C

Style and coding standards

By

SCOPE

2016

ABSTRACT

It describes a recommended coding standard for C programs. The scope is coding style, not functional organization.

The document is created using following reference documents

1. “C STYLE GUIDE” Document published bySEL organization of NASA
2. “Recommended C Style and Coding Standards” paper published byIndian Hill.

C style and coding standards

5 STATEMENTS AND CONTROL FLOW

1 INTRODUCTION

"Good programming style begins with the effective organization of code. By using a clear and consistent organization of the components of your programs, you make them more efficient, readable, and maintainable."

– Steve Oualline, *C Elements of Style*

1.1 Purpose

Purpose of this document is to have code with following points

- Organized
- Easy to read
- Easy to understand
- Maintainable
- Efficient

1.2 Audience

This document is written specifically for programmers in SCOPE T&M PVT. LTD. Pune (India), although the majority of these standards are generally applicable to all environments. In the document, we assume that you have a working knowledge of C, and therefore we don't try to teach you how to program in C. Instead, we focus on pointing out good practices that will enhance the effectiveness of your C code.

1.3 Approach

This document provides guidelines for organizing the content of C programs, files, and Functions. It discusses the structure and placement of variables, statements, and comments. The guidelines are intended to help you write code that can be easily read, understood, and maintained.

2 PROGRAM ORGANIZATION

This section discusses organizing program code into files.

1. A project will contain basically two types of program files. One is **source file** and 2nd is **header file**. See this example.

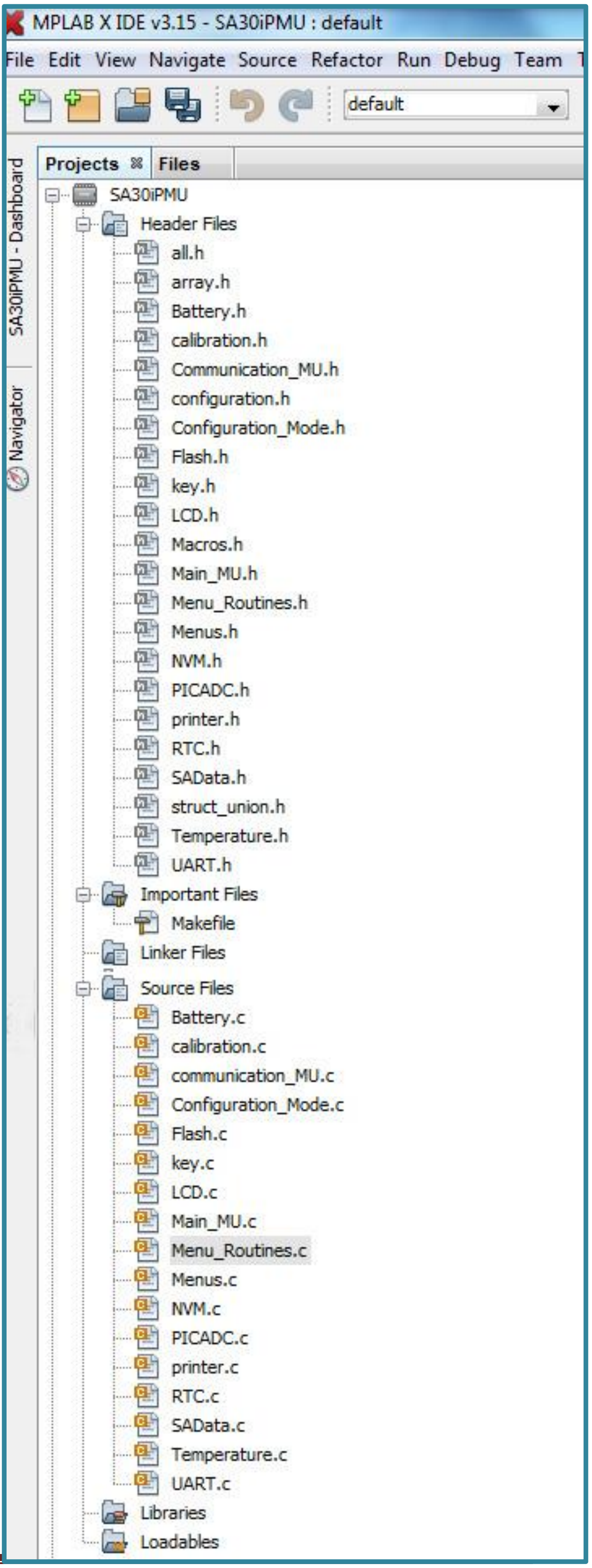
2. Use source files to group logically related functions.

3. Extension of Source file is .c

4. Use header files to write declarations of functions used in source file.

5. Extension of header file is .h

6. Source files also called as Module files.


MPLAB X IDE v3.15 - SA30iPMU : default
File Edit View Navigate Source Refactor Run Debug Team
default
Projects
Files
SA30iPMU - Dashboard
Navigator

```
SA30iPMU
  Header Files
    all.h
    array.h
    Battery.h
    calibration.h
    Communication_MU.h
    configuration.h
    Configuration_Mode.h
    Flash.h
    key.h
    LCD.h
    Macros.h
    Main_MU.h
    Menu_Routines.h
    Menus.h
    NVM.h
    PICADC.h
    printer.h
    RTC.h
    SAData.h
    struct_union.h
    Temperature.h
    UART.h
  Important Files
    Makefile
  Linker Files
  Source Files
    Battery.c
    calibration.c
    communication_MU.c
    Configuration_Mode.c
    Flash.c
    key.c
    LCD.c
    Main_MU.c
    Menu_Routines.c
    Menus.c
    NVM.c
    PICADC.c
    printer.c
    RTC.c
    SAData.c
    Temperature.c
    UART.c
  Libraries
  Loadables
```

2.1 Portability

1. ‘‘Portable’’ means that a source file can be compiled and executed on different machines with the only change being the inclusion of possibly different header files and the use of different compiler flags. New ‘‘machine’’ may be of different hardware, a different operating system, a different compiler, or any combination of these.

2. Organize Separate files like the **machine-independent files and the machine-dependent files** or **driver files and application files**. If the program is to be moved to a new machine, it is a much easier task to determine what needs to be changed.

   Example:

   Driver files

   SPI_PIC32MZ.c
   I2c_PIC32MZ.c
   UART_PIC32MZ.c
   NVM_PIC32MZ.c
   ADC_PIC32MZ.c
   USB_PIC32MZ.c
   Timer_PIC32MZ.c
   Ext_Interrupts_PIC32MZ.c

   Application files

   RTC_DS1307.c
   Printer.c
   ADC7176.c
   BatteryPercentage.c
   TemperaturePT100.c
   RF_MavenSys.c
   PC_communication.c
   Bluetooth_MC.c
   USB_FTD232.c
   GSMsim900.c
   LCD.c
   KeyBoard.c
   Menu.c
   DFT.c
   DigitalFilter.c
   DDS_AD9833.c

3. The C preprocessor provides a better alternative, namely **conditional compilation.** Lines of source code that may be sometimes desired in the program and other times not, are surrounded by `#ifdef`, `#endif` directive pairs as follows:

```
#ifdef DEFAULT_FACTOR
    CalFactor_DCvtg    = 1 ;
    CalFactor_DCCurr   = 1 ;
    CalFactor_ACvtg    = 1 ;
    CalFactor_ACCurr   = 1 ;
     ...
#endif
```

If `DEFAULT_FACTOR` exists as a defined macro, like `#define DEFAULT_FACTOR` in the program, then statements between the `#ifdef` directive and the `#endif` directive are retained in the source file passed to the compiler.
If `DEFAULT_FACTOR` does not exist as a macro, then these statements are not passed on to the compiler.
See following another example.

UART_PIC32MZ.c is source file which includes following functions. So while writing functions make them

```
/*********************************************/
#ifdef   UART1
/*********************************************/
  void UART1_Initialization (void)
     {
      Body of the function
     }
/*********************************************/
void UART1_TransmitByte (char c)
     {
      Body of the function
     }
/*********************************************/
char UART1_RecieveByte (void)
     {
      Body of the function
     }
/*********************************************/
#endif
/*********************************************/
```

```
/*********************************************/
#ifdef   UART2
     Functions for UART2 same as above
#endif
/*********************************************/
/*********************************************/
#ifdef   UART3
     Functions for UART3 same as above
#endif
/*********************************************/
/*********************************************/
#ifdef   UART4
     Functions for UART4 same as above
#endif
/*********************************************/
```

Define any uart module in a header file name as CONFIGURATION.H . this file will contain following statements.

```
#ifndef CONFIGURATION_H        // to prevents accidental double-
#define CONFIGURATION_H        // inclusion
     #define UART1
     #define UART2
     // #define UART3          // not in use
     // #define UART4          // not in use
     #define SPI1
     #define SPI2
     // #define SPI3           // not in use
     // #define SPI4           // not in use
     #define TIMER1
     #define TIMER2
     #define TIMER3
     // #define TIMER4        // not in use
     // #define TIMER5        // not in use
     // #define TIMER6        // not in use

#endif
```

4. Optimizations for one machine may produce worse code on another.

5. When editing someone else’s code, always use the style used in that code.

2.2 Source Files

The suggested order of sections for a source file is as follows:

1. First in the file is a **prologue** that tells what is in that file. The prologue may optionally contain author(s), revision control information, references, etc. see below example.

2. **System include files** like stdio.h should be included before **user include files**. See below example.

3. Use header file name between angle-brackets <> like #include <file_name>, for system include files or standard library files and use header file name between quotes ("") like #include "user_file" for user include files.see below example.

4. After user defined header files inclusion, define **macros** if any. One normal order is to have ‘‘constant’’ macros first, then ‘‘function’’ macros, then typedefs and enums.

5. Next come the **global (external)** data declarations.see below example.

6. Lines longer than 79 columns are not handled well by all terminals/editors and should be avoided if possible. Excessively long lines are often a symptom of poorly-organized code.

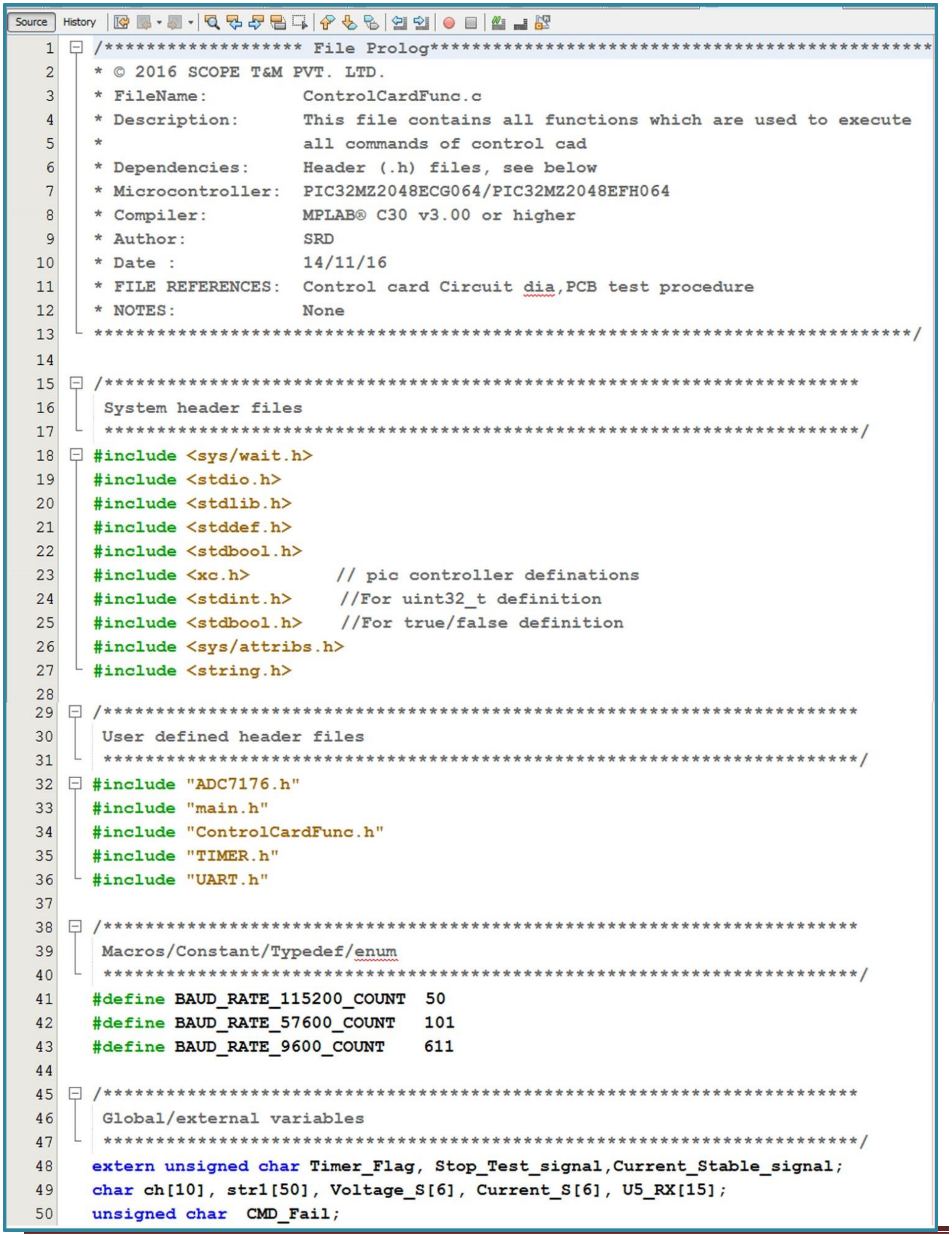
7. There is no maximum length limit for source files, files with more than about 1000 lines are difficult to use it.

8. The functions come last

9. If the file contains the main program, then the main( ) function should be the first function in the file.

10. Place logically related functions in the same file.

11. Put the functions in some meaningful order.



```
/******************** File Prolog****************************************************
* © 2016 SCOPE T&M PVT. LTD.
* FileName:          ControlCardFunc.c
* Description:       This file contains all functions which are used to execute
*                    all commands of control cad
* Dependencies:      Header (.h) files, see below
* Microcontroller:   PIC32MZ2048ECG064/PIC32MZ2048EFH064
* Compiler:          MPLAB® C30 v3.00 or higher
* Author:            SRD
* Date :             14/11/16
* FILE REFERENCES:   Control card Circuit dia,PCB test procedure
* NOTES:             None
******************************************************************************/

/**************************************************************************
 System header files
 **************************************************************************/
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdbool.h>
#include <xc.h>          // pic controller definations
#include <stdint.h>      //For uint32_t definition
#include <stdbool.h>     //For true/false definition
#include <sys/attribs.h>
#include <string.h>

/**************************************************************************
 User defined header files
 **************************************************************************/
#include "ADC7176.h"
#include "main.h"
#include "ControlCardFunc.h"
#include "TIMER.h"
#include "UART.h"

/**************************************************************************
 Macros/Constant/Typedef/enum
 **************************************************************************/
#define BAUD_RATE_115200_COUNT   50
#define BAUD_RATE_57600_COUNT    101
#define BAUD_RATE_9600_COUNT     611

/**************************************************************************
 Global/external variables
 **************************************************************************/
extern unsigned char Timer_Flag, Stop_Test_signal,Current_Stable_signal;
char ch[10], str1[50], Voltage_S[6], Current_S[6], U5_RX[15];
unsigned char  CMD_Fail;
```

2.3 Header Files

1. Write prolog at start of file.In the prolog for a header file, describe what other headers need to be includedfor the header to be functional. See below example.

2. The code between #ifndef and #endif directives is only compiled if the specified identifier has not been previously defined so it prevents accidental double-inclusion.

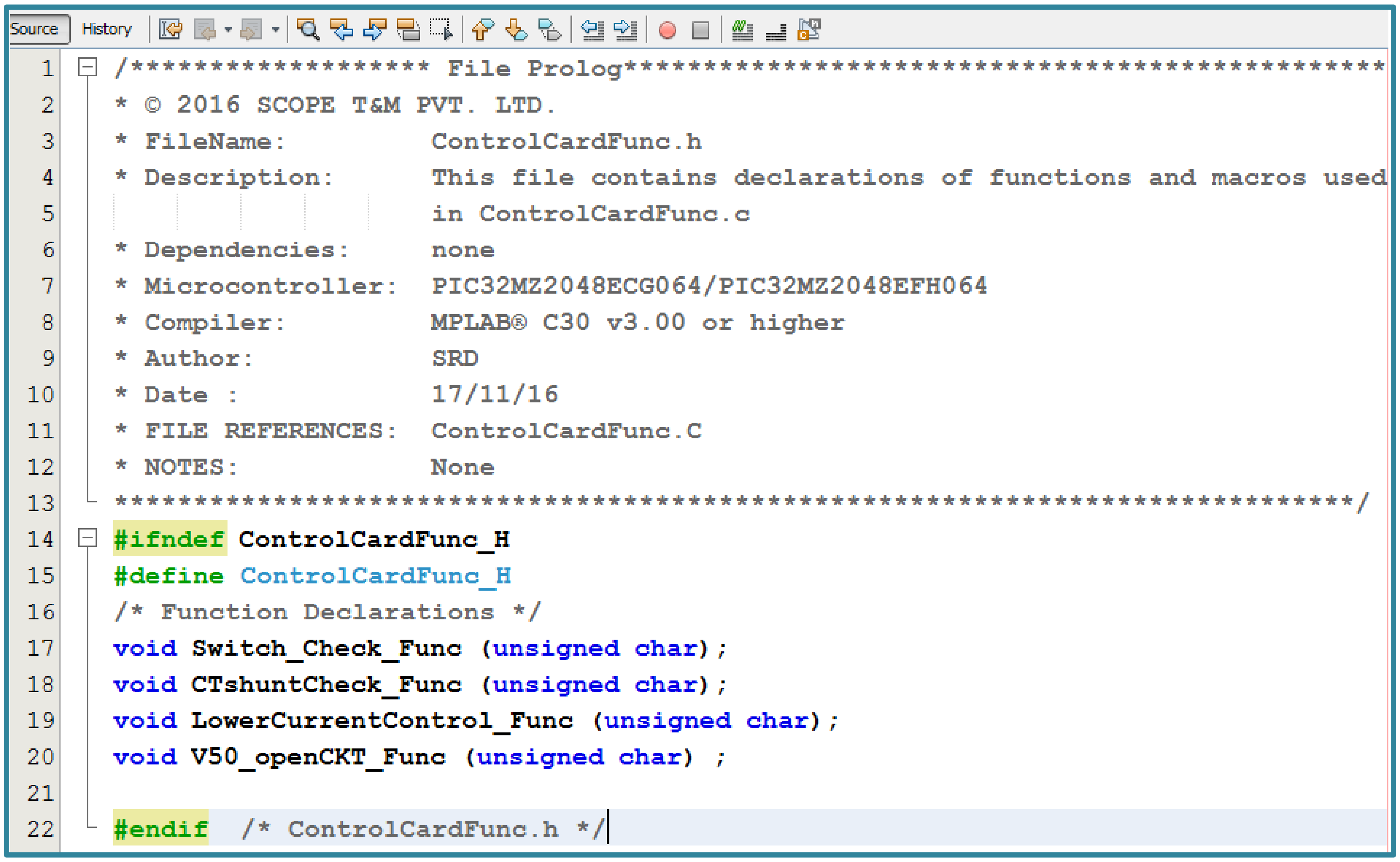
   Example:
   ```
   #ifndef EXAMPLE_H
   #define EXAMPLE_H
   ... /* body of example.h file */
   #endif /* EXAMPLE_H */
   ```

3. Header files are included in C source files before compilation.

4. Declarations for separate subsystems should be in separate header files. If a set of declarations is likely to change when code is ported from one machine to another, those declarations should be in a separate header file.

   Example: `PIC32MZCONFIGURATIONBITS.h`

5. Avoid private header filenames that are the same as standard library header filenames.



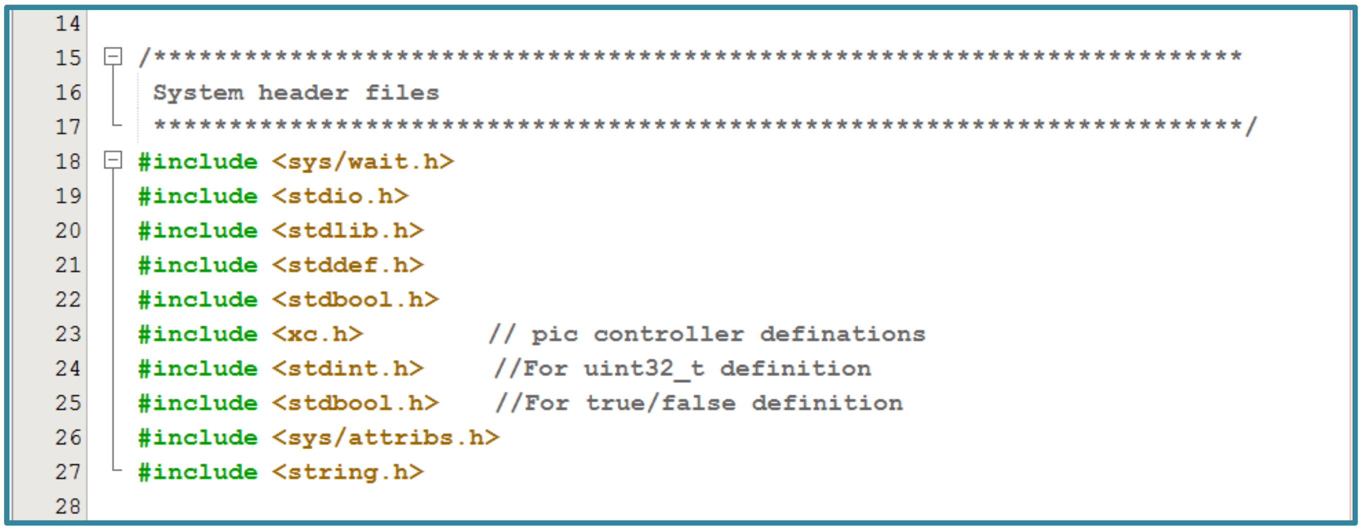
```
/******************* File Prolog************************************************
 * © 2016 SCOPE T&M PVT. LTD.
 * FileName:          ControlCardFunc.h
 * Description:       This file contains declarations of functions and macros used
                      in ControlCardFunc.c
 * Dependencies:      none
 * Microcontroller:   PIC32MZ2048ECG064/PIC32MZ2048EFH064
 * Compiler:          MPLAB® C30 v3.00 or higher
 * Author:            SRD
 * Date :             17/11/16
 * FILE REFERENCES:   ControlCardFunc.C
 * NOTES:             None
*******************************************************************************/
#ifndef ControlCardFunc_H
#define ControlCardFunc_H
/* Function Declarations */
void Switch_Check_Func (unsigned char);
void CTshuntCheck_Func (unsigned char);
void LowerCurrentControl_Func (unsigned char);
void V50_openCKT_Func (unsigned char) ;

#endif  /* ControlCardFunc.h */
```

Example: math.h, time.h are standard library files.

6. Defining variables in a header file is often a poor idea.

7. Header files should not be nested.

8. Large number of header files are to be included in several different source files, it is acceptable to put all common #includes in one include file.
   For example: these are system header files can declare in a header file name as "SystemHeaderFiles".

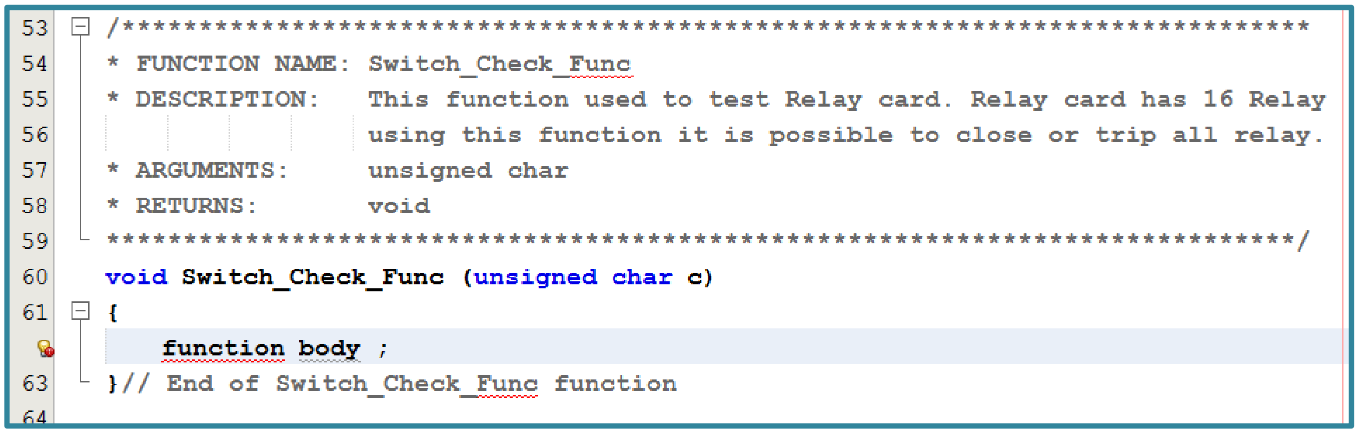


```

/***********************************************************************
 System header files
 ***********************************************************************/
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdbool.h>
#include <xc.h>         // pic controller definations
#include <stdint.h>     //For uint32_t definition
#include <stdbool.h>    //For true/false definition
#include <sys/attribs.h>
#include <string.h>

```

3 FUNCTION ORGANIZATION

3.1 Function Prolog

1. Every function should have a function prolog to introduce the function to the reader. The function prolog should contain the following information:



```
/***************************************************************************
 * FUNCTION NAME:  Switch_Check_Func
 * DESCRIPTION:    This function used to test Relay card. Relay card has 16 Relay
                   using this function it is possible to close or trip all relay.
 * ARGUMENTS:      unsigned char
 * RETURNS:        void
 ***************************************************************************/
void Switch_Check_Func (unsigned char c)
{
    function body ;
}// End of Switch_Check_Func function
```

2. If the function does not return a value then it should be given return type void.

Example:	`void TransmittReadingsToPC(void)`
Don't write it as	`void TransmittReadingsToPC()`

3. The function name (and the formal parameter list) should be alone on a line.

   Example:

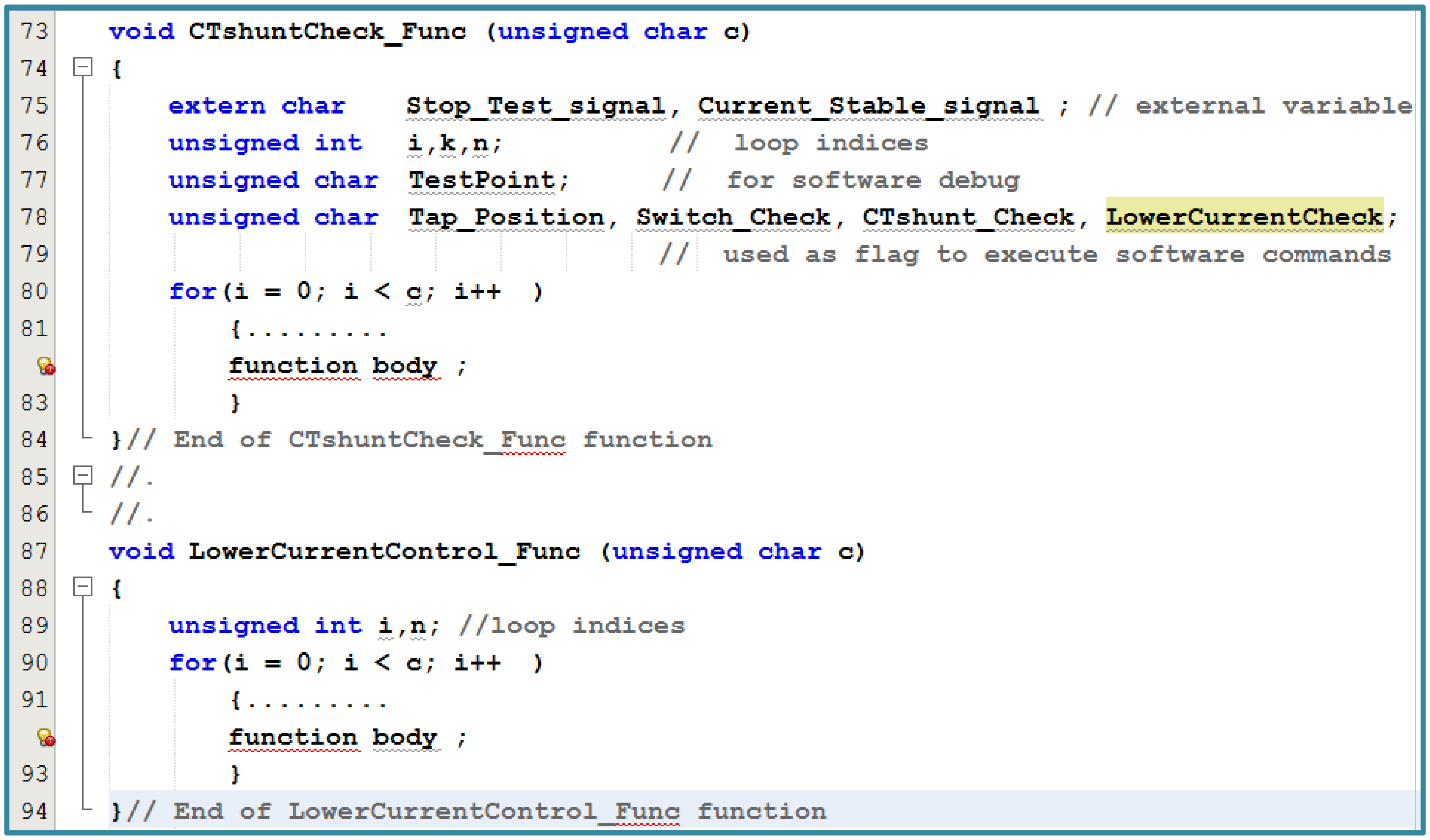
   ```
   Int NVMquadWordWrite(int address_wr, int data1,int data2)
   ```

4. Function body should be tabbed over one stop. See above example.

3.2 Variable Declarations

1. Declare **external variables** immediately after the opening brace of the function block. See below example.
2. **Internal variables** (also known as local variables) should be defined after the external variables.

3. Align variable declarations so that the first letter of each variable name is in the same column.



```
void CTshuntCheck_Func (unsigned char c)
{
    extern char     Stop_Test_signal, Current_Stable_signal ; // external variable
    unsigned int    i,k,n;            //  loop indices
    unsigned char   TestPoint;        //  for software debug
    unsigned char   Tap_Position, Switch_Check, CTshunt_Check, LowerCurrentCheck;
                                      //  used as flag to execute software commands
    for(i = 0; i < c; i++  )
        {.........
        function body ;
        }
}// End of CTshuntCheck_Func function
//.
//.
void LowerCurrentControl_Func (unsigned char c)
{
    unsigned int i,n; //loop indices
    for(i = 0; i < c; i++  )
        {.........
        function body ;
        }
}// End of LowerCurrentControl_Func function
```

4. When declaring variables of the same type, declare each on a separate line.
   Group related variables. Place unrelated variables, even of the same type, on separate lines.

   Example:
   ```
   int year, month, day;      // related variables
   int bufferINT ;
   int x, y, z;
   ```

   never write it as
   ```
   int year, month, day, x, y, z, bufferINT ;
   ```

5. Add a brief comment to variable declarations:

   Example:
   ```
   int  bufferINT;          /* comment */
   int  x, y;               /* comment */
   ```

6. In separate functions, variables that share the same name can be declared. However, the identical name should be used only when the variables also have the identical meaning.
   Example: in above example unsigned int i, n ; are used for loop iterations.

7. If a group of functions uses the same parameter or internal variable, call the repeated variable by the same name in all functions. Call the repeated variable by the same name in all functions.
   Example: use "i" as int datatype for every for loop in function.

   (Conversely, avoid using the same name for different purposes in related functions.)
   Example: don't use "i" as character buffer in UART communication.

8. When naming internal variables used by a function, do not duplicate global variable names. Duplicate names can create **hidden variables**, which can cause your program not to function as you intended.
   See following example

   Example: hidden variable

```
int total;
int func1(void)
{
     float total;         /* this is a hidden variable */
     ...
}
```

   Example: no hidden variable

```
int total;
int func1(void)
{
     float grand_total;   /* internal variable is unique */
     ...
}
```

9. There should be at least 2 blank lines between the end of one function and the comments/prolog for the next function.

10. There should be only one statement per line unless the statements are very closely related.

11. Use vertical and horizontal whitespace.

12. Always declare the return type of functions. Do not default to integer type(int). If the function does not return a value, then give it return type void.

4 READABILITY &MAINTAINABILITY

4.1 Encapsulation and Information Hiding

1. Encapsulation and information hiding techniques can help you write better organized and maintainable code.

2. **Encapsulation** means grouping related elements.
   You can encapsulate on many levels:
   • Organize a program into files.
   • Organize files into functions.
   • Organize functions into logically related groups.

3. **Information hiding** refers to controlling the visibility (or **scope**) of program elements.
   You can use C constructs to control the scope of functions and data.

   • Encapsulate related information in header files, and then include those header files only where needed. For example, #include <time.h> would be inserted only in files whose functions manipulate time.

   • A variable defined outside the current file is called an **external variable**. An external variable is only visible to a function when declared by the extern declaration, which may be used only as needed in individual functions.

4.2 White Space

Adding white space in the form of blank lines, spaces, and indentation will significantly improve the readability of your code.
The following examples illustrate how to use individual spaces to improve readability and to avoid errors.

Example: good spacing

```
*average = *total / *count;  /* compute the average */
```

Example: poor spacing

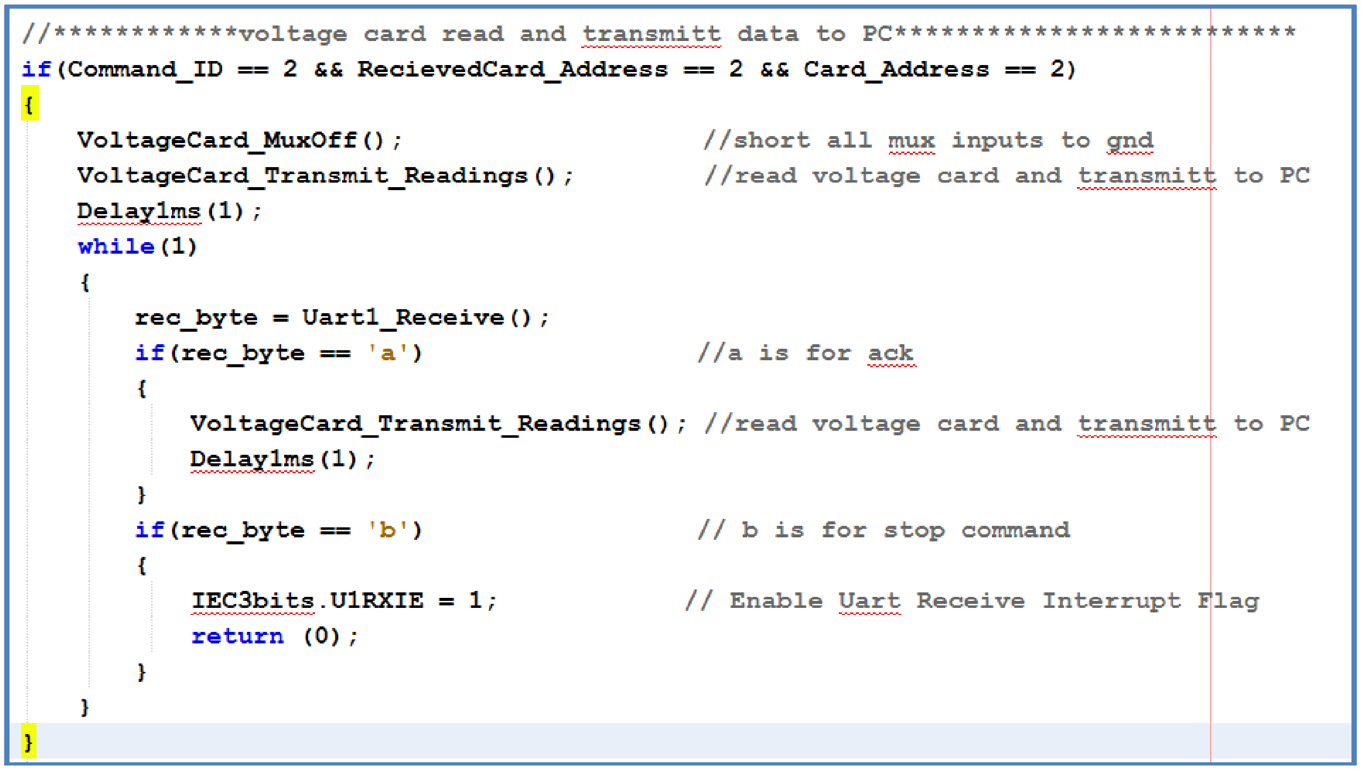
```
*average=*total/*count; /* compute the average */
```

^ **begin comment end comment** ^

Example: comma spacing

```
concat(s1, s2)
```

Use indentation to show the logical structure of your code. Research has shown that **four spaces or one tab** is the optimum indent for readability and maintainability. See below example.



```c
//************voltage card read and transmitt data to PC**************************
if(Command_ID == 2 && RecievedCard_Address == 2 && Card_Address == 2)
{
    VoltageCard_MuxOff();                   //short all mux inputs to gnd
    VoltageCard_Transmit_Readings();        //read voltage card and transmitt to PC
    Delay1ms(1);
    while(1)
    {
        rec_byte = Uart1_Receive();
        if(rec_byte == 'a')                 //a is for ack
        {
            VoltageCard_Transmit_Readings(); //read voltage card and transmitt to PC
            Delay1ms(1);
        }
        if(rec_byte == 'b')                 // b is for stop command
        {
            IEC3bits.U1RXIE = 1;           // Enable Uart Receive Interrupt Flag
            return (0);
        }
    }
}
```

4.3 Comments

1. Comments can be added at many different levels.
   • At the program level, **README** file that provides a general description of the program.
   • At the file level, **file prolog** that explains the purpose of the file
   • At the function level, a comment can serve as a **function prolog**.
   • Throughout the file, add comments to explain the purpose of the variables.

   Example: boxed comment prolog

```
/*****************************************************
* FILE NAME *
* *
* PURPOSE *
* *
*****************************************************/
```

   Example: section separator

```
/*****************************************************/
```

   Example: block comment

```
/*
* Write the comment text here, in complete sentences.
* Use block comments when there is more than one
* sentence.
*/
```

   Example: short comments

```
double ieee_r[4];                  //array of IEEE real*8 values
unsigned char ibm_r[5];            //string of IBM real*8 values
int count;                         //number of real*8 values
```

   or

```
double ieee_r[4];                  /* array of IEEE real*8 values */
unsigned char ibm_r[5];            /* string of IBM real*8 values */
int count;                         /* number of real*8 values */
```

   Example: inline comment

```
switch (ref_type)
{
     /* Perform case for either s/c position or velocity
     * vector request using the RSL routine c_calpvs */
     case 1:
     case 2:
     ...
     case n:
}
```

1. If more than one short comment appears in a block of code or data definition, start all of them at the same tab position and end all at the same position.

4.4 Meaningful Names

Choose names for files, functions, constants, or variables that are **meaningful** and **readable**.
The following guidelines are recommended for creating element names.

1. Choose names with meanings that are precise and use them consistently throughout the program.

Example:

```
DelayTimer180ms();
Delay100us(1);
int VoltageSampleArray[1000];
```

2. Follow a uniform scheme when abbreviating names. For example, if you have a number of functions associated with the "delay" you may want to prefix the functions with "Delay".

Example:

```
DelayTimer180ms();
Delay5ms(1);
Delay100us(1);
```

3.Avoid abbreviations that form letter combinations that may suggest un intended meanings. For example, the name "inch" is a misleading abbreviation for "input character." The name "in_char" would be better.

4. Use underscores within names to improve readability and clarity:

Example:

```
PC_Communication_Packet[]
```

4. Assign names that are unique (with respect to the number of unique characters permitted on your system).

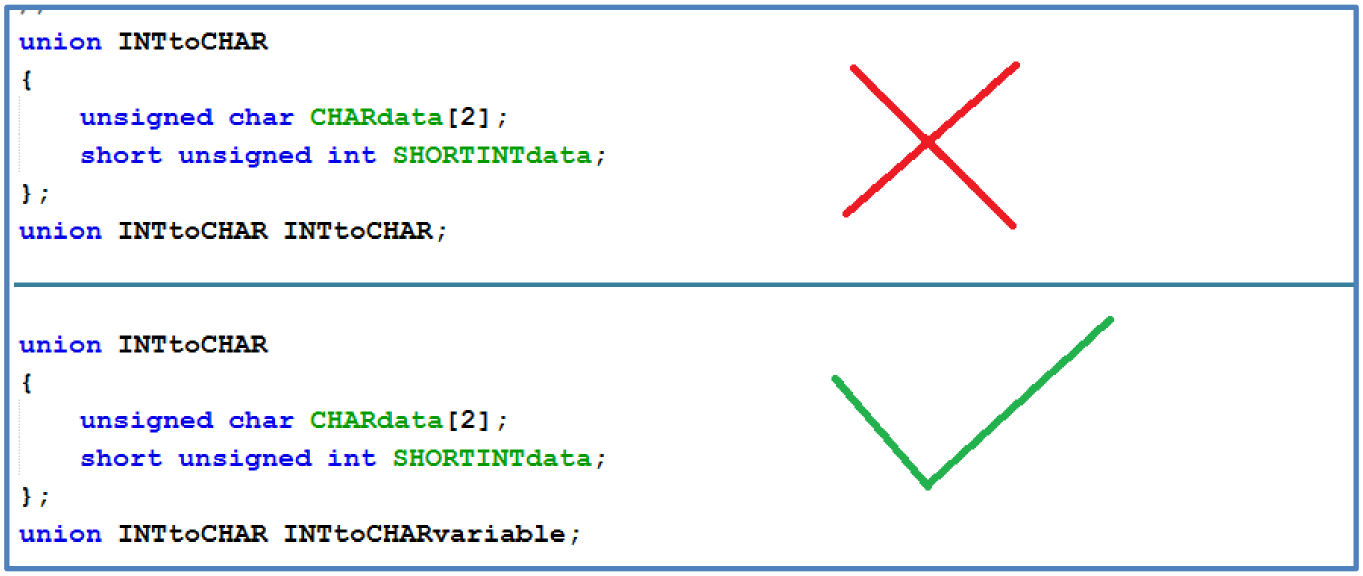
5. Use longer names to improve readability and clarity. However, if names are too long, the program may be more difficult to understand and it may be difficult to express the structure of the program using proper indentation.

6. Names more than four characters in length should differ by at least two characters. For example, "systst" and "sysstst" are easily confused. Add underscores to distinguish between similar names:

systst	should be	sys_tst
sysstst	should be	sys_s_tst

6. Do not rely on letter case to make a name unique. Although C is case sensitive(i.e., "LineLength" is different from "linelength" in C), all names should be unique irrespective of letter case. Do not define two variables with the same spelling, but different case.

8. Do not assign a variable and a typedef (or struct) with the same name, even though C allows this. This type of redundancy can make the program difficult to follow.



```
union INTtoCHAR
{
    unsigned char CHARdata[2];
    short unsigned int SHORTINTdata;
};
union INTtoCHAR INTtoCHAR;
```

✗

```
union INTtoCHAR
{
    unsigned char CHARdata[2];
    short unsigned int SHORTINTdata;
};
union INTtoCHAR INTtoCHARvariable;
```

✓

4.4.1 Standard Names

Some standard short names for code elements are listed in the example below.

Example: standard short names

c	characters
i, j, k	indices
n	counters
p, q	pointers
s	strings

Example: standard suffixes for variables

_ptr	pointer
_file	variable of type file*

4.4.2 Capitalization

The following capitalization style is recommended

• **Variables:** Use words separated by underscores.It’s better to use all lower case words.

Example:
```
pc_communication_packet[50]
payload_length
recievedcard_address
```

• **Function names:** Capitalize the first letter of each word; Try to write name without underscores.

Example:
```
HandshakeRutine();
CyclicRedundancyCheck();
OLTC_ControlFunc();
```

• **Constants/Macros:** Use upper-case words separated by underscores.

Example:
```
MAX_COUNT
#define   BAUD_RATE_115200_COUNT   50
#define   BAUD_RATE_57600_COUNT    101
```

4.5 Type Conversions and Casts

Type conversions occur by default when different types are mixed in arithmetic expression or across an assignment operator.
Use the cast operator to make type conversions explicit rather than implicit.

Example: explicit type conversion (recommended)

```
float   f;
int     i;
...
f = (int) i;
```

Example: implicit type conversion

```
float   f;
int     i;
...
f = i;
```

4.6 Operator Formatting

1. Do not put space around the **primary operators:** -› , . , and [ ] :

   Example: p->count
   buff.count
   ComPortBuff[i]

2. Do not put a space before **parentheses** following function names. Within parentheses, do not put spaces between the expression and the parentheses:

   Example: TransmittReadings(2, x);

3. Do not put spaces between **unary operators** and their operands:

   Example: !p
   ~b
   ++i
   -n
   *p
   &x

4. **Casts** are the only exception. do put a space between a cast and its operand:

   Example: IntData = (int)   FloatData

5. Always put a space around **assignment operators & conditional operators**:

   Example: c1 = c2 ;
   z = (a > b) ? a : b ;

6. **Commas** should have one space (or newline) after them:

   Example: strncat(t, s, n);

7. **Semicolons** should have one space (or newline) after them:

   Example: for (i = 0; i< n; ++i)

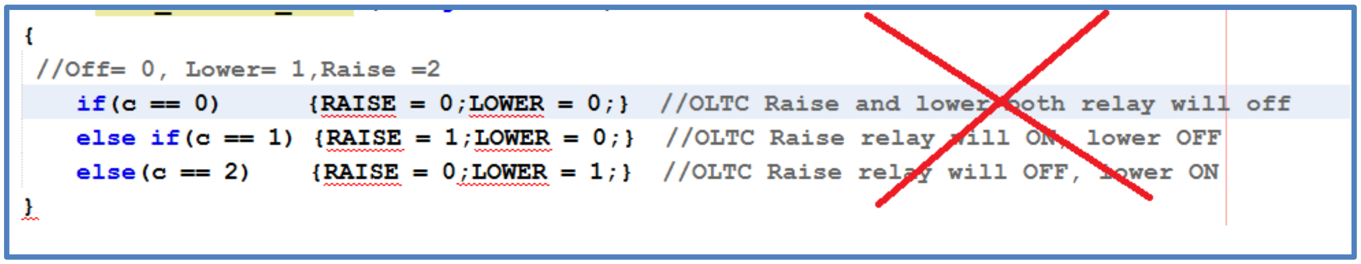
Note: MPlab IDE has facility convert your written code in standard conventions code. Go to Tools -> Options -> Editor -> Formatting. Select language as C and customize you coding style.

5 STATEMENTS& CONTROL FLOW

5.1 Statement Placement

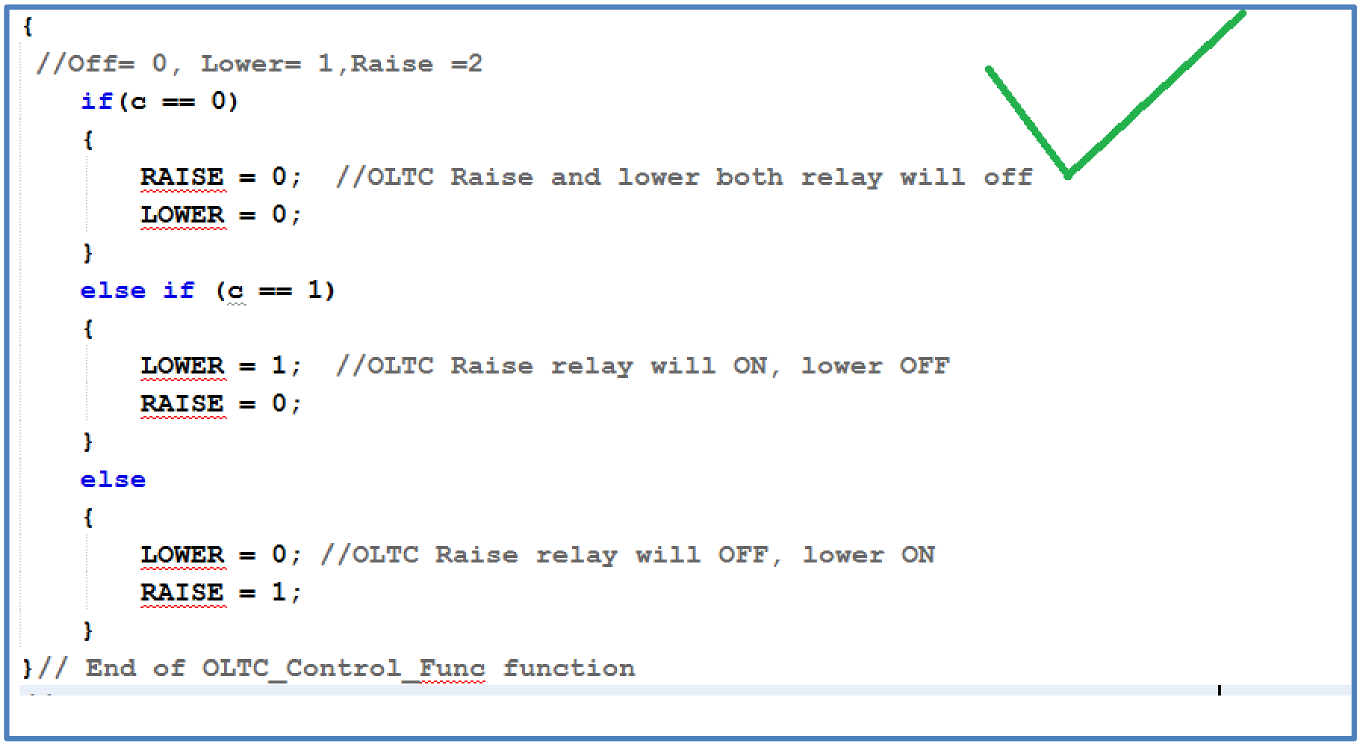
1. Put only one statement per line

2. Limit the complexity of statements, breaking a complex statement into several simple statements it



```
{
 //Off= 0, Lower= 1,Raise =2
    if(c == 0)       {RAISE = 0;LOWER = 0;}  //OLTC Raise and lower both relay will off
    else if(c == 1) {RAISE = 1;LOWER = 0;}  //OLTC Raise relay will ON, lower OFF
    else(c == 2)     {RAISE = 0;LOWER = 1;}  //OLTC Raise relay will OFF, lower ON
}
```

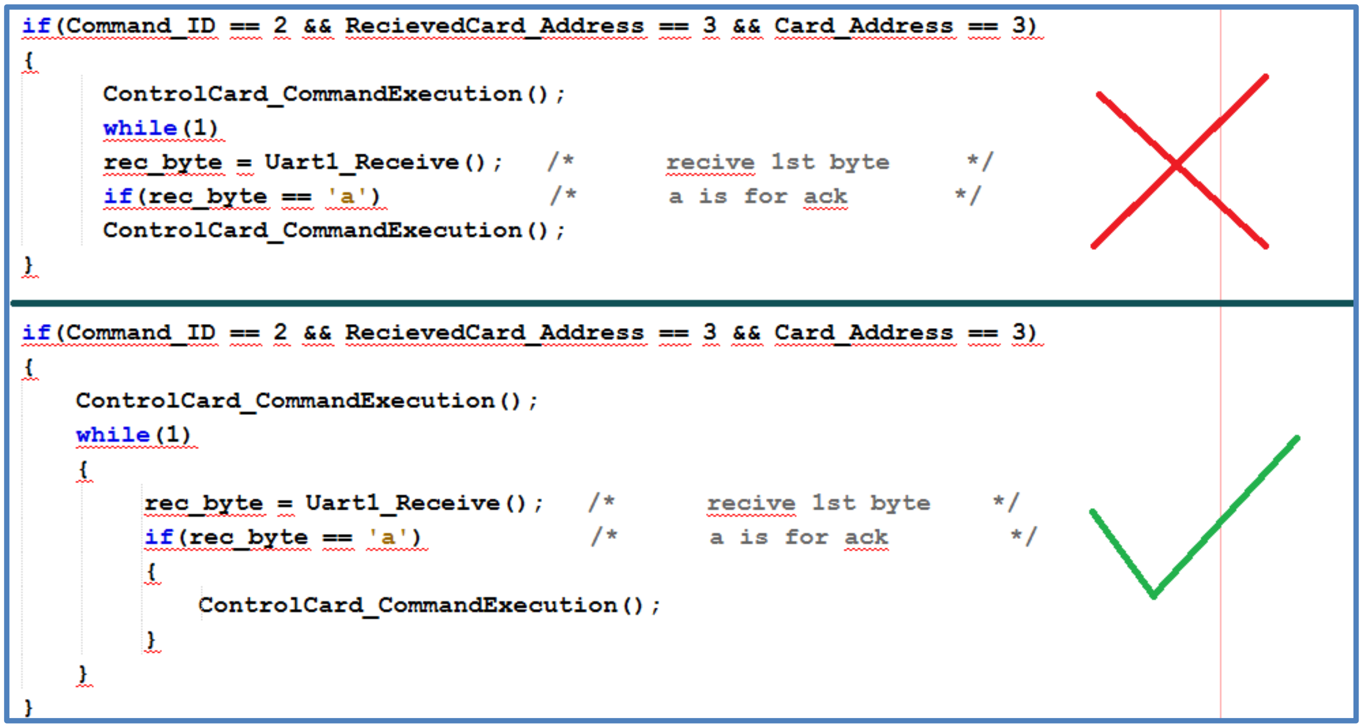
makes the code clearer to read.



```
{
 //Off= 0, Lower= 1,Raise =2
    if(c == 0)
    {
        RAISE = 0;  //OLTC Raise and lower both relay will off
        LOWER = 0;
    }
    else if (c == 1)
    {
        LOWER = 1;  //OLTC Raise relay will ON, lower OFF
        RAISE = 0;
    }
    else
    {
        LOWER = 0; //OLTC Raise relay will OFF, lower ON
        RAISE = 1;
    }
}// End of OLTC_Control_Func function
```

5.2 Braces

Compound statements, also known as blocks, are lists of statements enclosed in braces.
The following examples show the same code with and without braces. We encourage the use of braces



```
if(Command_ID == 2 && RecievedCard_Address == 3 && Card_Address == 3)
{
        ControlCard_CommandExecution();
        while(1)
        rec_byte = Uart1_Receive();   /*      recive 1st byte      */
        if(rec_byte == 'a')           /*      a is for ack         */
        ControlCard_CommandExecution();
}
```

```
if(Command_ID == 2 && RecievedCard_Address == 3 && Card_Address == 3)
{
    ControlCard_CommandExecution();
    while(1)
    {
        rec_byte = Uart1_Receive();   /*      recive 1st byte      */
        if(rec_byte == 'a')           /*      a is for ack         */
        {
            ControlCard_CommandExecution();
        }
    }
}
```

to improve readability. Use your own judgment when deciding whether or not to use braces,

5.3 Guidelines for Performance

1. If performance is not an issue, then write code that is easy to understand instead of code that is faster.

For example,   replace: d = (a = b + c) + r;
               with: a = b + c;
                          d = a + r;

2. Free allocated memory as soon as possible.

3.To improve efficiency, use the automatic increment ++ and decrement operators -- and the special operations += and *= (when side-effect is not an issue).

4. ANSI C allows the assignment of structures. Use this feature instead of copying each field separately.

5. When passing a structure to a function, use a pointer. Using pointers to structures in function calls not only saves memory by using less stack space, but it can also boost performance slightly.

7. Nested functions can create stack overflow. So try to make nested functions chain as small as possible. In other words, return to the main function as often as possible.
.
8. Do not use go to in a code unless absolutely necessary.

9. Main function should contain only function calls. It should have minimum logical statements.