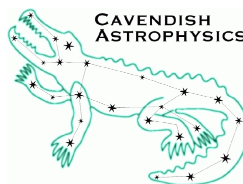
MRO Delay Line

Coding and Documentation Guidelines for Prototype Delay Line Software

John Young

rev 0.5
21 June 2007



Cavendish Laboratory
Madingley Road
Cambridge CB3 0HE
UK

Objective

To propose a set of guidelines for writing software code and associated documentation, which if followed, will enhance the readability, maintainability and re-use potential of the prototype delay line software.

Scope of this document

This document is not a review of previously published programming standard documents. The proposed guidelines are presented with only minimal justification.

1 Introduction

We have decided to use ANSI C (C99) for the prototype delay line code, except for the code for visualisation/analysis of recorded telemetry, which will be written in Matlab script (version 7).

We have decided against C++ because the Cambridge team has limited C++ experience. However, where appropriate (e.g. for much of the Workstation code and some libraries), we will write object-oriented code in C. A simple example of how this might be done is given in Sec. 4.

The guidelines presented here attempt to follow the spirit of the C++-based standards suggested by Tom Coleman at MRO
(see `http://www.possibility.com/Cpp/CppCodingStandard.html`).

1.1 Portability Requirements

We don't have strong requirements for portability, but we should:

- Ensure that only minor changes to Makefiles will be needed when porting to a different Linux distribution (i.e. possibly different version of gcc, possibly different library versions and file locations)
- Ensure that only *localised* changes to code will be needed when porting to a different flavour of Realtime Linux
- For Matlab scripts, ensure cross-platform compatibility where possible
- Facilitate porting of the visualisation/analysis code to later versions of Matlab:
  - Don't use functions which are designated to become obsolete in future versions
  - Don't modify Matlab functions or scripts unless required to fix bugs

  - Don't rename Matlab functions or scripts
  - Don't include multiple copies of scripts in path definition
  - Maintain subdirectory structure

- Write modular and well-commented code to facilitate partial re-use and/or reverse-engineering by MRO

2 Documentation Standards

Standalone documentation such as user manuals and architecture descriptions will be written in OpenOffice 2.x using suitable document templates. We anticipate that preliminary versions of documents will be passed to MRO at appropriate stages of the development process; such interim documents will be provided in PDF format. When the final software is delivered this will be accompanied by editable documents in native OpenOffice format (or converted to Word format if MRO prefers).

C code will include Doxygen markup as described in Sec. 3.8. We will automatically generate reference documentation in HTML and PDF formats from the marked-up code.

3 Coding Guidelines

3.1 Write portable Makefiles

Following the guidelines below will facilitate building a package on a different system:

- Use macro definitions to restrict platform-specific content to the beginning of the makefile
- Use standard targets (`all`, `install`, `clean`, ...)
- Use pkg-config (See `http://pkgconfig.freedesktop.org/wiki/`) or equivalents to avoid hard-wiring library and include-file paths into makefiles

Since we are only trying to support standard Linux and various flavours of Realtime Linux, and there will not be a large user community trying to build and install our software, there is no case for using GNU Autotools (autoconf/automake). However, use of Autotools is permitted where this expedites the development process.

3.2 Use a revision control system

We will use one of

- Subversion (`http://subversion.tigris.org/`)

- CVS (`http://www.nongnu.org/cvs/`)

A delay-line-specific source code repository will be set up on the Workstation.

3.3 Write modular code

As a general rule, there should probably be no more than 50 lines of C code in a function.

3.4 Use available code checkers

Write makefiles to invoke gcc with the `-Wall` switch. Run lint regularly during development to check for common coding errors
(see `http://www.pdc.kth.se/training/Tutor/Basics/lint/index-frame.html`). However, be aware that some lint warnings only indicate *potential* bad practice.

For Matlab code, use the supplied M-Lint code checker.

3.5 Source code formatting

C code should conform to the following formatting rules. These correspond to a variant of "Allman Style" (also known as "BSD Style").

- No embedded tab characters.
- Two spaces per indentation level.
- Two empty lines between large sections of distinct blocks of code.
- One empty line between small sections of code.
- Lines should not exceed 78 characters.
- The opening brace for a block of code is on a separate line immediately following the block initiator. It is not indented relative to the block initiator.
- The closing brace for a block of code is on a separate line immediately following the last non-empty line within the block. It is not indented relative to the block initiator.

Matlab code should use the formatting style implemented by the Matlab IDE.

3.6 Naming styles

The following naming styles are generally recognised:

- CapWords: initial letter capitalised, first letter of each word capitalised
- mixedCase: as CapWords but with lower case initial letter
- lower_case_with_underscores
- UPPER_CASE_WITH_UNDERSCORES

Don't use the following styles:

- Cap_Words_With_Underscores
- lowercase: no spaces or underscores
- UPPERCASE: no spaces or underscores

Don't use all-uppercase abbreviations within any name. The case of the initial letter of the abbreviation should match the particular naming style:

- Use `CalcCrcChecksum` rather than `CalcCRCChecksum`
- Use `wait_hz_tick` rather than `wait_HZ_tick` or `wait_Hz_tick`

3.6.1 C code

Use the following styles for function names, variable names, constants, and derived types:

- Function names: lower_case_with_underscores
- Variable names: mixedCase
- `#define`d and `enum` Constants: UPPER_CASE_WITH_UNDERSCORES
- Derived Types: CapWords

For object-oriented programming in C (see Sec. 4), use the following naming styles:

- "Class" names: CapWords
- "Method" names: CapWords
- "Attribute" names: mixedCase

3.6.2 Matlab code

Matlab function names use lowercase (no spaces or underscores). We should use the following styles:

- User-defined function names: mixedCase
- Variable names: lower_case_with_underscores
- Constants: UPPER_CASE_WITH_UNDERSCORES

3.7 Use informative function/variable names

Use informative names, particularly for functions (including methods) and global variables.

Where applicable, function names should indicate what the function returns, e.g. `data_is_valid()` rather than `check_data()` for a function returning TRUE or FALSE, `get_formatted_msg()` rather than `format_message()` for a function returning a string.

However, long variable names can impair readability to no advantage where the purpose of the variable is clear from nearby code. For example, `i` is probably better than `loopCounter` if there are not too many lines of code inside the loop.

Don't use variable naming conventions to indicate the types of variables, e.g. `sUserName`, `iCounter`. However, use of the following naming convention to identify pointers is encouraged:

```
Message msg, *pmsg, **ppmsg;
```

3.8 Use Doxygen to write self-documenting code

The MROI standard for documenting source code is to use the source code processing tool "Doxygen" to generate cross-referenced documentation from source code files.

See: `http://www.stack.nl/~dimitri/doxygen/index.html`

The markup conventions should follow JavaDoc style (set `JAVADOC_AUTOBRIEF = YES` in the Doxygen configuration file):

- `http://www.stack.nl/~dimitri/doxygen/docblocks.html#specialblock`
- `http://java.sun.com/j2se/javadoc/writingdoccomments/index.html`

Comment blocks in C (and C++) should precede each section of code and start with two *'s, like this:

```
/**
 * Defines the base class widget class. Used by extended widgets and
 * other widgets for operations related to foobar.
 */
class Widget ...
```

Here is a simple example (which doesn't use the exception-handling system!):

```
/**
 * Provides the mechanism for starting the widget. This includes a lazy
 * instantiation of the other objects owned by this widget.
 *
 * @param when The time at which this widget should start.
 * @return ErrorCode The completion status.
 * @retval OK Succeeded in starting the widget.
 * @retval ERROR Bad. Very Bad. Really really bad.
 */
ErrorCode Start(Time when) ...
```

3.9 Use DFB's exception-handling system

The exception-handling system is described in the document "An exception-handling system for C software".

The delay line messaging protocols (see protocols document) allow error reports returned by `ExcExplanation()` to be included in status messages. The user can thus be informed about exceptions caught on any sub-system.

We will consider unhiding the ability to classify exceptions if we encounter a situation where this functionality would be useful.

3.10 Miscellany

3.10.1 Note "gotchas" in comments

Use `:TODO:`, `:BUG:`, `:KLUDGE:` etc.

3.10.2 Prevent common mistakes with switch/case

Any statements following a case keyword (but not a final default) must end with a `break` statement, or `/*FALLTHROUGH*/` comment (to indicate that falling through is deliberate – this particular form should work as a lint directive, suppressing a warning).

```
        switch (expr)
```

```
{
  case ABC:
  case DEF:
    statement;
    break;
  case UVW:
    statement;
    /*FALLTHROUGH*/
  case XYZ:
    statement;
    break;
}
```

4 Object-oriented coding in C

As stated in the introduction, we will write object-oriented code where appropriate, but in ANSI C rather than C++.

Constructor and destructor function names shall have the prefixes `Create` and `Destroy` respectively (to match the existing serialise library). We now show the header and source code files that implement a simple example "class" (with some comments and error checking removed for brevity). Doxygen markup is included.

```
/**
 * @file MsgStack.h
 * Message stack definitions.
 */

#ifndef MSGSTACK_H
#define MSGSTACK_H

#define MAX_MSGLEN 200

/**
 * Single message within linked list, used by MsgStack.
 * ...
 */
typedef struct _MsgNode
{
  /* @publicsection */
  /* these attributes are accessed by MsgStack "methods" */
  /** Message text. */
  char msg[MAX_MSGLEN];
  /** Pointer to next message in linked list. */
  struct _MsgNode pNextNode;
  /* @privatesection */
  /* (nothing private) */
```

```
} MsgNode;

/**
 * List of messages.
 * ...
 */
typedef struct
{
  /* @publicsection */
  /* (nothing public) */
  /* @privatesection */
  /** Number of messages in stack. */
  int num;
  /** Pointer to first message. */
  MsgNode *pFirstNode;
  /** Pointer to last message. */
  MsgNode *pLastNode;
} MsgStack;

/* Function prototypes of MsgStack "methods" */
MsgStack *CreateMsgStack(void);
void AddMsg(MsgStack *, char *);
char *GetAllMsg(MsgStack *);
char *GetLastMsg(MsgStack *);
void DestroyMsgStack(MsgStack *);

#endif /* #ifndef MSGSTACK_H */


/**
 * @file MsgStack.c
 * Functions operating on MsgStack instances.
 */

#include <stdlib.h>
#include <string.h>

#include "MsgStack.h"

/**
 * Constructor.
 * ...
 */
MsgStack *CreateMsgStack(void)
{
  MsgStack *this;

  this = malloc(sizeof(MsgStack));
  this->num = 0;
  this->pFirstNode = NULL;
```

```
  this->pLastNode = NULL;
  return this;
}

/**
 * Add a message.
 * ...
 */
void AddMsg(MsgStack *this, char *msg)
{
  MsgNode *pNewNode;

  pNewNode = malloc(sizeof(MsgNode));
  strcpy(pNewNode->msg, msg);
  pNewNode->pNextNode = NULL;
  if (this->num == 0)
  {
    this->pFirstNode = pNewNode;
  }
  else
  {
    this->pLastNode->pNextNode = pNewNode;
  }
  this->pLastNode = pNewNode;
  this->num++;
}

/**
 * Return all messages, concatenated into a single string.
 * ...
 */
char *GetAllMsg(MsgStack *this)
{
...
}

/**
 * Return most recently-added message.
 * ...
 */
char *GetLastMsg(MsgStack *this)
{
...
}

/**
 * Destructor.
 * ...
 */
```

```
void DestroyMsgStack(MsgStack *this)
{
  ... /* traverse linked list, free nodes */
  free(this);
}
```

4.1 Inheritance in C

A “class” that inherits from MsgStack (usually referred to as a child class or sub-class) would be implemented as follows:

```
/**
 * @file ErrorStack.h
 * Error message stack definitions.
 */

#ifndef ERRORSTACK_H
#define ERRORSTACK_H

/**
 * List of error messages.
 * ...
 */
typedef struct
{
  /* @publicsection */
  /* (nothing public) */
  /* @privatesection */
  /** Instance of parent struct. */
  MsgStack *parent;
  /**  Destination for error messages. */
  FILE *errorStream;
} ErrorStack;

/* Function prototypes of ErrorStack "methods" */
ErrorStack *CreateErrorStack(void);
int HaveError(ErrorStack *);
...
void DestroyErrorStack(ErrorStack *);

#endif /* #ifndef ERRORSTACK_H */
```

```
/**
 * @file ErrorStack.c
 * Functions operating on ErrorStack instances.
 */

#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>

#include "MsgStack.h"
#include "ErrorStack.h"

/**
 * Constructor.
 * ...
 */
ErrorStack *CreateErrorStack(void)
{
  ErrorStack *this;

  this = malloc(sizeof(ErrorStack));
  this->parent = CreateMsgStack();
  this->errorStream = stderr;
  ...
  return this;
}

/**
 * Return TRUE if at least one error message in stack.
 * ...
 */
int HaveError(ErrorStack *this)
{
  return (this->parent->num > 0);
}

...

/**
 * Destructor.
 * ...
 */
void DestroyErrorStack(ErrorStack *this)
{
  DestroyMsgStack(this->parent);
  free(this);
}
```