Altran Coding Guidelines

ALTRAN

Objectives

- Need of coding guidelines
- Understanding Aricent Coding Guidelines
- Applying the ACG in all assignments and PRISM project
- Importance of doing code review

Facts and Questions

- Engineering projects have huge number of lines of code
- The code will change. In furture, new developers will need to understand it, and maintain it.
- Can you understand 40000 lines of code which do not use well-meaning menemonic names for variables/functions/symbols?
- *Can you maintain a large code base without proper comments in the source files?*

Why Coding Conventions

- Improve readability
- Improve understandability
- Improve maintainability

Altran Coding Standards

ALTRAN

Altran Coding Standards

- Document available on quality group home page

(Home Page -> Quick Links -> QMS -> Click on Coding Phase -> Guidelines)

- Presents coding standards for C, C++ and Java
- Projects may augment these standards
  - Projects may use local/client specific Work Instructions document

Altran Coding Standards

- Covers
  - Presentation style
    - File Headers, Function headers, indentation, etc.
  - Naming conventions
    - Naming of variables, functions, macros, etc…
  - Language usage
    - C features like typedefs, pointers, macros etc…

Presentation Style

ALTRAN

C Source File Structure

- File Header
- Preprocessor statements
- Include files
- External declarations – (should be in a header file)
- Initialization of global data
- Local function definitions
- Main function

File Header

- Every file starts with a comment header
- Comment header includes
  - The name of the file
  - Description of the file contents
  - Revision history
  - Copyright notice

File Header

```
/*****************************************************************************
 **    FILENAME      :    ss_esl_refer_ext.c
 **
 **    DESCRIPTION   :    This file defines the functions which send
 **                       external messages for Refer.
 **
 **    Revision History   :
 **    DATE       NAME              REFERENCE          REASON
 **    ----------------------------------------------------
 **    11 Sept 2002      Mayank Rastogi          SPR 1204    New code for RY feature
**              31 May 2012         Vikas Nagpal            SPR 1211    Fixed the cause of unhappiness
**                                                                      by commenting out complaints
**

 **    Copyright © 2019 Altran  Group  All Rights Reserved
 **
 *****************************************************************************/
```

Include File Structure

- File header
- Embedded include file references
- #defines, macros
- Type definitions
- “extern” variable declarations
- “extern” function declarations

Include File Issue

What are the problems in case same file included twice in some C/CPP file?

Compiler will give compilation error indicating re-declaration of already existing types/symbols/etc.

How do we solve it?
A header file can “indirectly” get included twice (Example : xyz.c includes a.h and b.h. Both header files a.h and b.h included a common headerfile common.h)

Hint 1 : Can you do something in the header file which protects the contents from being included twice?

Hint 2 : Can you use conditional compilation (#ifndef) ?

Example (Header File)

Technique for preventing multiple inclusion of include file

```
#ifndef   __SS_ESL_REFER_MACRO_H

#define  __SS_ESL_REFER_MACRO_H

/* Notice that the macro name is derived from the file name itself. That makes it unique. Underscores can be
added to the macro name to avoid accidental clash with some other name.

*/

#include "ss_esl_traces.h"

#include "ss_esl_macros.h“

#define ESL_REFER_INVALID_ARG 0


#ifdef ESL_TRACE_ENABLED

#define ESL_REFER_TRACE(trc_id, no_int_args, arg1, arg2, arg3, arg4) {

/*Write definition here*/}

#else

#define ESL_REFER_TRACE(trc_id, no_int_args, arg1, arg2, arg3, arg4)

#endif /* end of ifdef ESL_TRACE_ENABLED */

#endif /* End of ifndef __SS_ESL_REFER_MACRO_H */
```

Function Header

- Precedes every function definition
- Includes
  - Function name
  - Description of what the function does
  - Description of arguments and return values
  - Description of arguments can be given adjacent to argument declaration too
  - Optional notes describing special considerations, warnings, unusual techniques, responsibilities of memory allocation/free, etc

Function Header

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

**FUNCTION NAME    :    ss_esl_refer_arm

**DESCRIPTION      :    In this function sends a call event

**                      request to SF for arming the specified event on the

**                      specified leg.

**

**RETURNS           :  ESL_SUCCESS, and in case of any error

*                      ESL_FAILURE with the corresponding ecode.

**

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

```
return_t ss_esl_refer_arm(
    U8 *p_msg; /* Pointer to message (Notice : argument meaning can be specified adjacent to its declaration ) */
    U16 len;) /* Length of message */
{
/* Function body */
}
```

Comment Style

- Block comments
  - Precedes a “block” (group) of related code statements
  - Should begin at same indentation level as code
- Single line comments
  - Placed to the right of the statement, if they fit on the same line
- Do NOT write redundant comments

Example (Comments)

```
.
.
.
/*
 * Special case: Check if name is the name of the caller.
 */
if (strcmp(runningTask->name, name) == 0)
{
   return (int) runningTask;
}

for (i = 0; i < 256; i++)
{
   /*
    * Check ready queue.
    */
   for (tcb = (WIND_TCB *) lstFirst(&readyQ[i]);
        tcb != NULL;
        tcb = (WIND_TCB *) lstNext((NODE *) tcb))
   {
      if (strcmp(tcb->name, name) == 0)
      {
            return (int) tcb;
      }
   }

   /* Check pending queue. */
   for (tcb = (WIND_TCB *) lstFirst(&pendQ[i]);
        tcb != NULL;
        tcb = (WIND_TCB *) lstNext((NODE *) tcb))
   {
      if (strcmp(tcb->name, name) == 0)
      {
            return (int) tcb;
      }
   }
}
.
.
.
.
```

Figure 5. Comment Style Example (C Language format)

Indentation

- Every nesting should be indented one “level” to the right
- Curly braces should be on a line of their own.

```
if (p_ccb != NULL)
{
    for (i = 0; i < num; i++)
    {
            send_message(p_ccb->array_items[i]);
    }
}
```

Note : Amount of spacing (“level”) should be used uniformly (e.g. 2 spaces/4 spaces/1 tab/etc.

Indentation

- Function arguments should be presented one per line, each indented at least one indentation level

```
return_t  do_validation(
   msg_t      *p_msg, /* Pointer to message */
   ecode_t   *p_error_code) /* To return error code in case of failure */
{
    if (p_msg != NULL)
    {
            /* Validations code */
    }
}
```

Example (Indentation)

Global & local variable indentation

```
extern int  employee_age;

static char middle_initial;

char * func(int data)

{

            int i;

            for (i = 0; i < MAX_RECORDS; i++)

            {

                  …

            }

}
```

Example (Indentation)

```
int task_var_get(
    int tid,     /* task identifier */
     int *pvar)                  /* pointer to variable */
{
    TASK_VAR *tvar;
 if (ERROR == taskidverify(tid))
    {
            return ERROR;                  /* no such task */
    }
    for (tvar = ((TCB *) tid)->ptask; tvar != NULL; tvar = tvar->next)
    {
            if (tvar->address == pvar)
            {
               return tvar->value;
            }
    }
    return ERROR;            /* no such variable */
}
```

Horizontal Spacing

- Use horizontal spaces
  - Before and after binary operator.

    xyz + abc
  - After a keyword

    for (..) or while (..) or if (..)
  - After a comma or semicolon.

    for (i = 0; i < 10; i++)

Horizontal Spacing

- **DO NOT** use spaces
  - Between a function name and opening parenthesis : `get_id(int x)`
  - After an opening parenthesis : `if (xremote == 0)`
  - `return (1);`
  - Before a closing parenthesis: `if (xremote == 0)`
  - Before or after an opening bracket. `int array[10]`
  - Between unary operator and it's operand. `i++`
  - Before or after a structure reference operator. `abc.xyz = 1`

Vertical Spacing

- Have a blank line after every logical paragraph of code

- Preferably, also give a small comment explaining intent of each paragraph

Bad_Vertical_spacin
g_No_Comments

Good_Vertical_Spac
ng_With_Comments

Example

Left parentheses directly after function name

```
void foo ();      /* not good */
void foo();       /* better */
```

Declaring many variables in same statement

```
 /* Not recommended */
 char* i,j;        /* i is declared pointer to char, j is char */
/* Better Way */
        char* i;
        char* j;
```

Example (Function Declaration)

Difficult to understand (Not reccommended)

```
 int mycomplicatedfunction(unsigned unsignedvalue, int
intvalue, char* charpointervalue, int* intpointervalue,
myclass* myclasspointervalue, unsigned*
unsignedpointervalue);
```

Easy to understand (Reccommended)

```
int my_complicated_function(
    unsigned unsigned_value,
    int int_value,
    char *char_pointer_value,
    int *int_pointer_value,
    myclass *my_class_pointer_value,
    unsigned int *unsigned_pointer_value);
```

Sizing

- Avoid long and complex functions
  - If a function is getting too long, possibly we can breakup into multiple simpler functions
- Ideal size 60 to 120 lines

Advantage of smaller functions

- Reduces complexity
- Improves readability and enhance testability
- Improves understandability
- Improves maintainability
- If error occurs at end of a long function, it is difficult for the function to clean up & "undo" as much as possible before reporting error to calling function

Naming Conventions

ALTRAN

Naming Conventions

- Use names that indicate the intent/meaning of the variables.

```
int a, b, c; /* Not meaningful names */

int num_employees, count, salary; /* Meaningful names */
```

- Enumeration (enum) values should be consistently, either in upper case or lower case. (Upper case preferred in most C projects)

```
typedef enum day_of_week { SUN, MON, TUE, WED, THU, FRI DAT} day_of_week_t;
```

- Use underscore or mixed case characters naming, in a consistent manner. sz_user_name OR szUserName (underscore preferred in most C projects)
- Use all upper case for macros and #define constants

```
#define MAX_NUM_EMPLOYEES 100

#define SQUARE(x) ((x) * (x))
```

Naming Conventions

- Do NOT use names that differ only by the case of characters. e.g. `SzName` and `szName`
- Do NOT start names with "_" or "__".
- Do NOT rename operations using macros. Example

```
#define EQ ==          /* avoid */
```

The reason why people like to define EQ is to avoid coding mistakes like **`if(A=B).`** But there are better ways to detect this problem. e.g **a)** Compile with –Wall **b)** Keep the constant to the left hand side of comparison

Example (Naming)

Choice of names (Avoid non-intuitive abbreviations)

```
int group_id;                          /* instead of grpid */

int name_length;                 /* instead of namln */

printer_status_t reset_printer;  /* instead of rstprt */
```

Avoid Ambiguous names

```
void term_process();  /* Terminate process or
                         Terminal process? */
```

Example (Naming)

Names with numeric characters can cause errors which are difficult to locate

```
int I0 = 13;        // names with digits can be
int IO = I0;        // difficult to read
```

Example – Non-intuitive Naming

Example, a boolean variable '`noZbuffer`'

```
noZbuffer == TRUE    //Z buffer not available
noZbuffer == FALSE   //Z buffer is available
```

To execute code following fragment....

```
if (!noZbuffer )
        { ...Z buffer code here... }
```

If there's NOT the absence of a `Zbuffer`, then there IS a `Zbuffer`, so this code runs when a `Zbuffer` is available

Better Alternative

- Instead, to name the boolean ‘is_buffer_available',

```
if (is_buffer_available)
{
   /*...Z buffer code here... */
}
```

Or

```
if (!is_buffer_available)
{
   /* ...Non-z code here... */
}
```

Naming

Similarly, use meaningful names for

- Functions
- MACROs
- Typedefs
- Enums
- and other symbols

The effort spent in choosing good names would result in reducing lot of maintenance effort, debugging costs, and software engineers' suffering

Language Usage

ALTRAN

Language Usage

- Declarations and constants
- Expressions and statements
- Functions and files

Declarations & Constants

- Global data should NOT be used as a means to avoid passing parameters

- All functions must be explicitly declared to return some type which may be void

Example (C Function)

- Functions which return no value should be specified as having the return type void

```
void strange_function(char *before, char *after)
{
    /*Do something here*/
}
```

Declarations & Constants

- `NULL` must only be used with pointers, `0` with integers, and `'\0'` with ASCII characters
- Names should NOT be redefined in inner blocks
- Pointers to un-typed objects must be of type `void *`
- If registers are used, declare registers in order of importance to ensure compiler assigns the most important ones if it runs out of **`register`**`s` to use.
- When defining macros, each of the parameters in the replacement text must be surrounded by parentheses, as well as the entire replacement text.

Macros

```
#define square(x) (x * x)    /* BAD */

int b= SQUARE(2+3);
/* b = (2+3*2+3) = 11 */

#define square(x) ((x) * (x))  /* GOOD */
```

Declarations & Constants

- All variables must be initialized (or assigned first, before reading)
- In general, minimize the use of global data
- Use unions only if you cannot avoid them
- Avoid bitfields (instead use bitwise operators with unsigned integers)

Declarations & Constants

```
/* Avoid magic numbers (hardcoding) */
int salary_array[80];

/* Better */
#define MAX_NUM_EMPLOYEES 80
int salary_array[MAX_NUM_EMPLOYEES];
```

Declarations & Constants

- Enumerators should be defined with `typedef` statement
- Any user defined type should be defined with `typedef`
- Pointer to pointers should be avoided wherever possible

Expressions & Statements

- Check every system and library call for error returns (unless you wish to ignore errors)
- Avoid goto statement
- Do NOT use the << and >> operators to perform multiplication and division only use them for bit wise operation
- Use of nested conditional expressions, makes programs harder to read and should be avoided.

  e.g. `? : ? :`

Expressions & Statements

- If a sub-expression changes the value of a variable, then that variable may not appear anywhere else within the expression, except where explicit evaluation order is guaranteed

EXAMPLE:

```
/*
 *below may be evaluated as either alist[1] = 1  or
alist[2] = 1
 */
int index = 1;
alist[index] = index++; /* Undefined result. Never do it */
```

- Sub-expressions which have side-effects (++ and --) may NOT be used in logical expressions.

Functions and Files

Put function declarations in header file

Don't use absolute pathnames for include files

#include “/project/include/abcd.h” /* No need of giving path */

Instead, include files like

#include “abcd.h” /* No need of giving path */

In the makefile, we can specify directories where to search for header files.

e.g. gcc -I/project/include calc_func.c

Functions and Files

- If function takes no arguments, use the word void as the argument
- The names of formal arguments to functions are to be specified and should be same in declaration and definition

Functions and Files

- Structures should not be passed to functions because the whole structure is pushed onto stack. Use pointers instead
- Place machine-dependent code in a special file so that it may be easily located when porting code from one machine to another
- Use “const” to specify that the argument is un-modified by the function

Example : char *strcpy(char *dest, const char *src);

Best Practices

- Provide a readme file for developer
  - A brief description of module or files in directory
  - Steps to compile and execute the code
  - Compilation and execution environment setup
  - Any dependencies
  - Last known compilation status

Best Practices

- For nested loops or conditional statements, mark end of loop for easy identification (No need in case of small number of statements in the loop)

```
while (a < b)
{ ...
   while (c > d)
  {
      /* Lot of code making it diffcult for
       * reader to match end of loop
       */
  } // End while (c > d)
} // End while (a < b)
```

Modifying Existing Code

ALTRAN

Modifying Existing Code

- Should be able to track the changes in code. Update revision history

- Do NOT leave commented code

- Always comment the change. Comment should mention SPR (Software Problem Report) number or feature name

```
/* SPR 12345 : fixed “off-by-one” bug (changed <= to <)*/
for (i=0; i < num; i++)
{
    /* Looping and doing something */
}
```

Modifying Existing Code

- Do sufficient “homework” before making any changes
- Maintain consistency of design, presentation, coding standards
- Make sure that a perceived localized change does not have global repercussions
- Configuration management is absolutely essential

Deviations

- Not compulsory to follow all the rules except for those listed in coding standards
- Can be broken for performance etc
- If you break a rule, document it

Assignment

- Refer the program “fresher.c”

  - Make it as a multi-file program with multiple directories
  - Follow Coding Guidelines
  - Add comments

- You may refer the other sample files provided.

Any Questions?

Thanks


ALTRAN