constants when the opcode is a data opcode. The binary data is organized in **sections**. These sections are added to an ELF object file that DVASM™ generates as output.

DVASM™ can generate binary data for machine instructions and data constants only after a section **has been started**. Any attempt to do so before starting a section will result in an error.

As the assembler reads opcodes that generate binary data (i.e. machine instructions or constants), it places the binary data generated in sequential order inside the active section. The total length, in bytes, of data already placed in the section is the location or **offset** of the binary data generated by the next statement. This is equivalent to say that the offset of the next opcode is equal to the offset of the current statement plus its length.

When a label is specified for a machine instruction or constant, the label name defines a **symbol**. Symbols have three attributes:

Value	For machine instruction and constant statements it is the statement offset. When defined with EQU directive, it can be integer of several types, a floating point number or a string (see expressions on page 19).
Length	For machine instructions and constants it is the length of the binary data generated. When generating an array of constants it is the length of a single data item in the array. When defined with the EQU directive, its value is zero.
Replication factor	Replication factor is always 1 except when defining an array of constants, in which case it is the size of the array.

Here is an example of a section that contains both machine instructions ,constants, and several symbols:

```
//  Sample function that add three integers
//  Registers 10, 11, 12 are parameter registers
//  Register 10 contains the value returned
//
            SETENV      "RISCV",    /> RISCV architecture
                        "RV64I:a,c,d,m,n,zicsr,zifencei", />
                        "LP64D", "linux"
//
            FrameworkDef            // Standard framework definitions
//
            EXPORT      Add3        // Export Add3 ENTRY symbol
//
Code        SECTION     ELF_SHT_PROGBITS,  /> Section for binary code
                        ELF_SHF_ALLOC+     />
                        ELF_SHF_EXECINSTR, />
                        8, ".text"  // Alignment 8, external name '.text'
//
            UTF_8       "Add3--->"  // Define eye catcher
//
Add3        ADD         10, 11      // Add second parameter to first
            ADD         10, 12      // Add third parameter to first
//
Exit        JALR        0, 0[1]     // Return to caller via link register 1
//
            END
```

In the code above, these are the symbols defined:

`Code`	This symbol defines the beginning of the section used. As such it has an offset value of zero and a length of zero, since no data is generated.
`Add3`	Symbol `Add3` defines the function entry and it is exported. Since it lays after the eye catcher which is 8 bytes long it has offset value of 8 and length of 4 which is the length of the machine instruction `ADD`.
`Exit`	Symbol `Exit` has offset value of 16, which is the offset value of symbol `Add3` plus the length of the two machine instructions that follow `Add3`. It also has a length of 4 which is the length of machine instruction `JALR`.

4.1 Sections

A section is started with the `SECTION` directive. More than one section can be specified in an input file. The `SECTION` directive terminates the current section, if any, and starts a new one. Sections have an internal name and an optional external name. When an external name is not used, the internal name is used externally, after upper casing. When specifying multiple section the external name can be reused multiple times, however the internal name must be unique.

Sections have attributes. DVASM™ allows programmers to specify all attributes supported by ELF object files.

4.2 MMAPs

`MMAP` stands for "memory map" and it is a section that does not generate any binary code. It is used to map a memory structure so that items in the structure can be addressed symbolically. When DVASM™ encounter the `MMAP` opcode the current `SECTION` or `MMAP` is terminated and the new `MMAP` is started. For those who are familiar with `C` language, `MMAP` is similar to a `C` structure defined using `typedef`.

4.3 Symbols

Symbols are always defined as labels in a statement. Symbols have a unique name and cannot be redefined. Also, symbols can be used as part of an expression.

When symbols are defined as labels of a statement with an opcode, they have an attribute of **offset** integer with length equal to the length of binary code generated for the statement and replication of 1 unless a different replication factor is specified. In this case symbols are associated to the `SECTION` or `MMAP` currently active for the statement.

When symbols are defined as labels of statements using the `EQU` directive, it can have other attributes. These are:

Absolute integer — This is the result of an expression which contains only integer constants or symbols previously defined as absolute integer only. The length is always zero and the replication factor is always one.

Displacement integer — This is the result of the difference of two offsets associated with the same `SECTION` or `MMAP`. The length is always zero and the replication factor is always one.

Base-Displacemet Integer — This is a displacement which is associated to a register number and can be used by machine instruction to access memory.

Floating point — This is the result of an expression that contains at least one floating point constant or a symbol previously defined as floating point. The length is always zero and the replication factor is always one.

String — This is the result of an expression which contains only string constants or symbols previously defined as strings only. The length is always zero and the replication factor is always one.

4.4 External Symbols

All symbols, created using a label preceding an opcode, are considered **local** symbols. However, if there is a need to address symbols created outside the current source files, these symbols are defined using the `EXTERNAL` directive, and are used using a relocation

qualifier (see relocations on page 21).

Normally external symbols are associated with a value, which most of the time is an address, and sometimes with size or length when the computer architecture ABI supports it.

Local symbols can also be reference using a relocation qualifier, when the symbol is defined in a section and is reference in a different section.

Chapter 5

Expressions

Opcode parameters and sub-parameters are expressions. Expressions are formed by combining symbols and constants with unary and binary operators. Parenthesis are allowed.

5.1 Constants

These are the constants that can be specified:

Absolute Integer — It is any positive or negative integer specified in base 10, base 16 or base 2. When using base 16 or base 2 an underscore character can be used to separate group of digits. For example the constant `1023` can be also be written as `0xFFF`, `0x0F_FF`, `0b111111111111` and `0b1111_1111_1111`. An Absolute Integer constant can also be specified as an ASCII character. For example `'a'` is internally converted to `97`.

Floating Point — Floating point constants are expressed using the decimal base only. It is composed of a positive or negative mantissa followed by a positive or negative exponent. When the mantissa is a fractional number the exponent is optional and defaults to zero. When the mantissa is an integer an exponent is required, otherwise the constant becomes and Absolute Integer. For example Floating Point number `-122E-02` can also be written

as `-1.22` or as `-12.2E-1`.

String — String constants are defined the same way Java string constants are. They are enclosed in double quotes, are internally represented as UTF-16 Unicode strings and are not silently null terminated as in the `C` language. If, for example, the string `"This is a string"` must be null terminated then `"This is a string\u0000"` must be used.

5.2 Symbols

Symbols can be used alone or followed by a qualifier. When used alone, the actual symbol value and value attribute is used when evaluating an expression.

Symbol qualifiers are used to specify additional values associated with the symbol. The following qualifiers can follow a symbol without interleaving spaces:

Length — When qualifying a symbol with length, the length associated with the symbol is used instead of its value. Symbols length are always non-negative Absolute Integers. Symbols are length qualified by appending the `.$LENGTH` string to the symbol name.

Replication — When qualifying a symbol with replication, the replication factor associated with the symbol is used instead of its value. Symbol replication factors are always positive Absolute Integers. Normally, replication factors are one. They are larger than one when using a data opcode to define an array. Symbols are replication qualified by appending the `.$REPL` string to the symbol name.

Size — When qualifying a symbol with size, the product of length and replication factor associated with the symbol is used instead of its value. Symbol sizes are always non-negative Absolute Integers. Normally, replication factors are one. They are larger than one when using a data opcode to define an array. Symbols are replication qualified by appending the `.$SIZE` string to the symbol name.

Relocation

When qualifying a symbol with relocation, the symbol value is not used. Instead DVASM produces information in the output ELF file for the linker or loader to put the actual value of the symbol at link or load time. The reason for using relocations is that the symbol value will be available at link or load time only. A symbol is relocation qualified by appending the symbol name with `.@relocation_type`. Relocation types that can be used are dependent on the computer architecture **and** ABI being used. `@relocation_type` is case insensitive. When there is a need to specify multiple relocations for the same symbol in an expression, a symbol can be appended with multiple relocation types such as `symbol_name.@relocation_type1.@relocation_type2`.

5.3 Operators

Operators can be unary or binary. Operator's operand(s) can be symbols, constants or expressions enclosed in parenthesis.

5.3.1 Operator Precedence

Like in any computer language each operator used in an expression has a precedence with respect to other operators and an association direction. Table 4.3 contains the complete list all operators with their precedence and association.

Precedence	Operator	Description	Association
1	+	Unary Plus	None
	-	Unary Minus	
	!	Unary Logical Not	
	~	Unary Binary Not	
2	**	Power	Right to Left
3	*	Multiply	Left to Right
	/	Divide	
	%	Reminder	
4	+	Add	Left to Right
	-	Subtract	

Table 5.1: Operator Precedence in Descending Order Cont'd

Precedence	Operator	Description	Association
5	«	Bitwise Shift Left	Left to Right
	»	Bitwise Shift Right	
6	>	Greater Than	Left to Right
	>=	Greater Equal	
	<	Less Than	
	<=	Less Equal	
7	==	Equal	Left to Right
	!=	Not Equal	
8	&	Bitwise AND	Left to Right
9	|	Bitwise OR	Left to Right
10	^	Bitwise XOR	Left to Right
11	&&	Logical AND	Left to Right
12	||	Logical OR	Left to Right

Table 5.1: Operator Precedence in Descending Order

5.3.2 Operators and Operands Types

Each operator, unary or binary, can produce different results with different combinations of operand(s).

For example the binary plus `"+"` operator will concatenate two strings when the two operands are Strings, and it will add two integers when the two operands are Absolute Integers. However it will generate an error when one of the operands is a String and the other is an Absolute Integer.

These are the most general rules when using an operator in an expression:

- Absolute and Displacement Integers are handled, for the most part, the same way. When both are used with a binary arithmetic operator, the result is normally a Displacement Integer;
- Bitwise binary operators can only be used with Absolute or Displacement Integers as operands in any combination;
- When an Absolute or Displacement Integer is used with a Floating Point as operands of a binary arithmetic operator, the result is always a Floating Point type;

- When two Offset Integers from the same Section or MMAP have the same sign and are subtracted from each other, or have opposite sign and added to each other the result is an Integer Displacement. In all other cases the result is an aggregation of separate Offset Integers (see below for a more detailed explanation);
- Comparison operators will accept Absolute Integers, Displacement Integers and Floating Point as operands in any order. The result is always an Absolute Integer with value of either `1 (true)` or `0 (false)`;
- Logical operators will accept only Absolute and Displacement Integers as operand(s) in any order, with the assumption that any non-zero value is `true` and zero value is `false`. The result is always an Absolute Integer with value of either `1 (true)` or `0 (false)`.

Integer Offsets aggregation are intermediate terms that arise during an expression evaluation. An expression whose final result is an Integer Offsets aggregation is in error.

Let us consider this example:

```
Off_044 + Off_024 - 34 - ( Off_032 + Off_10 )
```

Symbols `Off_010`, `Off_024`, `Off_032`, and `Off_044` are all offset from the same section with respective values of `10`, `24`, `32`, and `44`. The first two term are first added and the result is:

```
Aggregate((Off_044, plusSign), (Off_024, plusSign)) - 34 -
    ( Off_032 + Off_10 )
```

Next the subtraction is evaluated, and the result is:

```
Aggregate((Off_044, plusSign), (Off_024, plusSign), -34) -
    ( Off_032 + Off_10 )
```

Thus Absolute Integer `-34` become part of the Integer Offset aggregate. Next the addition in parenthesis is evaluated and the result is:

```
Aggregate((Off_044, plusSign), (Off_024, plusSign), -34) -
    Aggregate((Off_032, plusSIgn), (Off_10, plusSign))
```

Finally the two aggregate are subtracted to each other. Since all offset are from the same section and all positive, those on the left of the minus operator are paired to those on the right and subtracted to each other as follows:

```
Off_044 - Off_032  +  Off_024 - Off_010  - 34
```

or

```
44 - 32  +  24 - 10  - 34
```

which results in an Integer Displacement of `-8`. It is important to note that if we change the sign in front of `Off_010` from plus to minus, in the original expression, Integer Offset `Off_024` cannot be paired anymore with Integer Offset `Off_010`, since subtraction of two Integer Offsets of opposite sign will result in a new aggregate. This in turn results in the expression to be in error.

A complete list of all valid operator/operand(s) combinations with corresponding results type is given in Table 4.2. Relocation qualified symbols are excluded. Any combination not listed will result in an error if used.

Operator	Operation	First Operand						Second Operand						Result
		ABS	DSP	B-DSP	OFF	FLT	STR	ABS	DSP	B-DSP	OFF	FLT	STR	
+	Unary Plus	x												ABS
			x											DSP
				x										B-DSP
					x									OFF
						x								FLT
-	Unary Minus	x												ABS
			x											DSP
					x									OFF
						x								FLT
~	Bitwise Not	x												ABS
			x											DSP
!	Logical Not	x												ABS
			x											DSP

Table 5.2: List of Operators and Operand Types Cont'd

Operator	Operation	First Operand						Second Operand						Result
		ABS	DSP	B-DSP	OFF	FLT	STR	ABS	DSP	B-DSP	OFF	FLT	STR	
**	Power	x						x						ABS
		x							x					DSP
			x					x	x					
						x		x	x					FLT
*	Multiply	x						x						ABS
		x							x					DSP
			x					x	x					
		x										x		FLT
						x		x						
						x						x		
/	Divide	x						x						ABS
		x							x					DSP
			x					x	x					
		x										x		FLT
						x		x						
						x						x		
%	Reminder	x						x						ABS
		x							x					DSP
			x					x	x					

Table 5.2: List of Operators and Operand Types Cont'd

Operator	Operation	First Operand						Second Operand						Result
		ABS	DSP	B-DSP	OFF	FLT	STR	ABS	DSP	B-DSP	OFF	FLT	STR	
+	Plus	x						x						ABS
		x							x					DSP
			x					x	x					
				x				x	x					B-DSP
		x	x								x			OFF
					x				x					
					x						x			
		x										x		FLT
						x		x						
						x						x		
							x						x	STR
-	Minus	x						x						ABS
		x							x					DSP
			x					x	x					
				x				x	x					B-DSP
		x	x								x			OFF
					x				x					
					x						x			DSP
		x										x		FLT
						x		x						
						x						x		

Table 5.2: List of Operators and Operand Types Cont'd

Operator	Operation	First Operand						Second Operand						Result
		ABS	DSP	B-DSP	OFF	FLT	STR	ABS	DSP	B-DSP	OFF	FLT	STR	
«	Shift Left	x						x						ABS
			x					x						DSP
		x							x					
«	Shift Right	x						x						ABS
			x					x						DSP
		x							x					
>	Bigger	x	x					x	x					ABS
		x				x		x				x		
<	Less	x	x					x	x					ABS
		x				x		x				x		
>=	Bigger Equal	x	x					x	x					ABS
		x				x		x				x		
<=	Less Equal	x	x					x	x					ABS
		x				x		x				x		
==	Equal	x	x					x	x					ABS
		x				x		x				x		
!=	Not Equal	x	x					x	x					ABS
		x				x		x				x		

Table 5.2: List of Operators and Operand Types Cont'd

Operator	Operation	First Operand						Second Operand						Result
		ABS	DSP	B-DSP	OFF	FLT	STR	ABS	DSP	B-DSP	OFF	FLT	STR	
&	Bitwise AND	x						x						ABS
			x					x						DSP
		x							x					
|	Bitwise OR	x						x						ABS
			x					x						DSP
		x							x					
^	Bitwise XOR	x						x						ABS
			x					x						DSP
		x							x					
&&	Logical AND	x	x					x	x					ABS
||	Logical OR	x	x					x	x					ABS

Table 5.2: List of Operators and Operand Types

5.4 Relocation Qualifier Restrictions

When an external or internal symbol appears in an expression and it is qualified by a relocatable attribute, the following restrictions apply:

- Only one relocation qualified symbol can appear in an expression and it must be the first operand in the expression;
- The relocation qualified symbol can be followed only by the binary plus or minus operators;
- The rest of the expression must resolve to an Absolute or Displacement Integer and represent the add-on of the relocation value.

5.5 Examples

5.5.1 Absolute Integers

In the following example, symbols `ABS001` and `ABS002` are defined as Absolute Integers:

```
//
//          Use ABS001 to define ASB002 by adding 64 and multiplying by 4
//
ABS002      EQU     ( ABS001 + 64 ) << 2
//
//          Define ABS001 as the product of two integers
//
ABS001      EQU     12 * 145
//
```

It is important to note the following:

- There is no requirement that symbols must be defined in the order they are used as in `C` and `C++`. Any order can be used as long as a circular unsolvable situation is not used. A circular situation is, for example, symbol `S001` is used to define symbol `S002` and later symbol `S002` is used to define symbol `S001`;
- Symbol `ABS002` is defined as Absolute Integer, since the expression, in its definition, contains only integer constants.
- Symbol `ABS001` is defined as Absolute Integer, since the expression, in its definition, contains only integer constant and a symbol that is defined as an Absolute Integer.

5.5.2 Displacement Integers

In the following example Displacement Integers are defined by subtracting one Offset from another:

```
//
//          Define the length of the data section as difference of two Offsets
//
LEN001      EQU     DATA002 - DATA001
//
//          Define number of words in data section
//
NWORD001    EQU     LEN001 / 4
//
DATA001     WRD     15[4]       // Define four words initialized to 15
//
            WRD     0[16]       // 16 more words set to zero
//
DATA002     WRD     0[0]        // Zero words (i.e. no data) to mark the end
//
```

It is important to note the following:

- Symbol `LEN001` is a Displacement Integer since it is the difference of two offsets within the same section or `MMAP`.
- Symbol `NWORD001` is also a Displacement Integer since the expression used to define it contains a Displacement Integer.

5.5.3 Base Displacement Integers

In the following example a Base Displacement Integer is used to load a word from memory using the RISC-V load word machine instruction.

```
//
            BASDEF                      // Define register symbols
//
//          At this point both registers X20 and X21 contain the address
//          of data mapped by MMAP DATA
//
//          Define X20 as Base Register for MMAP DATA with ID BASEID
//
            BASESET BASEID:DATA, X20
//
//          Define x21 as Base Register for MMAP DATA without any ID
//
            BASESET DATA, X21
//
//          Load first BB word within MMAP DATA into register X5
//          using base register X20
//
            LW      X5, BASEID:BB
//
//          Load third BB word within MMAP DATA into register X6
//          using base register X21
//
            LW      X6, BB+BB.$LENGTH*2
//
//          Define MMAP
//
DATA        MMAP
AA          WORD    0[2]            // Two words (4 bytes each)
BB          WORD    0[4]            // Four more words (4 bytes each)
CC          DWRD    0               // One double word (8 bytes)
//
```

It is important to note the following:

- Register `X20` is defined as the base of `DATA` MMAP using a base ID. Thus register `X20` is used as base register only when the addressed memory symbol is prefixed with the base ID used in the BASESET directive.
- Register `X21` is defined as the base of `DATA` MMAP without using any base ID and is used as the default base register for any symbol in MMAP `DATA` within the displacement rage of the machine instruction `LW`.

5.5.4 Floating Point

In the following example Floating Point numbers are used to create 64 bits IEEE floating point constants in memory.

```
//
//          Define PI using EQU directive
//
PI          EQU     3.141592653589793
//
PIDW        DWRD    PI              // Create 64 bit floating point number
                                    // containing PI
PIDWX2      DWRD    PI * 2          // Create 64 bit floating point number
                                    // containing PI times 2
//
EULERNUM    DWRD    2.718281828459045
                                    // Create 64 bit precision point number
                                    // containing Euler number
//
```

It is important to note the following:

- When `PI` is defined it becomes a Floating Point number since the expression used is a single number with a decimal point. The value is internally defined by DVASM™ framework with 256 decimal digit precision.
- When `PIDW` storage is defined, a IEEE 64 bit number is created since the `PI` symbol, which is a Floating Point number, is used in the defining expression. The same applies for symbol `PIDWX2` and symbol `EULERNUM`.

5.5.5 Strings

In the following example String symbols are defined and used to create a UTF8 character string in memory.

```
//
//          Define first name, middle initial, and last name of a person
//
FName       EQU     "John"
MInit       EQU     "X"
LName       EQU     "Doe"
//
//          Create UTF8 string
//
Str01       UTF_8   "User " + FName + " " + MName + ". " + />
                    LName + " is not in the database\U0000"
//
```

It is important to note the following:

- Symbols `FName`, `MInit` and `LName` are all defined as Strings since the defining expressions are Strings.
- UTF8 string is created in memory by concatenating String symbols with String constants. The UTF8 string is null terminated by inserting the unicode character `\u0000` which maps to a single byte null character.
- UTF8 string is created using label `Str01`. Symbol `Str01` is an Offset and **not** a string.

Chapter 6

ELF Output Files

DVASM™ produces object files in the ELF format only. ELF has become the standard format for Linux and any flavor of UNIX, and is now supported by MS Windows. If your target operating system does not support ELF format you can always use the Linux `binutils` to convert an ELF object file to almost any object format known on this planet.

ELF support two formats: object and executable. DVASM™ produces only objects. Executables are produced using a linker such as the Linux `ld` command to combine one or more object files into an executable module.

If `C` code object files are combined with DVASM™ object files, an executable module can be created by first assembling DVASM™ source files, and than using the `C` compiler to compile `C` source code and create the executable in one shot.

The format of ELF object files is beyond the scope of this manual, however it should be noted that the content of any ELF file can be shown in readable format using the `readelf -a` command.

6.1 ABI

When writing DVASM™ assembly code, the first directive in the input file is `SETENV`. One of the parameters to `SETENV` is the ABI being used. ABI stands for Application Binary Interface. The ABI specifies, many things such as register and stack usage when calling an external function, or when being called by another function. The only thing which is

relevant to DVASM™ is the part of the ABI that specifies values and formats of Relocatables, since this has an effect on how the ELF object output is generated.

6.2 Relocatables

When an assembly program references an external variable or function it uses relocatables. Relocatables are memory areas, within the code generated, that contain addresses, in complete or partial form, and either absolute or relative, to access those external variables or functions.

The formats of these addresses vary by architecture and to some degree by ABI. RISC-V architecture with System V ABI is used in all the following examples.

In the following example, a 64 bit constant within the code is created to contain the address of function `xtrfunct`.

```
//
//          Store address of function xtrfunct in a 64 bit constants
//          using RISC-V relocatable of type R_RISCV_64
//
xtrfuncadr  DWRD    xtrfunc@_64
//
```

Since the address will be known only at execution time when the module is being loaded by the operating system loader, the 64 bits containing function `xtrfunc` will be set by the loader itself, and it will be different every time the module is loaded at a different location in memory

In the following example, the address of function `xtrfunc` is loaded into register `10` using PC-relative (i.e. Program Counter relative) relocatables.

```
//
//          Load address of xtrfunct in register 10 using PC-relative
//          relocatables using RISC-V relocatables of type
//          R_RISCV_PCREL_HI20 and R_RICSV_PCREL_LO12_I
//
lbl001      AUIPC   10, xtrfunc@_PCREL_HI20
            ADDI    10, lbl001@_PCREL_LO12_I
//
```

Both machine instructions have, stored in their immediate fields, the displacement between

function `xtrfunc` and memory location of machine instruction `AUIPC`. For the first machine instruction only the high 20 bits are stored in its immediate, while for the second machine instruction only the low 12 bits are stored. At execution time the `AUICP` adds its immediate value, shifted left by 12 bits, to the current value of the program counter, and store it in register 10. The second machine instruction adds to the same register the missing 12 bits to form the address of function `xtrfunc`. Since the displacement between function `xtrfunc` and the first machine instruction is known at link time, the immediates of the two machine instructions can be set by the linker at link time. This is true only if function `xtrfunc` is part of the module and does not belong to an external shared object.

The use of PC-relative relocatables makes loading a module faster since fewer relocatables need to be reset at load time.

However when writing a boot loader or an OS kernel, the use of PC-relative relocatables is a requirement since no loader is available to reset relocatable at boot time.

Chapter 7

Framework Data Opcodes and Directives

DVASM™ provides both data opcodes and directives.

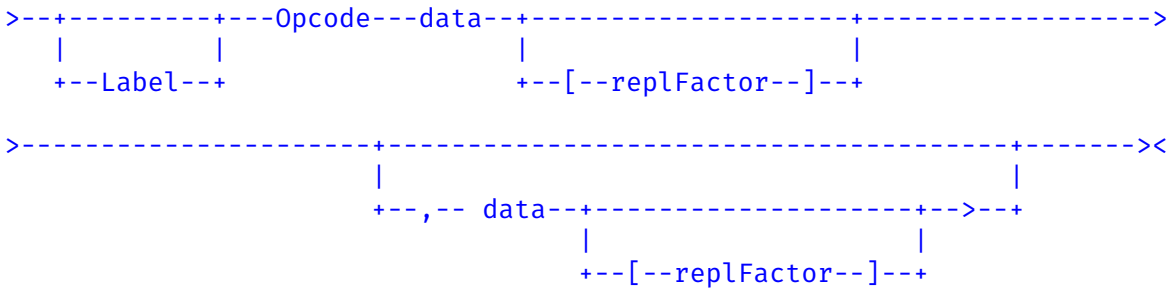
Data opcodes tell the assembler to reserve memory space and initialize it to some constant value. Directives tell the assembler how to behave under certain circumstances.

7.1 Data Opcodes

A data opcode specifies that a certain number of bytes at the current location must be reserved and set to a specified constant. Data opcodes can only be used when a SECTION or an MMAP is active. When a SECTION is active the bytes reserved will be initialized as specified by the data opcode parameter(s). When an MMAP is active no initialization takes place since an MMAP is a memory template with no binary code being generated. There are two types of data opcodes: words and Unicode strings.

7.1.1 Word Data Opcodes

Data opcode statements are coded as follows:



`Label` Optional label used to refer to this data.

`Opcode` Name of the data opcode (see table 7.1).

`Pos. Parameters`

`data[replication], ..., data[replication]`
Required, comma separated parameter list of up to 256 parameters, each specifying a constant and optionally a replication factor. The constant is used to initialize the reserved memory area. The replication factor is defaulted to one, and it specifies how many areas, initialized to the same constant value, must be reserved in memory.

`Key-word Parameters`

`None`

Words length are a power of two, starting from a length of 1 byte to a length of 64K bytes. Word data opcodes whose name starts with `BYTE_` followed by the length value all have an alignment value of 1. Word Data opcodes whose name starts with `WRD_` followed by their length value have an alignment value equal to their respective length. Word Data opcodes can have, as parameters, `INTEGER`s of type `ABSOLUTE` or `DISPLACEMENT`, and relocatables when allowed by the architecture. Also, for word data opcodes with lengths of 2, 4, 8, 16, and 32 bytes a `FLOATING POINT` parameter type can be specified, which is encoded using the IEEE floating point specifications.

`INTEGER`s are encoded using two's complement encoding. If the `INTEGER` is positive and its value overflows on the sign bit, it will be encoded as unsigned. Any other overflow, for positive or negative `INTEGER`s, will results in an error.

`INTEGER`s and `FLOATING POINT` values are encoded using the endianness (big or little) being active when the data opcode is parsed by DVASM™ .

Table 7.1 contains the complete list of data opcodes, with alias names, parameter types supported, length and alignment.

Name	Aliases	Parameter Types	Length	Alignment
BYTE_1	WRD_1, BYTE	Integers	1	1
BYTE_2		Integers, Floating Point	2	1
WRD_2	HWRD	Integers, Floating Point	2	2
BYTE_4		Integers, Floating Point	4	1
WRD_4	FWRD	Integers, Floating Point	4	4
BYTE_8		Integers, Floating Point	8	1
WRD_8	DWRD	Integers, Floating Point	8	8
BYTE_16		Integers, Floating Point	16	1
WRD_16	QWRD	Integers, Floating Point	16	16
BYTE_32		Integers, Floating Point	32	1
WRD_32		Integers, Floating Point	32	32
BYTE_64		Integers	64	1
WRD_64		Integers	64	64
BYTE_128		Integers	128	1
WRD_128		Integers	128	128
BYTE_256		Integers	256	1
WRD_256		Integers	256	256
BYTE_512		Integers	512	1
WRD_512		Integers	512	512
BYTE_1K		Integers	1K	1
WRD_1K		Integers	1K	1K
BYTE_2K		Integers	2K	1
WRD_2K		Integers	2K	2K
BYTE_4K		Integers	4K	1
WRD_4K		Integers	4K	4K
BYTE_8K		Integers	8K	1
WRD_8K		Integers	8K	8K
BYTE_16K		Integers	16K	1
WRD_16K		Integers	16K	16K

Table 7.1: List of Word Data Opcodes Cont'd

Name	Aliases	Parameter Types	Length	Alignment
BYTE_32K		Integers	32K	1
WRD_32K		Integers	32K	32K
BYTE_64K		Integers	64K	1
WRD_64K		Integers	64K	64K

Table 7.1: List of Word Data Opcodes

When a label is specified for a word data opcode, the length of the symbol is set to the length of a single word, the replication factor is set to the sum of the replications factors of each parameter specified. If no replication factor is specified for a given parameter, it is set to one. If the replication factor for a given parameter is set to zero, no data is generated for that parameter. Zero replication is normally used to for alignment data to the corresponding alignment of the word used.

7.1.2 Word Data Opcode Examples

In the following example a 4 byte word is defined and initialized to an `ABSOLUTE INTEGER` value:

```
Wrd01       FWRD        4096*100        // Set value to buffer size
```

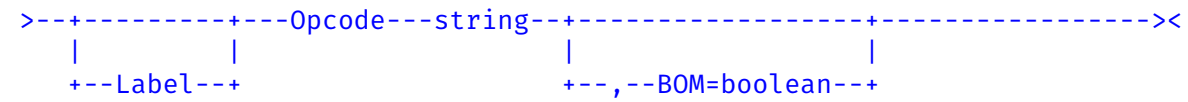
In the following example a null terminated string is initialized one byte at a time:

```
Str01       BYTE        'T', 'h', 'i', 's', ' ', 'i', 's', ' ',     />
                        'a', ' ', 's', 't', 'r', 'i', 'n', 'g', 0
```

It is important to note that since the string is defined as a collection of separate bytes, the length `Str01.$LENGTH` of symbol `Str01` is set to 1. The replication factor `Str01.$REPL` is set to the number of bytes which is 17, and the total size `Str01.$SIZE` is set to 17 (i.e. word length time replication factor).

7.1.3 Unicode Data Opcodes

Unicode data opcode statements are coded as follows:



`Label`	Optional label used to refer to this data.
`Opcode`	Unicode opcode: `UTF_8`, `UTF_16` or `UTF_32`.
`Pos. Parameters`	
`string`	Required, one expression that resolve to a string, without any subparameter (i.e. no replication factor).
`Key-word Parameters`	
`BOM=boolean`	Specifies if the Unicode string should be encoded starting with a BOM character.

All three Unicode formats are supported by DVASM™ . Unicode UTF-8 is supported by opcode `UTF_8`, Unicode UTF-16 is supported by data opcode `UTF_16` and Unicode UTF-32 is supported by data opcode `UTF_32`. Alignment for the encoded storage is `1` for `UTF_8`, `2` for `UTF_16` and `4` for `UTF_32`.

All Unicode data opcodes have the key-word parameter `BOM=` with possible values of `1` for true and `0` for false, with default set to false. When `BOM=` is set to true the the Unicode BOM character sequence is inserted in the Unicode character encoding generated.

When characters are encoded with more than one byte, the byte order used is the endianness (big or little) being active when the data opcode is parsed by DVASM™ .

7.1.4 Unicode Data Opcode Examples

In the following example an ASCII string, null terminated to make it compatible with `C` language library, is defined:

```
Str01       UCF_8       "Number of users is \t135\n\u0000"
```

Since the string contains only ASCII characters it is encoded as an ASCII string (i.e. all

characters are encoded as one byte). Also both special characters tab and new-line are used. The terminating null character is specified by using the Unicode value `\u0000`.

In the following example a UCF 16 Unicode string is specified with a `BOM` character at the beginning to identify the endianness used for encoding:

```
Str02       UCF_16      "This is a Unicode string encoded using UCF 16", />
                        BOM=1
```

The string will be encoded using the endianness active when the the statement is parsed and the `BOM` character will be set accordingly. No null terminator is specified.

In the following example a Unicode UCF 16 string to be transmitted over a network connection is encoded using big endianness:

```
            BIGENDIAN                   // Set data constants to BIG ENDIAN
Str02       UCF_16      "This is a Unicode string " +   />
                        "encoded using UCF 16 - big endian"
            DEFENDIAN                   // Reset endianness to default
```

The string is encoded using UCF 16, big endian, since endianness is switched to big using the `BIGENDIAN` directive. Endianness is reverted to default, after encoding the string, using the `DEFENDIAN` directive. It is important to note that this example will work regardless of what is the default architecture endianness, since it uses the correct endianness to encode the string, and restores the default endianness afterword.

7.2 Control Directives

In this section all control directives are documented in alphabetic order.

7.2.1 ASSERT Directive

Verify that the given expression resolves to true. If not, raise an error.

```
>----------------ASSERT------expression-------------------------------><
```

`Label`	Not allowed.
`Name`	`ASSERT`.
`Pos. Parameters`	
`expression`	Required, must resolve to an `INTEGER` of type `ABSOLUTE` or `DISPLACEMENT`.
`Key-word Parameters`	
`None`	

In the following example a MMAP is defined to map data in an working area whose addressed is passed to a function by the caller. The working area is 64 bytes long. The `ASSERT` directive is used to make sure that the MMAP length is no longer that 64 bytes, now and in the future if it will be modified during maintenance.

```
//
//WAMAxLength   EQU     64          // Define work area length
//
//WArea         MMAP
//
//WAVect1       FWRD    0[8]        // Vector of eight words
//
//WAVext2       DWRD    0[4]        // Vector of 4 double words
//
WALength        EQU     $PC-WArea   // Define MMAP length
//
                ASSERT  WALength <= WAMaxLength
                                    // Verify that MMAP does not overflow
//
```

7.2.2 BASECLEAR Directive

Drop all base registers previously defined.

```
>----------------BASECLEAR-----------------------------------------><
```

`Label` Not allowed.

`Name` `BASECLEAR`.

`Pos. Parameters`

None

`Key-word Parameters`

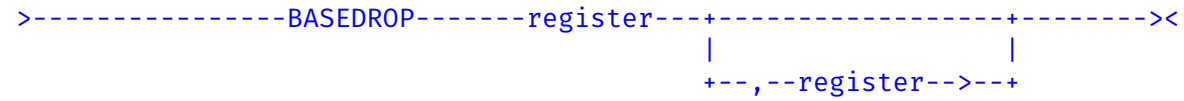
None

In the following example three registers, previously defined as base registers, are dropped using the `BASECLEAR` directive.

```
//
//          Define three base registers
//
            BASESET     Data1,      R10
            BASESET     Data2,      R12
            BASESET     ID3:Data3,  R18
//
//  Insert code that use the three registers here
//
            BASECLEAR           // Drop all previously define base registers
//
```

7.2.3 BASEDROP Directive

Drop a previously define base register.



`Label`	Not allowed.
`Name`	`BASEDROP`.
`Pos. Parameters`	
`register`	Required, list of up to 256 expressions, each resolving to `INTEGER ABSOLUTE`. These are the base registers that must be dropped.
`Key-word Parameters`	
`None`	

In the following example three registers are defined as base registers, and two of them are than dropped using the `BASEDROP` directive.

```
//
//          Define three base registers
//
            BASESET     Data1,      R10
            BASESET     Data2,      R12
            BASESET     ID3:Data3,  R18
//
//  Insert code that use the three registers here
//
            BASEDROP    R12, R18    // Drop base register R12 and R18
//
```

7.2.4 BASESET Directive

Set a base register using an OFFSET or a BASE DISPLACEMENT in the form ID:OFFSET.

```
>----------------BASESET--------offset,---register--------------------><
```

Label	Not allowed.
Name	BASESET.
Pos. Parameters	
offset	Required, must resolve to either type OFFSET or type BASE DISPLACEMENT. This is the memory location pointed by the base register.
register	Required, must resolve to type INTEGER ABSOLUTE. This is the base register.
Key-word Parameters	
None	

In the following example, two registers are defined as base registers using the BASESET directive and they are used to load two words from storage. Both words are within the range of both base registers via a positive or negative offset (RISC-V supports negative offsets). In both cases DVASM™ will choose the smallest offset in absolute value.

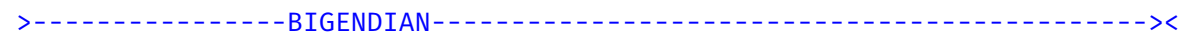
```
//
//          Define three base registers
//
            BASESET     Data1,  R12 // Define R12 as base register for Data1
            BASESET     Data2,  R18 // Define R18 as base register for Data2
//
            LW          R21, Data1+8
                                    // Load third word in Data1
                                    // Base register R12 is used with
                                    // offset +8.  If R12 were defined as
                                    // base R18 would be used with
                                    // offset -44
//
            LD          R20, Data2+8
                                    // Load second double word in Data2
                                    // Base register R18 is used with
                                    // offset +8. If R18 were not defined
                                    // as base register R12 would be used
                                    // with offset +60
//
Data1       FWRD        13[8]       // Define word vector
//
Data2       DWDR        9[4]        // Define double word vector
//
```

In the following example, two different registers are set as bases for the same MMAP using different base IDs. Each register points to a different instance of the MMAP in memory. The use of base IDs allow to specify which instance of the MMAP is being accessed.

```
//
//          Define three base registers
//
            BASESET     ID1,DataCell, R12   // Define R12 as base register
                                            // for DataCell first instance
                                            // using base ID ID1
            BASESET     ID2:DataCell, R18   // Define R18 as base register
                                            // for DataCell second instance
                                            // using base ID ID2
//
            LW          R20, ID1:DataCell+8 // Load third word in first
                                            // instance of DataCell
//
            SW          R20, ID2:Data1+4    // Store it in second word of
                                            // second instance of DataCell
//
DataCell    MMAP
            FWRD        13[8]               // Define word vector
//
```

7.2.5 BIGENDIAN Directive

Set byte encoding of following Data opcodes to Big Endian.



`Label` Not allowed.

`Name` `BIGENDIAN`.

`Pos. Parameters`

None

`Key-word Parameters`

None

An example on how to use `BIGENDIAN` is given in subsection describing directive `DEFENDIAN` on page 59.

7.2.6 CNTRES Directive

Reset Program Counter to default value, which is the current SECTION or MMAP length.

```
>----------------CNTRES-------------------------------------------------><
```

`Label` Optional.

`Name` `CNTRES`.

`Pos. Parameters`

None

`Key-word Parameters`

None

An example on how to use `CNTRES` is given in subsection describing directive `CNTSET` on page 55.

7.2.7 CNTSET Directive

Set Program Counter to the offset specified.

```
>---------------CNTSET-------offset----------------------------------><
```

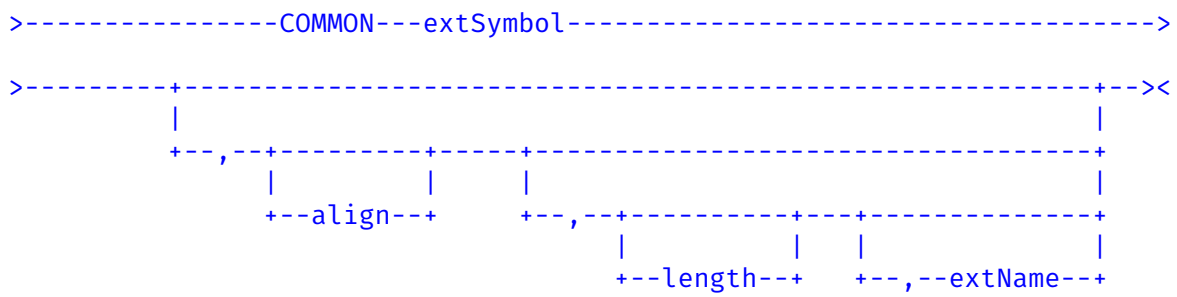
`Label`	Optional
`Name`	`CNTSET`.
`Pos. Parameters`	
`offset`	Required, it specifies the new offset for the Program Counter within the current `SECTION` or `MMAP`.
`Key-word Parameters`	
`None`	

In the following example the directives CNTSET and CNTRES are used to overlay different type of data within an MMAP.

```
//
//          Start the MMAP here
//
Data        MMAP
//
Vect01      FWRD        0[8]        // Define 8 word vector
//
            CNTSET      Vect01+4*2  // Reset program counter to third word
//                                  // of Vect01
Str01       BYTE        0[8]        // Define eight byte vector overlaying
//                                  // word 3 and 4 of Vect01
HVect       HWRD        0[4]        // Define vector of 8 half words
                                    // Overlaying words 5 and 6 of Vect01
//
FVect       DWRD        0[2]        // Define 2 double word vector
//                                  // Overlaying words 7 and 8 of Vect01
//                                  // and extending MMAP by 8 more bytes
//
            CNTSET      Str01       // Reset program counter to Str01 which
//                                  // is the same as 3 word of Vect01
//
Addr01      DWRD        0           // Overlay Str01 and words 3 and 4 of
//                                  // Vect01 with a double word
//
            CNTRES                  // Set program counter to current length
//                                  // of MMAP which is the size of Vect01
//                                  // plus 8 bytes
//
```

7.2.8 COMMON Directive

Define a COMMON storage area. Common block are normally used by programs written in the FORTRAN language.



`Label`	Not allowed.
`Name`	`COMMON`.
`Pos. Parameters`	
`extSymbol`	Required, expression must contain only one token which refers to an `EXTERNAL` symbol. It defines the external symbol used to access the COMMON from within the source code.
`align`	Optional, must be of type `INTEGER ABSOLUTE`. It defaults to a value of `1`. It defines the alignment of the COMMON.
`length`	Optional, must be of type `INTEGER ABSOLUTE` or `INTEGER DISPLACEMENT`. It defaults to a value of `0`. It defines the length of the COMMON.
`extName`	Optional, must be of type `STRING`. It defaults to a null string. It defines the external name used in the ELF files. When defaulted to a null string, parameter 1 is used instead.
`Key-word Parameters`	
`None`	

In this example a COMMON block is defined with internal name `COMMDATA` external name `CommonData` of 1 megabyte length and aligned on a page boundary (4096 bytes).

```
//
//          Define common block
//
            COMMON  COMMDATA, 4096, 256*4096, "CommonData"
//
```

7.2.9 DEFENDIAN Directive

Set byte encoding to the default endianness, which is defined by the the architecture/ABI combination used in the SETENV directive.

Label	Not allowed.
Name	DEFENDIAN.
Pos. Parameters	
None	
Key-word Parameters	
None	

In the following example the directive `BIGENDIAN` and `DEFENDIAN` are used to map a header of a network message with integers in network order endianness, and then reset endianness to the default value.

```
//
//          Switch endianness to Big Endian
//
            BIGENDIAN           // Set endianness to BIG
//
DataType    FWRD        35      // Define data type
//
DataLength  FWRD        1024    // Define data length
//
            DEFENDIAN           // Reset endianness to default
//
```

The above example can be embedded in a macro. The macro itself can be used in multiple platforms, some using Big Endian, and some using Little Endian. The directive `DEFENDIAN` will reset the endianness to the default for the platform being used, without the need to know the platform specific endianness.

7.2.10 END Directive

End input file.

```
>----------------END-------------------------------------------------------><
```

`Label` Not allowed.

`Name` `END`.

`Pos. Parameters`

None

`Key-word Parameters`

None

The `END` directive indicates the end of the source file. Every source file must have one `END` directive at its end. After the `END` directive only comments are allowed, anything else will result in an error. In the following example the `END` directive is used to terminate input.

Comments are inserted after the `END` since it is allowed by DVASM™ .

```
//
//          Insert code here
//
            END                 // End of code
//
//  We use here some comments, for example as a summary of maintenance
//  Only comments are allowed to followed the END statement
//
```

7.2.11 EQU Directive

Define a new symbol.

```
>------Label-----EQU-----expression-------------------------------------><
```

`Label` Required.

`Name` `EQU`.

`Pos. Parameters`

`expression` Required, must resolve to any type except an external symbol or a relocatable expression.

`Key-word Parameters`

`None`

In this example symbol `Data01` is defined as an `INTEGER ABSOLUTE`

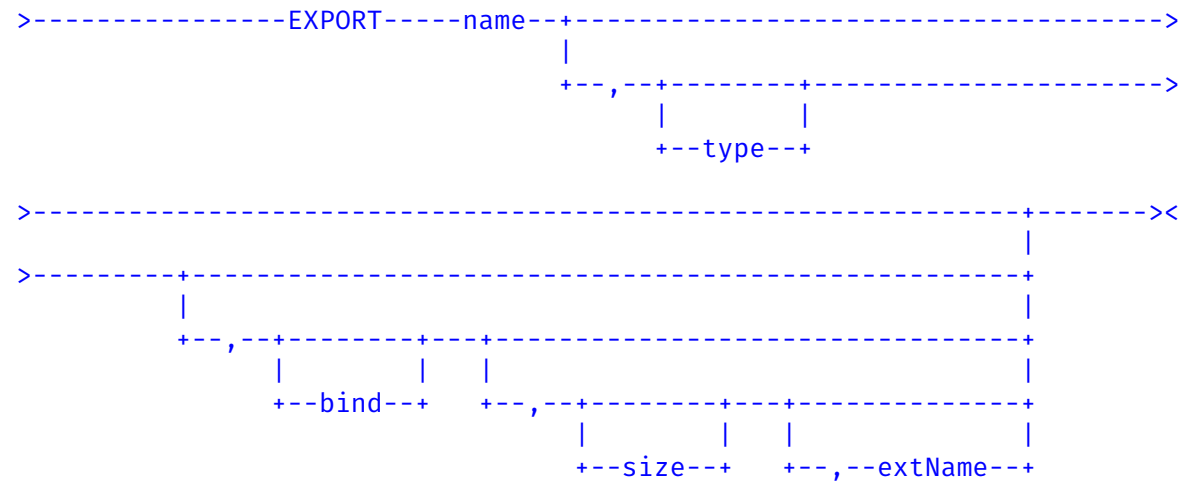
```
//
Data01      EQU     1 << 10         // Define 1024
//
```

In this example symbol `XXXLEN` is defined as a `INTEGER DISPLACEMENT`

```
//
XXX         FWRD    0[4]            // 4 word vector
//
YYY         FWRD    0[0]            // End of vector
//
XXXLEN      EQU     YYY-XXX         // Define length of vector which is 16
//
```

7.2.12 EXPORT Directive

Export a local symbol.



`Label` Not allowed.

`Name` `EXPORT`.

`Pos. Parameters`

`name` Required, expression must contain only one token which refers to a symbol to be exported.

`type` Optional, must be of type `INTEGER ABSOLUTE`. It default to a value of `1`. It defines the exported symbol type to be used in the ELF file.

`bind` Optional, must be of type `INTEGER ABSOLUTE`. It defaults to a value of `1`. It defines the exported symbol bind type to be used in the ELF file

`size` Optional, must be of type `INTEGER ABSOLUTE` or `INTEGER DISPLACEMENT`. It defaults to a value of `0`. It defines the exported symbol size in bytes to be used in the ELF file.

`extName` Optional, must be of type `STRING`. It defaults to a null string. It defines the name to be used in the ELF files for the symbol being exported. When defaulted to a null string, parameter 1 is used instead. The usage of parameter 1 as the external name is case sensitive, even though the symbol it refers to is

case insensitive. For example if a symbol, originally defined as `Abcd`, is spelled as `AbCd` in the `EXPORT` directive, than `aBcD` is used as the exported name default.

`Key-word Parameters`

None

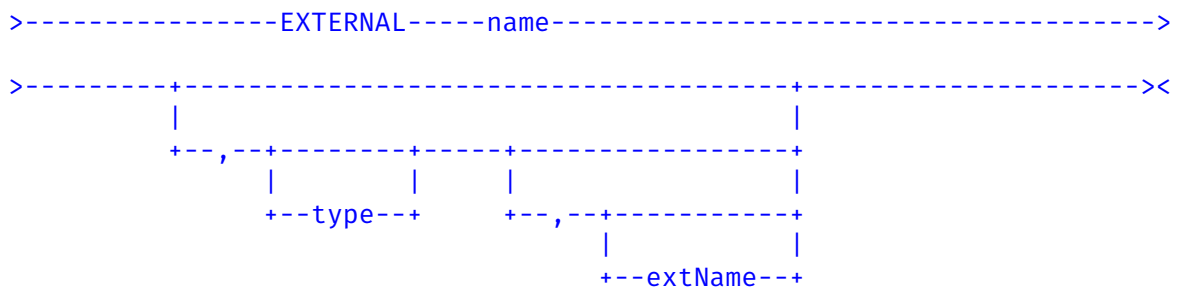
In the following example, run time default parameters are defined in a block of data, and its symbol /ttxDefParm is exported. exported.

```
//
//          Define default parameter block
//
DefParm     DWRD    0[0]
//
NumThreads  FWRD    12              // Default number of thread
DefMem      FWRD    1024*6          // Default memory in megabyte (6 GB total)
//
DefPort     HWRD    3999            // Default port
DefParmLen  EQU     $PC-DefParm     // Define parm length
//
            EXPORT  DefParm, , , DefParmLen, "DefaultParameters"
                                    // ELF symbol name is "DefaultParameters"
                                    // Type and bind are defaulted to 1
                                    // Size is set to length of DefParm
//
```

All possible values for ELF types, attributes, and bind for a section or an external/exported symbol are defined in the framework macro FrameWorkDef.mac.

7.2.13 EXTERNAL Directive

Define an external symbol for internal use.



`Label` Not allowed.

`Name` `EXTERNAL`.

`Pos. Parameters`

`name` Required, expression must contain only one token which refers to the name used internally to refer to the external symbol.

`type` Optional, must be of type `INTEGER ABSOLUTE`. It defaults to a value of `1`. It defines the external symbol bind type.

`extName` Optional, must be of type `STRING`. It defaults to a null string. It defines the name to be used in the ELF files for the external symbol. When defaulted to a null string, parameter 1 is used instead. In this case, even though the external symbol name used internally is case insensitive, the external name, as stored in the ELF file is case sensitive.

`Key-word Parameters`

`None`

In the following example external symbol `DefaultParameters`, which refers to a block of data containing run time default parameters, is defined as external, with name `DefParm` used internally.

```
//
        EXTERNAL    DefParm ,, "DefaultParameters"
                                    // DefParm is used internally only
                                    // Bind type is defaulted to 1
//
```

All possible values for ELF types, attributes, and bind for a section or an external/exported symbol are defined in the framework macro FrameWorkDef.mac.

7.2.14 LITTLEENDIAN Directive

Set byte encoding to little endian.

`Label` Not allowed.

`Name` `LITTLEENDIAN`.

`Pos. Parameters`

None

`Key-word Parameters`

None

An example on how to use `LITTLEENDIAN` is given in subsection describing directive `DEFENDIAN` on page 59.

7.2.15 MMAP Directive

Start a new Memory Map template.

`Label` Required.

`Name` `MMAP`.

`Pos. Parameters`

`None`

`Key-word Parameters`

`None`

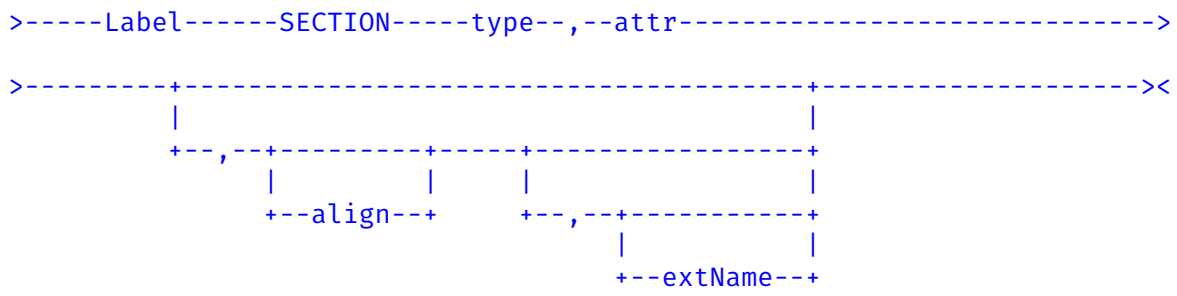
In this example a new MMAP is started. If a SECTION or another MMAP is active, it is terminated.

```
//
ParmBlock   MMAP                    // Define mapping of parameter block
//
Type        FWRD        0           // Type of request
RC          FWRD        0           // Return code
DataArea    DWRD        0[63]       // Data area
//
```

Initialization constants can be set to any valid value but are ignored.

7.2.16 SECTION Directive

Start a new Section.



`Label` Required.

`Name` `SECTION`.

`Pos. Parameters`

`type` Required, expression must resolve to `INTEGER ABSOLUTE` and specify the ELF section type.

`attr` Required, expression must resolve to `INTEGER ABSOLUTE` and specify the ELF section attribute type.

`align` Optional, expression, when specified, must resolve to `INTEGER ABSOLUTE` and specifies the section alignment. When not specified it is set to `1`. The final alignment value for the section is set to the largest values between the value specified or defaulted in the `SECTION` directive and the largest alignment value for each data or machine instruction component specified inside the section.

`extName` Optional, expression, when specified, must resolve to `STRING` and specifies the external name for the section, that is the name stored in the ELF file. When not specified the label name will be used. In this case, even though the label defines a symbol that is case insensitive, the external name, as stored in the ELF file is case sensitive.

`Key-word Parameters`

`None`

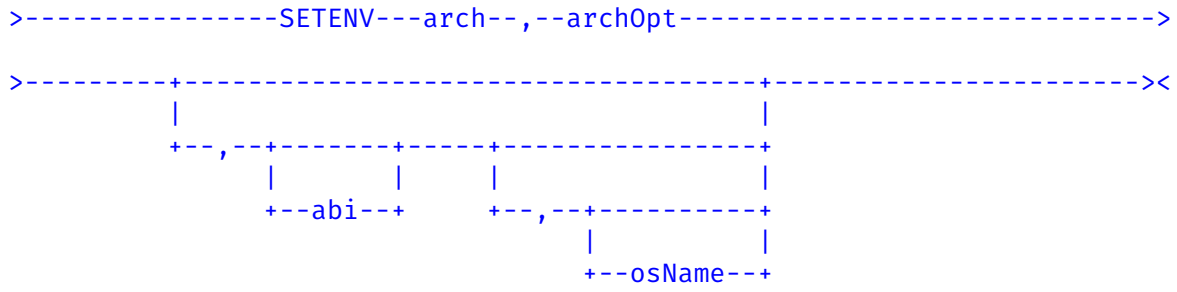
In the following example, directive `SECTION` is used to specify a new section.

```
//
Code        SECTION         1,      /> Section data is inside ELF file
                            2+4,    /> Executable code
                            16,     /> Align on quad word
                            ".text" // Section name used in ELF file
//
```

All possible values for ELF types, attributes, and bind for a section or an external/exported symbol are defined in the framework macro FrameWorkDef.mac.

7.2.17 SETENV Directive

Informs DVASM™ what computer architecture, architectural options, ABI and OS are to be used for this input file. Directive `SETENV` must be used only once in the input file before any `SECTION` or `MMAP` directive has been specified.



`Label`	Not allowed.
`Name`	`SETENV`.
`Pos. Parameters`	
`arch`	Required, expression must resolve to `STRING` which contains the computer architecture being used.
`archOpt`	Required, expression must resolve to `STRING` which contains the architecture options being used.
`abi`	Optional, expression, when specified, must resolve to `STRING` which contains the ABI being used.
`osName`	Optional, expression, when specified, must resolve to `STRING` which contains the OS being used. This parameter is not used by DVASM™ framework, but is passed, on request, to macros, so that they can be OS dependent.
`Key-word Parameters`	
`None`	

In the following example, directive SETENV is used to specify RISC-V architecture for Linux.

```
//
    SETENV      "RISCV",                            /> RISC-V architecture
                "RV64I:a,c,d,m,n,zicsr,zifencei",   /> RISC-V options
                "LP64D",                            /> 64 bits ABI
                "linux"                             // Linux OS
//
```

When the SETENV directive is encountered, DVASM™ will search for the Java module supporting the architecture. If found it will insert the macros that come with the architecture into the macro search path between the framework macros, which are last, and the application macros.

The first parameter string is converted to upper case and used to search for the Java module supporting the architecture. If misspelled the search will fail.

When assembling multiple input files with one command, the parameter used by the SETENV directive must be **identical in each input file**.

To avoid errors and simplify updates to SETENV parameters, the directive should be embedded in an application macro used by all input files of the project.

7.2.18 SETFILE Directive

Direct DVASM™ to generate a record in the ELF output file containing the input file name.

```
>----------------SETFILE-------------------------------------------><
```

Label	Not allowed.
Name	SETFILE.
Pos. Parameters	
None	
Key-word Parameters	
None	

In this example the SETFILE directive is used to tell DVASM™ to insert a source file name record into the generated ELF file.

```
//
        SETFILE             // Store source file record inside the ELF file
//
```

The SETFILE directive can be used only once, within an input file.

Chapter 8

Macro Processor

Assembly macros are programs written by the coder and executed by the assembler when their name is encountered in place of opcodes. The purpose of macros is code reuse, just like in-line C functions. For example if a program needs to repetitively load a number from memory into a register, add it to another register, and store the result back to memory, it is very convenient to use a macro and achieve in one line what it would take three lines otherwise. This results in shorter and more readable source code, without any loss of performance.

Macros can be short and simple or very complex and generate a lot of code. For example, the DVASM™ support macros for the RISC-V architecture, include a macro that implements heap sort, and a set of macros that implement, insert, delete and search of AVL trees.

When DVASM™ reads an opcode, it checks if a macro with that name exists in its macro search path. If one exists, it pass control to the macro processor which read the macro parameters, and execute the macro in a sandbox.

It is important to remember that all DVASM™ macros are executed **before** the assembler resolve symbols. For this reason information such as symbols' values, lengths, etc. are not available when the macro is being executed. In its essence a macro is a program that process input parameters and generates output statements, i.e. strings.

DVASM™ macros are coded in JavaScript®, and are executed within a JavaScript® sandbox. DVASM™ specific functions are available to the macro JavaScript® code to interact with DVASM™ and direct DVASM™ to generate code.

A DVASM™ macro is composed of two parts: A header and the actual `JavaScript®` program. Each DVASM™ macro must have a header. The header is used to name the macro and its parameters, if any, both positional and key-word parameters.

8.1 A simple example

A simple example, for the RISC-V architecture, is the best way to introduce the reader on how to write DVASM™ macros. Macro `ADDMEM` generates code to load a content of a word, add to it a value and store it back. To be used macro `ADDMEM` must be stored in a file named `ADDMEM.mac`, case insensitive, and store in a directory which is in the search path DVASM™ uses to search for macros.

The source code of macro `ADDMEM` is:

```
//MACRO     ADDMEM      MemLoc,    Value,  WReg,    WType=!""
//
    var ld          // Load machine instruction
    var st          // Store maachine instruction

    //
    //  Check if word type is specified;
    //
    if (WType !== "")                   // Validate WType
        if ([ "H", "W", "D" ].indexOf(WType.toUpperCase()) < 0)
            return DVASM.putError(
                "Value of key-word parameter [" + WType + "] is invlid.  " +
                "Possible values are: 'H', 'W', 'D'");
    else {
        eval(DVASM.getEnv()[4]);        // Get env Values
        WType= abiWID;                  // Override WType
    }

    ld= "L" + WType;                    // Set load MI
    st= "S" + WType;                    // Set store MI

    //
    //  Generate code
    //
\#Label     #ld     #WReg, #MemLoc      // Load memory content
\           ADDI    #WReg, #Value       // Add value
\           #st     #WReg, #MemLoc      // Store result back
```

8.1.1 The Header

In the above example, the first line is the macro header.

The line is a JavaScript® line comment followed by the word MACRO with no interleaving blanks. Parameters both positional and/or key-word follow.

In this case there are three positional parameters MemLoc, Value and WReg. Parameter MemLoc is the memory address of the word to be added to, parameter Value is the value being added, and parameter WReg is a working register.

When the macro is executed the header line is changed by DVASM™ so that JavaScript® variables MemLoc, Value and WReg are each initialized to the values coded as parameters when invoking the macro. Also if the macro is invoked with a label, JavaScript® variable Label is set to its value, otherwise it is set to null.

The key-word parameter WType specifies the type of word being addressed for the addition and it is defaulted to a null string. It can be set to either 'H' for half word, 'W' for full word, and 'D' for double word. When it is defaulted to null the macro uses the architectural word length, either 32 or 64 bits, as a target.

8.1.2 The Code

The code in the above example is divided into two parts. In the first the correct machine instruction opcode for load and store are set in JavaScript® variables ld and st depending on the value of WType.

If key-word parameter WType is set, it is validated.

If WType is not set, the code execute the line:

```
eval(DVASM.getEnv()[4]);
```

This line uses function DVASM.getEnv() to get an array of string each containing environment information. The 5th and last string in the array is an architecture/ABI specific string containing semicolon separated JavaScript® statements. Each statement initialize a variable, each starting with prefix "abi", to an architecture/ABI specific value. This string is than executed with the JavaScript® eval function so that all variable in the above string are assigned. Variable abiWID in particular, contains either the value 'W', when the RISC-V architecture is 32 bits, and 'D' when it is 64 bits. In this case, it

is used by the macro, to override key-word parameter `WType` to the word address size used by the architecture.

The last two lines before code generation section set `JavaScript®` variables `ld` and `st` to the correct machine instruction opcodes.

The last three lines of macro `ADDMEM` generate the actual code. Each line starts with the backslash character in position 1, and each string starting with the pound character is substituted with the value in the corresponding `JavaScript®` variable.

8.2 The Header

In this section we are going to list the complete specification of a macro header.

8.2.1 Header Format

The macro header is composed of one or more lines at the start of the macro. Each line must start with a JavaScript® line comment // starting in position one. The first line must start with the string //MACRO followed by one or more blanks. The macro name and parameters follows in the first line and/or the following lines.

When using multiple lines, each line except the last must terminate with the continuation comment /> optionally followed by a comment string. The last line can include a comment at the end after another line comment //.

Here is an example of a one line macro header:

```
//MACRO    TestMacro  Parm1, Parm2{integer}, keyParm{integer}[2]= [13, 15]
```

Here is the same macro coded on multiple lines with comments:

```
//MACRO     TestMacro                       /> Macro name
//          Parm1,                          /> First positional parameter
//          Parm2{integer},                 /> Second positional parameter
//          keyParm{integer}[2]= [13, 15]   // Key word parameter
```

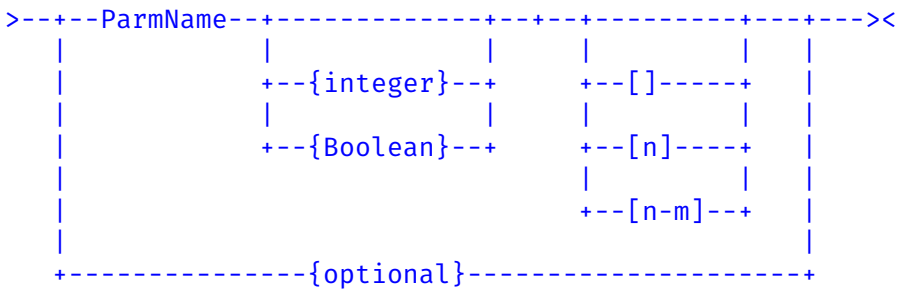
8.2.2 Header Parameters

Two type of parameters can be defined in a macro header: positional parameters and key-word parameters. All parameters are comma separated. Blanks are allowed.

Positional Parameters

Positional parameter are matched with positional arguments used when invoking the macro by their position. Each positional parameter has a name that must be a valid JavaScript® variable. JavaScript® variable Label must not be used since it is reserved to pass the name of the label used, if any, in the statement invoking the

macro. Each argument value at invocation is assigned by DVASM™ macro processor to a `JavaScript®` variable with name equal to the matched parameter name. This is the coding diagram for positional parameter definitions:



In the above diagram the words in curly brackets indicates if the parameter must be of type integer or Boolean. The default type is string. When integer is specified a valid signed integer must be used as corresponding argument at invocation time. When Boolean is specified, any of the four the case insensitive strings `yes`, `true`, `no`, and `false` must be used as corresponding argument at invocation time. Strings `yes` and `true` are mapped to a `JavaScript®` value of true, while the remaining two are mapped to false.

Following the type specification there are three type of list specification. The first `[]` indicates a list of any length, the second `[n]` indicates a list of fixed length `n` and the third `[n,m]` indicates a list with a minimum length of `n` and a maximum length of `m`. In `JavaScript®` a list variable is a variable that contains a list of values that can be accessed using an index (i.e. same as a `C` language array). When no list specification is used, the parameter is defaulted to single value and not a list.

No blanks are allowed in between the parameter name, the attribute and the list specification.

A dummy positional parameter specification, coded as `optional`, can be inserted in the list of positional parameters to separate the parameters that are required from the ones that are optional. When an optional parameters are not used, at macro invocation, the corresponding argument is set to `null`. `{optional}` can be used only once and must be followed, in the parameter list, by at least one positional parameter. This example shows how to use `{optional}`:

```
//MACRO     MacroName   Parm1{integer}, {optional}, Parm2
```

In the above case, when invoking the macro, `Parm1` must be matched by an argument, while `Parm2` if not matched is defaulted to `null`.

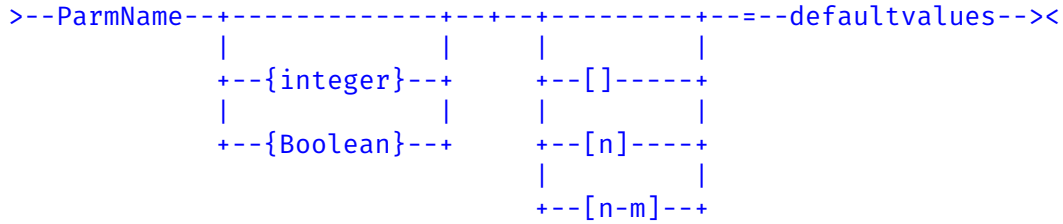
Key-word Parameters

Key-word parameters differ from positional parameters as follows:

- An argument is matched, at invoking time, by name and not by position;
- A key-word parameter must be followed by default values used when a matching argument is not used. Default values are verified to follow the rules established by the parameter attribute and list specifications;
- Key-word parameters can be placed anywhere in the parameter list without any effect on the relative position of positional parameters.

Otherwise, key-word parameters and positional share the same attribute and list specification.

This is the coding diagram for key-word parameters:



No blanks are allowed in between the parameter name, the attribute, the list specification and the equal sign. After the equal sign blanks are allowed.

More about Attributes and Lists

The reason the coder is given the option to specify attribute and list specifications in the macro header is that it minimize the coder need to verify an argument correctness. For example if a parameter has attribute Boolean, the coder can rely on the fact that the DVASM™ macro processor has already verified that the `JavaScript`® variable containing the value of the matching argument is a valid Boolean variable. This is a time saver since verifying variable type and value in `JavaScript`® can be time consuming.

8.3 Invoking a Macro

A macro is invoked just like an opcode or a directive. The only difference is the way parameters are coded. The general rules are as follows:

- Al positional parameters defined in the macro header must be specified, except for those parameter that follows the `{optional}` clause in the header;
- Key word parameter are optional;
- Parameter type as specified in the macro header must be followed.
- When the parameter type is string, if a string is passed enclosed in single or double quotes either quotes are passed as part of the parameter string;
- If the parameter type is string, and there is a need to enclose it in quotes, but the quotes must not be part of the parameter value passed, use double quote where the first quote is preceded by the character `'!'`. For example, if parameter `!"This is a test"`, the value passed will be `This is a test`. A null string (i.e. zero length) can be passed as `!""`.
- If the parameter type is Boolean these four case insensitive values can be used: `true`, `false`, `yes`, and `no`;
- If the parameter is defined as a list of any type in the macro header, a comma separated list enclose in square brackets must be passed. If the size of the list is specified in the header, it must be matched by the number of values passed in the list, otherwise, any number is acceptable including the case of an empty list.

For example for a macro defined with header:

```
//MACRO   TSTMCR   p1{Boolean}, p2[], p3{integer}[3]
```

can be invoked as follows:

```
TSTMCR     true,                                              />
           [test, "With double quotes",                       />
               !"Transparent quotes to keep      spacing"],   />
           [15, 31, -56]
```

A special short hand can b e used when a list of string must be passed, whose values use the same prefix followed by an integer of increasing value. In this case the parameter can be specified as follows:

```
[PrefixStartingInteger-EndingInteger]
```

For example macro `COMP` is defined with header:

```
//MACRO   COMP   p1, p2, p3, WReg[5]
```

The parameter `WReg` is used to pass working registers, in this case 5 of them. These are examples on how macro `COMP` can be invoked using the short hand notation and the corresponding regular notation

```
COMP    s3, s5, s2, [t1-5]                  // Short hand
COMP    s3, s5, s2, [t1, t2, t3, t4, t5]    // Regular

COMP    s3, s5, s2, [a1-2, t1-3]            // Short hand
COMP    s3, s5, s2, [a1, a2, t1, t2, t3]    // Regualar

COMP    s3, s5, s2, [a1, t2-5]              // Short hand
COMP    s3, s5, s2, [a1, t2, t3, t4, t5]    // Regular
```

8.4 Execution Environment

DVASM™ macros are written in regular `JavaScript`® language, the same language normally used for WEB applications. The macro `JavaScript`® code is executed in a sandbox by GrallVM, and it can communicate exclusively with DVASM™ .

When DVASM™ decide that the opcode encountered is a macro, it reads the macro code and modifies it as follows:

- Substitute the first line with a sequence of semicolon separated statement that do the following:
  - Define a function statement to enclose the code in a function;
  - Execute a statement that initialize all parameter variables to the correct values.
- Convert every code generating line, starting with a backslash in position one, to a `DVASM.formatLine` or `DVASM.formatLineLeft` function call. See page 92 for more information;
- Add, at the bottom of the code, a new line with a a right curly bracket to end the function, followed by call to the same function.

DVASM™ than asks GraalVM® to compile the macro code. GraalVM® , in turn, checks if it already has a compiled copy to reuse, and if not, it compiles it. Than executions starts.

8.4.1 DVASM™ Macro Functions

Each DVASM™ macro communicates with DVASM™ via a set of predefined functions. These function allows a macro to query the environment, generate code and access some selected Java functions. All these function are implemented in Java and not in `JavaScript®` . GraalVM® converts parameters passed by a function call within the macro to the correct object type in Java whenever possible, and it is important that parameter passed match the definition given for each function.

Function **`DVASM.insertLine`** is defined as follows:

```
public      void    DVASM.insertLine(String line)
```

This function generates a single line of code which is the string passed as an argument. The line is left unchanged, with no attempt to indent it, or format it in any way.

Function **DVASM.formatLine** is defined as follows:

```
public      void    DVASM.formatLine(String lbl, String opCode,
                                     String parm, String comm)
```

This function generates a single, formatted line of code which is formed using the four strings passed as arguments:

- Argument `lbl` is the line label. If none is is being passed a zero length string must be used;
- Argument `opCode` is the line opcode. If none is is being passed a zero length string must be used;
- Argument `parm` is a string with all the opcode parameters. If none is is being passed a zero length string must be used;
- Argument `comm` is a string containing a comment. If none is is being passed a zero length string must be used.

The line being generated is formatted by putting the label at offset 0, and putting the remaining three arguments at predefined tab location plus the current indentation. Function **DVASM.formatLineLeft** is defined as follows:

```
public      void    DVASM.formatLineLeft(String lbl, String opCode,
                                         String parm, String comm)
```

This function work the same as `DVASM.formatLine`. The only difference is that it uses the current indentation decreased by one.

Function **DVASM.setIndentIn** is defined as follows:

```
public      void    DVASM.setIndentIn()
```

This function increase the indentation setting by one.

Function **DVASM.setIndentOut** is defined as follows:

```
public      void    DVASM.setIndentOut()
```

This function decrease the indentation setting by one.

Function **DVASM.putInfo** is defined as follows:

```
public      void    DVASM.putInfo(String msg)
```

This function add an informational message associated with the statement that invoked the macro being executed. The message will show in the output listing.

Function **DVASM.putWarning** is defined as follows:

```
public      void    DVASM.putWarning(String msg)
```

This function add a warning message associated with the statement that invoked the macro being executed. The message will show in the output listing.

Function **DVASM.putError** is defined as follows:

```
public      void    DVASM.putError(String msg)
```

This function add an error message associated with the statement that invoked the macro being executed. The message will show in the output listing. Issuing an error message will stop DVASM™ from executing the following phases of the assembly process, such as symbols' resolution and code generation.

Function **`DVASM.putJSGlobal`** is defined as follows:

```
public      void    DVASM.putJSGlobal(String name, Object object)
```

This function allows the executing macro to store any JavaScript object as a tag and value pair for later retrieval and use by another macro.

Function **`DVASM.getJSGlobal`** is defined as follows:

```
public      Object  DVASM.getJSGlobal(String name)
```

This function retrieve JavaScript object that has been previously stored using the `putJSGlobal` function.

Function **`DVASM.dropJSGlobal`** is defined as follows:

```
public      Object  DVASM.dropJSGlobal(String name)
```

This function erase a JavaScript object that has been previously stored using the `putJSGlobal` function.

Functions `putJSGlobal`, `getJSGlobal` and `dropJSGlobal` are intended to be used to save states needed to be shared by multiple macros, such as control flow macros `IF`, `ELSE`, WHILE, etc. Another possible use is to cache results from CPU intense calculations, such as compiling regular expressions.

Function **DVASM.getNewLabel** is defined as follows:

```
public      String  DVASM.getNewLabel()
```

This function allows the executing macro to get a unique symbol to be used when generating code. The symbol returned is a string starting with `"Asm"` followed by a six decimal digit. Everytime this function is called, the 6 digit number is increased by one, in order to generate a unique symbol at each call. TO assure uniqueness the coder must refrain from ever using symbols starting with `"Asm"` followed by six decimal digits.

Function **DVASM.getEnv** is defined as follows:

```
public      String[]    DVASM.getEnv()
```

This function returns a string array of size 5. The first 4 elements in the array contain the 4 parameters passed to the `SETENV` directive. The fifth element is architecture dependent and contains a set of `JavaScript`® statements, semicolon separated, that should be executed with the `JavaScript`® function `eval`. Upon execution, several `JavaScript`® variable will be initialized to values that further describe specific attributes of the architecture being used, such as the number and size of available registers. For the `JavaScript`® variable names used, and what values they contain, the specific architecture manual should be consulted.

Function **DVASM.getEvalParm** is defined as follows:

```
public      String  DVASM.getEvalParm()
```

This function returns a string which contains several JavaScript® statements, semicolon separated. The string is executed using the JavaScript® `eval` function to initialize the JavaScript® macro parameter variables to the arguments specify by the macro invocation. This function is inserted by DVASM™ in the first line of the macro so that parameter JavaScript® variables have the correct values when execution is started. This function can also be used by the coder for debugging purposes, by printing the returned string, when he/she thinks that parameter values are not what they are expected to be.

Function **DVASM.getCode** is defined as follows:

```
public      String  DVASM.getCode()
```

This function returns a string which contains the whole macro JavaScript® code after DVASM™ has pre-processed it, just before execution. The purpose of this function is for debugging and the string returned by it should be printed for inspection by the coder.

8.4.2 Generating Code

There are two ways to generate code in a DVASM™ macro. The first is to directly use function `insertLine`, `formatLine` and `formatLineLeft`. The second is to use template lines which start with `'\'` character and contain `JavaScript®` variable prefixed with the `'#'`. for example the following macro code:

```
      var reg1= "s5";
      var reg2= "a1",
      var opCode= "MV";
      var lbl= getNewLabel();

  \\#lbl      #opCode     #reg1, #reg2        // Copy register #reg2 to #reg1
```

will be converted by DVASM™ during pre-processing to the following `JavaScript®` code:

```
      var reg1= "s5";
      var reg2= "a1",
      var opCode= "MV";

      formatLine(lbl, opCode, reg1 + ", " + reg2,
          "// Copy register " + reg2 + " to " + reg1);
```

The code generated will look like:

```
      Asm000001    MV      s5, a1      // Copy register a1 to s5
```

The following should be noted:

- Template line are converted to code using formatting and, as a consequence, indented to the current indentation;
- If formatting and indentation is not wanted, the line should be generated by using the `insertLine` function directly;
- `JavaScript®` variable substitution applies everywhere in the line, including in the comment section;
- `JavaScript®` variable substitution applies only to non array `JavaScript®` variables. Array variable will not be recognized. If an array variable is specify such as `#data[3]`, it will be substituted as `data + "[3]"`, that is, it expects non-array variable `data` to exists and assumes that `[3]` following variable `data` is just a string constant;

- When experiencing problem with template lines code generation, function `getCode` should be used to check the code generated by DVASM™ during macro pre-processing.

8.4.3 Concatenation of **JavaScript**® variables in template lines

When using template lines to generate code, it is sometimes necessary to concatenate the value in a `JavaScript`® variable with a constant. For example give the `JavaScript`® variable `abc` whose value must be concatenated with constant `xyz`, if the template line contains `#abczyx` the `JavaScript`® interpreter will look for the value in variable `abcxyz` instead of variable `abc`. For such cases the dot character '.' is used to indicate concatenation, and the correct coding will be `#abc.xyz`. If the value of the variable `abc` is `R1` the resulting generated string will be `R1xyz`. If the constant being concatenated is `.xyz`, the correct coding is `abc..xyz` and the result will be `R1.xyz`.

Chapter 9

Output Listing

DVASM™ generate output listing in `HTML` format. The listing can be viewed using any web browser, by entering the full path name of the listing file prefixed with the string `file://`.

The reason for using the `HTLM` format is that when using a browser the data in the listing can be shown in a relatively concise form, but that any detail needed by the programmer is only one click away.

There are several sections in the listing but certain rules apply to all of them.

- When the data length in a field is longer than the width allocated for the same field, the data is truncated. However, when clicking on the field, the data is expanded to its full length, and the following data is shifted to the right. Clicking again will restore that data to its truncated form. When truncated data can be expanded, its color changes when the mouse cursor is hovering over the data.
- Some lines in the listing can be expanded vertically. These line have a `[+]` at the very beginning of the line, and when clicked on it, the line is expanded vertically to show additional information, and the `[+]` field change to `[-]`. When clicking on the `[-]` field, the expanded lines disappear and the clicked field change back to `[+]`. As an alternative to clicking the `[+]`, `[-]` fields, **the users can right click the line anywhere**.

9.1 Header

At the very top of the listing file there is a header composed of several clickable fields:

Reload	Reload the listing file being browsed. As an alternative most browsers will do the same when pushing the F5 key on the keyboard. This function can be used by developers to view a new listing after assembling the same source file.
Expand All	Ex[and recursively all code generated by macros. To undo this function, simply reload the listing file.
Statement Listing	Scroll vertically to statement listing.
Section Listing	Scroll vertically to section listing.
Symbol Listing	Scroll vertically to symbol listing.
Relocation Listing	Scroll vertically to relocation listing.
DVASM Manual	Start a new browser tab to display this manual from the DVASM™ WEB site.
Dark Background	This field is a dark square, and when selected, it will display the listing with a dark background (the default).
Light Background	This field is a light square, and when selected, it will display the listing with a light background.
Print	Print the listing file exactly as it appears at the moment of printing.

9.2 General Information Section

The general information section display the input file name, assembly date and time, and a log with the sequence of steps the assembler took to generate code.

9.3 Source Code Listing

This section shows the source code and corresponding code generated by each opcode and its offset.

When clicking on an opcode, the line is expanded to show a detail description of the opcode and its fields, such as registers and immediates. When clicking on the opcode again, the details disappear. If the opcode is a macro, the location of the macro is shown.

When the opcode is a macro, the line starts with a `[+]`. This field can be selected to show the macro expansion. When the macro code expansion itself contains one or more macros, each macro can be expanded recursively.

When an opcode is the result of multiple nested macros, the list of macro names, from the outer to the inner, is shown on the right, following the statement number, with the macro names separated by the colon character '`:`'.

9.4 Sections

This section list all the code section that have been generated with attributes that have been stored in the ELF file.

9.5 Symbols

This section list all the symbols that have been that have been used in the source code, or generated by macros. For each symbol all attributes and values are shown, with the name of the owning code section when applicable.

Each line can be expanded vertically to show where in the code the symbol was defined, and where it was used.

9.6 Relocations

The relocation section show all symbols that are relocatables, with owning section, external name, relocation ID and addend value.

Appendix A

Summary of Case Sensitive Rules

This is a short summary of the rules about what is case sensitive and what is not.

Everything in the source code is **not** case sensitive, and it is internally converted to upper case. This includes symbols, opcode, directives, macro names, key-word parameter keys, and symbol qualifiers. For symbols that are external or are exported and are not all upper case, there is always an option to specify and external name which is case sensitive, when using the `EXTERNAL`, `EXPORT` and `COMMON` directives.

Everything inside a macro is case sensitive since `JavaScript`® is a case sensitive language, with the exception of the macro name in the macro header following the string `//MACRO`.

Special care should be paid with macro names. Macro names are mapped to the name of the file containing them by appending file extension `".mac"` to the macro name. In a file system with file names that are not case sensitive (e.g. MS Windows) there are no problem.

However, when using a file system with file names that are case sensitive (e.g. Linux, UNIX, etc.) two file names, spelled the same, but with different cases for each character, will map into the same macro name. For example consider the case of two macros with file names `Macro.get.mac` and `Macro.GET.mac` and residing in the same directory. When DVASM™ scan the directory, and compares the names in case insensitive mode, it will detect a duplicate and, it will discard the second one found.

Another area that needs to be clarified is the mapping of a macro key-word parameter into

the corresponding argument in the macro header. If, for example, a key-word parameter is defined in a macro header as `WReg=`, the source code invoking the macro can use any case combination of `WReg=` such as `wreg=` or `WREG=`. However, inside the macro, only the `JavaScript®` variable `WReg` will contain the value passed as a parameter, or its default value.