Computer Systems, Inc.
MERCURY

Coding Standards and Design Guidelines

FM3TR Waveform Reference Implementation

SDR Forum Contract

March 23, 2007

Revision 1.0

Table of Contents

1 Introduction

1.1 Purpose

This document provides the guidelines for coding conventions and practices that will be used by the MCS SDRF SCA Reference Waveform Implementation (hereafter referred to as SCARWI) software engineering team. These guidelines are expected to be followed in all future software design and development work.

1.2 Goals

This document was written with the following goals in mind:

- Quality – Many aspects of established conventions are the disciplines required to create reliable and maintainable software. Without some common discipline, software will degrade in quality.
- Maintainability – Using consistent coding conventions will make the software more maintainable.

2 Class declarations

2.1 General practices

- Each include file will contain “include guards” to prevent multiple inclusion. The include guards will use the following naming convention “NAMESPACE_FILENAME”.

2.2 File Header Conventions

Header files shall not contain any namespace import or namespace aliasing statements.

Header files shall be self-contained, i.e., they shall not require a user to include other files before itself. E.g., all include files that reference “std::string” shall include <string>.

All include files and source files should contain a header with the following format.

```
/*
 * Abstract:
 *     This section should contain a text that describes the contents of the file
 *
 * Revision History:
 *    This section will contain the file revision history with the following format.
                Date:     the date the file was modified
                Revision Detail:  details on the modifications
 *
*/
```

File creation, including the date, should be the first entry in the revision history.

2.3 Naming conventions

2.3.1 File extensions

All C++ include files will have a ".h" extension and all C++ source files will have a ".cxx" extension.

2.3.2 File Names

Source files should have the same name as their corresponding include file with a different extension.

If a file contains nested namespaces it is encouraged that the file name reflects the namespace with additional prefixes.

2.3.3 Class Names

- Class Names Should start with a capital letter
- Each word in a class name should start with a capital letter, example "HopDecoder"

2.3.4 Class methods

- Class method names should start with a lower case letter
- Each word in a class name should start with a capital letter, example "this->getFrame()"

2.3.5 Class members

- Class members should have an “m_” prefix, and after that, should start with a lowercase letter, with each following word starting with a capital letter, e.g., “m_myParentClass”.

2.3.6 Non-Class Static data

- Data declared static in a .cxx file that is not class member data should have a “s_” prefix and after that, should start with a lowercase letter, with each following word starting with a capital letter, e.g., “s_myStaticArray”.

2.3.7 Constant declarations

- Constants should follow namespace conventions.
- Constant names should be upper case and each word in the name should be separated with an underscore, example “MAXIMUM_NUMBER_OF_NODES”.

2.3.8 Global Data

- Global data should be avoided. However if there is a legitimate need for a global class instance or global variable, it should start with “g_”.

2.4 Comments

The declaration of each class will contain a block comment with a detailed description of the class.

Example:

```
/************************************************************
 * This class is responsible for decoding errors in a received data frame
 ************************************************************/
Class A {

};
```

Each class method should also have a block comment similar to the one above and indented at the same level as the method. In addition, each parameter of the method should have a comment on separate lines which will include a description of the parameter along with an indication of whether the parameter is an input or an output parameter.

Example:

```
/*********************************************************
 * Creates a new circuit within a connection based upon the source
  * port set and destination ports set(s)
  *********************************************************/
 virtual Circuit* CreateCircuit(
                Connection* connection,      // In - Connection to create the circuit in
                PortOrdinal src_ports[],     // In - Null terminated array of source ports to be used
                                             // by the circuit.  If this is Null then all source
                                             // ports in the connection are used.
                PortOrdinal dest_ports[]);   // In - Null terminated array of dest ports to be used
                                             // by the circuit.  If this is Null then all dest
                                             // ports in the connection are used.
```

3 C++ Coding Conventions

3.5 General Conventions

- Constant declarations should use the C++ "const" declaration instead of the #define.
- C++ style "casting" should be used over C style typecasting.
- Class members should always be private or protected. Access to members should be provided through public access methods.
- Forward referencing should be used wherever it is safely possible to minimize #include dependencies and reduce compilation times.
- Implementation details of complex class declarations should be moved to the class definition to reduce #include dependencies and to enhance the ability to modify class implementation without changing the class declaration. In other words if you have a class with a lot of detail in the header (perhaps due to members that require include files instead of forward references), then one approach is to have a forward reference to a structure that exists in the .cxx file that actually contains the details of the member data. So, the header file in made to have less details and very few include dependencies.
- The class declaration shall not contain any inline implementations. It shall be a pure declaration of the class interface. Inline implementations can be provided, if desired, after the class declaration.

3.6 Braces

- All "if" and "else" statements should contain braces even if they contain a single line of code.

3.7 Function and Method Signatures

- The documentation for functions and methods shall clearly indicate the direction of each parameter.

3.8 Exceptions

- Instances should be used as exception types. When throwing pointers, it is easy to leak memory, e.g., in a "catch (...)" clause.

- Exceptions should be caught using a reference to const, e.g., “catch (const std::string &)”. If an exception was caught by value, a temporary object would be created.
- Code should limit itself to throwing a small set of exceptions. Keeping in mind that callers need to handle different exceptions separately, different exceptions shall only be thrown if users of the method would react to these exceptions differently. E.g., when calling a function to start a process, most callers of that function do not care about the specific reason of failure. In many cases, an exception of type “std::string” may be sufficient, so that a human-readable message can ultimately be reported to the user.

4 CORBA Usage Conventions

Always use “_var” types to store dynamically allocated data. This way, all memory is released at the end of a method, or in case of an exception. To return a value from an “_var” type, use its “._retn()” method.