**Initialization:**

1. All local variables shall be initialized, at the time of declaration.

Reason: Avoid the chances of reading garbage values.

1. All global variables shall be initialized, at the time of declaration, even if the initialization is zero
2. All pointer variables shall be initialized at the time of declaration

**Variable Naming:**

1. All local variables shall have the suffix “\_l” to signify that that variable is local to a block
2. All global variables shall have the suffix “\_g” to signify that the variable has global access
3. All variable names shall follow the following syntax:

<Module Name>\_ <variable unit identifier ><variable description>\_<global or local suffix>

1. All variable names shall follow the camelCase naming convention. This rule applies primarily to the variable description in the variable name but may also apply to the Module Name depending on the name of the module

This is to have uniformity in naming in case multiple developers work on the code.

**Modules:**

1. Every software shall be segregated clearly into modules.

For example, if the software uses a temperature sensor, an RTC and UART communication, the functionalities of these three shall be in different modules. One for temperature sensor, one for RTC and one for UART communication.

There can be yet another module in this example which controls the operations of the modules mentioned above.

1. Every software shall clearly identify the modules and allocate appropriate names (or abbreviations) for the modules. These module names shall be used in naming the variables owned by these modules. <Please refer to>

For example: The module that controls the RTC may be named **RTC**. Any variable used by the RTC module shall start with the Module Name RTC

RTC\_<variable unit identifier ><variable description>\_<global or local suffix>

1. Module Name shall always start with an upper case.
2. Module Names shall be self descriptive. Similar to the RTC example mentioned above, the module Temperature Sensor may be named **TmpSnsr**
3. At this point it is important to mention that the software department of the organization, as a good practice, define a list of words that may be used in Module Names and variable descriptions. This list of words shall be accompanied by approved abbreviation for the word.

For example:

Temperature – Tmp

Sensor – Snsr

1. The list of words shall be updated regularly with newly identified words and their abbreviations.
2. The identified abbreviations shall be unique

**Variable Unit Identifier**

The <variable unit identifier> identifies the unit of the data which a variable holds

1. The software team shall identify a list of units for the different types of data they deal with.
2. Every identified unit shall be accompanied by an approved identifier.

For example:

Temperature – t

Time – ti

1. The unit identifier shall always be in lower case
2. The approved identifier for a unit shall be used in the field <variable unit identifier>

For example: A variable in the Temperature Sensor Module which stores the temperature of the environment and is local to the module may be named as follows:

TmpSnsr\_tEnv\_l

This example assumes that the abbreviation “Env” is defined for the word “Environment”.

1. Dynamic memory allocation shall never be used

Reason: Embedded systems work on limited resources which include the working memory. Dynamic memory allocation “depends” a lot on chance. In case sufficient requested memory is not available, there is a chance for the requirement to not function properly.

**Use of Macros:**

1. There shall be no “magic numbers” in the code. Magic numbers are the numbers that appear randomly in the code and do not convey the purpose. Consider the example below:

*for(i = 0; i < 30; i++)*

*{*

*//some code here*

*}*

Although the “zero” in the initialization is well understood, the value “30” doesnot signify its use. It becomes difficult for the people maintaining the software. Moreover, if this value 30 is related to one entity, for example, to read the number of samples from a sensor, there are high chances that the value might appear else where. An example could be that we are defining a buffer size to hold the samples, reading the samples in the above example, and somewhere in the code, processing them.

So, any change in the value will need to reflect at all places. Missing at one place shall result in misbehavior.

Hence, to avoid this issue, it is always good to define such values as Macros and then use the macros.

Example:

#define NO\_OF\_SAMPLES 30

Benefits:

1. Better Readability
2. Better Maintainability
3. Reduced chances of bugs
4. Every word in a macro shall be separated by an underscore (\_)
5. Macros shall all be in upper case
6. Macros shall NOT be used to replace any keywords of C or brackets.

#define begin {

#define end }

if(some\_condition)

begin

//statements

end

The above implementation shall be illegal per the guidelines mentioned in this document.

Arrangement:

1. Every line shall contain at most only one statement
2. The source code shall be indented properly
3. In a series of variable declarations, the first character of the variable name shall in aligned
4. The members of a structure or union shall have the first character of their names aligned.