



easm Language Reference

- 1 IMPORTS.....3**
 - 1.1 OVERVIEW3
 - 1.2 DECLARING AN IMPORT3
 - Example.....3
 - 1.3 CALLING CONVENTION3
 - Example.....4
- 2. CONSTANTS5**
 - 2.1 OVERVIEW5
 - 2.2 DECLARING A CONSTANT5
 - Constant Declaration Examples5
- 3 STRUCTURE DECLARATIONS.....6**
 - 3.1 OVERVIEW6
 - 3.2 DEFINING A STRUCTURE.....6
 - Example.....6
 - 3.3 USING STRUCTURES7
 - Example.....7
- 4 DATA DECLARATIONS8**
 - 4.1 OVERVIEW8
 - 4.2 DECLARING A VARIABLE.....8
 - Variable Declaration Syntax.....8
 - Variable Declaration Examples8
 - 4.3 ACCESSING A VARIABLE.....9
 - Access Examples9
 - 4.4 VARIABLE STORAGE.....9
 - Examples10
- 5 FUNCTION DECLARATIONS.....11**
 - 5.1 OVERVIEW11
 - 5.2 DECLARING A FUNCTION11
 - Function Declaration Syntax11
 - Type Information11
 - Function Termination.....12
 - The return Instruction.....12
 - Function Examples12
 - 5.3 LOW LEVEL SEMANTICS.....12
 - Common Executable Layout12
 - easm Executable Layout12
 - Example Code Section Layout.....13
- 6 LANGUAGE FACILITIES14**
 - 6.1 OVERVIEW14
 - 6.2 THE STACK14
 - Stack Operations14
 - easm's Stack Access14
 - Examples15
 - 6.3 ASSIGNMENT16

Pointer Dereferencing.....	16
Examples	16
6.4 ITERATION	17
Example.....	17
6.5 COMPARISON.....	18
The Compare/If Instructions	18
Examples	18
Available Expressions	18
6.6 INVOCATION	19
Literal Arguments.....	19
String Arguments	19
Function Pointers	20
Function Pointer Example	20
6.7 ARITHMETIC OPERATIONS	21
Addition & Subtraction	21
Examples	21
Multiplication.....	21
Examples	21
Division.....	21
Examples	21
Increment And Decrement	22
Examples	22
6.8 BITWISE OPERATIONS	23
The Bitwise Operators	23
Examples	23
Constraints	24
6.9 INCLUDE LIBRARIES.....	24
Example.....	24



1 Imports

1.1 Overview

An import can be described as a function whose declaration has been pre-defined in a foreign module - commonly in a dynamic link library (DLL). At runtime, the executable loader resolves any imported functions and makes them available for invocation.

The **from** and **import** instructions allow an import to be declared in the specified imports section of an easm source code file. Once declared, an import function may be invoked either from within the main code section or from within a user-defined function via the **call** instruction.

1.2 Declaring an Import

Import functions can be declared using the **from** and **import** directives respectively. Following the **from** directive, the library name must be specified and following the **import** directive the symbol name must be specified.

Example

```
section imports

    from user32.dll import MessageBoxA
    from kernel32.dll import ExitProcess
```

The example above shows how two functions from different dynamic link libraries can be imported into an easm executable. The first import takes the function **MessageBoxA()** from **user32.dll** and makes it available for calling. Likewise, the second import takes the function **ExitProcess()** from **kernel32.dll**.

Subsequent to the functions being declared in the import section, they are then available to the rest of the program and can be invoked from either the main code section or a user-defined function.

1.3 Calling Convention

In the example above, the two functions are assumed to be **stdcall** functions. That is, they are assumed to be using the standard calling convention. In some cases however, a function will need to be imported which is not **stdcall**, such as the **printf()** function from the C standard library.

The **printf()** function uses the **cdecl** calling convention, which differs to **stdcall** in that stack clean-up is not delegated to the function itself, but the caller of the function.

To elaborate let's take the example of two functions **a** and **b**. The former is a **stdcall** function and the latter is a **cdecl** function. If we were to call function **a** and pass 2 parameters on the stack to this function, we can assume that the function itself will clear up these 2 parameters from the stack before returning. In contrast, if we were to call function **b**, passing it 2 parameters, these parameters



would **not** be removed from the stack by the function and would remain there for the caller of the function to remove.

For this reason, **easm** needs to know if a particular import function is a **cdecl** function so that it can generate the necessary instructions to remove any parameters from the stack when the function returns. This can be achieved through the use of the **using** directive, which follows the import declaration and has one mandatory operand which specifies the new calling convention.

Example

```
section imports
```

```
    from msvcrt.dll import printf using cdecl
    from kernel32.dll import ExitProcess
```

Here, **printf()** has been imported from msvcrt.dll as a **cdecl** function and **ExitProcess()** has been imported from kernel32.dll as a **stdcall** function. Necessary code for removing the parameters from the stack will be generated by the **easm** assembler for any calls to **printf()** but not for calls to **ExitProcess()** (since **ExitProcess()** is **stdcall**).



2. Constants

2.1 Overview

Constants exist to allow program code to become more readable and maintainable. Typically arbitrary values are replaced by more meaningful aliases (or constants). These are more correctly referred to as symbolic constants and literally replace each instance of their mapping in a macro style.

Commonly, constants are used where a value itself is not meaningful. A good example is **PI** being a symbolic constant for the value 3.14159265359. Instead of instances of the number 3.14159265359 in source code, instances of **PI** are much easier to read.

Another important use of symbolic constants relates to maintainability, such that using a symbolic constant for many instances of a single value means that a change to the particular value only requires a change being made to the value of the constant, instead of trawling through the entire source code file looking for values that need replacing.

2.2 Declaring a Constant

Constants are not mandatory and should only be used when they are necessary. Constants must be declared in the constants section of an easm source code file and should have unique identifiers.

A constant declaration should be of the form:

```
const <identifier> = <value>
```

Commonly, a constant identifier is upper-case to symbolise the fact that it is a constant - although this is a trend rather than a rule.

Constant Declaration Examples

```
const HELLO_WORLD = "Hello World!\n"  
const HANDLE = dword  
const NULL = 00h
```



3 Structure Declarations

3.1 Overview

Structures are a form of composite data type allowing the grouping of related data declarations under a common name. A structure might be declared to hold *Client Details* and be composed of 3 single entities, *Name*, *Address* and *Telephone Number*.

3.2 Defining A Structure

Structures must be defined in the **structures** section of an easm source code file. A structure definition takes the following form:

```
structure <identifier>

    <data-type> <identifier>
    string <identifier> <size>
    ...
```

The structure identifier must be unique and *fields* declared within the structure must be unique within the scope of the enclosing structure. That is, a structure cannot contain two *fields* that have the same identifier.

Example

```
section structures

    structure RECT

        dword left
        dword right
        dword top
        dword bottom

    structure POINT

        dword x
        dword y
```

The example above defines two structures named **RECT** and **POINT**. **RECT** contains four member *fields* named *left*, *right*, *top* and *bottom* respectively. **POINT** contains two member *fields* named *x* and *y* respectively.

Subsequent to their definitions, RECT and POINT now exist as composite data types within the application that defines them. In the data section of the source code file, a declaration of type RECT or POINT may now be made.



3.3 Using Structures

Once a structure has been defined, a named instance of the structure may be declared just as normal variables are declared (in the data section). The declaration of a structure instance should be of the following form:

```
struct <type> <identifier>
```

The keyword **struct** should be followed by the name of a pre-defined structure (from the structures section) and then by the desired identifier.

Example

section data

```
struct RECT rcSomeRect
```

The example above declares **rcSomeRect** as a named instance of the **RECT** structure. The net effect of this declaration is that the following four individual declarations are made.

```
dword rcSomeRect.left  
dword rcSomeRect.right  
dword rcSomeRect.top  
dword rcSomeRect.bottom
```

On a final note, when referring to the address of the structure, you must use the address of the first structure field. Using the example from above, the address of the structure is given by the following statement:

```
&rcSomeRect.left
```



4 Data Declarations

4.1 Overview

A data declaration in the context of easm refers to the creation of some amount space in the memory of the executable for a user-defined value to be stored in - more commonly referred to as a variable.

The amount of space allocated to the variable depends on the type of data that the variable indicates it will hold.

There are four different data types which may be used in easm: **byte** (8-bit) data, **word** (16-bit) data, **dword** (32-bit) data and **string** data. The first three data types [byte, word and dword] have a fixed size, but the last data type [string] will vary in size.

The size of a string variable can be specified in two ways. If the string is initialised, the size of the string is the sum of all the containing characters plus 1 (for a string termination character). If the string is an un-initialised string, then the size must be declared along with the variable declaration. In the latter case, a specified number of bytes are reserved in the memory of the executable for holding the variable's value.

Additionally, each byte of the un-initialised string is set to 0 to prevent any errors occurring from random data left behind.

4.2 Declaring a Variable

Variables must be declared in the specified data section of an easm source code file. A variable declaration will be of the following form:

Variable Declaration Syntax

```
<data-type> <identifier> [= value]  
string <identifier> <size>  
string <identifier> <= value>
```

Note: items that are enclosed in <> are mandatory, items that are enclosed in [] are optional.

Variable Declaration Examples

```
string szFirstMonth = "January"  
string szBuffer [255d]  
  
dword dwSomeDword = deadbeefh  
word wSomeWord = abcdh  
byte bSomeByte
```

The five examples above show the varying way in which variables can be declared in easm. The first two declarations create two string variables.



The first string variable is created and initialised with the string "January" and will also contain a terminating NULL character - resulting in a total size of 8 bytes. The second string variable is created with 255 bytes of data reserved for it to use. When declaring a string variable with a specific size, there is an upper limit of 4096 bytes (4KB) per string variable. If more than 4 KB's of storage is required, it would be better to allocate it dynamically and this is the reason for the limit.

The third dword data declaration creates a four-byte variable and initialises it to the value 0xdeadbeef. Likewise the fourth data declaration creates a two-byte variable and initialises it to the value 0xabcd. Finally, the last data declaration creates a one-byte variable which is by default initialised to the value 0x00.

Note: when creating a data declaration, the identifier must be unique or easm will generate a 'duplicate identifier' error.

4.3 Accessing a Variable

Variables can be accessed using their identifier as a reference to them in a particular instruction. To access the address of a particular variable, an ampersand (&) should precede the variable identifier.

Access Examples

```
section code
```

```
set bMyAge = 21d  
set eax = &bMyAge
```

The first statement shows how the variable bMyAge can be accessed in order to store the value 21. The second statement shows how the address of bMyAge can be retrieved and stored in the eax register.

4.4 Variable Storage

Variables declared in the data section of an easm source code file are ultimately stored in the executable data section of the final executable. The executable is memory-mapped into RAM when the application is loaded by the Operating System and hence the variables actually exist in virtual memory for the execution of the application.

Variables occupy physical space in the executable and as such, the more variables that are declared in an application, the larger the application will be. This means that if a particular easm application creates a 1024 byte (1KB) string variable, 1KB of data will be reserved in the executable and the final executable will be 1KB larger in size.

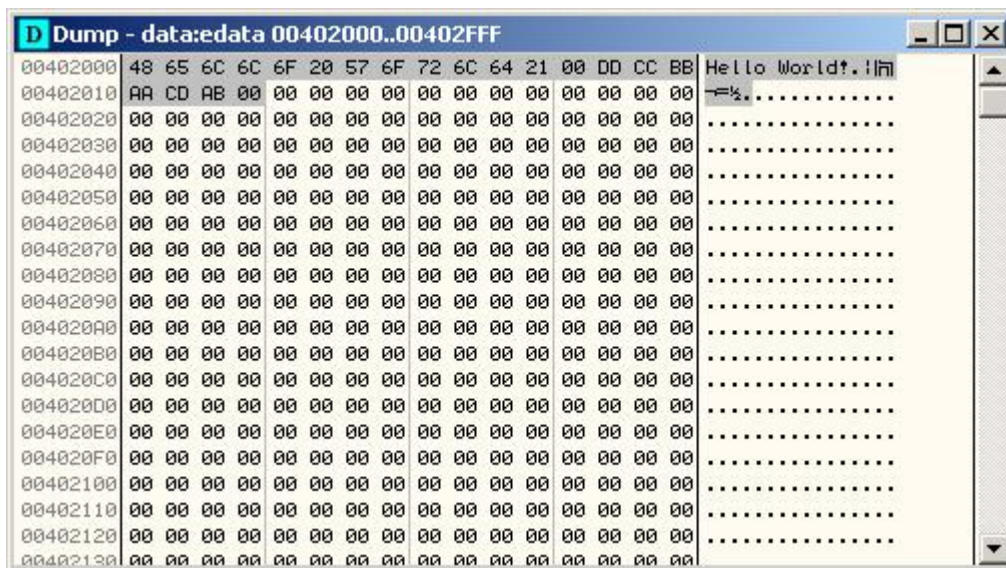
Of course, it is possible for an application to dynamically allocate memory by making import declarations for the malloc() and free() functions and then allocating memory from the heap via calls to these functions. This approach will not result in data being added to the executable.



Examples

The examples below show the mapping of four variable declarations to their physical contents in the final executable:

```
string szSomeString = "Hello World!"  
dword dwVar = aabbccddh  
word wVar = abcdh  
byte bVar
```





5 Function Declarations

5.1 Overview

The following provides an accurate description of what a function is:

"In computer science, a subroutine (function, procedure, or sub program) is a sequence of code which performs a specific task, as part of a larger program, and is grouped as one or more statement blocks; such code is sometimes collected into software libraries. Subroutines can be "called", thus allowing programs to access the subroutine repeatedly without the subroutine's code having been written more than once."

(Wikipedia.org, 2007)

Functions are supported in easm in the same way as the description above explains. Functions may be declared in the specified functions section of an easm source code file. The functions section should come before the primary code section in order to ensure that any functions are known of before they are invoked.

Functions may or may not return a value to the caller and once they are declared they are available for invocation from within a function (including from within itself: recursion) or from the main code section. A function can also be declared with or without parameters.

5.2 Declaring a Function

Functions must be declared in the specified functions section of an easm source code file. Function identifiers must be unique and a function may either specify a `dword` or `void` return type. A function declaration should be of the form:

Function Declaration Syntax

```
function <identifier> <return-type> ([<identifier>:<parameter-size>, ...])  
    ...  
end
```

Items enclosed in `<>` are mandatory syntax elements. Items enclosed in `[]` are optional syntax elements. `<identifier>` must be unique in both the context of the function identifier and the context of a parameter identifier. Function identifiers must be unique among other declared functions and parameter identifiers must be unique among other parameters declared within the same function.

Type Information

`<return-type>` specifies the type of data that is returned by the function and must either be `dword` or `void`, meaning returns a value or doesn't return a value respectively. `<parameter-size>` specifies the size of the parameter and must be `1`, `2` or `4` representing byte, word or dword sized parameters respectively.



Function Termination

A function declaration should be terminated with the end directive which causes the function to return control back to the point where it was called from. A function may also make use of the **return** keyword which effectively does the same as end but can also return a value.

The return Instruction

The **return** instruction can be used in two different ways depending on the data type being returned by the function. If the function is not returning a value (i.e. the return type is `void`) then the **return** instruction must be used without an operand. If the function is returning a value (i.e. the return type is `dword`) then the **return** instruction must also specify a value that is to be returned.

Function Examples

```
function dword ReturnParameter (Parameter:4)
    return Parameter
end

function void TerminateCurrentProcess ()
    call ExitProcess (00h)
    return
end
```

5.3 Low Level Semantics

There are two reasons why user-defined functions must be declared in a section which precedes the main code section. The first reason is so that when a call is made to a function, the function has already been encountered and is recognised. The second reason is due to the physical layout of the final executable.

Common Executable Layout

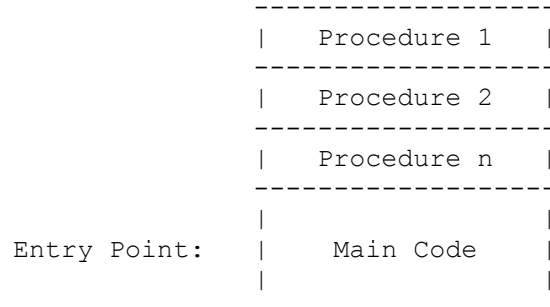
Commonly the code section is organised with any code belonging to a procedure occurring subsequent to the main code. This means that procedure code will only be reached if an explicit call is made to it. The entry point will always point to the start of the code section. The code section in an easm executable is organised slightly differently however.

easm Executable Layout

In an easm executable, code that belongs to a procedure occurs before the main code and the executable entry point is adjusted to point past any declared procedure code.



Example Code Section Layout





6 Language Facilities

6.1 Overview

The code section of an easm source code file is the effective entry point to an easm application. The first instruction present in the code section is the first instruction that will be executed when the application runs.

Likewise, the last instruction in an easm source code file should be the terminating instruction. Commonly, a call to **TerminateProcess()** is used to ensure that the primary thread of the application is gracefully terminated.

If the last instruction does not cause termination of the process, execution will continue into memory which is likely to lead to an access violation and certainly isn't the suggested method of ending an easm application.

6.2 The Stack

A stack is essentially a data type used for storing and retrieving data values in a *Last In First Out* (LIFO) manner. The stack plays an important role within an executable, being used for local variables, function parameters and general runtime storage.

Stack Operations

The two most useful stack operations are named push and pop and allow data to be put on and taken off the stack respectively. easm provides access to these two operations through the **push** and **pop** instructions which require additional operands to specify the exact type of push or pop.

easm's Stack Access

easm breaks down pushing and popping into more specific types of push and pop. In terms of pushing data onto the stack, easm considers three different types of push:

- Pushing a literal value onto the stack
- Pushing a register onto the stack
- Pushing a variable onto the stack

In each of these cases, an additional qualifier is needed to specify which type of push is required. The qualifiers **literal**, **register** and **variable** are used respectively. The qualifier must succeed the instruction and precede the instruction operand. Additionally, when the **literal** qualifier is being used, another qualifier is required which specifies the size of the literal value.

The following three types of literal may be pushed onto the stack:

- byte-sized literals
- word-sized literals
- dword-sized literals



In terms of popping data off the stack, the pop instruction requires one of two possible qualifiers - specifying the destination size of the pop operation.

Examples

```
push literal byte abh
push literal word abcdh
push literal dword abcdabcdh

push register eax
push register ebx
push register ecx
push register edx
push register ebp
push register esi
push register edi

push variable wSomeVariable
push variable dwSomeVariable
push variable &dwSomeVariable

pop register eax
pop register ebx
pop register ecx
pop register edx
pop register ebp
pop register esi
pop register edi

pop variable wSomeVariable
pop variable dwSomeVariable
```



6.3 Assignment

Assignment concerns the setting of data in a particular memory location. Assignment is achieved through the use of the **set** instruction. The **set** instruction should be of the form:

```
set <identifier> = <identifier>
```

easm does not support memory to memory assignment and the left-hand-side (LHS) of the set instruction must not be a value type or the assembler will generate an appropriate error.

Pointer Dereferencing

easm supports pointer dereferencing of any 32-bit quantity occurring on the right hand side of a **set** instruction. This includes 32-bit parameters, 32-bit user defined variables and registers.

easm uses the asterisk operator to indicate that the identifier should be de-referenced. Similarly, the address of a user defined variable or function can be retrieved through the use of the ampersand 'address-of' operator.

Examples

```
// Get the value of the variable and store in eax.
set eax = dwSomeVariable

// Get the de-referenced value of the variable and store in eax.
set eax = *dwSomeVariable

// Get the address of the variable and store in eax.
set eax = &dwSomeVariable

// Store the value of eax in the variable.
set dwVariable = eax

// Store the de-referenced value of eax in the variable.
set dwVariable = *eax

// Store the function address in the variable.
set dwAddress = &SomeFunction
```



6.4 Iteration

Iteration can be described as follows:

"Iteration in computing is the repetition of a process within a computer program. It can be used both as a general term, synonymous with repetition, and to describe a specific form of repetition with a mutable state."

(Wikipedia.org, 2007)

Iteration in a high level programming language is more commonly known as looping, for which there exists many different types (for, do, while). `easm` however, is a low level programming language and achieves iteration through conditional jumping based on testing and decrementing/incrementing a particular value.

Note: The example shown below makes use of conditional jumping and comparison in achieving a looping construct similar to that of a **for** loop in C. You may need to read the sections on conditional jumping and comparison before being able to fully understand the example.

Example

```
start:    set dwCounter = 100d
          // Perform loop processing.

          dwCounter -= 01h

          compare dwCounter with 00h
          if (==) jump done

          jump start

done:     // Loop has terminated.
```



6.5 Comparison

Comparison combined with conditional jumping is the construct that allows execution to follow different paths depending on the result of particular tests. Comparison in `easm` concerns two separate instructions. The **compare** instruction allows a test to be made and the **if** instruction allows execution to be directed based on the result of the previous comparison.

The Compare/If Instructions

The **compare** instruction allows two data elements to be compared, while the **if** instruction allows the thread of execution to branch to a specific location based on the result of the test. The **compare** and **if** instructions should be of the form:

```
compare <identifier> with <identifier>
if (expression) jump <label>
```

Examples

```
compare eax with dwSomeVariable
if (==) jump done

compare dwSomeVariable with &SomeFunction
if (!=) jump not_equal

compare edi with esi
if (<=) jump less_or_equal
```

Available Expressions

overflow	overflow flag is set
!overflow	overflow flag is not set
parity	parity flag is set
!parity	parity flag is not set
zero	zero flag is set
!zero	zero flag is not set
carry	carry flag is set
!carry	carry flag is not set
signed	result is signed
unsigned	result is not signed
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
+<	less than (signed)
+>	greater than (signed)
+<=	less than or equal to (signed)
+>=	greater than or equal to (signed)



6.6 Invocation

Invocation in `easm` is achieved through use of the **call** instruction. The **call** instruction should be of the form:

```
call <identifier> (<argument 1>, <argument 2>, <argument n>)
```

<identifier> should specify the name of the routine which is to be invoked. The routine may be a locally defined function or an imported function. Each argument should represent a necessary value that needs to be passed to the routine in order for it to function correctly.

Arguments are not checked by `easm` and as a result, if one too few or one too many arguments are specified, the stack will be inconsistent with how the routine may expect to find it which could lead to application failure.

Literal Arguments

When passing a literal argument to a function, an ambiguity exists in determining the intended size of the literal argument. Take the following example:

```
call SomeFunction (00h)
```

In this example, one argument is being passed to the function. The problem is that the size of this argument could be one of three possibilities. It might be a **byte** (0x00), a **word** (0x0000) or a **dword** (0x00000000).

This issue is resolved in `easm` through the use of call-time qualifiers. The call-time qualifier is appended onto the end of the literal argument and used to specify the argument size (as shown below).

```
call SomeFunction (00h:byte)
call SomeFunction (00h:word)
call SomeFunction (00h:dword)
```

As a result of the call-time qualifiers, the three call examples shown above will push a **byte** argument, **word** argument and **dword** argument respectively.

Note: If no call-time qualifier is found, the argument is assumed to be a **dword** and as such, the following is perfectly legal (assuming the function expects a **dword** argument):

```
call SomeFunction (00h)
```

String Arguments

For functions that expect a string argument, the string may be declared at the same time as it is passed - by means of a declarative argument:

```
call printf ("Hello World\n")
```



Function Pointers

A function may be invoked through a function pointer by making a call to an identifier that contains the address of the said function.

```
call eax
call dwSomeVariable
call dwParameter
```

The three examples above show how a function can be called indirectly through the `eax` register, a user defined variable and a function parameter respectively.

Function pointers are commonly used in the implementation of a 'call-back' system in which a pointer to a function is stored and invoked to indicate the completion of some event.

Function Pointer Example

```
section functions

function void DoWorkAndThenNotify (functionPointer:4)

    // Do some work
    call functionPointer

end

function void NotifyComplete ()

    // We have been notified of completed work

end

section code

call DoWorkAndThenNotify (&NotifyComplete)
```

The example shows how the function *DoWorkAndThenNotify* indirectly calls the *NotifyComplete* function through the function pointer stored in the parameter.



6.7 Arithmetic Operations

Arithmetic in easm concerns the following four mathematical operations: addition, subtraction, multiplication and division. The operators `+=`, `-=`, `*=` and `/=` provide access to these operations respectively.

Addition & Subtraction

The addition and subtraction operators should be of the form:

```
<identifier> += <identifier>
<identifier> -= <identifier>
```

Currently, memory to memory arithmetic is not supported and as such, the *LHS* and *RHS* of an arithmetic statement cannot both be a variable. The *LHS* of an arithmetic addition or subtraction may be either a register or a variable.

Examples

```
eax += 01h
esi -= edi
dwAddress += &SomeFunction
```

Multiplication

easm handles multiplication in a very similar fashion to addition and subtraction but with an additional constraint. The multiplication operator `*=` must not reference a variable on the *LHS* of the statement and is constrained to registers only.

Examples

```
eax *= ebp
ecx *= abcdh
ebx *= &dwVariable
```

Division

Division in easm is achieved through the use of the `/=` operator. Division is constrained to the `eax` register and is only in scope for the accumulator. The division operator will not be recognised if used with any other register or variable.

Note: The `edx` register is used as the higher 32 bits of the total sum to be divided; therefore you must ensure `edx` is cleared if you only want to divide the value in `eax`.

Examples

```
eax /= 02h
eax /= &SomeFunction
eax /= ebx
```



Increment And Decrement

Increment and decrement operations are supported by easm through the ++ and -- operators. The increment or decrement operator should directly follow the operand it is operating on.

Examples

```
eax ++  
dwVariable ++  
dwParameter ++  
ebx --  
wVariable --  
wParameter --
```

The first three examples add 1 to the `eax` register, a 32-bit user defined variable and a 32-bit function parameter respectively. The last three examples subtract 1 from the `ebx` register, a 16-bit user defined variable and a 16-bit function parameter respectively.



6.8 Bitwise Operations

The Bitwise Operators

A bitwise operation is one which operates on the binary pattern of its operand. In easm, bitwise operations are supported through the following bitwise operators:

Binary Operation	easm Operator
AND	&=
OR	=
XOR	^=
NOT	~
LEFT SHIFT	<<
RIGHT SHIFT	>>

Examples

```
// Perform bitwise NOT on the eax register, a 32-bit variable
// and a 32-bit function parameter.
~ eax
~ dwVariable
~ dwParameter

// Perform a bitwise left shift on the eax register, a 32-bit
// variable and a 32-bit function parameter.
eax << 0fh
dwVariable << 0fh
dwParameter << 0fh

// Perform a bitwise right shift on the eax register, a 32-bit
// variable and a 32-bit function parameter.
eax >> 0fh
dwVariable >> 0fh
dwParameter >> 0fh

// Perform a bitwise AND on the eax register, a 32-bit
// variable and a 32-bit function parameter.
eax &= ebx
dwVariable &= ebx
dwParameter &= ebx

// Perform a bitwise OR on the eax register, a 32-bit
// variable and a 32-bit function parameter.
eax |= ebx
dwVariable |= ebx
dwParameter |= ebx
```



```
// Perform a bitwise XOR on the eax register, a 32-bit
// variable and a 32-bit function parameter.
eax ^= ebx
dwVariable ^= ebx
dwParameter ^= ebx
```

Constraints

Bitwise left and right shift may operate on any of the 7 registers, variables and parameters but are only allowed to use the 4 registers `eax`, `ebx`, `ecx` and `edx` and literal values as the shifting operand.

This means that you may shift the contents of a register by the value in `eax`, `ebx`, `ecx`, `edx` or by some constant number but not the value in `ebp`, `esi`, `edi`, a user defined variable or parameter.

6.9 Include Libraries

It is possible to create libraries in `easm` with support for include libraries. An include library is simply a regular source code file without a `code` section. The source code file may declare anything from constants to functions which can later be reused in subsequent applications.

Once an include library is implemented (and tested), it may be included in a primary source code file by using the **include** directive followed by the path to the file, enclosed within double quotation marks.

Example

```
subsystem cui

include "inc\stdio.easm"
include "inc\win32.easm"

section data
```

The example above creates two include references, causing the declarations made within the two files `stdio.easm` and `win32.easm` to be included in the current file.