

This Documentation is not complete, it is more of first draft. The Language may change and nothing in this document is set in stone.

MasC (Multi-platform Assembly Source Code)

Introduction:

MasC is to be a form of grouping code to be loaded into a internal representation to be passed to code generators which have different tasks, including being converted to C and compiled and linked. It is to be designed to work along with the GNU GCC compiler collection and binutils as well as custom virtual machine.

The following is the Layout for the language and reference notes on how I think It could be implemented.

Notation Notes:

Hex is prefixed by 0x (same as C)
language comments are same a C

The language instead of operating on register's, it operates on variables, so the registers are more like keywords.

A special register `stkptr` which stands for the current stack pointer.

First is the keyword **object** :

The object keyword (followed by a object *name*) will group together data and text sections of the code in different named sub object **sections**. Override able sections through inheritance will be available as well. However all objects contain a var section, and cannot be compiled without it. This var section is very important it is where your variables are declared. All variables are initialized to zero by default.

A Example object:

```
object ExapleProgram {  
  
section var {  
  
}  
  
}
```

Declaring variables using the following types, byte, word, dword, real, string (magic built in type you can use arrays of bytes for lower level string manipulation). You can also make an instance of an object declared before your object. Creating variables: the **define, type, and array** keyword.

Expressions are encased in `[$[]]` brackets and can be used for initializers and on instructions, and for array size dimensions.

legal:

```
define byte x = $[5+5]
define string str = "test\n";
define type ExampleProgram prog
define type array ExampleProgram prog $[10]
define byte z = $[x+5]
```

arrays are indexed using the `[]` operator and the `+` operator and its index.

Example:

```
mov x, [x+5] this would load array index five into register one
```

Inheritance:

Inheritance from classes can be done through the `uses` block. You may inherit from as many classes as you need (Multiple Inheritance). You may also dynamically inherit from any other object (dynamic binding).

Example of `uses` block:

```
object example {
uses { ExampleProgram, ClassTwo }
}
```

driver keyword:

The `driver` keyword lets the compiler know this object contains the `begin` and `end` blocks, and will be the entry point for the application. (same as static main function in Java).

Example:

```
driver object example {
uses { ExampleProgram }
begin {
}
```

```
end {  
  
}  
  
}
```

Object Methods:

There are two types of methods for msvc objects. Virtual methods, and standard methods. Virtual methods are the same as pure virtual functions in C++, and you can not make a direct instance of a class that contains them. They must be inherited or dynamically inherited.

```
method f  
{  
start:  
    mov x, 1  
  
}
```

The method would be called with the instruction:

```
call f
```

A virtual method is declared the following way:

```
object handler {  
  
empty method NameOfEmptyProcedure  
  
}  
  
object Instance {  
  
uses { handler }  
  
section var {  
  
define dword y = 10  
  
}
```

```
overload method NameOfEmptyProcedure
{
start:
    mov y, 0
}
}
```

```
object instance2 {
```

```
uses { handler }
```

```
section var {
```

```
define dword y = 10
```

```
}
```

```
overload method NameOfEmptyProcedure {
```

```
start:
```

```
    mov y,0
```

```
}
```

```
}
```

Methods in the language can take a few different forms.

```
empty method m_F
```

a empty method makes the object abstract, and any object which inherits from this object must overload the method.

```
overload method m_F {
```

```
}
```

for normal methods it is a simple syntax

```
method m_F {
```

```
}
```

but hopefully I hope the language will eventually permit:

```

object displayHello {
    section var {
        define string hello = "Hello World"
    }
    empty method display
}

object sayHelloOutSide {
    uses { displayHello }

    overload method display {
        output(" growl outside ", hello , "\n");
    }
}

driver project Object {
    section var {
        define type sayHelloOutSide hey
    }

    begin {
        call hey->display
    }

    end {
    }

}

```

The output function is a unusual part of the language added for ease of use, and is using a masc type called a parameter list.

Section Overloading:

You can override a section like the begin, end, var, uses blocks, by simply placing a new section in the object. By default var sections merge.

Object Persistence:

Objects will support persistence and will be able to be saved and load with ease.

Expressions:

in MasC expressions will be enclosed in `$()` blocks. Expressions can be used on instruction operands, initializers (section var block), and procedure templates. All standard operators are allowed, `+`, `-`, `*`, `/`, `&`, `!`, `^`, `!`, etc. and work with floating point values, as well as integer values.

The following is a example:

```
driver object TestProgram {  
  
  section var {  
  
    define dword v = $[25*5]  
    define dword x = $[v*5]  
  
  }  
  
  begin {  
  start:  
  load r1, $[x+v]  
  mov r1,r2  
  str r1. v  
  
  }  
  
  end {  
  
  /* clean up here */  
  
  }  
  
  }  
}
```

Example Program:

```
driver object Test {  
  
section var {  
  
define byte array bt $[5]
```

```
define byte len = 5
}
```

```
method add {
```

```
start:
load r1, len
loop:
load r2, [bt+r1]
inc r2
str r2, [bt+r1]
inc r1
cmp r1, 5
jne loop
```

```
}
```

```
begin {
```

```
call add
call add
```

```
}
```

```
end { }
```

```
}
```

Instruction Set:

s

Instructions will include a unique instruction set masked in pseudo mnemonic easy to use form, to be translated to a variety of platforms and languages. The object feature should help grouping of code and increase reuse of routines.

if you have any interest in this project please contact me jared@lostsidead.biz

<http://lostsidead.com>