**Version History**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***Ver. No.*** | ***Authors*** | ***Date*** | ***Reviewers*** | ***Review Date*** | ***Release Date*** |
| 1.0 | Application Development Team | 27-Aug-2018 | QMF | 31-Aug-2018 | 03-Sep-2018 |
| 2.0 | Application Development Team | 10-Dec-2019 | QMF | 13-Dec-2019 | 16-Dec-2019 |
| 3.0 | Application Development Team | 02-Nov-2020 | QMF | 06-Nov-2020 | 10-Nov-2020 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Change History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Ver. No.*** | ***Section*** | ***Date*** | ***Change Information*** | ***RFC No.*** |
| 1.0 | All | 03-Sep-2018 | New Release | - |
| 2.0 | All | 03-Sep-2019 | New Release | - |
| 3.0 | All | 02-Nov-2020 | Annual Release | - |
|  |  |  |  |  |
|  |  |  |  |  |

**STANDARDS FOR CODING IN C**

1. **Objectives**

The objective of this document is to define coding standards for programs developed in ‘C’.

1. **Scope**

This standard applies to programs developed in ‘C’.

1. **References to (checklists, forms, guidelines, lists, standards, Templates, other processes)**

| ***Item*** | ***Description*** | ***ID*** |
| --- | --- | --- |
| *Checklists* | *--* | *--* |

1. **Document Structure, Glossary of Terms, etc.**

This document describes various formats for:

* Format for the Source Code Files
* Rules for Indentation
* Globals
* Function Size
* Naming Conventions
* Conventions for Comments in the Code
* Source Code Compatibility
* Error Messages
* 14. Debug Messages
* Additional Conventions
* A sample C code file

1. **Format for the Source Code Files**
2. The format for files containing source code and for headers will be as follows:

* Header
* #include Statements
* Literal Definitions
* MACRO Definitions
* Local Structure Definitions
* PRIVATE Variable Definitions
* Function Definitions

1. It is recommended that conditional compilation statement be put around the #include statements in the source code files

## Format of the Source Code File Header

1. A C source code file should contain the following information at the start of the file as Program Header. The information appears as a comment

* Copyright Notice
* The name of the file
* The initials of the programmer who wrote the file and the group to which he/she belongs to
* Date of creation
* The development platform
* The target platform(s)
* Names of the functions in the file that are entry points
* List of the macros that are used in this file and where they are defined
* Dates of updations with a brief description of the changes made and the initials of the person making the changes. Entries in this section should be numbered (see Commenting Changes to the Code) and ordered chronologically

## Copyright Notice

1. The copyright notice should contain the following text:

Copyright (c) 1994 <Company Name>

All Rights Reserved.

THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF

<Company Name>

The copyright notice above does not evidence any actual or

Intended publication of such source code

## Additional Header Files Conventions

1. The header file should define a pre-processor symbol when compiled. The header file code should be compiled only if that symbol is not defined previously

#ifndef \_MYHEADER\_H

#define \_MYHEADER\_H

the contents of myheader.h

#endif

1. The header file should not contain any executable C statements
2. **Rules for Indentation**
3. Each block of code should be indented with respect to the outer block. The indentation should be 4 places. Do not use tabs for indentation. Large program blocks should end with endblock comment. The switch statement should be indented the following way :

switch(<expression>)

{

case <value1> :

<statement>

.

.

case <value2> :

<statement>

.

} /\* endswitch \*/

1. Try to avoid using one switch statement within another
2. **Function Size**
3. The average function size should be sixty source code lines. Under no circumstances should the function size be more than hundred source code lines
4. Additionally following things should be kept in mind to decide the optimum function size
5. The level of indentation should not exceed six levels
6. A single block of code should not exceed twenty to thirty lines
7. **Globals**
8. All global variable definitions must be centralized into a single compiled source file, or atleast a set of source files that contain only global variables and a function that initializes them
9. Comments should accompany declaration of every global variable stating the usage of the variable. Try to use as few global variables are possible. If a global is used by functions inside a single file, restrict its use by making it static
10. **Naming Conventions**
11. The name of the variable should reflect the data stored/referred by the variable. Hungarian Notation should be used

## *Variables & User Defined Data Types*

1. The format of a C variable or a C function should be: type format name
2. The type component gives the data-type of the value represented by the variable or the value returned by the function. The format component should be used if the value represented or returned is an array or a pointer. The name component is the user defined name of the variable. It is recommended to be as meaningful as possible

|  |  |
| --- | --- |
| ***Data Type*** | ***Notation*** |
| char | c |
| float | f |
| double | d |
| int | i |
| long int | l |
| short int | s |
| void | v |
| structure | t |
| class | o |
| widget | w |
| swidget | Sw |
| x-type | X |

|  |  |
| --- | --- |
| ***Format*** | ***Notation*** |
| pointer | P |
| array | A |

1. Avoid using underscores in the variable name. Instead make first letter of each word in the name capital
2. If a variable is unsigned add the prefix u to it
3. The names of the global variables should start with g
4. Use the variables i, j ... n as loop counters
5. Do **not** use similar looking names in the same scope
6. Do **not** use similar sounding names in the same scope
7. The length of the name of the variable should not exceed 30
8. Structure names should have the prefix tag
9. Enumerated type names should have the prefix enm

## *Macros*

1. All macro names should be in capital letters. Words inside the name should be separated by an underscore
2. **Conventions for Comments in the Code**
3. All comments should be in plain English and written as you write code, or even before you write code. Make sure that the comment and the code agree
4. The comment should be indented at the same level at which the corresponding code is indented. Each comment will start with a code indicating the type of the comment

|  |  |
| --- | --- |
| ***Comment Type*** | ***Code*** |
| (1) Module Header | MH |
| (2) Globals Description | GL |
| (3) Function Header | FH |
| (4) Code Comment | CC |
| (5) Change Comment | NC |

## Commenting a Function

1. Each function will have the following information associated with it:

* Prototype
* Purpose
* Input arguments and their semantics
* Modifiable arguments (try to use as few of them as possible) and their semantics
* Return value and its semantics
* Algorithm (optional) is provided if the function is big (say more than 10 lines)
* The logic is complicated
* The logic is not apparent (by just looking at the purpose and the code)

1. Side effects (optional) Side effects should also list the global variables that are modified by this function.
2. The names of the functions that are called from this function
3. The names of the functions that call this function (the list may not be exhaustive)
4. Items 2 to 9 are collectively called as function header

**Example:**

/\*

prototype

\*/

int iProcessData(int iCommand, char \*cpData, int \*ipDatalen);

/\*

Purpose : This function performs compresses/uncompresses the data supplied.

Inputs : Command denotes the operation to be performed on the data. Legal values are

CRUNCH\_IT

EXPAND\_IT

Modifiable Values Supplied:

cpData pointer to the buffer where the data is stored. will contain the processed data on return.

ipDatalen points to a location where length of the data is stored. will contain

the length of the processed data on return.

Return Value :

The function will return an integer indicating the result of the operation.

It returns :

SUCCESS the operation succeeded

INVALID\_COMMAND the command parameter is not correct

FAILED could not carry out the operation

Algorithm :

SWITCH command

CASE CRUNCH\_IT

compress the spaces in the data

FOR every word in the line

IF the word is not in the hash table

add the word to the hash table

IF unsuccessful

RETURN FAILED

ENDIF

ENDIF

replace the word by its table entry number

ENDFOR

update the size of the compressed text

RETURN SUCCESS

CASE EXPAND\_IT

WHILE there is more data to uncompress

get the next hash entry number from the data

IF entry does exists in the hash table

RETURN FAILED

.

.

ENDIF

replace the entry number in the data with the word

ENDWHILE

RETURN SUCCESS

DEFAULT

RETURN INVALID\_COMMAND

ENDSWITCH

Side effects:

1. If the function fails to perform the specified operation the data buffer and the data lengths remain in inconsistent state. The hash table gets updated with every new word found in the data

This routine calls the following routines:

strtok available in the library

gather\_spaces from the file utils.c

find\_word

add\_word from the file hash.c

1. This function is called by:

compress\_file to compress a file line by line in file

compress.c

expand\_file to expand a compressed file line by

line in file expand.c

\*/

## Commenting Iterative Statements

1. Each iterative statement may contain a comment header at the start of the looping statement. This comment header is optional and should be included for complex loops (large loop body, complex or non-apparent looping condition)
2. The comment header should contain:

* The precondition
* What each iteration does
* The post condition

1. If it is not clear from the code how the post condition is reached from the precondition, it should be explained

## Commenting the IF Statement

1. Each conditional statement may also be commented. The commenting is necessary for conditional statements with complex or non-apparent conditions
2. The comment should include:

* The condition
* What happens if it is true
* What happens if it is false

## Commenting Changes to the Code

1. All changes made to the code should be properly commented using the change comment. In the change comment, NC should be followed by a number. This number should be the number under which this change is recorded in the module header
2. If code is to be added, then

* A comment should precede the added code stating the date of the change and brief explanation

1. If code is to be removed, then

* It should not be physically deleted from the file. Instead it should be commented out with date and explanation

1. If code is to be changed, then

* The previous code should first be duplicated and then one copy is commented out with date stamp and the explanation and the other copy should be modified

1. **Source Code Compatibility**
2. The source code is expected to be portable across operating systems and their flavors, compilers and machine architectures
3. The following points need to noted in this regard

* Path names of non-standard header files
* Size of an integer variable
* Size of structures when structure members are aligned on the word boundary
* Default character type (signed or unsigned)
* Non-standard library functions

1. The #ifdef pre-processor statement should be used to include/exclude non-portable part of the source code
2. Enabling the compiler switch for ANSI Compatibility helps make code more portable
3. **Error Messages**
4. All the error messages should be written to stderr. If a system call fails, the function perror (provided by the standard C library) should be called. The name of the function that failed should be passed to perror
5. Following information should accompany messages displayed when software aborts

* Name of the source code file (given by the macro \_\_FILE\_\_)
* The line number inside the source code file where it is aborting (given by the

macro \_\_LINE\_\_)

* The name of the function in which it is aborting

1. **Debug Messages**
2. All debug messages should be written to stdout. The debugging statements should be conditionally compiled if DEBUG is defined. The debugging statements should not be physically removed from the code
3. **Additional Conventions**
4. Every statement in the code should start on a new line.
5. All compiler errors and warnings should be enabled. The final version of the source code should not generate any errors or warnings when compiled.
6. No errors should be reported by lint on the source code. Try to remove warnings sounded by lint to the best extent possible.
7. All keywords should be preceded and followed by a space.
8. All operators except structure member dot and arrow operators, unary minus operator, the increment and decrement operators, array member index operator, the & and the \* operators for address of and contents pointed by, the round brackets should be preceded and followed by a space.
9. An explicit typecast should be followed by a space.
10. Space should be inserted after a comma, in parameter list for a function call for example
11. All names (e.g. name of the author) should be initials of the person in the upper case.
12. Magic numbers should be avoided as far as possible. They should always be defined as macro and then used.
13. Enumerated variables should be used when integer variables can take only certain predefined values only.
14. Try to use as few exit points from a function as possible.
15. Try to keep the code as simple as possible.
16. Use library functions to perform a task, if such a function is available.
17. Try to minimize use of temporary variables.
18. Use parenthesis to avoid precedence ambiguity.
19. Organize the functions inside a file hierarchically. The rule is, if you are calling another function, the called function should appear later in the file.
20. Make sure that you always initialize a variable before using it.
21. Watch out for rounding off errors, especially if you are doing floating point calculations.
22. Do not try to optimize at the code level. If you have any suggestions regarding algorithmic efficiency, talk to the project manager so that it can be included in the design after due deliberations.
23. System header files should be included first. They will be followed by user created header files. There should be a blank line between these two blocks.
24. Do not use nested comments.
25. Do not use goto statement.
26. Do not use the ternary operator (?:).
27. Do not initialize variables in the header files.
28. Each line in the source code should be less than or equal to 80 characters
29. Avoid using register variables.
30. Avoid using pointers to unknown types (void \* variety).
31. Some linkers are not case sensitive.
32. Always trap defaults in the switch statement. Place error sounding/aborting code there if the control is not expected to come there.
33. Avoid using fall through in the switch statement.
34. Avoid complicated conditional statements. Use explicit comparisons in the conditional statements.
35. Always parenthesize parameters to the macro in the expansion
36. **A Sample C Code File**

/\*MH

CLIENT.C

Copyright (c) 1994 <Company Name> All Rights Reserved. THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF

<Company Name> The copyright notice above does not evidence any actual or intended publication of such source code.

Author : SSM

Date : 04/04/1994

\*/

#include <stdio.h>

#include "client.h"

#include "fs\_clnt.h"

/\*GL

the variable to store the current state

\*/

static int giState=NOT\_CONNECTED;

/\*GL

the connection endpoint socket descriptor

\*/

static int giCfd=-1;

/\*FH

Purpose : This function asks the server to open the specified directory.

Input :

cpPathName the path name of the directory to be opened

Return Value :

The function returns an integer which indicates :

0+ on success, denotes the directory descriptor

FS\_NOCONNECT not connected

FS\_COMERROR communication error

FS\_NODFD too many open directories

FS\_NOACCESS no access

FS\_NODIR no such dir. exists, invalid directory name pointer directory path name too long

Side Effects :

The gloabl variable giState is set to CONNECTED if the conect succeeds. The gloabl variable giCfd stores the socket descriptor if the conect succeeds.

This routine calls :

strlen

memcpy from standard library

send

recv from socket library

\*/

int iFsOpenDir ( char \*cpPathName )

{

unsigned char caBuf[MAX\_BUFFER\_LEN];

if ( !cpPathName )

{

return FS\_NODIR;

}

if ( (int) strlen( cpPathName ) > ( MAX\_BUFFER\_LEN - 3 ) )

{

return FS\_NODIR;

}

if ( giState == NOT\_CONNECTED )

{

return FS\_NOCONNECT;

}

caBuf[0] = LF\_OPENDIR;

caBuf[1] = strlen( cpPathName );

(void) memcpy( caBuf + 2, cpPathName, strlen( cpPathName ) );

if ( send( giCfd, (char \*) caBuf, caBuf[1] + 2, 0) == -1 ) return FS\_COMERROR;

if ( recv ( giCfd, (char \*) caBuf, MAX\_BUFFER\_LEN, 0 ) == -1 )

{

return FS\_COMERROR;

}

switch ( caBuf[0] )

{

case LF\_SUCCESS :

break;

case LF\_NODFD :

return FS\_NODFD;

case LF\_NOACCESS :

return FS\_NOACCESS;

case LF\_NODIR :

return FS\_NODIR;

default :

/\* should not come here \*/

return FS\_COMERROR;

} /\* endswitch \*/

if ( caBuf[1] != 1 ) return FS\_COMERROR;

return (int) ( caBuf[2] );