**9 Naming Convention:**

* 1. **Files Names**

Filenames should be all lowercase with words separated by underscores (\_). Follow the convention that your project uses.  
C files should end in .c and header files should end in .h.

dse\_<Module name>\_config.h

dse\_< Module name >.h

dse\_< Module name >.c

* 1. **Function names**

All functions must contain as a prefix: dse\_func\_<function name> as follow:

dse\_func <get\_max\_count>

* 1. **Variable names**

Use all lower case letters.

Use '\_' as the word separator.

Contain the prefix:  dse\_<variable name>, for instance:

uint32\_t dse\_variable1;

bool dse\_variable2;

* 1. **Global Variables**

Global variables should be prepended with a 'g\_'.

Global variables should be avoided whenever possible.

Justification

It's important to know the scope of a variable.

Example

uint8\_t g\_dse\_variable1;

1. **Macro Definitions**

All definitions and macros shall be defined with upper letters.

1. **Data Types, Structures and User defined Types.**

**Structures**

Use underbars ('\_') to separate name components

When declaring variables in structures, declare them organized by use in a manner to attempt to minimize memory wastage because of compiler alignment issues, then by size, and then by alphabetical order. It may be useful to use a meaningful prefix for each member name.

All structures shall contain dse\_<Struct Name> in toggle case.

**Enumerations**

All enums shall contain dse\_<state/funct>\_<state name> prefix.

Labels All Upper Case with '\_' Word Separators

This is the standard rule for enum labels. No comma on the last element.

Enum must contain a DSE\_<STATE/FUNC>\_MAX\_<VALUE>

**User Defined Types**

Type defined shall contain the <variable\_name>\_T indicating this is a type defined by user.

**Threads names**

dse\_<Name>\_thread

dse\_adc\_thread

**CallBacks**

dse\_<Name>\_callback

dse\_timer\_control\_callback

**Interrups**

dse\_<Name>\_interrupt

dse\_adc\_interrupt

1. **Project Structure:**

**Comments in Files**

Commenting Out Large Code Blocks

Sometimes large blocks of code need to be commented out for testing.

Using #if 0

void

example()

{

great looking code

#if 0

lots of code

#endif

more code

}

In case code be pending for the implementer, it shall indicate with:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
@todo:

what remains to be done  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

In case of issue known or pending to be solved, it shall contain a description as bug word.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
@bug:

report a bug found in the piece of code  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

All the comments allowed shall be block comments.

File Comments: Every .h and .c file should have a high-level comment at the top describing the file's contents, and should include your name(s) and the date.

Function Comments: Every function (in both the .h and the .c files) should have a comment describing:

what function does;

what its parameter values are

what values it returns (if a function returns one type of value usually, and another value to indicate an error, your comment should describe both of these types of return values).

In header files, function comments are for the user of the interface. In a source file, function comments are for readers of the implementation of that function. Because of this, function comments in C source files often additionally include a description of how the function is implemented. In particular, if a function implements a complicated algorithm, its comment may describe the main steps of the algorithm.

My advice on writing function comments: write the function's comment first, then write the function code. For complicated functions, having a comment that lists the steps of the algorithm, will help you

When commenting stick to a particular style. For example:

/\*

\* Function: approx\_pi

\* --------------------

\* computes an approximation of pi using:

\* pi/6 = 1/2 + (1/2 x 3/4) 1/5 (1/2)^3 + (1/2 x 3/4 x 5/6) 1/7 (1/2)^5 +

\*

\* n: number of terms in the series to sum

\*

\* returns: the approximate value of pi obtained by suming the first n terms

\* in the above series

\* returns zero on error (if n is non-positive)

**Header Files**

Header comment must be the same as the source files.  
Consider the following sections in the \*.h document:  
/\* ======== Includes ========\*/

/\* ======== Global Variables ========\*/

/\* ======== Macros =======\*/

/\* ======== Structures ========\*/

/\* ======== Function Prototypes ========\*/

**Source Files**

Every file shall contain the following comment at the top:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file: hid.c

\* @purpose:

\*

\* @version: V1.00

\* @date: 10/16/2019

\* @author: Mabe QTC

\*----------------------------------------------------------------------------

\*

\* Copyright (C) 2019. All rights reserved.

\*

\*----------------------------------------------------------------------------

\* Author Date Description

\* SY 03/29/14 added Timer2 handler for future use.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Consider the following sections in the \*.c document:

/\* ======== Includes ========\*/

/\* ======== Global Variables ========\*/

/\* ======== Macros =======\*/

/\* ======== Structures ========\*/

/\* ======== Function Prototypes ========\*/

1. **Applied Norms:**

MISRA C:2012 was published on 18 March 2013. MISRA C:2012 extends support to the C99 version of the language whilst maintaining guidelines for C90. Other improvements, many of which have been made as a result of user feedback, include: better rationales for every guideline, identified decidability so users can better interpret the output of checking tools, greater granularity of rules to allow more precise control, a number of expanded examples and integration of MISRA AC AGC. A cross reference for ISO 26262 has also been produced.