# MUI2C

**COLLABORATORS**

| | *TITLE* :<br><br>MUI2C | |
|---|---|---|

| *ACTION* | *NAME* | *DATE* | *SIGNATURE* |
|---|---|---|---|
| WRITTEN BY | | July 20, 2024 | |

**REVISION HISTORY**

| NUMBER | DATE | DESCRIPTION | NAME |
|---|---|---|---|
| | | | |

# Contents

**1 MUI2C** **1**

1.1 Table of contents . . . 1
1.2 Author . . . 1
1.3 Aim . . . 1
1.4 Installation . . . 2
1.5 Requirements . . . 2
1.6 Use . . . 2
1.7 Syntax . . . 3
1.8 Future . . . 5
1.9 History . . . 6
1.10 Distribution . . . 7
1.11 Acknowledgements . . . 7

# Chapter 1

# MUI2C

## 1.1 Table of contents

MUI2C: A MUI Preprocessor.
    by Jason Birch

Aim
Installation
Requirements
Use
Syntax
Future
History
Distribution
Acknowledgements

## 1.2 Author

Author

I'm a PhD student at The University of Western Australia. Those of you who have followed Usenet may have noticed that I've been a PhD student for quite a while now -- 4 1/2 years, in fact. Hopefully I'll finish this year. :-)

For more information on me or my research (ie. what I should be doing instead of this :) you can visit my homepage at http://www.cs.uwa.edu.au/~jasonb/. You can also contact me via email at jasonb@cs.uwa.edu.au or the MUI mailing list, mui@sunsite.informatik.rwth-aachen.de.

## 1.3 Aim

Aim

My aim in writing this program was to ease the burden on the MUI custom class writer. There is a fair amount of tedious housekeeping required when creating a custom class (keeping track of methods and attributes you've created, assigning

them non-overlapping numbers, making sure you don't forget any in the dispatcher, etc) and I decided that this was something the computer should probably be doing, so MUI2C was born.

## 1.4 Installation

Installation

Installation couldn't be much simpler -- just place MUI2C in your path somewhere.

## 1.5 Requirements

Requirements

None, unless you want to recompile it. As it stands you'll need flex and SAS/C, although gcc probably works fine. Flex can be found on Aminet:

```
Flex252.lha        dev/c      320K  20+GNU Flex - lexical scanner gen. (Amiga p
```

mui.l uses a few features of flex that aren't present in lex (exclusive start states, <<EOF>>) so it'd be best to stick with the former.

## 1.6 Use

Use

```
MUI2C <filename>[.m]

MUI2C will parse <filename> and produce three files:
    <filename>.c    -- C code for your custom class. This should be ready to ↩
       compile.
    <filename>.h    -- public header file for your custom class, ready for use by ↩
       others.
    <filename>p.h   -- private header file for your custom class, used by < ↩
       filename>.c.
```

MUI2C was written as a preprocessor, which means it scans the input file and parses the bits it recognizes. It makes no attempt to verify that the rest of the code is correct, that's for your compiler to check. It tries to verify that the "language" it recognizes is correct, but it's probably quiet easy to fool. Let me know if you want this improved. All the strings that it outputs (currently in C code) are in id.c, so it should be fairly easy to convert it into a preprocessor for other languages, such as E or Oberon (I hope :). The only other thing that might require looking at is mui.l -- you might find that some of the tokens I've defined have special meaning in your language and the preprocessor might incorrectly detect them as intended for it. (C++ programmers: Don't worry about MUI2C inadvertantly grabbing a "::" from your code -- it only checks for that token if you precede it with "Class ( <somename> " before you've defined any MUI class.)

All the error messages are in English, and hardwired that way. If you want to produce a localized version of this, please let me know and send me the changed sources so I can include it. Also let me know if you decide to change the language I've defined.

Current limitations are that you can only define one custom class per file, only private custom classes are generated, and the number of IDs you can create for one particular class is fixed (NOOFIDS in id.c, currently 256). If you want to change any of these, feel free. :-) I'll assume you have a C compiler, because this program isn't of much use without one.

## 1.7 Syntax

Syntax

MUI2C understands the following syntax:
(Note: [] surrounds optional tokens, <> surrounds descriptions of something you supply.)

```
Class (<classname> :: <superclassname>) : <tagbase>

Data (<instancedata>)

Method[*] (<methodname>)[: public]
{
    <method code>
}

Method (OM_GET)
{
    ...
    Attributes
    {
        <attribute>[*][: public]
        {
            <get attribute code>;
        }
        ...
    }
    ...
}

Method (OM_SET)
{
    ...
    Attributes
    {
        <attribute>[*][: public]
        {
            <set attribute code>;
        }
        ...
    }
    ...
}
```

<classname> = the name you wish to call your class.
Should be less than CLASSNAMELENGTH chars (currently 64).

<superclassname> = the name of your superclass, as used in a call to
MUI_CreateCustomClass(). Should be less than
SUPERCLASSNAMELENGTH chars (also 64).

<instancedata> = the name of your instance data. Should be less than
DATANAMELENGTH chars (again, 64).

<methodname> = the name of a method. Methods you are creating (which
therefore need an entry in one of the header files) as
opposed to those you are overriding (which don't) are
distinguished by their names; your methods should all have
MUIM_<classname>_ at the front. Should be less than
NAMELENGTH chars (64).

<attribute> = the name of an attribute. Attributes you are creating rather
than overriding are distinguished by having MUIA_<classname>_
at the front. Also less than NAMELENGTH chars.

<tagbase> = the base value to be used for tags of your class. Method and
attribute IDs will be formed by adding a number to this.
Ideally it should be equal to

TAG_USER | (<your_mui_serial_no> << 16)

plus any number less than 0xFFFF (the sum of this and the
number of all your methods and attributes must be less than
TAG_USER | (<your_mui_serial_no+1> << 16)). The number may be
octal (recognized by having a leading "0"), hexadecimal
(recognized by a leading "0x" or "0X"), or decimal
(recognized by a leading digit other than "0"). It would be
wise to look at the range of values used by other custom
classes you have created and avoid reusing numbers in those
ranges, although currently only strictly necessary if one
class is an ancestor of another.

:public = optional suffix that indicates this method or attribute is to
be placed in the public header file rather than the private
one. Making an attribute public in either OM_GET or OM_SET is
enough to make it publically available; not making it public
in the other won't protect it from outside use. Perhaps one
day MUI2C will automatically do this, but for now you'll have
to work around it yourself if you want something gettable but
not settable to the outside world, for example, but still
settable by yourself in order to trigger notifications.

MUI2C will automatically create the dispatcher function, along with a creation function of the form: struct MUI_CustomClass *<classname>_Create(void);

It will also create the switch statements in OM_GET and OM_SET methods, the for loop in the OM_SET method, the call to the superclass in all methods, and the returning of the appropriate value in all methods. OM_NEW, MUIM_AskMinMax, MUIM_Setup, MUIM_Show, and MUIM_Draw are all recognized as special cases and have slightly different setup/cleanup code generated. If you have a method with

unusual requirements, such as an InputHandler method, you can tell MUI2C to not generate setup/cleanup code for it by using "Method*" to declare it rather than "Method".

Within a Method, you automatically have access to the following variables:

```
struct IClass   *cl     = your class pointer
Msg             msg     = pointer to the message you received
struct Object   *obj    = your object pointer
<instancedata>  *data   = pointer to your instance data
```

In addition, OM_GET also has:

```
ULONG           *store  = pointer to area to place result
```

while OM_SET has:

```
ULONG   tag->ti_Data    = data you were given to place in the current attribute.
```

Method* methods only have cl, msg, and obj defined, and they don't automatically call the superclass. If you wish to obtain the data for your object, you can use:

```
<instancedata>  *data   = GetData();
```

To call your superclass, you can use the "Super()" macro.

After each attribute in an OM_GET MUI2C automatically generates a "return (TRUE);", and after each attribute in an OM_SET it generates a "break;", unless the attribute name is followed immediately by a "*". As with Method, appending the "*" instructs MUI2C to not add extra code. Look at the source code generated for more information.

I've included a couple of examples which I hope will make things clear. The first, test.m, illustrates what an input file can look like. The second, Number.m, is actually a real (albeit simple) class, which subclasses MUIC_Text to add the ability to display double-precision numbers, although I don't think it will compile in gcc as-is because of the header files.

## 1.8 Future

Future

Well, that's largely up to you. If you make any changes, please email me and let me know and I'll consider incorporating them into my code. Localization of the error messages and the ability to produce other than C source code would be two prime candidates. Also let me know if you find any errors.

If you have any suggestions about the language MUI2C understands or the code it generates, feel free to send them to me. Personally I would like to be able to do something like

```
    object.MethodName();
```

and have it automatically converted into a DoMethod() call, or

```
    var = object.Attribute;
```

and have that converted into a get(), but that requires quite a bit of parsing of the source code to establish types, etc, unless we want to precede each such statement with a special keyword.

You'll probably notice that I precede each function with SAVEDS. This isn't really necessary unless the function is going to be called from another task, but I decided to play it safe. You can remove this if you wish. Also, the series of #define's included to define ASM, etc, were lifted straight out of the demo.h header file that comes with MUI in an attempt to make the generated code GCC and DICE friendly. I don't really know if it works or not, as I don't use either. :-) I've tried to avoid using SAS/C-specific functions in my source code in case GCC didn't have them, but a few may have slipped through.

Finally, feel free to send me suggestions on what else should be done automatically (perhaps add the standard header files so that the programmer doesn't have to?).

Oh, one more thing -- you may be wondering why I didn't use yacc (or bison). Well, I tried. I found that, while they make writing a language from scratch really easy, they seemed to make writing a preprocessor just as difficult as doing it entirely in flex. The amount of state information I had to pass back and forth was so great I ended up opting for the method I found simplest.

## 1.9 History

History

Version 1.2:

- Added ability to not generate the automatic "return (TRUE);" at the end of an OM_GET attribute or the automatic "break;" at the end of an OM_SET attribute by appending a "*" to the attribute name.

Version 1.1:

- Put ability to generate setup/cleanup code for methods back in. Now MUI2C will automatically call the superclass and return the appropriate result for the programmer.

- Added recognition for five new special cases which have unique setup/cleanup requirements: OM_NEW, MUIM_AskMinMax, MUIM_Setup, MUIM_Show, and MUIM_Draw.

- Added new keyword "Method*" to indicate a method that MUI2C is to do no variable setups, superclass calls, or returns for automatically. This was added to allow special cases where a method didn't fit the standard format.

Version 1.0:

- Removed ability to generate setup/cleanup code for methods automatically.

- Added ":<basecode>" field to class definition to allow the programmer to set the base code to be used for generating all method and attribute

values.

- Rewrote the parser to simplify it and reduce the number of states. Sacrificed some error-detection ability and comment-placement freedom to do so. New rules are simpler, however, so adding it back in should be easier.

- Altered syntax for defining attributes to allow code to be placed before and after the attribute definitions.

Version 0.5:

- First release, intended to test the waters and see if people liked the idea.

## 1.10 Distribution

Distribution

The contents of this archive are freely distributable with no fees, royalties, or anything like that, but I would prefer if the contents remained intact. You may change the source to suit your own needs, but I would appreciate you letting me know so that others might benefit. If you want to publish your changed version, I would also appreciate you using a name other than "MUI2C" so that we don't have two incompatible parsers out there with the same name. :-)

## 1.11 Acknowledgements

Acknowledgements

Thanks to:
Gilles MASSON (masson@alto.unice.fr), Ellis Pritchard (ellis@cam-ani.co.uk), Flavio Stanchina (flavio@iestn.inet.it), Wessel Dankers (wsldanke@cs.ruu.nl), Rob van der Veer (robv@dataweb.nl), and any others who provided feedback and suggestions.

Jody Garnet (jgarnett@sfu.ca) for the use of the icons.