Rockwell Collins CETC Avionics Co., Ltd Coding Standards for the C and Assembly Programming Languages

RCCAC-ENG-S-001

Rev. 1.0

Rockwell Collins CETC Avionics Co., Ltd.

Approval

	Name	Title	Approval	Date
Prepared by:	Hu Yinjie	Software engineer	On File	09/13/2017
Reviewed by:	Heng Zhong	TPM	On File	09/13/2017
Reviewed by:	Yang Luo	Product Line Manager	On File	09/13/2017
Approved by:	Richard Hackett	CTO	On File	09/25/2017

Revision History

Revision	Originator	Description	Date
1.0	Hu Yinjie	New Release	09/13/2017

Table of Content

1 Introduction

1.1 Purpose

This document presents coding standards for the C and Assembly programming language. It is expected that the user of this document is already familiar with the ANSI C programming language. ANSI C was designed to support the development of high quality, reliable, portable, reusable software.

ANSI C was also designed by committee that provided a vehicle by which many de facto language features made it into the final product. This document provides limits and guidelines to be used during software development.

1.2 Applicability

This standard applies to all software developed using the C and Assembly programming languages, as referenced in the project’s Software Development Plan.

1.3 Terminology

Throughout this document the following terminology applies to the use of the words shall, should, may, will, can, is, are

shall: This is the only word that defines a mandatory, binding requirement.

should, may, will: These are examples of words which are used in the commentary typically to further explain a requirement(s) or to define a recommendation. Recommendations should be followed when possible, but deviations are permissible with justification.

can, is, are: These are examples of words which are used in commentary to declare a fact or definition.

1.4 General Guidelines

Legacy code (older, inherited code) and third-party code *may* not be expected to follow this standard.

New code added to legacy code *is* expected to follow this standard.

Machine generated code from code generators, screen designers, etc., *is* not expected to follow this standard either. Ideally machine generated code *should* not be manually modified unless the code being modified *is* between protected regions that the code generator recognizes. It *should* be regenerated using the original tool. Code that *is* manually inserted in these modules *should* follow as much of the standard as applies.

2 Reference Documents

[1] Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*, First Edition, Prentice Hall, Inc., Englewood-Cliffs, NJ, 1988.
[2] Harbison, Samuel P. and Cuy L. Steele Jr. *'C' A Reference Manual*, Third Edition, Prentice Hall, Inc., Englewood-Cliffs, NJ, 1991.

3 Scope

3.1 Code Formatting

Code formatting affects how the code looks, not how the code functions. The physical layout of source text has a strong influence on its readability. The most important guideline is to be consistent throughout the project. If all code looks the same, code reviews are more productive since team members can concentrate on code functionality rather than code formatting. Source code "beautifiers" exist and can be employed to enforce formatting rules.

3.1.1 Horizontal Spacing

At least one blank should be left before and after binary operators.
No spaces should be left before or after parentheses delimiting argument lists.
Spaces may be left before or after parenthesis in any other case.
No spaces should be left before or after square brackets delimiting array indices, unless followed by binary operators.

Example 1. Horizontal Spacing

```
Interface_Buffer[a] = Get_Rcv_Register(port_1);
if((Interface_Buffer[a] >= 0) &&
    (Interface_Buffer[a] <= 9))
{
    Interface_Buffer[a] += 30h;
}
```

3.1.2 Indentation

Control structures should be consistently indented to highlight the control.
Function calls should vertically align parameters when continuing on multiple lines.
Statement continuations should logically reflect the operation.
Indentation should be two or three spaces for each level and should be consistent throughout the file.

Example 2. Indentation

```
/* function call */
error_code = Set_Data_Word (parameter_1,
                             parameter_2,
                             parameter_n);
/* statement continuation                                                   */
if(((control_word & reset_word)    == reset_word) ||
    ((control_word & resync_word)   == resync_word))
{
    statement sequence;
}
```


3.1.3 Alignment of Operators

Vertical operator alignment should be used to emphasize local program structure.

Example 3. Alignment of Operators

```
if(slotA >= slotB)
{
    temporary  = slotA;
    slotA      = slotB;
    slotB      = temporary;
}

/* quadratic equation                                                       */
radicand       = square(b) - (4 * a * c);
denominator    = 2 * a;
if((denominator   != 0) &&
    (radicand      >= 0))
{
    first_solution    = (b + square_root(radicand)) / denominator;
    second_solution   = (b – square_root(radicand)) / denominator;
}
x = (a * b) +
    (c * d) +
    (e * f);

y = (a * b) + c) -        /* basic equation                                 */
    (3.5) +               /* error factor                                   */
    ((2 * d) - e);        /* BS factor                                      */
```

3.1.4 Alignment of Declarations

Each declaration should be made on a separate line.

When "const" is used, it shall be written to the right of the type for constant pointers and to the left of the type for constant integers.

Declarations should be vertically aligned.

Example 4. Alignment of Declarations

```
/* declarations                                                             */

typedef double user_type;                 /* user_type is same as double    */
typedef int *int_ptr_type;                /*                                */

int        i;
int        *iPtr;
int        MatrixVar[H_SIZE][V_SIZE];
char       *cPtr;
double     d;
```



```
/* constant declarations                                                   */

int const     *const_pointer;      /* constant pointer to an integer       */
int * const   const_pointer;       /* again, constant pointer to an        */
                                   /* integer                              */
const int     const_integer;       /* constant integer                     */
const int     *const_integer;      /* pointer to a constant integer        */

int_ptr_type const   const_ptr;
User_Type            Its_Type;

static struct s
{
   float x,
   float y;
}

extern void Print_Manager();

void Queue_Manager();
```

3.1.5 Alignment of Braces, Brackets, and Parenthesis

Begin braces, brackets, and parenthesis should be aligned with their corresponding end braces, brackets, and parenthesis.

3.1.6 Function Argument Alignment

Vertical alignment of function arguments should be used when continuing on multiple lines.

Example 5. Function Argument Alignment

```
/* function declaration                                                    */

Error_Code Enqueue_Error
(
   Message *Msg_Ptr,        /* message pointer                             */
   Queue_Id que_id,         /* queue identifier                            */
   Priority_Type priority   /* message priority                            */
);

/* subsequent function call                                                */
Queue_Error = Enqueue_Error(Printer_Msg_Ptr,
                            Printer_que_id,
                            priority_3);

/* or possibly                                                             */
```



```
/* function declaration                                                     */

void Assign_To_Top
(
   Stack_Id Task_Stack_Id,  /* stack identifier                             */
   Priority_Type priority   /* task priority                                */
);

/* subsequent function call                                                 */

Assign_To_Top(Task_Stack_Id, priority);
```

3.1.7 Blank Lines

Blank lines should be used to group logically related lines of text.
Blank lines should be included to delineate major blocks of code from each other.
Note: See Example 35 Sample Source Code, Receive_Arinc_Data function for example.

3.1.8 Number of Statements per Line

Each statement **shall** be started on a new line.
When appropriate, compound statements *should* be broken across multiple lines.

3.1.9 Source Code Line Length

The source code line length should not exceed 102 characters and should be consistent throughout the File.

4 Readability

4.1 Commentary

The structure and function of well written C code should be clear without commentary. Obscure and badly structured code is hard to understand, maintain, or reuse irrespective of the amount of commentary.
In addition, during maintenance, commentary must be updated along with the code it explains, which is a potential source of error. Commentary should reveal to a reader only information which is difficult to extract from the program text.

4.1.1 General Comm

Minimize comments by creating code structured to be easily understood.
Where a comment is required, the comment *should* be grammatically correct sentences and **shall** be clearly identified by comment delimiters.

Example 6. General Commentary

```
In C:
/* Comment statements are placed to explain a functional group              */
/* of statements whenever additional explanation is required.               */
```

4.1.1.1 Commentary for Expressions

Any expression, whether logical or mathematical, **shall** be commented if it has a complexity factor greater than 2. The comment **shall** provide clarification of the expression and its intent.

4.1.2 Special Tags

There are special tags that may be added for certain sections of the files so that these sections may be easily identified using a software tool. Data in special tagged sections may be extracted to aid in creating documentation and traceability.
{@CB} .. {@CE}
These delineate the section at the top of each source code file that contains the description/purpose of the file.

{@FDB} .. {@FDE}
These delineate the section preceding each function that contains the name and description of the function.

{@RTB} .. {@RTE}
This section contains the requirements traceability requirement identifiers. There *should* be one of these sections embedded within each {@FDB}..{@FDE} section.

{@HB} .. {@HE}
These delineate the section at the top of each header file that contains the description/purpose of the file.

Note: When a header file (.h) contains function definitions (i.e. the actual code for the function such as inline functions), the function definition tags ({@FDB}..{@FDE}, including the {@RTB}..{@RTE} tags) *should* be included in the header file for each function. Otherwise, the function definition header *should* be in the source file. This ensures that every function has an associated traceability section.

4.1.3 File Headers

Every file that contains source code **shall** be documented with the standard file header that provides information on the file and its contents.

A .c standard file header:

- Special tag, {@CB} *should* be included.
- File name: Exact file name as it appears on the disk, including case.
- Purpose: The file purpose shall be text describing the overall purpose of the file. Why is it here? In general, what does it do?
- Notes: This section is optional. Any design considerations that are file wide but do not properly belong in the function header can be included here. This text can be extensive, for example, describing data formatting, or interface structure.
- Copyright: copyright information
- When the source file contains function definitions, such as inline functions, the function definition tags (i.e., FDB}..{@FDE}) including the requirement tractability tags (i.e., {@RTB}..{@RTE}) should be included.
- Special tag, {@CE} should be included.

A .h standard file header:

- Special tag, {@HB} *should* be included.
- File name: Exact file name as it appears on the disk, including case.
- Purpose: The file purpose shall be text describing the overall purpose of the file. Why is it here? In general, what does it do?
- Notes: This section is optional. Any design considerations that are file wide but do not properly belong in the function header can be included here. This text can be extensive, for example, describing data formatting, or interface structure.
- Copyright: copyright information
- When the header file contains function definitions, such as inline functions, the function definition tags (i.e., FDB}..{@FDE}) including the requirement tractability tags (i.e., {@RTB}..{@RTE}) *should* be included.
- Special tag, {@HE} *should* be included.

Example 7. .c File Header

```
/*
{@CB}
-
***************************************************************************
-* FILE NAME: cs_arinc_hw.c
-*
-* PURPOSE:
```



```
-* The ARINC I/O Group will receive and transmit ARINC 429 data.
-* Received data will be stored into queues for multi-word data such as
-* ARINC 615 data, and in a single word list for single word data such as
-* tune frequency. Transmit data is stored in queues for multi-word data
-* and in a list for single word data. Data is retrieved from and stored
-* to the group by use of a unique identifier. Serial ARINC Hardware
-* contains files that interface with the ASIC (i.e. Snoopy).
-*
-* NOTES:
-*
-* COPYRIGHT NOTICE:
-* Copyright 2001 Rockwell Collins Inc. Proprietary Information
-*
-
*************************************************************************
{@CE}
*/
```

Example 8. .h File Header

```
/*
{@HB}
-
*************************************************************************
-* FILE NAME: cs_serial_arinc_pub.h
-*
-* PURPOSE:
-* The Serial ARINC Public header file defines macros, type
-* declarations and function declaration that are used within serial
-* ARINC and presents the interface for outside the group.
-*
-* NOTES:
-*
-* COPYRIGHT NOTICE:
-* Copyright 2001 Rockwell Collins Inc. Proprietary Information
-*
-
*************************************************************************
{@HE}
*/
```

4.1.4 Section Headers

The following section headers *should* be included in each .c file:

File Prologue - contains the file header

- File Inclusion - contains all of the files to be included
- Macro Definition - contains all of the #defines
- Type Declarations - contains all of the typedefs
- Object Declarations - contains all of the global, external, and private data declarations
- Function Declarations - contains all of the function declarations

The following section headers *should* be included in each .h file:

File Prologue - contains the file header

- File Inclusion - contains all of the files to be included
- Macro Definition - contains all of the #defines
- Type Declarations - contains all of the typedefs
- Object Declarations - contains all of the external data
- Function Declarations - contains all of the function prototypes
- Note: The .h file *should* not declare any storage.

Example 9. .c File Section Headers

```
/*............................ FILE PROLOGUE
..............................*/

/*............................ FILE INCLUSION
.............................*/

/*............................ MACRO DEFINITION
...........................*/

/*............................ TYPE DECLARATIONS
..........................*/

/*............................ OBJECT DECLARATIONS
........................*/

/*............................ FUNCTION DECLARATIONS
.......................*/
```

Note: Include each section header regardless of whether there is corresponding data or not.

Example 10. .h File Section Headers

```
/*............................ FILE PROLOGUE
..............................*/

/*............................ FILE INCLUSION
.............................*/

/*............................ MACRO DEFINITION
...........................*/
```



```
/*............................ TYPE DECLARATIONS
.........................*/

/*............................ OBJECT DECLARATIONS
.......................*/

/*............................ FUNCTION DECLARATIONS
.....................*/
```

Note: Include each section header regardless of whether there is corresponding data or not.

4.1.5 Function Headers

The following function headers **shall** be included in each .c file:

- Special tag, {@FDB} *should* be included.
  - Name: function name
  - Syntax: function invocation syntax
  - Arguments: type and description of each argument
  - Returns: return value type and description
  - Description: description of the function
  - Special Notes: special considerations *(optional)*.
- Special tag, {@RTB} *should* be included.
  - Function ID: unique identifier for the function
- Special tag, {@RTE} *should* be included.
- Special tag, {@FDE} *should* be included.
-

Example 11. C Function Header

```
/*
{@FDB}
-
***********************************************************************
****
-* Name: Receive ARINC Data (public)
-*
-* Syntax: void Receive_ARINC_Data(void)
-*
-* Arguments: NONE
-*
-* Returns: NONE
-*
-* Description:
-* This procedure services the ARINC 429 receiver when data is received.
-* The receiver status registers are read to determine which receiver
-* FIFO has new data available and where to obtain the newly received
```



```
-* input data. The receiver's port register address and the received
-* label are then used to determine the proper ARINC 429 Multiword Rx
-* Msg Queue or ARINC 429 Single Word Rx List location in which to
-* place the received data. Based on the label and receiver channel,
-* the ARINC Application Specific Data for the ARINC word is used to
-* determine if the word is dependent on the SDI bits or not. If it
-* is SDI dependant, then the received word's SDI is compared to the
-* Unit's SDI via a call to Check Input SDI. If received word SDI code
-* matches the unit's SDI strapping or the ARINC word is not SDI
-* dependant, the word is accepted for further processing.
-*
-* If the accepted word is designated as re-mapped then the re-mapping
-* is done. This involves substituting the received label for the
-* re-map label in the ARINC Application Specific Data.
-* If the accepted word is designated for one of the Single Word
-* Lists, the word's ARINC 429 Single Word Rx Counter is reset and
-* the word is written into the ARINC 429 Single Word Rx List. The
-* word's freshness flag is set to FRESH.
-*
-* If the accepted word is designated as port selectable and the port
-* select has selected the port on which the word was received, then
-* the word is also written to the port select location in the ARINC
-* 429 Single Word Rx List. The port selectable word's counter is
-* reset and the freshness flag is set to FRESH.
-*
-* If the accepted word is designated to go into one of the ARINC 429
-* Multiword Rx Msg Queues and there is room in that designated queue
-* then the data is transferred to the queue via the Put Queue
-* function. If there is no room in the designated queue, then the
-* current new input data will be discarded.
-*
-* The ARINC 429 Multiword Rx Msg Queues users call the Get
-* ARINC Multiword function to retrieve the queued data.
-*
-* Special Notes:
-*
-* {@RTB}
-* Function ID: ARINC_HW_01
-* {@RTE}
-*
-
****************************************************************************
{@FDE}
*/
```


4.1.6

Commentary *should* be used to highlight and explain violations of programming guidelines.

Commentary **shall** be used to highlight and explain unusual or special code features.

Example 12. Strategic and Tactical Comments

```
/* This is a strategic comment describing the following block of code */
{
    ...
}
int Insanely_Great_And_Complicated_Function(int i)
{
    int index = i++ + ++i * i-- - --i; // TACTICAL COMMENT
    return (index);
}
```

Note: The "//" form for the tactical comment *is* allowed provided the "C" Compiler being used accepts this form.

4.2 Naming Convention

4.2.1 Numbers

Numbers **shall** be represented in a consistent fashion.
Literals **shall** be represented in a radix appropriate to the problem.
When using scientific notation, the "E" **shall** be upper case.
In an alternate base, the alphabetic character *should* be represented in lower case.

Example 13. Numbers

```
2.345E6 /* scientific                                                    */
0xffe23 /* hexadecimal                                                   */
```

4.2.2 Identifier Names

The names used to reference variables, functions, and other user-defined elements *are* known as identifiers. "Self-documenting" names require fewer explanatory comments. Comprehension *is* enhanced when identifier names *are* pronounceable and not overly long.

- Naming conventions **shall** be uniform.
- Names **shall** use as many characters as are allowed to clarify the meaning of the element up to a maximum of 40 characters.
- Identifier names **shall** be unique within overlapping scopes and *should* be unique, otherwise.
- In general, identifiers *should* be spelled out using lower case or mixed case, when “word” separation is desired, using alphanumeric characters with the first letter of each “word” optionally represented as upper case (e.g., PreviousTemperature).
- Words in a multiple word name *should* be concatenated with the underscore, "_", character or by capitalizing the first letter of each word following the first word (e.g., Previous_Temperature).
- The first character of the identifier name **shall** be an alpha-character.

- All alpha-characters in symbolic constant and "*#define*" names *should* use upper case.
- In general, function names *should* be verbs or phrases that describe an action, variable names *should* be adjectives and nouns that describe something tangible.

Example 14. Identifier Names

```
TimeOfDay    or Time_Of_Day  instead of tod
OriginX      or origin_x     instead of originX
GetData(x)   or Get_Data(x)  instead of getData(x)

#define identifiers are an exception in that, by convention, all upper
case alphabetic characters are used.

#define DEFAULT_DRIVE 1      instead of #define defaultDrive 1
```

4.2.3 Element Identification

Element names *should* be formed from words and phrases suggesting elements in natural language.
Common nouns *should* represent non-boolean objects.
Predicate clauses *should* represent boolean objects.

Example 15. Element Identification

```
/* non-boolean elements                                                 */
Users        Current_User
Schedules    class_schedule

/* boolean elements                                                     */
bool User_Is_Available
bool List_Is_Empty
bool Empty
```

4.2.4 Program Unit Identification

In general, procedures *should* be given identifiers that contain a transitive, imperative verb phrase describing the action.
In general, boolean functions *should* be given predicate-clause names and non-boolean functions *should* be given noun names.

Example 16. Program Unit Identification

```
/* procedural routines                                                 */
void rx_obtain_next_token()
void util_create_new_group()

/* boolean valued functional routines                                  */
bool Que_Is_Full(Queue_Id)
```

```
bool port_device_is_ready(Device_Id)
```

4.2.5 Constants and Named Numbers

Symbolic parameters *should* be used in lieu of specific numeric values to represent constants, relative locations within a table, and size of data structures.
Enumerated variables *should* be used to ease readability and type checking.
An exception to these rules *will* be when a simple operation such as initializing a memory block to zero is occurring, or subtracting 1 from a variable.

Example 17. Constants and Named Numbers

```
PI            instead of 3.141592653589793
MAX_IDU_WIDTH instead of 30
ASCII_BEL     instead of 0x7
ASCII_STX     instead of 0x2
```

4.2.6 Booleans

Boolean names *should* clearly describe the TRUE or FALSE state that the boolean represents.

Example 18. Booleans

```
Msg_Received (not Receive_Msg)
```

4.3 Using Typing

4.3.1 Declaring Types

"*typedef*" declarations *should* be used to improve program readability.

Example 19. Typedef

```
/* boolean type declaration                                              */
typedef unsigned short bool;
```

4.3.2 Enumeration Types

Enumeration types *should* be used instead of numeric encodings.
Bit field definitions **shall** match requirements of external devices.

5 Program Structure

5.1 Program Structure

Program structure can have a significant effect on maintainability. Well structured programs are easily understood, enhanced, and maintained. Poorly structured programs are frequently restructured during maintenance just to make the maintenance job easier.
In C, a program can be created in small, manageable, independently tested pieces. To create a program with multiple files, functions in one file must access functions and data in other files. A declaration tells the compiler, "This function or this piece of data exists somewhere else, and here *is* what it *should* look like." A definition tells the compiler, "Make this piece of data here." or "Make this function here.".

5.1.1 File Names

A consistent file naming convention *should* be used.
File names **shall** be formatted as <filename>.<extension>. Where "filename" should not exceed 28 characters and the first 13 characters *should* be unique. The "extension" is ".h" for header files and ".c" for source files.
Header files generally contain declarations, whereas source files generally contain definitions.
Each file **shall** be uniquely named.

Example 20. File Names

```
Bite_Output_Diagnostic_Word.c /* Source file containing definitions.
        */
Bite_Output_Maintenance_Word.h /* Header file containing declarations.
        */
```

5.1.2 Include Files

Header files **shall** be used to tie declarations together for large programs. This guarantees that all source files *are* supplied the same definitions and variable declarations. Changing a header files causes all source files that "*#include*" it to be recompiled.
Header files **shall** be clearly identified by having a filename extension of ".h".
Header files *should* conditionally "*#include*" other dependent header files.
Header files **shall** protect themselves from having their contents declared multiple times by using the "*#ifndef-#define-#endif*" preprocessor directives. These directives check whether the specified item is currently undefined in the preprocessor.
Include statements **shall** not contain directory paths, either absolute or relative.
No storage **shall** be declared in an include file.
All global data **shall** be preceded by "extern".
Include statements for header files *should* be specified in the order shown below:
System header files
Local project header files

Example 21. Include Files

Compiler directives used to avoid multiple declarations:

```
/* assume the filename is queues.h, then the first non comment lines        */
#ifndef queues_h
#define queues_h

/* Notice that the define preprocessor variable name is the same as the
filename without the period, surrounded by two underscores on each
side.
*/

/* All header file text.                                                      */

#endif /* queues_h                                                            */
```

Include ordering rules:

```
/* First system header files.                                                 */
#include <stdio.h>

/* Last project local header files.                                           */
#include "local.h"
```

5.1.3 Macro Substitutions

The directive "*#define* macro_identifier(<argument_list>) <token_sequence>" defines a macro identifier, an optional argument list and an optional token sequence. The token sequence is substituted for the macro identifier each time it is encountered in the source file. In practice, this directive is often used to implement in-line functions.

- Macro usage *should* be kept to a minimum as usage makes code size estimation more difficult, thus impacting the project planning process.
- Macro identifiers *should* consist of all upper case characters.
- Macro definitions **shall** not exceed 512 bytes.
- The number of macro arguments *should* not exceed 8.
- All macro arguments *should* be enclosed in parentheses.

Example 22. Macro Substitutions

```
/* In-line reciprocal function. */
#define RECIPROCAL(x) (1/(x))

/* In-line square function. */
#define SQUARE(x) ((x) * (x))
```

5.1.4 Functions

All function calls *should* have a prototype.

All functions that are not declared as void **shall** return a value.

All functions **shall** have their parameters declared within the function's parentheses.

Calling sequences **shall** agree in number and type of arguments both where called and where received.

Arguments *should* be ordered - first input, then output.

Return types for variables passed to called routines *should* be explicitly declared in the calling routine.

Functions *should* not exceed 60 lines of source code excluding comments.

Function prototypes declared within include files *should* use the "extern" keyword.

5.1.5 Functional Cohesion

A file **shall** serve a single purpose.

A file **shall** contain functionally related data, types, and routines.

5.2 Visibility

5.2.1 Minimization of Interfaces

Implementation details *should* be hidden from the user.

The number of parameters *should* be minimized within a routine.

The manipulation of global data *should* be minimized and rigidly controlled.

5.2.2 Restricted Visibility

Only those header files containing necessary declarations *should* be "*#include*"ed.

5.3 Portability

5.3.1 Avoiding Dependency

Units **shall** not depend on absolute memory locations except as dictated by the target computer architecture.

Dependencies on internal numeric or character representations *should* be avoided except in functions performing output conversions.

5.3.2 Physical Device Dependency

Code which tends to bind the software to unique physical device characteristics *should* be concentrated in low level device interface driver units.

5.3.3 Scan Formats

Scan formats **shall** contain only the following three components, the first two of which *are* optional:

- An asterisk to specify that the converted value is not to be stored.
- A field width to specify the maximum number of input characters to match when determining the conversion field. The field width is an unsigned decimal integer.
- A conversion specifier to determine the type of any argument, how to determine its conversion field and how to convert the stored value.

5.3.4 Compiler Dependency

Function parameters **shall** not be given "extern" storage class.

Preprocessor arguments **shall** not exceed 256 bytes.

6 Programming Practices

6.1 Data Structures

6.1.1 Variable Declarations

When bit field declarations *are* used, they *should* be declared as "*unsigned*" using integer data types.
Each variable *should* be declared on a separate line in its own statement followed by an optional comment.
Variables shall be declared to limit their scope to an appropriate level.
Every variable declared shall be used at least once.
Variables *should* be initialized prior to use.

Example 23. Variable Declarations

```
/* Initialize a two dimensional array element by element.                   */
void init_array()
{
    unsigned int i; /* array row index                                      */
    unsigned int j; /* array column index                                   */
    for(i = 0; i < ROW_MAX; ++i)
    {
        for(j = 0; j < COLUMN_MAX; ++j)
        {
            init_array_element(i, j);
        }
    }
}
```

6.1.2 Program Structure

Definitions for a given global storage identifier **shall** be identical in each declaration of that identifier.
Unique global variable names **shall** be assigned to each global variable in each block.
Multiple access and modification of global data *should* be rigorously controlled.
Members of structures **shall** be accessed through member names.
No size assumptions *should* be made when a structure is accessed.
Structure definitions *should* be performed via "typedef".

6.1.3 Arrays

Arrays **shall** be initialized with the exact number of array elements as the array is declared to contain.
Indices to arrays **shall** be within the array bounds.

6.1.4 Unions

Unions *should* be used with proper care taken to assure that the proper element is being accessed.
Word alignment *should* be carefully observed when declaring unions.
Example 24. Union Declaration

```
typedef union
{
   unsigned long ArincWordLong;

   struct
   {
      unsigned int ArincWordHigh;
      unsigned int ArincWordLow;
   }ArincWordInts;

   struct
   {
      unsigned int Parity   : 1;  /* Parity of Arinc Word.          */
      unsigned int Ssm      : 2;  /* Sign Status of Arinc Word.     */
      unsigned int Pad      :19;  /* Data Bits.                     */
      unsigned int Sdi      : 2;  /* SDI of Arinc Word.             */
      unsigned int Label    : 8;  /* Label of Arinc Word.           */
   }ArincBitField;

} Arinc429WordType;
```

6.2 Expressions

6.2.1 Parenthesized Expressions

Parentheses *should* be used to ensure the order of evaluation for compound statements.
Parentheses **shall** be used to force an operation or a set of operations to a higher level of precedence.

6.2.2 Positive Forms of Logic

Compound negative boolean expressions *should* be avoided.

6.2.3 Boolean Value Testing

Testing the boolean value of a variable *can* be done explicitly or via the unary '!' operator.
Testing the boolean value of an assignment *should* be explicit.

Example 25. Boolean Value Testing

```
if(var != 0)            /* or */           if(var)
if(var == 0)            /* or */           if(!var)

if((var1 = var2) != 0)  /* instead of */  if(var1 = var2)
if((var1 = var2) == 0)  /* instead of */  if(!(var1 = var2))
```

6.2.4 Type Casting

An expression *can* be forced to a specific type using a construct called a cast.

Mixed mode operations (e.g., arithmetic operations with both real and integer values) *should* be avoided. Comments **shall** document exceptions.
The cast operator **shall** be used to force the desired mode of operation in mixed mode operations. Care *should* be taken that data is not unintentionally truncated when using a cast.
A cast from a portion of a "struct" to another "struct" *should* be avoided.

Example 26. Type Casting

```
y = (int) (((long)x * (long)COEF1 + (long)y * (long)COEF2) >> 15);
```

6.2.5 Accuracy of Operations

All arithmetic operations *should* be analyzed to ensure computational errors do not result.
Outputs *should* not provide more significant digits than warranted by the computational method.
Divide-by-zero errors *should* be avoided by checking all divisors before the divide operation *is* executed.

6.3 Statements

An expression becomes a statement when it is followed by a semicolon, referred to as a statement terminator. An opening and closing brace pair, "{" and "}", is used to group declarations and statements together into a compound statement or block. A block is syntactically equivalent to a simple statement.

6.3.1 Nesting

More than ten levels of nesting *should* be avoided.
Nesting of Boolean expressions beyond six (five operators) levels *should* be avoided.

6.3.2 Self Modifying Code

Run-time self modifying code **shall** be avoided.

6.3.3 Conditional Statements

Statement braces *are* required for multiple lines following a control statement.
Statement braces *may* be used, but *are* not required, when only one line follows a control statement.

Example 27. Conditional Statements

```
if(expression)
{
    statement;
}
if(expression)
{
    statement;
    statement;
}
else
{
    statement;
    statement;
```



```
}

if(expression_1)
{
    statement;
}
else if(expression_2)
{
    statement;
}
else
{
    statement;
}
```

6.3.4 Switch Statements

Switch statements *should* always use a default.
The default *should* be biased toward error detection.
Each case within a switch statement with unique code *should* include either a “break” or “continue” statement. Consecutive case statements with no unique commands do not require a “continue” statement but **shall** be documented with a commentary in order to show the intention of the design.

Example 28. Switch Statements

```
switch(expression)
{
    case constant_1:
        statement sequence;
        break;

    case constant_2:
        statement sequence;
        break;
    .
    .
    .

    case constant_n:
        statement sequence;
        break;

    default:
        statement sequence;
        break;
}
```


6.3.5 Ternary Operator Expressions

Use of the ternary operator ***shall*** be avoided.

Use if-then-else conditional statements instead of ternary operator expressions.

Care *should* be taken when using structural coverage analysis tools (CodeTest, VectorCast, etc.) which may not correctly instrument ternary expressions. Check the Tool Qualification Data Archive for the results and expectations when ternary expressions are used.

Example 29. Ternary Operator Expressions

```
/* Ternary Conditional Operator format.                                    */
variable = ((expression) ? value_1 : value_2);

/* if-then-else conditional statement format.                              */
if (expression)
{
    variable = value_1; /* true case in ternary expression                 */
}
else
{
    variable = value_2; /* false case in ternary expression                */
}
```

6.3.6 Loops

The loop index *should* not be changed indiscriminately within a loop.

Use "for(;;)" instead of "while(1)" when coding loops with no terminating condition.

Outer loops *should* not be exited from within inner loops

Example 30. Loops

```
/* While loop format.                                                      */
while(expression)
{
    ...
    statement;
}

/* For loop format.                                                        */
for(init_expression; test_expression; increment_expression)
{
    ...
    statement;
}

/* Do-while loop format.                                                   */
do
{
```

```
    ...
    statement;
} while(expression);
```

6.3.7 Safe Programming

All pointers *should* be explicitly declared before they are used.

Pointers **shall** not be passed between processors.

Pointer Arithmetic *should* be avoided.

Pointer usage *should* avoid indirection in excess of one level. Pointer usage beyond one level of indirection **shall** be documented.

Static pointers **shall** not reference local variables.

The “signal” system call *should* be avoided. Use of the “signal” system call **shall** be documented.

Keywords of the C programming language **shall** not be redefined.

6.3.8 Entry/Exit Statements

A calling function *should* use a "(*void*)" cast to explicitly ignore the returned value of the called function.

If a function *is* not declared as "*void*", it must return a value.

Return types for variables passed to called procedures *should* be explicitly declared in the calling routine.

Each function *should* have a single entry point and a single exit point.

Example 31. Entry/Exit Statements

```
if(expression_1)
{
    statement sequence;
    return_val = code_1;
}
else if(expression_2)
{
    statement sequence;
    return_val = code_2;
}
else /* error condition                                              */
{
    statement sequence;
    return_val = errorCode;
}
return(return_val);
```

6.3.8.1Recursion

No recursive functions **shall** be used.

6.3.9 Goto’s and Labels

"*goto*" statements **shall** not be used.

Labels **shall** not be used for execution control by the operational software.

Labels *may* be inserted to aid in testing and debugging.

6.3.10 Calling Sequences

Calling sequences **shall** agree in number and type of arguments both where called and received.
Calling sequences **shall** avoid making assignment statements within the function arguments.

6.4 Debugging

6.4.1 Compiler-Generated Warning Messages

Every compiler-generated warning *should* be investigated to determine that the program is not doing something of questionable validity.

6.4.2 Exceptions

Exception handlers are used to promote a fault-tolerant system and typically provide recovery from the following conditions:

- anticipated conditions during the normal course of program execution
- anticipated but uncommon terminating conditions
- hardware failures
- detection of invalid input
- anticipated and unanticipated errors

Detected errors caused by software bugs **shall** at worst generate a reset of the program, and at best handle the error (e.g., exception handlers and software traps), and archive relevant troubleshooting information.

6.4.3 Asserts

An "assert" *is* a macro that expands to an **"***if***"** statement. If the expression evaluates to zero, it causes the program to output debugging information before aborting program execution. The "assert" syntax *is* as follows

assert(expression);

An "assert" can be easily activated by defining or deactivated by undefining the indicated macro definition.
Note that even though "assert" *is* a macro it uses lower case characters.
Properly coded "assert"s cut debugging time.
"*assert*"s *may* be used at any point that the programmer *is* making an assumption about the state of the system.
"*assert*"s *may* be used to determine information about items that are assumed but not known.
"*assert*"s *should* be removed for final production version of the code.

Example 32. Asserts

```
/* Good example of using asserts.                                                  */
void Function_Name
(
    char *(string_ptr)
)

{
/* Has a valid pointer been passed?                                                */
    assert(stringPtr != NULL);
    .
```



```
    .
    .
}

/* Bad example of using asserts.                                             */
void Function_Name(char *string_ptr)
{
    int max_rows = 7;
/* Was max_rows initialized?                                              */
    assert(max_rows == 7);
    .
    .
    .
}
```


7 Assembly Guidelines

Assembly code *should* be produced by expanding the PDL statements into a functionally equivalent implementation.
Code **shall** be designed and written to be easily understood.
Use of coding “tricks” and special features should not be allowed unless specific optimizations *are* needed to meet performance requirements.
Use of such features **shall** be appropriately documented within the source code using comments.
Procedures *should* be kept under 100 lines of executable code.
The design and code of Assembly **shall** be extensively documented, including blocks of comments introducing and explaining major functions, any subsections of a function and comment lines to the right of the code statements, to supplement and explain the logic statements.

Example 33. General Commentary

```
; Comment statements are placed to explain a functional group
; of statements whenever additional explanation is required.
```

7.1 File Header

A .asm standard file header **shall** contain the following:

- Special tag, {@CB} *should* be included.
  - File name: Exact file name as it appears on the disk, including case.
  - Purpose: The file purpose shall be text describing the overall purpose of the file. Why is it here? In general, what does it do?.
  - Notes: This section is optional. Any design considerations that are file wide but do not properly belong in the function header can be included here. This text can be extensive, for example, describing data formatting, or interface structure.
  - Copyright: copyright information
  - When the assembly file contains function definitions, the function definition tags (i.e., FDB}..{@FDE}) including the requirement tractability tags (i.e., {@RTB}..{@RTE}) *should* be included.
- Special tag, {@CE} *should* be included.

Example 34. .asm File Header

```
;{@CB}
;*************************************************************************
;* FILE NAME: Read_FLASH.asm
;*
;* PURPOSE:
;* This file contains the low-level function that reads data stored
;* in the FLASH memory.
;*
;* NOTES: None
;*
;* COPYRIGHT NOTICE:
;* Copyright 2001 Rockwell Collins Inc. Rockwell Proprietary Information
```



```
;*
;**********************************************************************
;{@CE}
```

7.2 Section Header

The following section headers *should* be included in each .asm file:

- File Prologue - contains the file header
- File Inclusion - contains all of the files to be included
- Macro Definition - contains all of the #defines
- Type Declaration - contains all of the typedefs
- Object Declaration - contains all of the global, external and private data declarations
- Function Declarations - contains all of the function declarations
-

Example 35. .asm File Section Header

```
*............................ FILE PROLOGUE .............................*

*............................ FILE INCLUSION ............................*

*............................ MACRO DEFINITION ..........................*

*............................ TYPE DECLARATIONS .........................*

*............................ OBJECT DECLARATIONS .......................*

*............................ FUNCTION DECLARATIONS .....................*
```

Note: Include each section header regardless of whether there *is* corresponding data or not.

7.3 Function Headers

The following header text **shall** be placed at the beginning of each function

- Special tag, {@FDB} *should* be included.
- Name: function name
- Syntax: Calling syntax: the “C” calling syntax of the function OR the assembly calling syntax. If the “C” calling syntax *is* specified, then the assembly calling syntax *is* implicit and need not be specified. If the function does not conform to “C” calling syntax, then the assembly calling syntax must be specified.
- Arguments: a description of each argument (parameter). If the “C” calling syntax *is* not specified, then this description must include the required placement of the parameter (register ID, accumulator ID or stack) before the call to the routine is made. If the “C” calling syntax *is* specified then parameter placement *is* implicit and need not be specified. If a complete register/accumulator usage breakdown *is* included under “special notes”, then this section need only reference the register/accumulator usage breakdown.
- Return parameter: if the “C” calling syntax *is* specified, then the placement of the return parameter *is* implicit and need not be specified. If the “C” calling syntax *is* not specified, then the description and placement (register ID, accumulator ID or stack) of the return parameter must appear here. If a

complete register/accumulator usage breakdown *is* included under “special notes”, then this section need only reference the register usage breakdown.

- Description: description of the function
- Special Notes: Any special notes or limitations that apply to the function. It is often helpful (for code maintenance) to include a complete breakdown of register and accumulator usage here (optional).
- Special tag, {@RTB} *should* be included.
- Function ID: unique identifier for the function
- Special tag, {@RTE} *should* be included.
- Special tag, {@FDE} *should* be included.

Example 36. Assembly Function Header

```
;{@FDB}
;***********************************************************************
;* Name: Read_FLASH
;*
;* Syntax:
;* Read_FLASH (unsigned long Address, unsigned int * value)
;*
;* Arguments:
;* Address : The physical flash address that will be read.
;* Located in the A register
;* value : The address where the value read from flash
;* will be written to.
;* Located at *SP(0)[i]
;*
;* Returns: None
;*
;* Description:
;* The FLASH address is passed to this function as a parameter along
;* with a pointer to the location where the FLASH data is going to
;* be stored. This function returns nothing.
;*
;* Special Notes : <the Breakdown of the register usage could be put
here>
;*
;* {@RTB}
;* FUNCTION ID: READ_FLASH_01
;* {@RTE}
;*
;***********************************************************************
;{@FDE}
```

7.4 Symbol Declarations

Each Assembly symbol *should* have a separate declaration statement.

The symbol declarations **shall** be as described in the specific assembly language documentation.

Only one symbol *should* be declared per line and each symbol declaration *should* include a short description of the declared symbol. Additional comments and descriptive text that clarify the usage of symbols *is* encouraged.

Appendix A C Coding Example

Example 37. **Sample Header File**

```
#ifndef CS_SERIAL_ARINC_PUB_H
#define CS_SERIAL_ARINC_PUB_H

/*............................ FILE PROLOGUE
...............................*/
/*
{@HB}
-
*****************************************************************************
-* FILE NAME: cs_serial_arinc_pub.h
-*
-* PURPOSE:
-* The Serial ARINC Public header file defines macros, type
-* declarations and function declaration that are used within serial
-* ARINC and presents the interface for outside the group.
-*
-* NOTES:
-*
-* Copyright:
-* Copyright 2001 Rockwell Collins Inc. Rockwell Proprietary Information
-*
-
*****************************************************************************
{@HE}
*/
/*............................ FILE INCLUSION
.............................*/
/*............................ MACRO DEFINITION
...........................*/
#define ENTER_TEST_MODE             0x4956 /* Start Snoopy loop back test.
*/
#define MULTIWORD_QUEUE_SIZE        32     /* Size of Multiword Queues.
*/
#define OUTPUT_CHANNEL_QUEUE_SIZE   16     /* Output Channel Queue Size.
*/
#define MAX_HS_BUS_THRESHOLD        22     /* Speed Detect HS threshold.
*/
#define NO_RX_BUS                   999    /* Indicator of no selected bus.
*/
#define LOW_PRIORITY                0      /* Queue priority Low.
*/
```



```
#define HIGH_PRIORITY             1      /* Queue priority Hi.
*/
#define ALL_CALL                  0      /* Radio configured All Call.
*/

/*............................ TYPE DECLARATIONS
..........................*/

/*
-
******************************************************************************
-* Word Format Type - word type is used were elements can be both CSDB
-* or Arinc words, i.e. used for bus selectable received words.
-
******************************************************************************
*/

typedef struct
{
unsigned int Word_High_16;
unsigned int Word_Middle_16;
unsigned int Word_Low_16;
} Word_Format_Type;

/*
-
******************************************************************************
-* Arinc 429 Word Type - union that provides access to arinc words for
-* element extraction.
-
******************************************************************************
*/

typedef union
{
   unsigned long Arinc_Word_Long;

   struct
   {
      unsigned int Arinc_Word_High;
      unsigned int Arinc_Word_Low;
   }Arinc_Word_Ints;

struct
```

```
{
unsigned int Parity            : 1;   /* Parity of Arinc Word.
*/
unsigned int SSM               : 2;   /* Sign Status of Arinc Word.
*/
unsigned int pad1              :13;   /* Blank on 16 bit boundary.
*/
unsigned int pad2              : 6;   /* Blank on 16 bit boundary.
*/
unsigned int Sdi               : 2;   /* SDI of Arinc Word.
*/
unsigned int Label             : 8;   /* Label of Arinc Word.
*/
}Arinc_Bit_Field;

} Arinc_429_Word_Type;

/*............................. OBJECT DECLARATIONS
..........................*/

typedef void (*Event_Driven_Proc_Type)(int ,Arinc_429_Word_Type);
typedef void (*Error_Handling_Proc_Type)(int Rxer_Number, unsigned
Error_Status);

/*............................. FUNCTION DECLARATIONS
.......................*/

void ARINC_Process_Word( unsigned int Snoopy_Rcver_Index,
                         unsigned int Snoopy_Index, Arinc_429_Word_Type
Arinc_Word);
unsigned int Check_Input_SDI(unsigned int, unsigned int );
void Config_ARINC_Txers_Rvers(void);
void Detect_Speed(void);

#endif
```


Example 38. Sample Source File

```
/*............................ FILE PROLOGUE
..............................*/
/*
{@CB}
-
****************************************************************************
-* FILE NAME: cs_arinc_hw.c
-*
-* PURPOSE:
-* The ARINC I/O Group will receive and transmit ARINC 429 data.
-* Received data will be stored into queues for multi-word data such as
-* ARINC 615 data, and in a single word list for single word data such as
-* tune frequency. Transmit data is stored in queues for multi-word data
-* and in a list for single word data. Data is retrieved from and stored
-* to the group by use of a unique identifier. Serial ARINC Hardware
-* contains files that interface with the ASIC (i.e. Snoopy).
-*
-* NOTES:
-*
-* COPYRIGHT NOTICE:
-* Copyright 2001 Rockwell Collins Inc. Rockwell Proprietary Information
-*
-
****************************************************************************
{@CE}
*/

/*............................ FILE INCLUSION
.............................*/

#ifndef   CS_DEFINES_H
#include  "cs_defines.h"
#endif
#ifndef   CS_SERIAL_ARINC_PUB_H
#include  "cs_serial_arinc_pub.h"
#endif
#ifndef   CS_SERIAL_ARINC_PRV_H
#include  "cs_serial_arinc_prv.h"
#endif
#ifndef   CS_SNOOPY_DEF_H
#include  "cs_snoopy_def.h"
#endif
```



```
/*............................. MACRO DEFINITION
...........................*/

/*............................. TYPE DECLARATIONS
..........................*/

/*............................. OBJECT DECLARATIONS
........................*/

extern ARINC_429_Multiword_Queue_Type
ARINC_429_Multiword_Rx_Msg_Queues[];
extern ARINC_429_Multiword_Queue_Type
ARINC_429_Multiword_Tx_Msg_Queues[];
extern ARINC_429_Out_Chan_Queue_Type      ARINC_429_LP_Out_Chan_Queue[];
extern ARINC_429_Out_Chan_Queue_Type      ARINC_429_Output_Channel_Queue[];
extern ARINC_429_Single_Word_Tx_List_Type ARINC_429_Single_Word_Tx_List[];
extern ARINC_429_Single_Word_Rx_List_Type ARINC_429_Single_Word_Rx_List[];
extern ARINC_429_Receive_Word_Dsc_Type    ARINC_429_Rx_Word_Dsc[];
extern ARINC_429_Receiver_Dsc_Type        ARINC_429_Receiver_Dsc[];
extern ARINC_429_Transmitter_Dsc_Type     ARINC_429_Transmitter_Dsc[];
extern Label_Screen_Type                  Label_Screen[];
extern Error_Handling_Proc_Type           Error_Handling_Proc[];

/* Extern variables requiring volatile for optimation
*/

extern volatile const int                Max_NonAudio_Rx;
extern volatile const int                Max_NonAudio_Tx;
extern volatile const int                Num_Receivers_Each_Snoopy;
extern volatile const int                Num_Transmitters_Each_Snoopy;
extern volatile const int                Number_Of_Snoopies;

/*
-
***************************************************************************
-* Rxer_Error_Status - is an array that will serve as a temporary variable
-* for the error status. This is needed since Snoopy Receiver Error
-* Register contains error status for 4 receivers. When the register is
-* read from the snoopy the error register is cleared, and error status for
-* the other three receivers is lost.
-
***************************************************************************
*/
Rxer_Error_Status_Type Rxer_Error_Status[NUM_ERROR_STATUS_REGS];
```



```
/*
-
*****************************************************************************
-* Max_Rx_Counter_Value and Max_Tx_Counter_Value - are variables that
provide
-* limits for counters to protect against wrap-around when the counter
arrays
-* are incremented.
-
*****************************************************************************
*/

unsigned int Max_Rx_Counter_Value;
unsigned int Max_Tx_Counter_Value;

/*
-
*****************************************************************************
-* Label_Remap - configuration of the snoopy label screen ram involves bit
-* manipulation on the nibble. Label_Remap provides that bit transformation
-* by providing the value to manipulate as an index it will access the
-* correct nibble oriented value.
-
*****************************************************************************
*/

unsigned int Label_Remap[SNOOPY_ARINC_RXERS] =
       {0x0, 0x8, 0x4, 0xc,
        0x2, 0xa, 0x6, 0xe,
        0x1, 0x9, 0x5, 0xd,
        0x3, 0xb, 0x7, 0xf
       };

/*
-
*****************************************************************************
-* Rate_Selectable_Buses - provides information on which receive bus is
-* rate configurable.
-
*****************************************************************************
*/
static unsigned int Rate_Selectable_Buses[MAX_NUM_RATE_SELECTABLE_BUSES];

/*............................ FUNCTION DECLARATIONS
```

```
......................*/
/*
{@FDB}
-
*****************************************************************************
-* Name: Receive ARINC Data (public)
-*
-* Syntax: void Receive_ARINC_Data(void)
-*
-* Arguments: NONE
-*
-* Returns: NONE
-*
-* Description:
-* This procedure services the ARINC 429 receiver when data is received.
-* The receiver status registers are read to determine which receiver
-* FIFO has new data available and where to obtain the newly received
-* input data. The receiver's port register address and the received
-* label are then used to determine the proper ARINC 429 Multiword Rx
-* Msg Queue or ARINC 429 Single Word Rx List location in which to
-* place the received data. Based on the label and receiver channel,
-* the ARINC Application Specific Data for the ARINC word is used to
-* determine if the word is dependent on the SDI bits or not. If it
-* is SDI dependant, then the received word's SDI is compared to the
-* Unit's SDI via a call to Check Input SDI. If received word SDI code
-* matches the unit's SDI strapping or the ARINC word is not SDI
-* dependant, the word is accepted for further processing.
-*
-* If the accepted word is designated as re-mapped then the re-mapping
-* is done. This involves substituting the received label for the
-* re-map label in the ARINC Application Specific Data.
-* If the accepted word is designated for one of the Single Word
-* Lists, the word's ARINC 429 Single Word Rx Counter is reset and
-* the word is written into the ARINC 429 Single Word Rx List. The
-* word's freshness flag is set to FRESH.
-*
-* If the accepted word is designated as port selectable and the port
-* select has selected the port on which the word was received, then
-* the word is also written to the port select location in the ARINC
-* 429 Single Word Rx List. The port selectable word's counter is
-* reset and the freshness flag is set to FRESH.
-*
-* If the accepted word is designated to go into one of the ARINC 429
-* Multiword Rx Msg Queues and there is room in that designated queue
```

```
-* then the data is transferred to the queue via the Put Queue
-* function. If there is no room in the designated queue, then the
-* current new input data will be discarded.
-*
-* The ARINC 429 Multiword Rx Msg Queues users call the Get
-* ARINC Multiword function to retrieve the queued data.
-*
-* Special Notes:
-*
-* {@RTB}
-* Function ID: ARINC_HW_01
-* {@RTE}
-*
-
****************************************************************************
{@FDE}
*/
void Receive_ARINC_Data(void)
{
   Error_Handling_Proc_Type Error_Handling_Procedure;   /* Pointer to Error
proc*/
   Arinc_429_Word_Type      Arinc_Fifo_Word_32;         /* Received arinc word.
*/
   unsigned int             Rx_Error;                   /* Reported rx error.
*/
   unsigned int             Error_Handling_Proc_Index;  /* Error proc. index.
*/
   unsigned int             ARINC_Receiver_Number;      /* Receiver Channel.
*/
   unsigned int             Word_In_Fifo_Status;        /* Fifo status.
*/
   unsigned int             Snoopy_Rcver_Index;         /* Snoopy reciever
index*/
   unsigned int             Snoopy_Index;               /* Snoopy id.
*/
   unsigned int             Arinc_Fifo_Index;           /* Receiver fifo index.
*/
   unsigned int             Is_Rxer_Data_Corrupted;     /* Error flag.
*/
   unsigned int             Error_Status_Index;         /* Error array index.
*/
   unsigned int             Rxer_Index;                 /* Receiver index.
*/
```



```
/*Clear the Freshness bit in the Rxer_Error_Status indicating that
*/

/*snoopy error status has not been read for that register
*/

    for(Error_Status_Index = 0; Error_Status_Index < NUM_ERROR_STATUS_REGS;
Error_Status_Index++)
    {
        Rxer_Error_Status[Error_Status_Index].Freshness_Flag = FALSE;
    }
/*For each of the non-audio receivers as defined by the Max Receivers,
*/
/*process the receiver. Also, update the Receiver Operating counter.
*/
/*This Counter provides a means of measuring the last time valid word
*/
/*received on ARINC Receiver Channel.
*/

    for(Rxer_Index = 0; Rxer_Index < Max_NonAudio_Rx; Rxer_Index++)
    {
/*      Translate the logical input channel to the physical location of
*/
/*      the receiver.
*/

        Snoopy_Rcver_Index = ARINC_429_Receiver_Dsc[Rxer_Index].Location;
        Snoopy_Index = ARINC_429_Receiver_Dsc[Rxer_Index].Snoopy_Index_Number;

/* Event Driven Process requires unique identifier for each receiver
*/
/* channel. Defines in cs serial arinc pub.h and cs serial csdb pub.h
*/
/* defines each receiver channel for both ARINC and CSDB receivers. These
*/
/* defines are based on each snoopy having 16 ARINC receivers and 2 CSDB
*/
/* receivers. Numbers 0-31 are for ARINC receivers, 0-15 for snoopy[0] and
*/
/* numbers 16-31 for snoopy[1]. Numbers 32-35 are for the CSDB receivers
*/
/* with 32-33 for snoopy[0] and 34-35 for snoopy[1].
*/
```



```
    ARINC_Receiver_Number =
                        (Snoopy_Rcver_Index + LOW_WORD_POS_LIMIT *
Snoopy_Index);

/* If the receiver's status register indicates receiver error and
*/
/* Error Handling Enabled, otherwise process word.
*/

    if(Snoopy[Snoopy_Index]->RX_429_Error_Status & (1 <<
Snoopy_Rcver_Index))
    {
        Snoopy_Delay();
/*      Error Status Index is used as an index into Rxer_Error_Status.
*/
        Error_Status_Index = (ARINC_Receiver_Number >> DIVIDE_BY_4_SHIFT);
/*      The receiver status are grouped four receivers per register.
*/
/*      To Index into the correct status register a divide by 4 is need
*/
/*      on the receiver index. Reading this register clears the error
*/
/*      conditions for those 4 receivers.
*/

    if(Rxer_Error_Status[Error_Status_Index].Freshness_Flag == FALSE)
    {
        Rx_Error = Snoopy[Snoopy_Index]->
                        ARINC_RX_Error[Snoopy_Rcver_Index >>
DIVIDE_BY_4_SHIFT];
        Snoopy_Delay();
        Rxer_Error_Status[Error_Status_Index].Freshness_Flag = TRUE;
        Rxer_Error_Status[Error_Status_Index].Rxer_Status = Rx_Error;
    }

    else
    {
        Rx_Error = Rxer_Error_Status[Error_Status_Index].Rxer_Status;
    }

/*  To read error status for receiver, mask out status of other
*/
/*  receivers and right shift.
*/
```



```
    Rx_Error = ((Rx_Error >> (ERROR_REG_SHIFT *
                      (Snoopy_Rcver_Index & BIT_SCREEN_3))) &
BIT_SCREEN_15);

/* Check which type of error received, if Parity Error or Frame Error
*/
/* then data is corrupted else if Word length/Bit Rate Error process
*/
/* data in Fifo.
*/

    if(Rx_Error & BIT_SCREEN_3)
    {
        Is_Rxer_Data_Corrupted = TRUE;

/*      Clear Receiver Fifo with corrupted Data
*/

        Snoopy[Snoopy_Index]->RX_Fifo_Clear = (1 << Snoopy_Rcver_Index);
        Snoopy_Delay();
    }

    else
    {
        Is_Rxer_Data_Corrupted = FALSE;
    }

    if(ARINC_429_Receiver_Dsc[Rxer_Index].Error_Handling_Enabled == TRUE)
    {

/*      Call the procedure specified in the Error Handling Proc
*/
/*      passing the receiver number and status read.
*/

        Error_Handling_Proc_Index =
ARINC_429_Receiver_Dsc[Rxer_Index].Error_Handling_Proc_Index;
        Error_Handling_Procedure =
Error_Handling_Proc[Error_Handling_Proc_Index];
        Error_Handling_Procedure( Snoopy_Rcver_Index, Rx_Error);

    }
}
else
```



```
{
    Is_Rxer_Data_Corrupted = FALSE;
} /* END if((Snoopy[Snoopy_Index]->RX_429_Error_Status */

/*  If no error then process word in snoopy fifo for that receiver.
*/

    if(Is_Rxer_Data_Corrupted == FALSE)
{

/* Check if there is data in the FIFO, process the data.
*/

    Word_In_Fifo_Status = Snoopy[Snoopy_Index]->RX_Fifo_Ready;
    Snoopy_Delay();

    Arinc_Fifo_Index = Snoopy_Rcver_Index * FIFO_INDEX_OFFSET;

    Word_In_Fifo_Status = (Word_In_Fifo_Status >> Snoopy_Rcver_Index) &
FIFO_RX_STATUS_MASK;
    Snoopy_Delay();

/*  If the receiver has no new word, check next receiver.
*/

    if(Word_In_Fifo_Status == TRUE)
    {

/*  Clear Receiver Operating Counter to indicate word received
*/

    ARINC_429_Rxer_Operating[ARINC_Receiver_Number] = 0;

/*  Read the data Lower 16 bit from the FIFO.
*/

    Arinc_Fifo_Word_32.Arinc_Word_Ints.Arinc_Word_Low =
        Snoopy[Snoopy_Index]->ARINC.Rx_Fifo[Arinc_Fifo_Index];
    Snoopy_Delay();

/* Read the data Upper 16 bit from the FIFO.
*/

    Arinc_Fifo_Word_32.Arinc_Word_Ints.Arinc_Word_High =
```



```
       Snoopy[Snoopy_Index]->ARINC.Rx_Fifo[Arinc_Fifo_Index + 1];
    Snoopy_Delay();
    ARINC_Process_Word(Snoopy_Rcver_Index, Snoopy_Index,
Arinc_Fifo_Word_32);
          } /* END if(Word_In_Fifo_Status == TRUE) */
       } /* END if(Is_Rxer_Data_Corrupted == FALSE) */
    } /* END for(Rxer_Index =0; Rxer_Index< Max_NonAudio_Rx */
} /* END of CS Receive ARINC Data */
```


Appendix B Assembly Coding Example

Example 39. Source File Example

```
*............................ FILE PROLOGUE ..............................*

;{@CB}
;*********************************************************************
;* FILE NAME: Read_FLASH.asm
;*
;* PURPOSE:
;* This file contains the low-level function that reads data stored
;* in the FLASH memory.
;*
;* NOTES: None
;*
;* COPYRIGHT NOTICE:
;* Copyright 2001 Rockwell Collins Inc. Rockwell Proprietary Information
;*
;*********************************************************************
;{@CE}

*............................ FILE INCLUSION .............................*

*............................ MACRO DEFINITION ...........................*

*............................ TYPE DECLARATIONS ..........................*

*............................ OBJECT DECLARATIONS ........................*

    .mmregs

*............................ FUNCTION DECLARATIONS .......................*

;{@FDB}
;*********************************************************************
;* Name: Read_FLASH
;*
;* Syntax:
;* Read_FLASH (unsigned long Address, unsigned int * value)
;*
;* Arguments:
;*           Address : The physical flash address that will be read.
;*                     Located in the A register
;*           value : The address where the value read from flash
```

```
;*                          will be written to.
;*                          Located at *SP(0)ii
;*
;* Returns: None
;*
;* Description:
;*     The FLASH address is passed to this function as a parameter along
;*     with a pointer to the location where the FLASH data is going to
;*     be stored. This function returns nothing.
;*
;* Special Notes : <the Breakdown of the register usage could be put here>
;*
;* {@RTB}
;* FUNCTION ID: READ_FLASH_01
;* {@RTE}
;*
;*************************************************************************
;{@FDE}

.global _Read_FLASH

_Read_FLASH:

    .if __far_mode
    MVDK *SP(2),*(AR3)

    .else
    MVDK *SP(1),*(AR3)

    .endif

    PSHM PMST ; save PMST

    SSBX 1,INTM ; Disable interrupts by setting INTM

    STM PMST,AR2
    ANDM 0ffdfh,*AR2 ; clear overlay bit
    NOP
    NOP

    READA *AR3
    POPM PMST ; restore PMST
```



```
RSBX 1,INTM ; Enable interrupts by clearing INTM

.if __far_mode
FRET
.else
RET
.endif
```

- Note that the placement of the parameter is at SP(0). This is where the parameter must be before this routine is called. The call will cause the return address of the calling procedure to placed onto the stack. Therefore, depending if the call was a near or far, the function itself will find the parameter at either *SP(1) if near, or *SP(2) if far.
- Note that the placement of the parameter is at SP(0). This is where the parameter must be before this routine is called. The call will cause the return address of the calling procedure to placed onto the stack. Therefore, depending if the call was a near or far, the function itself will find the parameter at either *SP(1) if near, or *SP(2) if far.

