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 translation table to be converted to multiple higher level languages or converted to assembly and assembled and linked. It is to be designed to work along with the GNU GCC compiler collection and binutils as well as a multiple Virtual Machines and interpreters.

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 /* */
s

Language contains 15 general purpose registers:

r1 – r15

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

*+ add more register information

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 **section**s. 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 initializeds 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 a instance of a 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:

```
load r1, [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.

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 masc 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 r1,0
}
```

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 }
overload method NameOfEmptyProcedure
{
start:
        mov r1, 0
}

}
```

Now this Is very similar to C++ however this is where dynamic inheritance comes into play.

Say you had another object named instance2 with the following:

```
object instance2 {

uses { handler }

overload method NameOfEmptyProcedure {

start:
        mov r1,0

}

}

object instance3 {

section var {

 define byte x = 100
define string hellostr=”Hello World”
}

method walk {
start:

        load r1, x
        mov r1, 20
        str r1, 0xFF
}

method talk {
start:

say hellostr
}
}
```

in your main driver class you could say the following:

```
driver object MainProgram {

section var {

define empty handler handle
define generic name
define generic obj
define type instance3 three

}

begin {

scat three->hellostr, “howdy world”

inherit handle, instance
call handle->NameOfEmptyProcedure
shed handle, instance
inherit handle, instance2
call handle->NameOfEmptyProcedure
shed handle,instance2

 /*  this is where it differs from dynamic binding */

inherit handle, instance3
setgen name, walk
call handle->(name)
setgen name, talk
call handle->(name)

/* you can even use the generic type for inheritance of a instance of a object */

setgen obj, three
inherit handle, obj
call obj->[name]

; this would attach a instance instance3 to handle and call its talk function which would contain the
; string howdy world instead of hello world, since it is modified at the top of the begin method scope.
}


end {

}

}
```

What this code does, is the empty handler, inherits the overloaded functions from instance, then It is called. Then a call to shed (which 'sheds' the instance like a snake shedding its skin) and is free to inherit another instance of the overloaded empty procedures. My seem similar to C plus plus 's dynamic_cast , however when the inherit keyword is used the object glues the inherited object to itself. Objects can inherit in real time objects that have no relation to the base object unlike dynamic_cast. Also differs from Objective-C dynamic binding, by allowing the attachment of a existing instance of a class to be attached, or a new instance. Also the generic type can be used to store method and object names as variables so the exact name of the method or object can change.

The most basic procedure you can create. No parameters, only the var section which is shared through out the object (the data section). To create a method template the language would

permit:

```
        method f (&z1,&z2) {

        mov z1,0xFF
        mov z2,0

        }
```

Note no temporary is created, z1 and z2 are pointers to the actual registers, or variables passed by the call to f.

Calling the method:

```
        instruct f  r1 , r2
```

Putting it together:

Calling a function from a object instance inside of a object.

```
object base {
section var {
```

```
define string x = “hello world”
}
proc printText {
}
}
driver object program {
section var {
define base b
}
begin {
call b->printText
}
end {}
}
```

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@lostsidedead.biz

http://lostsidedead.com