3.0 SOURCE CODE FORMAT STANDARDS

Try not to be redundant with the SDS. Note that you can merge the two documents into one if you choose as long as you clearly describe your coding standards. What follows is a good set of standards to follow but ensure you enhance with your own corporate standards, especially if they are stricter.

While the <doc-ref#>_SDS included some coding guidelines that intersected with design guidelines, this section is more explicitly tuned to the standards that will be used for the code development itself.

3.1 General Code Formatting Guidelines

The general code formatting guidelines shown in Table 3-1 shall be met during coding.

Table 3-1. General Code Formatting Guidelines

Category	Guideline
Line Length	Maximum 132 characters per line. 80 characters per line or less is preferred.
Indentations	Spaces Only, in sets of 3 or 4 (Never Use Tabs)
Blank Lines	Blank Line Usage: Blank lines should be used liberally for readability o Use one (1) blank line inside functions. o Use two (2) blank lines between functions. o Use three to four (3 - 4) blank lines between file sections. o Avoid more than four (4) blank lines together.
Spaces	Other Use of Spaces: o Place a single space before and after each operator or control statement. o Place a single space after each comma in a function parameter list. o Be consistent with the use of spaces in the opening and closing parentheses of a function call or function declaration. Prefer placing a single space after each open parenthesis and before each close parenthesis
Naming Conventions	See section 4.0 "NAMING CONVENTIONS."

3.2 Required Code Format

The following code standard template shall be used for all code development.

```
// ---------------------------------------------------------------------------
//  <PRODUCT>, <PROCESSOR>
// ---------------------------------------------------------------------------

// ---------------------------------------------------------------------------
// NOTICE:  All rights reserved.
//
```

```
// ------------------------------------------------------------------------

// ----------------------------------------------------------------------
//
// Project   : Project Name
// Subsystem : Subsystem name (if applicable)
//
// Filename  : filename.c
// Author    : John W. Smith
// Revision  : 1.0
// Updated   : dd-mmm-yy
//
// This module contains…
//
// ----------------------------------------------------------------------

// ----------------------------------------------------------------------
//
// Revision Log:
//
// Rev 1.0  : dd-mmm-yy, jws -- Original version created.
//
// Rev 1.1  : dd-mmm-yy, jws -- <Change Description>
//
// ----------------------------------------------------------------------

// ------------------------------------------------------------------------
//
// Design Details and Explanations:
//
// o The original...
//
// o This task still makes use of...
//
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
//
// Implementation Notes:
//
// o There is a "sticky" design issue that occurred to me...
//
// o Still need to deal with...
//
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
//                              Include Files
// ------------------------------------------------------------------------

#include <std.h>
#include <sys.h>
#include <que.h>
```

```
#include "sysdefs.h"
#include "fault.h"

// ------------------------------------------------------------------------
//                              Definitions
// ------------------------------------------------------------------------

// This defines the rate (in milliseconds) in which the CVR
// Status word will be output via the ARINC 429 output port

#define STATUS_OUTPUT_RATE   1000

// ------------------------------------------------------------------------
// Local Conditional Compilation Switches
// ------------------------------------------------------------------------
// Note: Certain of these switches are intended only for debug and analysis
// of conditions observed during development.
// ------------------------------------------------------------------------

#define DEBUG_TRACE         0  // Nonzero = Enable Debug Trace output
// ------------------------------------------------------------------------
//                            Global Variables
// ------------------------------------------------------------------------

// This contains the DLR serial number as received from the
// SMP via the TDM bus during the SMP/FDP startup exchange.

char DLR_serial_num[10];

// ------------------------------------------------------------------------
// Begin Code
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
//
// GetWord - Get word at specified bit offset
//
// This function...
//
// Params : bit_offset - This is the offset from the beginning of the
//                       buffer to start assembling the bits for the word
//                       to be returned.
//
// Returns: This routine returns...
//
// ------------------------------------------------------------------------

unsigned int GetWord( long bit_offset )
{
   code...
}
```

Figure 3-1. Standard Code Template

3.3 Formatting of Existing Modules

If no discernable style is present in existing code, the file shall be reformatted according to the Coding Standards presented in this document.

If a discernable style *is* present in the existing code, but the style differs from the Coding Standard, consider reformatting the code to follow the Coding Standard.

Normally this is not an issue; however, code may be inherited into the project that was not produced by the local development team. Anything not explicitly standardized in this Coding Standard is left to the discretion of the engineer. When it does not conflict with the Coding Standards, prefer following nuances of existing code format.

4.0 NAMING CONVENTIONS

Update with your own naming conventions. Be as specific as possible.

The naming convention guidelines specified in Table 4-1 shall be followed in all code development.

Table 4-1. Naming Conventions Used in Coding

Category	**Guideline**
Abbreviations	Abbreviations shall be used if they are either commonly used in the application domain (e.g., FFT for Fast Fourier Transform), or they are defined in a project-recognized list of abbreviations. Otherwise, it is very likely that similar but not quite identical abbreviations will occur here and there, introducing confusion and errors later (e.g., track_identification being abbreviated trid, trck_id, tr_iden, tid, tr_ident, and so on).
Names	Names shall be chosen from the usage perspective and use adjectives with nouns to enhance local (context specific) meaning. Names shall also agree with their types.
Case	Names that differ only by (upper/lower) case shall not be used.
Underscores	The use of two underscores ('__') in identifiers shall not be used as it is reserved for the compiler's internal use according to the ANSI-C standard. Underscores ('_') are often used in names of library functions (such as "_main" and "_exit"). In order to avoid collisions, do not begin an identifier with an underscore.
Long Names	Identifiers shall not be extremely long, to reduce the risk for name collisions when using tools that truncate long identifiers.
Spelling	English words in names shall be spelled correctly and conform to the project required form of U.S. English. This is equally true for comments.
Class	A common noun or noun phrase in singular form shall be used to give a class a name that expresses its abstraction. Use more general names for base classes and more specialized names for derived classes. Classes shall be named so that "object.function" is easy to read and appears to be logical
Function	Verbs or action phrases shall be used for functions and methods. Use adjectives (or past participles) for functions returning a Boolean (predicates). For predicates, it is often useful to add the prefix "is" or "has" before a noun to make the name read as a positive assertion. This is also useful when the simple name is already used for an object, type name, or an enumeration literal. Be accurate and consistent with respect to tense. Negative names shall not be used as this can result in expressions with double negations (e.g., !isNotFound) making the code more difficult to understand. In some cases, a negative predicate can also be made positive without changing its semantics by using an antonym, such as "isInvalid" instead of "isNotValid".