

Document Number 2630020
WSR-88D ROC
7 February 2008
Version - Final

Radar Product Generation System

(RPG)

C CODING STANDARDS

(WPI0051)

**SUBMITTED AND
APPROVED FOR
USE AS PRODUCT
BASELINE BY:**

Date: _____

**Steve Smith
Engineering Team Lead
WSR-88D Radar Operations Center**

TABLE OF CONTENTS

Introduction.....	3
Purpose.....	3
Definitions and Conventions.....	3
Document Organization.....	3
File Organization.....	4
Heading Section.....	4
Prologue.....	4
System / Local Include Files.....	5
Constant Definitions.....	5
Type Definitions.....	5
External Global.....	6
External Static.....	6
Function Prototypes.....	6
Program Section.....	7
Function Organization.....	7
Function Heading Block.....	7
Function Return.....	8
Include Files.....	8
Readability and Maintainability.....	9
Brace Style.....	9
Whitespace.....	10
Horizontal Spacing.....	11
Vertical Spacing.....	12
Indentation.....	12
Comments.....	12
Block Comments.....	12
One-Line Comments.....	13
In-Line Comments.....	13
Naming Conventions.....	13
Task/Library Filename Conventions.....	14
"main" Source Files.....	14
Ancillary Source Files.....	14
Library Public Header Files.....	15
Task/Library Private Header Files.....	15
Task/Library Module Name Conventions.....	15
API Functions.....	13
Intratask/Intralibrary Public Functions.....	16
File-Scope Functions.....	16
Definitions / Declarations.....	16
Constants.....	16

#define.....	17
Const Modifier.....	17
Enumeration Type.....	17
Macros.....	18
Structures.....	18
Variables.....	18
Numbers.....	19
Functions.....	19
Statements and Program Control.....	20
Statements.....	20
Program Control.....	21
Loops.....	21
if, if-else, and switch.....	21
Tests.....	23
Portability.....	24
References.....	25
Appendix A. Infrastructure Task/Library Prefixes.....	27
Appendix B. RPG Task/Library Prefixes.....	28

RPG C Coding Standards

1 Introduction

The RPG C Coding Standards (CCS) defines a set of coding standards and common practices related to developing software for inclusion into the Radar Product Generation (RPG) software baseline using the C programming language. The standards and practices defined in this document closely follow the ANSI C and POSIX standards.

1.1 Purpose

This document provides a road map to direct the RPG scientific applications software developers and implementers as they develop and implement their applications using the Common Operations Development Environment (CODE). Developing software using standard principles and practices increases portability reduces software maintenance costs and improves source code readability.

This document is not intended to be a C programming tutorial. It is assumed the reader has a sound understanding of the C language, its terminology, constructs and syntax.

1.2 Definitions and Conventions

For this document, we define *function* as any unit of code that exists in one C source file. A file may contain one or more functions. Functions within a file are generally logically connected. Include files are the exception since this type of file generally do not contain functions.

An identifier defined or declared outside all functions in the file is known as an *external* variable. If access to an *external* variable is restricted to the file in which the variable is defined, the variable is a *static* variable and is said to have *file scope*. An *external* variable that can be accessed across files is a *global* variable and is said to have *global scope*. A *task* or *program* consists of functions that are grouped together for a specific purpose.

1.3 Document Organization

The CCS begins with the general issues of file organization, readability, and maintainability. It continues with definitions/declarations, statements, and program control. Following that, the CCS addresses the issue of portability. Lastly, it includes references to documents used to create this document.

RPG C Coding Standards

2 File Organization

A source file or module contains one or more logically connected functions. Each function is organized into two sections: a heading section and a program section. Each section is separated by at least one blank lines or other type of delineation such as a line of asterisks beginning and ending with the comment symbol.

2.1 Heading Section

Below is the recommended ordering of the parts that comprise a program's heading section.

```
/* Prologue */  
/* System include files / Local include files */  
/* Constant definitions / Macro definitions / Type definitions */  
/* Global variables / Static global variables */  
/* Internal static functions */
```

2.1.1 Prologue

The prologue describes the file's contents. It should include a statement of the purpose of the file and its relation to the overall program. The prologue should mention any use or requirements of external files and any other information or dependencies on which the source code file or module relies. The following is the preferred template for the prologue portion of the heading section.

```
/*  
*****  
Description:  
    Describe the overall purpose of the functions within  
    this file and their relation to the overall program.  
  
Dependencies:  
    Describe any dependencies or requirements to  
    external files or information.  
  
Notes:  
    Describe any other pertinent information that would  
    be useful to a maintainer of this file.  
*****  
*/
```

Since Resource Control System (RCS) is used, the following information is at the start of the heading section.

RPG C Coding Standards

```
/*
 * RCS info
 * $Author: $
 * $Locker: $
 * $Date: $
 * $Revision: $
 * $State: $
 * $Log: $
 *
 */
```

The RCS template fields are automatically populated by RCS when the file is integrated into the RPG baseline.

2.1.2 System / Local Include Files

A list of included header files follows the prologue. Included header files have file-wide scope, so placing this information at the top of the file underscores this influence. The developer should comment any include file whose purpose is not obvious. Include only those header files that are necessary for that file. List system include files first, followed by local (developer created) include files.

2.1.3 Constant Definitions

The file scope constants should appear after the include files. Placing definitions at the top of the file allows for easier location and identification when searching for them. Constant definitions should be defined using *#define*. Constant names should be descriptive and appear in all upper case capitalization.

Example:

```
#define          INDEX_OF_REFRACTION    1.21
```

2.1.4 Type Definitions

Data type definitions (*typedef*) local to the file should follow the constant definitions. Place data type definitions in a header file when those types are global, used within several files. Type definition names should be appended with *_t*.

RPG C Coding Standards

Example:

```
typedef struct node {  
    int *data;           /* Pointer to node data. */  
    Node_t *next;       /* Pointer to next node in list. */  
} Node_t;
```

2.1.5 External Globals

External (to file) global variables are *global-scope* variables. Declare them in a file where they are initialized, all other files in which the variables are referenced should precede the storage class identifier with *extern*. Use descriptive names and at a minimum, the leading character of the name should be capitalized. To further emphasis these are global variables, it is good practice to identify these variables with leading characters identifying the task.

Example:

```
extern int PBD_start_of_volume_time; /* Used by pbd to store volume  
                                     time. */
```

2.1.6 External Static

External (to function) static variables are *file-scope* variables. Declare them in the module in which they are used. Precede the definition with the storage class identifier *static*. Use descriptive names and at a minimum, the leading character of the name should be capitalized.

2.1.7 Function Prototypes

Function prototypes inform the compiler about the existence of functions before they are defined. This allows the code to compile without having to ensure all functions are defined before they are called. Place function prototypes after the external static variables.

Functions used only in the module in which they are defined are static functions. Include the keyword *static* in the function prototype. Function names should be descriptive and at a minimum the first character of the function name should be capitalized.

Functions used in multiple files should be declared in a header file. Function names should be descriptive. The leading character(s) of functions used in multiple files should be capitalized and denote the file in which they are defined.

Example:

RPG C Coding Standards

```
PBD_process_header(); /* Function used by Process Base Data task. */
```

2.2 Program Section

The program section immediately follows the heading section. It contains the body of the source code in the file. It may also contain code for the one or more functions defined in the module.

2.2.1 Function Organization

2.2.1.1 Function Heading Block

Include a heading block before the code of every function. A function with little or no inputs whose purpose is obvious such as *is_time_valid ()* may not appear to need a heading block. However, information such as time units, time for what, and valid time limits can make future software modification or maintenance more intuitive and easier. The function heading block contains the information shown in the function heading block template given below.

```
/******  
  Description:  
  
  Input:  
  
  Output:  
  
  Returns:  
  
  Notes:  
  
******/
```

Following is the description of the function heading block items:

- A description of the function including its purpose.
- A list of function input arguments with descriptions, including any constraints on input arguments. Should also list any global variable input to the function which are not explicitly listed as a function argument.
- A list of function output arguments with description, including constraints on output arguments. Any argument which has its value(s) modified should be listed as an output argument. Should also include any global variable modified in the function which are not explicitly listed as a function argument.
- A description of the function's return value to the calling function. The description should include the meaning of special return values (e.g., error codes returned as a result of error conditions).
- Notes.

RPG C Coding Standards

The *Notes* section should contain the following:

- A discussion of the algorithm used, if applicable.
- The exception handling scheme used in the function.

Items in the function heading block that are not applicable need not be included. If included but are not applicable, state N/A.

2.2.1.2 Function Returns

The function return type should appear on the same line as the function name:

```
int MLB_open( int identifier, int flags, MLB_attr *attr );
```

Do not default the function return type to type `int`. If a function does not return a value, give it return type `void`.

2.3 Include Files

Include (header) files contain global constants, type definitions, variable definitions, function prototypes, etc. which are used by one or more C functions. Organize the header files functionally. For example, put definitions for separate subsystems or libraries in separate header files. Place non-portable code (i.e., code that would probably change if ported to different hardware) in a separate header file. Do not name header files with the same name as a system header file.

Use the ‘<’ and ‘>’ symbols to include system header files in your C source module and use double quotation for all other header files:

Example:

```
#include <stdio.h>           /* System header */
#include "my_header.h"      /* Application header file */
```

Do not use absolute or relative path names to point to your header files:

```
#include "/home_dir/inc/my_header.h"    /* Don't do this */
#include "../..../another_header.h"     /* Don't do this, either */
```

It is much better to use the `-I<dir>` C compiler command line parameter to inform the compiler the location of your header files:

```
gcc ... -I/my_header_dir ...
```

RPG C Coding Standards

Header files, particularly system header files, often include other header files. When this occurs, one or more header files may be included more than once. This is called multiple inclusion and is poor programming practice. To prevent multiple header file inclusion, use the following construct:

```
#ifndef    MULTIPLE_EXCLUSION_SYMBOL
#define    MULTIPLE_EXCLUSION_SYMBOL

    /* Body of header.h file here */

#endif    /* MULTIPLE_EXCLUSION_SYMBOL */
```

The multiple exclusion symbol is the header file name in uppercase with the suffix `_H`. For example, for header file `com_defs.h` the multiple exclusion symbol would be `COM_DEFS_H`.

3 Readability and Maintainability

3.1 Brace Style

The essential purposes of using braces are to delimit blocks of code, provide a limited scope to local variables, and to make a program's purpose and structure clear.

Two brace styles are common. One example is:

```
if ( nodeptr->data != 0 ) {
    nodeptr = nodeptr->forward_link;
}
```

The opening brace follows to the right of the condition statement, and the closing brace is aligned with the beginning of the condition statement. Code inside the braces is appropriately indented. This style is expanded when using compound block statements, as:

```
if ( nodeptr->data != 0 ) {
    nodeptr = nodeptr->forward_link;
}
else {
    nodeptr->data = 1;
}
```

Another popular style is:

```
if ( nodeptr->data != 0 )
{
    nodeptr = nodeptr->forward_link;
}
```

and

RPG C Coding Standards

```
if ( nodeptr->data != 0 )
{
    nodeptr = nodeptr->forward_link;
}
else
{
    nodeptr->data = 1;
}
```

For reasons of consistency and clarity, choose one of these two styles and adopt it when writing code.

NOTE: When modifying code that others have written, be careful to pay strict attention and follow the style already in use within the module or library. Having multiple coding styles within a function or file reduces readability of the code.

3.2 Whitespace

Use whitespace to make reading through source code easier, and to reflect such program constructs as block structure. Be judicious in your use of white space.

3.2.1 Horizontal Spacing

Appropriate spacing enhances the readability of variables, operators and statements. Place one space on either side of binary operators:

```
*speed = *miles / *hour;
```

In the next example, we remove the white space and not only is the code more difficult to read, but we also introduce an error, as the compiler will interpret `/*` as a beginning comment symbol.

```
*speed=*miles/*hour;
```

Use white space between comma-separated arguments in a list:

```
My_function ( int nyquist_vel, int unamb_range );
```

However, macro definitions with arguments must not have a white space between the name and the left parenthesis, or the C preprocessor will not recognize the argument list.

Splitting long strings of conditional operators onto separate lines and using appropriate whitespace also enhances code readability. Similarly, break elaborate `for` loops onto different lines. Notice the white space surrounding the parentheses:

RPG C Coding Standards

```
if ( radial[waveform_type] != CONTIGUOUS_SURVEILLANCE &&  
    radial[waveform_type] != CONTIGUOUS_DOPPLER &&  
    radial[waveform_type] != STAGGERED_PRF )  
{  
}
```

and

```
for ( cell = index_begin, data_start = 0;  
      cell < MAX_RANGE_CELLS;  
      cell++, data_start++ )  
{  
}
```

3.2.2 Vertical Spacing

Group together related program statements and use blank lines to separate them from other program statements. Use one or more blank lines or a combination of blank lines to separate the source code of one function from another.

```
int get_average (int num_1, int num_2)  
{  
    int average;  
  
    average = (num_1 + num_2) / 2;  
    return (average);  
}  
  
/*****/  
Description:  
    Does something.  
  
Inputs:  
    score_1 - ...  
    score_2 - ...  
  
Outputs:  
  
/*****/  
  
void do_something (int score_1, int score_2)  
{  
    #define MINIMUM 0  
    #define MAXIMUM 100  
    int test;
```

RPG C Coding Standards

```
int      largest = MINIMUM;
int      smallest = MAXIMUM;

test = get_average (score_1, score_2);

if (test > largest)
{
    largest = test;
}
else if (test < smallest)
{
    smallest = test;
}
}
```

3.2.3 Indentation

Indentation helps reveal the logical structure of your code. Use two, three, or four spaces when indenting your code. Be consistent with whatever number of spaces you select. The examples in this document use three spaces for indentation. Avoid using tabs to indent.

3.3 Comments

Comments in source code should describe what is happening, how it is being done, what parameters mean and any restrictions or (known) bugs. Avoid comments that simply restate the code. Comments that disagree with the commented code are more confusing than helpful. Use short comments to describe what ("compute mean value"), rather than how ("sum of values divided by n").

Justify unusual code by commenting it; explain the unusual behavior and describe why you used the particular approach.

3.3.1 Block Comments

Different forms of comments are appropriate for different places in source code. Within a code segment, the following two forms of block comments are preferred:

```
/*
    This is a comment in block form.
    Comment text should be aligned uniformly.
    Opening slash-star / closing star-slash are alone on a line.
*/
```

or

RPG C Coding Standards

```
/*  
 * This is a comment in block form.  
 * Comment text should be aligned uniformly.  
 * Each line begins with an asterisk'.  
 */
```

Outside of code segments or functions, the following form is preferred:

```
/*  
 * This is a comment in block form. *  
 * Comment text should be aligned uniformly. *  
 * Surround comment with asterisks (trailing asterisk optional *  
 * on lines 2 through n-1). *  
 */
```

Never use “//” to denote comments. Although this may be acceptable with some compilers, it is not ANSI C compliant and is not portable.

3.3.2 One-Line Comments

Another comment form is a one-line comment. Generally, one-line comments are short, and describe a short code fragment:

```
/* Check if Bypass Map Request was solicited. */  
if ( bypass_map_request_pending )  
    [...]
```

3.3.3 In-Line Comments

Comments that are very short often appear on the line of code to which they refer. Use spaces to separate those comments from the code itself. It is good practice to align several short comments associated with a block of code, as shown

```
if ( argc <= 2 ){  
    error_value = INCOMPLETE; /* Wrong number of arguments */  
}  
else{  
    error_value = SUCCESSFUL; /* Has minimum # of arguments*/  
}
```

All local variables, except trivial ones, should be explained with in-line comments. Static and external variables should always be explained with in-line comments.

3.4 Naming Conventions

There are some general rules and guidelines for the RPG project:

RPG C Coding Standards

- Names with leading and trailing underscores are reserved for system use, and should not be used by the programmer in that form, or as part of a user-created name.
- Constants defined by *#define* should be in all CAPITAL letters. The same is true for constants within an enumerated (*enum*) block.
- Names that differ by case only, such as bar and Bar, should be avoided.
- Do not use names that might conflict with standard library names.
- Type defined (*typedef*) names should have an appended *_t* to denote they are of a defined type (e.g., *node_t*).
- Use starting capital letters for global variables.
- Do not use initial capital letters for non-global variables.
- Use initial capital letters for module level global variables and function names.
- Use more than one initial capital letter (typically 2 or 3) for truly global variables and function names.
- Words in variable names should be separated by underscores (not by capital letters) and the variable names should be descriptive.
- Local variables serving as loop indices and the like do not have to be descriptive but short, descriptive names may prove to be helpful.
- The maximum length of variable names should be limited to 24 characters.

3.4.1 Task/Library Filename Conventions

This section describes the convention for naming the C source (*.c) and header (*.h) files that comprise a given Task or Library. A key element of this convention is the use of unique alphanumeric prefixes assigned to each Task or Library. A current list of these prefixes is maintained in Appendix A and Appendix B.

3.4.1.1 "main" Source Files

The Task C source file that includes the obligatory "main" function will be named either

prefix.c OR *prefix_main.c*

where "prefix" is as described above.

Examples:

psv_main.c - the "main" for the Product Server task
hci.c - the "main" for the Human Computer Interface (HCI) task

3.4.1.2 Ancillary Source Files

For a given Task or Library, all ancillary C files should be named as follows:

prefix_.c*

RPG C Coding Standards

where "*" cannot be "main".

Examples:

```
psv_process_events.c  
hci_timer_proc.c
```

3.4.1.3 Library Public Header Files

Public header files are typically associated with Libraries (rather than Tasks). If a Library includes a public header file, that file shall be named as follows:

```
prefix.h
```

Example:

```
rpgc.h -RPG C Algorithm Support Library
```

3.4.1.4 Task/Library Private Header Files

Any header files that are "private" to a given Library or Task should be named as follows:

```
prefix_*.h
```

Examples:

```
psv_def.h  
hci_product_colors.h
```

3.4.2 Task/Library Module Name Conventions

This section describes the convention for naming the modules within a given source file. A key element of this convention is the use of unique alphanumeric prefixes assigned to each Task or Library. Current lists of these prefixes are maintained in Appendix A and Appendix B.

3.4.2.1 API Functions

Since "public" functions must be documented by an Application Programming Interface (API), these functions are referred to as "API" functions. APIs allow you to program to a pre-constructed interface (the API) instead of programming a device or piece of software directly.

RPG C Coding Standards

These are typically functions provided for the use of software outside the scope of the current Library.

API function names must incorporate an uppercase prefix as follows:

```
PREFIX_*
```

Example:

```
RPGC_get_inbuf_by_name()
```

3.4.2.2 Intratask/Intralibrary Public Functions

A given Task or Library will typically include functions that are "public" to the software that comprises the Task or Library but are not intended to be used by software outside the scope of the current Library.

These public functions should incorporate a non-API uppercase prefix as follows:

```
LOCALPREFIX_*
```

Example:

```
WAN_initialize();
```

3.4.2.3 File-Scope Functions

In keeping with other sections of these Guidelines, file-scope functions should be named with uppercase first letter only.

Example:

```
Open_all_lb(); /* defined in orpgda.c */
```

4 Definitions / Declarations

This section describes the rules to follow when you define or declare constants, macros, variables and structures. Apply the *static* keyword to all global variables and functions local to single files.

4.1 Constants

Avoid hard coding numerical constants. Using a descriptive name instead of a numerical

RPG C Coding Standards

constant makes software maintenance much easier since constant values can be changed at one place. Using the numerical constants -1, 0 and 1 are acceptable when initializing variables. Use uppercase characters for constant names and separate words in the name with underscores. Include comments when the meaning of the constant is not clear. Use *NULL* when initializing pointers or when testing a pointer (rather than 0). Define simple character constants as character literals rather than numbers. If non-text characters are unavoidable, use their octal or hexadecimal form (e.g.: '\014' (octal); '\x0c' or 0x0c (hexadecimal)). Still, such usage is most often machine-dependent; be aware of the word sizes on the architecture in question.

4.1.1 #define

The *#define* construct (constant macro) instructs the C preprocessor to replace subsequent instances of the identifier with its value.

```
#define INIT_VALUE      0      /* Initial value */
#define MAX_STR_LEN    80      /* Maximum string length */
```

Use the *#define* preprocessor directive for values needed to dimension arrays, otherwise use the *const* qualifier. By doing so, better type checking is provided by the compiler and the value is visible to the debugger.

4.1.2 Const Modifier

One advantage of defining constants with the *const* qualifier is that it allows a debugger to access the constant value. However, a constant defined with *const* may not be used to dimension arrays.

```
const int MAX_DAYS = 7;
const char WELCOME_MSG[] = "Glad you're aboard";
```

4.1.3 Enumeration Type

You may use an *enum* to place constants together in a logical group or when actual values for the constants are unimportant. The first value in an *enum* is 0 (zero) unless otherwise specified. As in the other constants, use uppercase characters to construct the constant name. Notice the alignment and indentation that improve readability.

```
enum waveform
{
    CONTIGUOUS_SURVEILLANCE, /* Value is 0, since not specified */
    CONTIGUOUS_DOPPLER,
    NO_AMBIGUITY_RESOLUTION,
    BATCH,
    STAGGERED_PRF           /* Value is 4 */
};
```

RPG C Coding Standards

```
enum buffer_operation_result
{
    UNKNOWN -2,          /* Assigned value -2 so ... */
    ERROR,
    EMPTY,               /* This constant's value is 0 and ... */
    FULL,
    OKAY                 /* This constant's value is 2 */
};
```

4.2 Macros

Avoid function macros and replace them with standard functions. Use of macros is acceptable if performance is an issue.

4.3 Structures

Structures enhance the logical organization of your data and are often good candidates for type definitions (*typedef*). Fully parenthesize structure initializations with braces (do not use the empty initializer { }). Make your defined type name have as a minimum a leading upper case character and append “_t” to the name to indicate the type definition.

```
typedef struct radial_type
{
    enum waveform wave;          /* processing type */
    float velocity_resolution;   /* 0.5 or 1.0 m/s */
    short pulse_width;          /* Long pulse(2) or short(1) */
} Radial_type_t;
```

4.4 Variables

Place unrelated declarations of variables of the same type on separate lines. Related variables may be grouped. Include a comment for each variable if the meaning of the declared variable is not obvious from its name. If access to an external variable is restricted to the module in which the variable is defined (i.e., the variable is static), you must begin the variable definition with the keyword *static*. An external identifier that can be accessed across modules is a global variable and usually declared with the keyword *extern*. It is good practice to align names, values, and comments underneath each other. Examples are shown below.

```
int alarm_priority;          /* Alarm ranking. Used by scheduler */
int alarm_state;            /* Is alarm off, on standby or active? */
int short x, y, z;         /* Cartesian coordinates */

static char *TaskName;     /* Name of the current task. The static
                           keyword restricts access to this
                           variable to the module where it is
                           defined. */
```

RPG C Coding Standards

```
extern long All_task_ids[]; /* List of ids for all tasks. Since this
                             is a declaration and not a definition,
                             no need to dimension array */
```

When defining a variable to a pointer, associate the pointer qualifier '*' with the variable name, not the variable type.

```
char *ptr_to_char = NULL; /* Good practice */
char* ptr_to_char; /* Bad practice */
```

Explicitly initialize a variable whose initial value is important. Do not assume that variables are automatically initialized to zero.

4.4.1 Numbers

Include at least one digit on either side of the decimal point for floating point variables. Start hexadecimal variables with 0x and use uppercase for A-F.

```
const float GRAVITY_EFFECT = 0.937;
const int   MAX_WEIGHT     = 0x1F4A;
```

When initializing long variables with constants, use an uppercase L:

```
long starting_slant_range = 1000L; /* Clearly a long 'L' */
long starting_slant_range = 1000l; /* Looks like the number 1000l' */
```

When initializing float variables with enumerated constants, use lowercase f.

```
float starting_slant_range = 1.0f;
```

4.5 Functions

Place the opening brace at the end of the function name or by itself on a separate line aligned with the first character of the function type. Use the keyword `static` to restrict access to a function to the module in which the function is defined. Indent local declarations within a function and separate them from the body of the function by appropriate whitespace. For example:

```
int MLB_open( int identifier, int flags, MLB_attr *attr )
{
    MLB_struct *mlb;
    int index;          /* MLB index */
    ...
}
```

Comment each local variable unless it is clear by name. Typically a loop counter such as 'i' is understood for what it is so commenting it is not necessary. Give careful thought to future software modification and maintenance before assigning any variable a nondescript name and

RPG C Coding Standards

not include a comment. You may define local variables within the block with smallest scope:

```
if( radial_status == GOOD )
{
    int i; /* i is defined for the "if" block */

    for( i = 0; i < LENGTH; i++ )
    {
        ...
    }
}
```

Do not use local declarations that override higher level definitions/declarations of the same name.

5 Statements and Program Control

5.1 Statements

Place each program statement on a separate line:

```
task_name = get_current_task (...); /* Retrieve name of current task */
push_name (task_name);           /* and place on name stack */
```

No individual program line should exceed 80 characters in length. If a program statement exceeds this limit, break it into multiple lines (not necessarily multiple statements).

An embedded assignment statement is a form of side effect. Rarely use embedded assignment statements. There are some instances, however, where there is no cleaner way to accomplish what you want without making the code bulky. The following are a couple of common examples of embedded assignments:

```
while ( (c = getchar()) != EOF )
{
}

If ( (obj_ptr = malloc (number, size)) == NULL )
{
}
```

Remember that the operators increment (++) and decrement (--) count as assignments. Although allowed, we discourage embedding increment and/or decrement assignments. They are often a source of error and can be difficult to debug.

Avoid using the *goto* statement. It can be usefully employed to break out of several levels of *for/while/switch* nesting (though you should rethink and redesign any code that meets this condition):

RPG C Coding Standards

```
for (...)
{
    while (...)
    {
        .
        .
        .
        if (disaster)
        {
            goto error;
        }
    }
}

error: /* clean up this mess */
```

If a `goto` is used, align it one tab stop to the left of the label that follows (so as to set it off and make it recognizable).

5.2 Program Control

5.2.1 Loops

Place null bodies of `for` or `while` loops on a separate line, and comment them so that it is clear a null loop body is intended.

```
for (i = 0; product_ids[i] != requested_product; i++)
{
}; /* Null Loop Body intended*/
```

Although not required, we strongly recommend the use of braces in `for` and `while` loops where simple statements follow the condition.

```
while (i < total) /* This is okay, but ... */
    part_codes[i] = tmp_codes[i];

while (i < total) /* ... this is recommended */
{
    part_codes[i] = tmp_codes[i];
}
```

The penalty is only one or two extra lines and the code is more readable and easier to interpret.

5.2.2 if, if-else, and switch

Just as with loops, we strongly recommend the use of braces in conditional statements (`if`, `if-else`, and `switch`) where simple statements follow the condition. Again, it improves readability

RPG C Coding Standards

and maintainability. However, if the statement following the condition is compound, all statements are “considered” compound and thus, should be surrounded by braces (called fully bracketed syntax).

```
if (signal_received == FALSE) /* While this is accepted ... */
    resend_signal ();

if (signal_received == FALSE) /* ... this is recommended */
{
    resend_signal ();
}

if (radar_state == INVALID) /* Braces used here since the */
{                             /* else clause is compound */
    change_state = TRUE;
}
else
{
    change_state = FALSE;
    retrieve_state_attributes ();
}
```

Always use braces with an if-if-else construct. This ensures that parsing is done correctly and clauses meant to be associated with one condition are correctly associated. In the following example, we want the *else* clause to associate with the first *if* statement and we indent it as such. However, because there are no braces to block the *if-if-else* properly, the compiler will place the *else* clause with the second *if* statement.

```
if (altitude > threshold) /* Too high, but check tmp 1st */
    if (temperature < min_temp) /* Temp is too low - not good */
        location = BAD; /* This location won't do */
else /* Altitude okay so this */
    location = POSSIBLE; /* location may work */
```

Instead, code it like:

```
if (altitude > threshold) /* Too high, but check tmp first */
{
    if (temperature < min_temp) /* Temp too low - not good */
    {
        location = BAD; /* This location won't do */
    }
}
else /* Altitude okay so this */
{ /* location may work */
    location = POSSIBLE;
}
```

The code above is clear to both people and the compiler. There is no question about where the *else* clause belongs.

RPG C Coding Standards

The *switch* statement can cause problems because it is possible to "fall through" to the next condition after one condition has been met. Sometimes this is a desirable feature. Make sure you comment its use.

```
switch ( expression )
{
    case A:
        statement;
        break;

    case B:
        statement;
        /* FALL THROUGH */
    case C:
        statement;
        break;

    default:
        break;
}
```

5.2.3 Tests

Defaulting tests for non-zero is generally a bad idea. We recommend that explicit tests against a set return flag be coded, as:

```
if ( (buff_ptr = carve_buffer_space (length)) != FAIL )
```

rather than

```
if ( (buff_ptr = carve_buffer_space (length)) )
```

because an explicit test will help later when the constant value for FAIL is changed from 0 to another value (-1, for example) or the return value for the function (*carve_buffer_space* in the example) is changed to 0 (zero).

Use explicit comparisons to reflect the numeric (not boolean) nature of a test:

```
if( (queue_ptr % MAX_QUEUE_SIZE) == 0 )
```

instead of

```
if ( !(queue_ptr % MAX_QUEUE_SIZE) )
```

One common practice for using a boolean type is to include one of the following in a globally included header file:

RPG C Coding Standards

```
typedef int    bool;

#define  TRUE    1
#define  FALSE   0
```

or

```
typedef enum { NO = 0, YES } bool;
```

Generally, we suggest you test boolean values for inequality with 0 (FALSE, etc.) rather than for equality with 1 (TRUE, etc.). Most functions, BUT NOT ALL, are guaranteed to return 0 (zero) if false, but only non-zero if true. For example,

```
if ( func() != FALSE )
```

is usually better than

```
if ( func() == TRUE )
```

However, when you explicitly define TRUE, you may use the second form (`func() == TRUE`) instead of the first form (`func() != FALSE`) if it "makes more sense" to do so. Renaming the function or macro expression may be better, where possible so that the meaning is obvious without a comparison to TRUE or FALSE (`isvalid()`, for example).

6 Portability

For this document, portable means that a source code file can be compiled and executed on different machines with the only changes being the possible inclusion of different header files and the use of different compiler flags. The header files contain `#define` and `typedef` constructs that may vary from machine to machine (new machine may mean different hardware, a different operating system, a different compiler, or any combination of these).

Some important issues:

- Recognize that some things are inherently non-portable. Try to avoid these wherever possible.
- Try to organize machine-independent code in separate files from machine dependent code.
- NULL pointers are not always stored with all bits zero, and so may not compare equally with a variable that has a value of zero.
- Data alignment is an important consideration. Various machines may begin addresses at even numbers, while others may do this but restrict the valid addresses to multiple-of-four addresses.
- Recognize that some machines are little-endian and some are big-endian. Byte-ordering in words (and further, word ordering) is important.

RPG C Coding Standards

- Bit shifts and bit masks are affected by word size. Do not assume all machines have the same convention.
- Be familiar with existing library functions and definitions, but do not depend on them. They can be changed any time.
- Use explicit casts when in doubt.

7 References

We used several documents, articles and books when creating the RPG C Coding Standards document. Some text is used verbatim and other text we reworded.

- The C Programming Language, 2nd Ed., Kernighan & Ritchie, Prentice Hall, 1988
- Tornado User's Guide (UNIX Version) 1.0, WindRiver Systems, 1995
- Advanced C Tips and Techniques, Paul Anderson and Gail Anderson, Hayden Books, 1989
- NASA's Software Engineering Laboratory's C Style Guide
- Indian Hill C Style and Coding Standards paper

The Bibliography from NASA's Software Engineering Laboratory's C Style Guide:

- Atterbury, M., ESA Style Guide for 'C' Coding, Expert Solutions Australia Pty. Ltd., Melbourne, Australia (1991)
- Computer Sciences Corporation, SEAS System Development Methodology (Release 2) (1989)
- Indian Hill C Style and Coding Standards, Bell Telephone Laboratories, Technical Memorandum 78-5221 (1978)
- Kernighan, B., and Ritchie, D., The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1978)
- Minow, M., A C Style Sheet, Digital Equipment Corporation, Maynard, Massachusetts
- Oualline, S., C Elements of Style, M&T Publishing, Inc., San Mateo, California (1992)
- Wood R., and Edwards, E., Programmer's Handbook for Flight Dynamics Software Development, SEL-86-001 (1986)

References from the Indian Hill C Style and Coding Standards Paper

- B.A. Tague, C Language Portability, Sept 22, 1977. This document issued by department 8234 contains three memos by R.C. Haight, A.L. Glasser, and T.L. Lyon dealing with style and portability.
- S.C. Johnson, Lint, a C Program Checker, Unix Supplementary Documents, November 1986.
- R.W. Mitze, The 3B/PDP-11 Swabbing Problem, Memorandum for File, 1273-770907.01MF, September 14, 1977.
- R.A. Elliott and D.C. Pfeffer, 3B Processor Common Diagnostic Standards- Version 1, Memorandum for File, 5514-780330.01MF, March 30, 1978.

RPG C Coding Standards

- R.W. Mitze, An Overview of C Compilation of Unix User Processes on the 3B, Memorandum for File, 5521-780329.02MF, March 29, 1978.
- B.W. Kernighan and D.M. Ritchie, The C Programming Language, Prentice Hall 1978, Second Ed. 1988, ISBN 0-13-110362-8.
- S.I. Feldman, Make -- A Program for Maintaining Computer Programs, UNIX Supplementary Documents, November 1986.
- Ian Darwin and Geoff Collyer, Can't Happen or /* NOTREACHED */ or Real Programs Dump Core, USENIX Association Winter Conference, Dallas 1985 Proceedings.
- Brian W. Kernighan and P. J. Plauger The Elements of Programming Style. McGraw-Hill, 1974, Second Ed. 1978, ISBN 0-07-034-207-5.
- J. E. Lapin Portable C and UNIX System Programming, Prentice Hall 1987, ISBN 0-13-686494-5.
- Ian F. Darwin, Checking C Programs with lint, O'Reilly & Associates, 1989. ISBN 0-937175-30-7.
- Andrew R. Koenig, C Traps and Pitfalls, Addison-Wesley, 1989. ISBN 0-201-17928-8.

RPG C Coding Standards

Appendix A. Infrastructure Task/Library Prefixes

CSS	Client/Server Support Library
CS	Configuration Support Library
DEAU	Data Element Attribute Utility Library
EN	Event Notification Services Library
LB	Linear Buffer Library
LE	Log Error Library
MALRM	Multiple Alarm Library
MISC	Miscellaneous Services
NET	Network Services Library
RMT	Remote Tool
RSS	Remote System Services
SMIA	Structure Metadata Information Applications Library
STR	String Manipulation Library
SUPP	Supplemental Services

RPG C Coding Standards

Appendix B. RPG Task/Library Prefixes

CRDA	Control RDA
HCI	Human/Computer Interface Task
IPI	Initialize Product Information
MNGRED	Manage Redundant
MNGRPG	Manage RPG
MRPG	Monitor RPG
ORPGADPT	Adaptation Data Support Library
ORPGADPTSV	Adaptation Data Support Library
ORPGADPTU	Adaptation Data Support Library
ORPGALT	Alert Threshold Table Support Library
ORPGCCZ	Clutter Censor Zone Support Library
ORPGCFG	Configuration Support Library
ORPGCMI	Communications Manager Interface Library
ORPGCMP	Data Compression Support Library
ORPGBDR	Base Data Replay Library
ORPGDA	ORPG Data Access Library
ORPGDAT	Data Attribute Table Support Library
ORPGINFO	RPG Info Datastore Access
ORPGDBM	Data Base Manager Support Library
ORPGEDLOCK	Edit Lock Support Library
ORPGGDR	Generic Radial Support Library
ORPGGST	RPG Status Support Library
ORPGTASK	Task Routines

RPG C Coding Standards

ORPGTAT	Task Attribute Table Access
ORPGLOAD	RPG Load Support Library
ORPGMGR	Manage RPG Support Library
ORPGMISC	RPG Miscellaneous Library
ORPGNBC	Narrowband Control Support Library
ORPGPAT	Product Attribute Table Support Library
ORPGPGT	Product Generation Table Support Library
ORPGPRQ	Product Request Support Library
ORPGRAT	RDA Alarm Table Support Library
ORPGRDA	RDA Status and Control Support Library
ORPGRED	FAA Redundant Support Library
ORPGSITE	RPG Site Information Support Library
ORPGSMI	Support Metadata Information Library
ORPGSUM	Scan Summary Data Support Library
ORPGVCP	VCP Support Library
ORPGVST	Volume Status Support Library
OTR	One Time Request
PBD	Process Base Data
PRM	Process Remove
RMS	Remote Monitoring Subsystem
MNTTSK	RPG-Specific Maintenance Tasks
UMC	User Message Conversion