



Software Development Plan

SDP

Version 01

July 22, 2019

Prepared by:

Juan Manuel Esquivel

Jorge Luis Oficial López

# Review and Approval

| **Reviewer Title** | **Reviewer Name** | **Approval Date** |
| --- | --- | --- |
| Software Engineer | Jorge Luis Oficial López | July 31, 2019 |
| Functional Safety Manager | Juan Manuel Ezquivel | July 31, 2019 |
| Lead Quality Assurance Engineer | Adbeel Alejandro Perez | July 31, 2019 |

# Revision Record

| **Revision Number** | **Revision Status\*** | **Date** | **Description / Reason for Revision** | **Author** |
| --- | --- | --- | --- | --- |
| 01\_01 | Draft | July 16, 2019 | First version of the SDP | Jorge Luis Oficial |
| 01\_02 | In-Review | July 25, 2019 | Updated to accommodate component | Jorge Luis Oficial |
|  |  |  |  |  |
|  |  |  |  |  |

# 3 Scope

The Software Development Plan (SDP) is a subset of the overall project plan, focusing on the tasks to be performed by the Software Engineers. This plan is intended to meet the requirements of the integrator project.

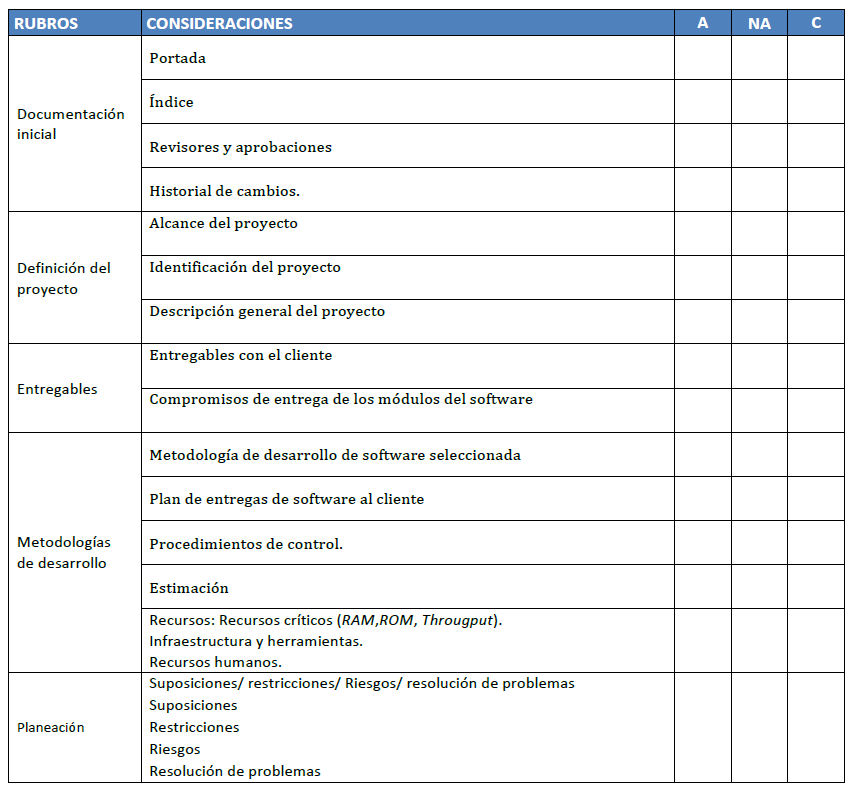
The SPICE will be tailored as methodology for this project. The tailored SPICE will be followed for only the software development portion, and therefore Systems and Program Management Activities will not be discussed in this SDP. SPICE SW process items that will not be part of this development process will be identified throughout this document.

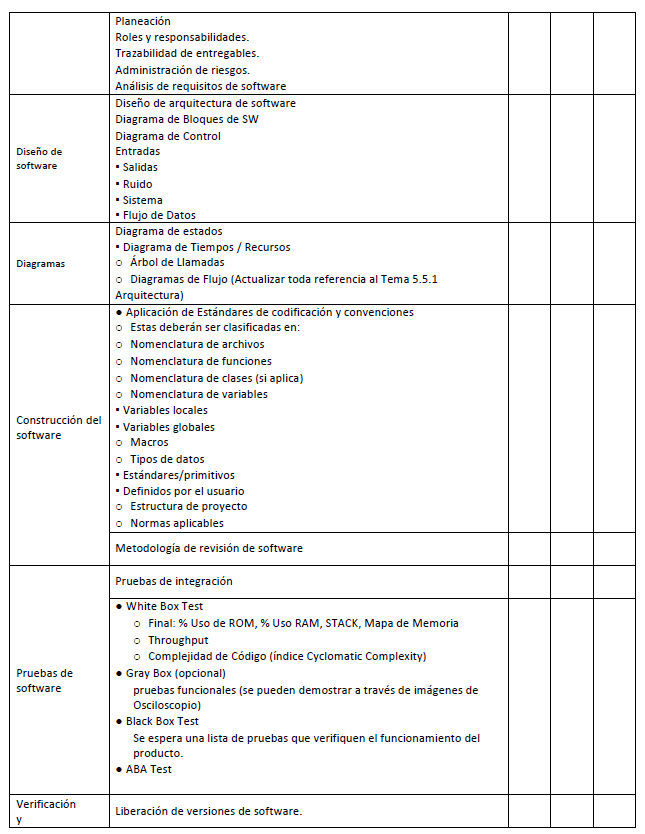
The objective and scope of this project is control the speed of a direct current engine through square signal which will varie in its pulse width whose working frequency will be 10 KHz. By using a hall effect sensor coupled to the motor rotor, the speed of the motor which will provide a series of pulses each time a complete lap is completed, this measurement.

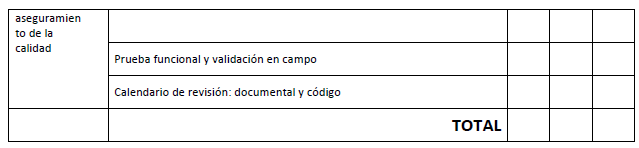
The feasibility of achieving the goals of the project will be / are covered in the SDP Peer Review, Project Reviews and Design Reviews as defined in Joint Review section of this document.

# 4 Deliverables

|  |
| --- |
| Release software executable package for Production Build – August 9, 2019 in UTEQ campus Queretaro |



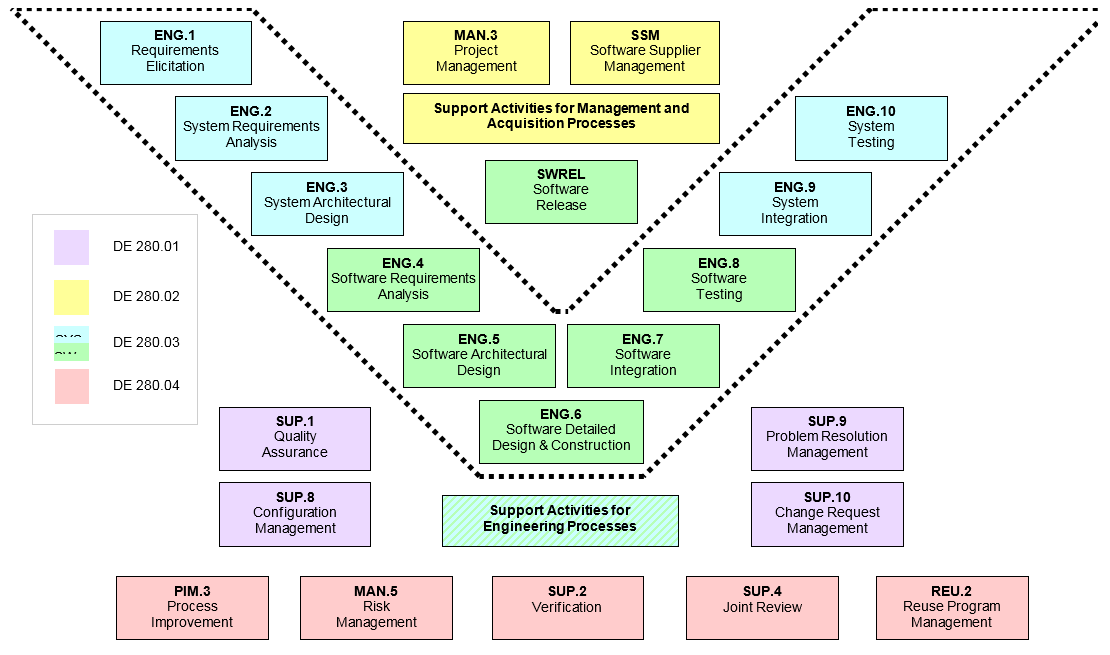




# Software Development Methodology

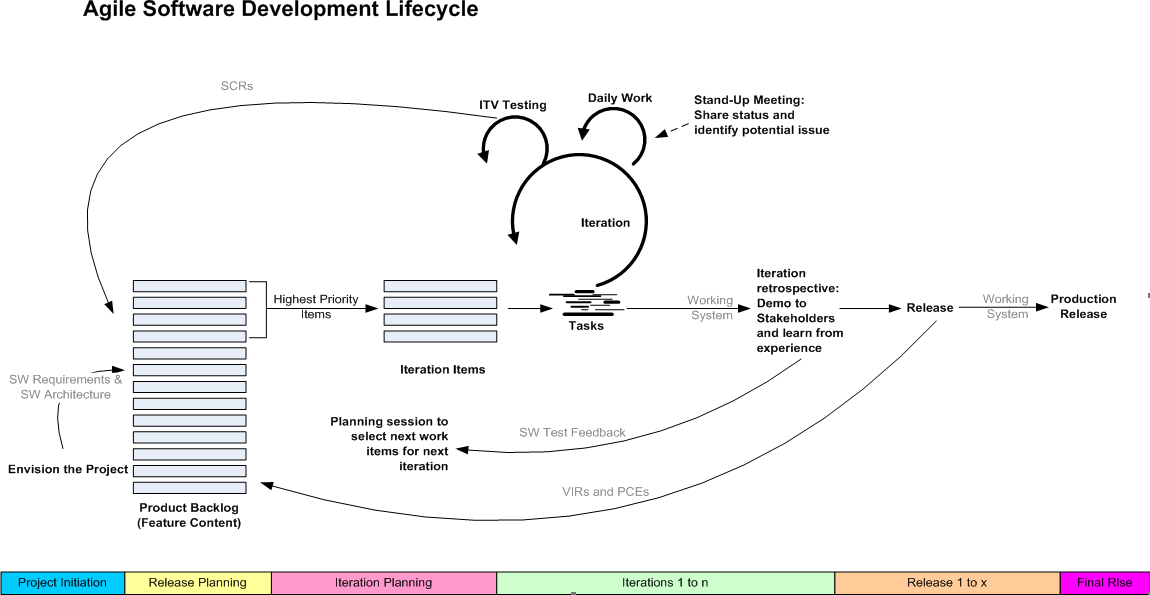
The Software Engineering work for this project will follow the SPICE Process as a framework to develop the product. The software lifecycle will follow the E&S Engineering Operating System (EOS) agile development lifecycle.

This lifecycle meets the requirements of the SPICE Process.



## Agile Methodology

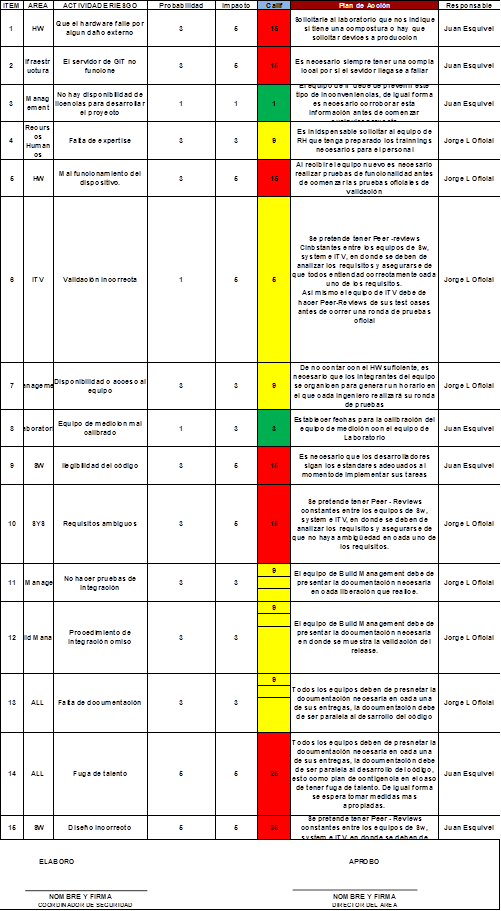
The development of this software will follow some aspects of Agile methodology. The diagram below shows the Agile Software Development Lifecycle that is to be used on this project



1. **Actively working with Systems team to** initially determinethe scope of the software system by understanding existing requirements and creating the initial high-level requirements for the software.
2. **Starting to build the team**.  Although the team may evolve over time, during Project Initiation the key team members are identified.
3. **Documenting the** initial software architecture.  Early in the project the team needs to have a general idea of how the software is going to be partitioned.  The software architecture will likely evolve over time, and it will not be very detailed.  The goal is to identify an architectural strategy.
4. **Setting up the environment**.  Identify development tools, hardware needs, etc., for the team and when they are needed

# Estimates

  Create an initial estimate of the software project based on the initial requirements, the initial architecture, and the skills of the team. This initial estimate is documented in this Software Development Plan. This estimate will evolve throughout the project, and any changes following the initial estimate will be documented in ReqView and in our main document for the Integrator Project.

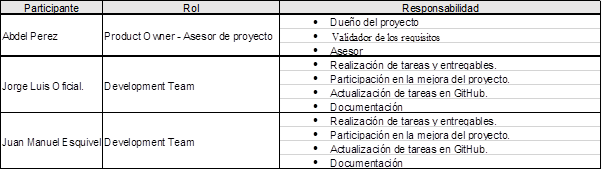


# > --Diagrama de GAN

# Planning

## 7.1 Release Planning

Using the software architecture design, releases will be planned that align with the Integrator Project based on feature/function content requirements. The output of release planning is a set of Product Backlog, i.e. feature/function content broken down by relative size and prioritized according to need. The Product Backlog (Feature Content) is determined by the Lead Systems Engineer and Software Project Managers and documented as Features in ZenHub once they are pulled from the Requirements located in ReqViw. These Features capture the high-level intent of the feature/function. The complete set of Product Backlog descriptions comprise all the feature/function content of the software project. Requirements in ZenHub capture the lower level SW requirement. ZenHub taks are created by the team to illustrate the actual tasks needed to be completed in order to satisfy the Requirements.

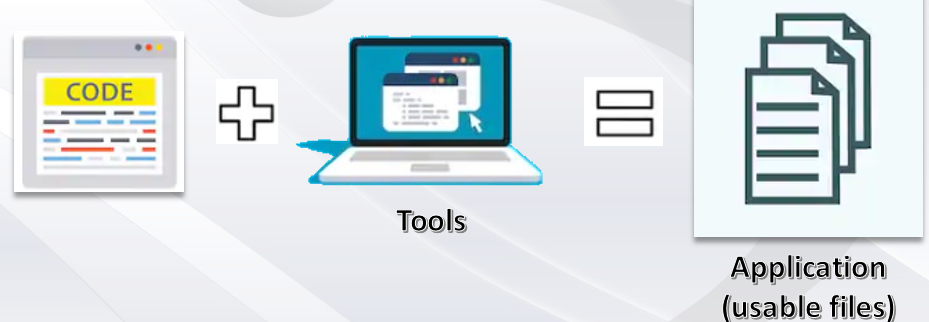


## Iteration Planning

Iteration Planning is accomplished by taking the Product Backlog and dividing up the Modules content for each release and assigning them by priority and hours estimation to an iteration. Each iteration will be planned with a frequency of 2 week intervals.

### Baselines

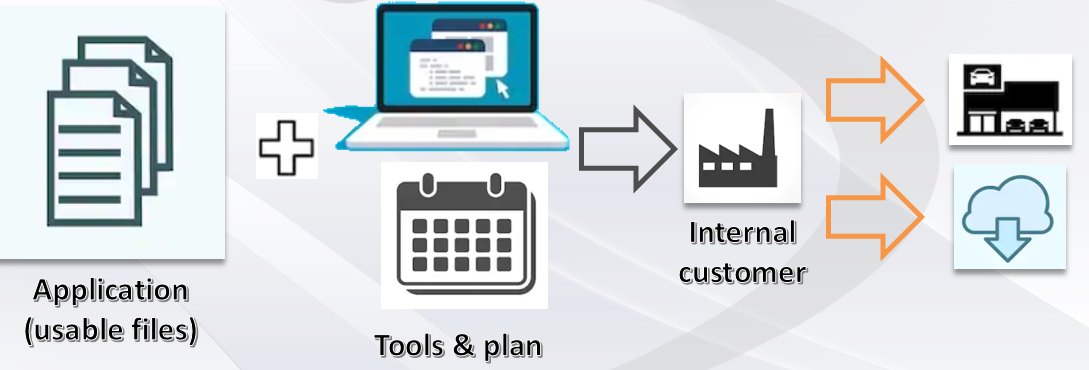
Integrate the Software code units to produce an executable application consistent 100% with its design.



* The usable files are meant to be loaded into a device which executes them for the expected purpose or functionality.
* Loading method is open to the defined requirement of the product such as Over The Air (OTA), USB stick, automotive network or so.

### Releases

Controls the launch of an executable application to the internal customer according to the delivery plan.



The internal customer prepares the product so that it is delivered to the final customer through a defined method

Releases are identified by Version. Releases for development intention will appear in github prject. Releases will be baselined in GitHub Tool under the corresponding repository.

### 7.2.3 Release Naming convention

Naming convention for each release will be identified with a sequential number, as shown below:

DSE\_G2\_IntegratorProyect\_DEV\_YYWW.NN

- YY Current year   
- WW Current Sprintyy   
- NN Release number

e.g.

DSE\_G2\_IntegratorProyect\_DEV\_1925.00

DSE\_G2\_IntegratorProyect\_DEBUG\_1925.00

DSE\_G2\_IntegratorProyect\_PROD\_1925.00

# Problem Solving Strategy

This process is based on the Problem Resolution Management process area of the GSCP

The purpose of the Problem Solving strategy process is to ensure that all discovered problems are identified, analyzed, managed and controlled to resolution. Following the GSCP methodology, Integrator Project will use a combined Problem Resolution Change Request Management where a member of the team will play the role as “Independent Test and Verification” department. This person shall apply different kind of test in order to validate that the outputs of the system make match with each one of the requirements.



If some issue is found during the Test Set, this shall be reported through a ticket in ZenHub Tool and assigned to another member of the team in order to be solved. Each ticket must describe the issue founded, as well as, the steps followed to do visible the issue. As in the next example:

*HW version: Renesas 3.0*

*SW version: DSE\_G2\_IntegratorProyect\_DEV\_1925.03*

*Test ID related: TEST-DSE-1620.1*

*Requirement related:* ***DSE- 1920.3 The Engine shall have works to 10Khz***

*Test procedure:*

*1.-Set the power supply voltage at 12 v  
2.-Microcontroller shall sent and square signal of 10kHz*

*Expected results:  
The engine shall receive a 10 Khz frequency*

*Observed results:  
The Engine is reveiving just 9Khz frequency according to the oscilloscope*

Software Engineers generate the code to implement the software requirements and the detailed design complying with C/C++ Coding Standards - SW REF 264.01E – and conducts a peer review of the output.

Software Engineers shall generate unit tests using Google Test to confirm that newly developed code executes as expected and conducts a peer review of the output. Functional code and unit tests may be reviewed in a single peer review.

Software Engineers shall perform static code analysis using the Coverity application. Any new static analysis issues found are resolved, as well as, addresses any items identified from the peer review and executes software unit tests.

Software Engineers shall complete the task in GitHub or integrates the task branch into the ‘Development’ branch in GitHub and marks the ZenHub tasks to “done” status.

## 8.1 Problem solving and soft skills

* + Troubleshooting
    - Required for immediate feedback on issues in the application
    - Supported by Fault Tree Analysis (FTA) and 3 Leg 5 Why (3L5W)
    - Supported by AMEF
  + Technical Writing and Structured Documentation
    - The nature of activities reside on continuous documentation of work products and evidences
  + Essential Leadership Competencies
    - Bias for action: Taking on new opportunities and tough challenges with a sense of urgency, high energy, and enthusiasm.
    - Steel trap accountability: Stepping up to address difficult issues, saying what needs to be said.
    - Process improvement: Knowing the most effective and efficient processes to get things done, with a focus on continuous improvement. Planning and prioritizing work to meet commitments aligned with organizational goals.
    - Collaboration: Building partnerships and working collaboratively with others to meet shared objectives.

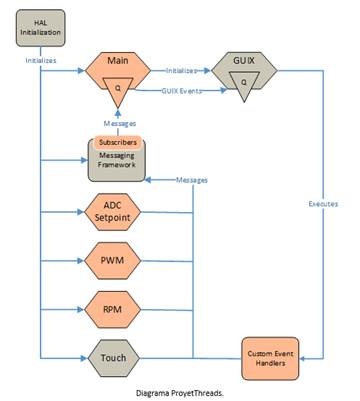
# 9 Software Design

The purpose of the Software Design process is to develop a detailed software design and produce verified software components that properly reflect the software design following standards and naming conventions defined by SW competency

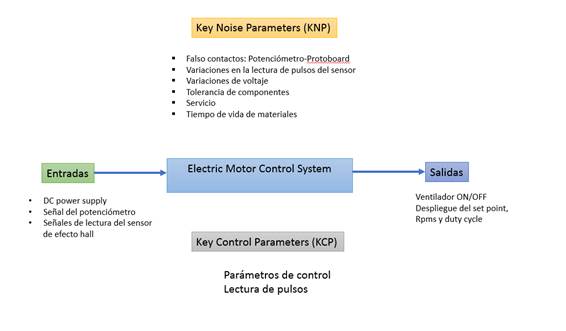
Software components are composed of software units. The strategy for Software Design is to create Software Detailed Design documents that implement the allocated software requirements for that design. Software detailed design documents will be used to describe each software function for the product. Software code and unit tests specifications will be developed from the detailed design. Software Static Analysis will be performed on the software code.

## 9.1 Diagrams

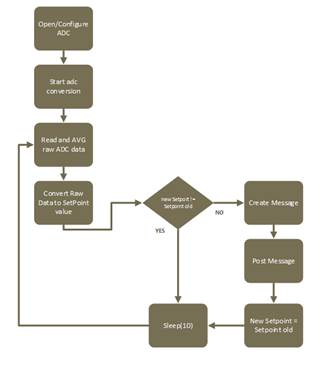
### 9.1.1 SystemDiagrams



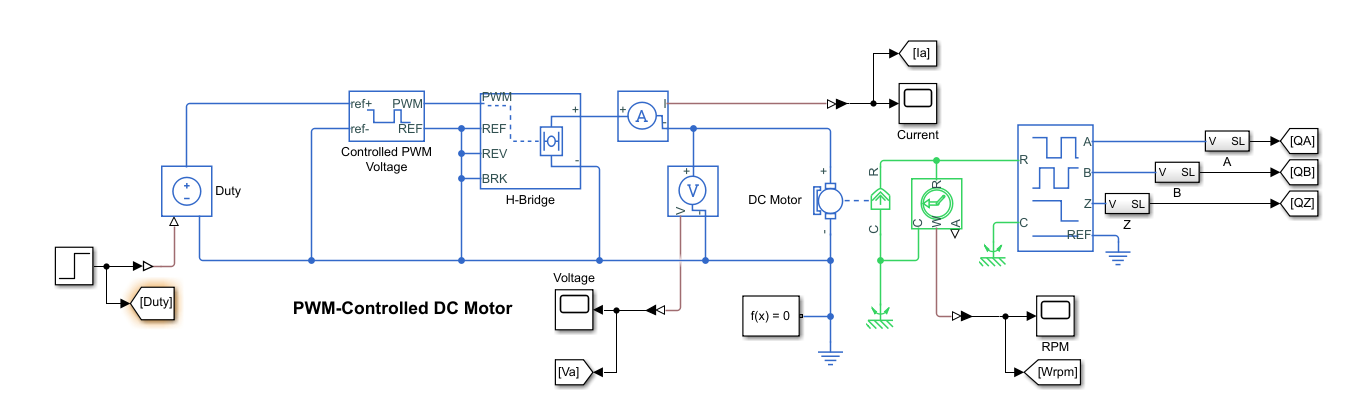
### 9.1.2 Parameter Control/ Pulse Reading



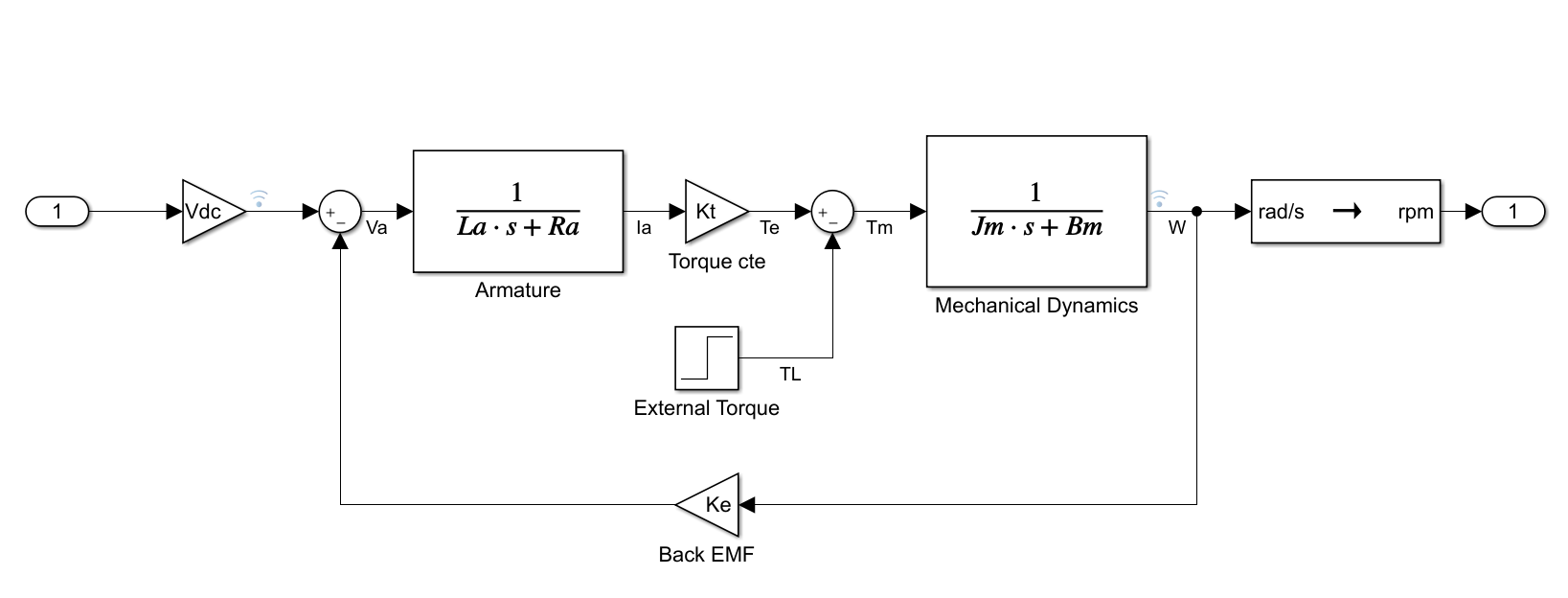
### 9.1.2 UML



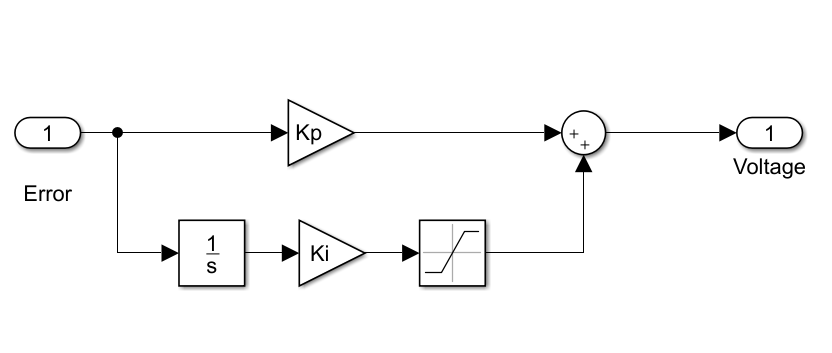
### 9.1.3 PWM-Controler DC Motor



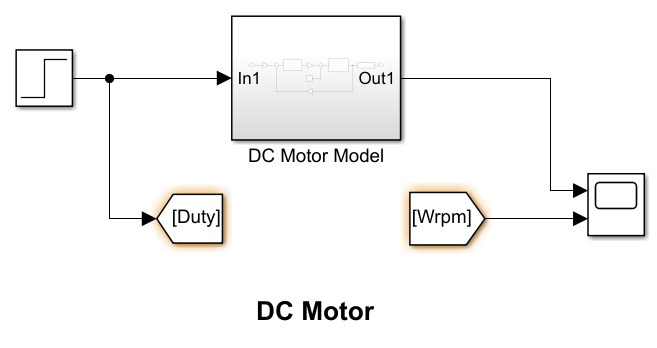
### Control System



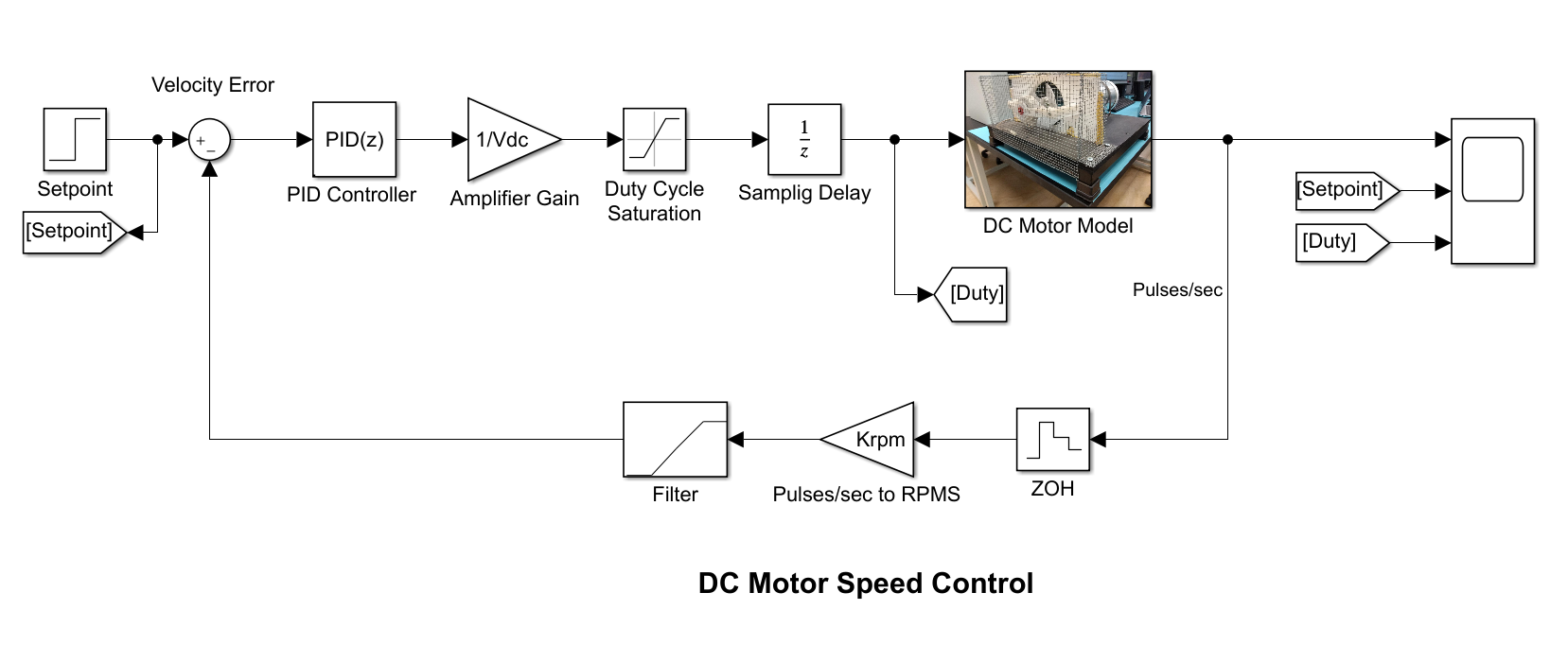
### Error Management First Order Function



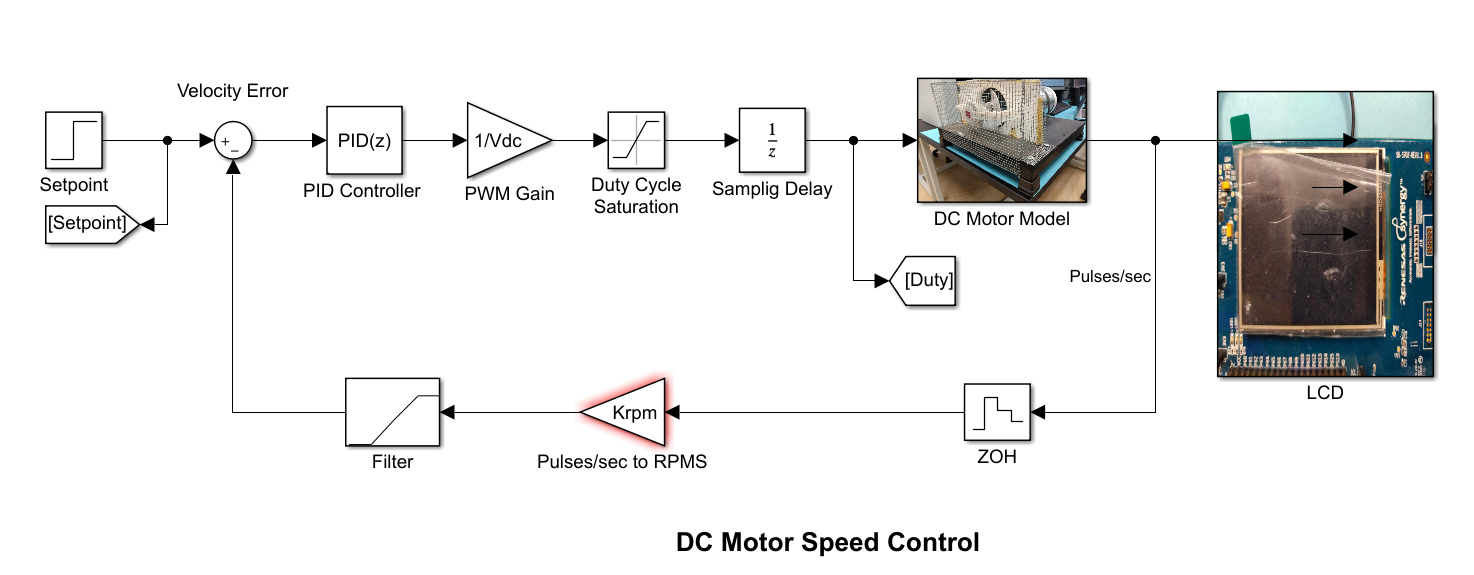
### Motor



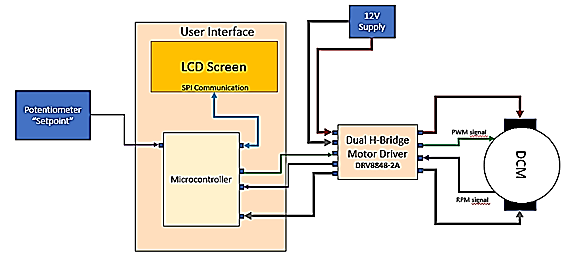
### Motor Speed Control



### Motor Speed Control And LCD System

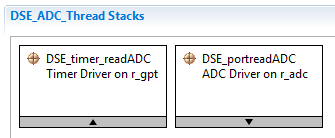


### Motor Speed Control And LCD System

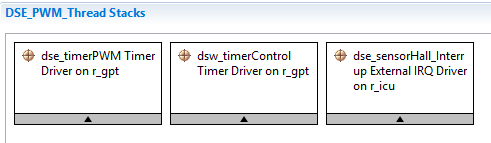


### Implementation

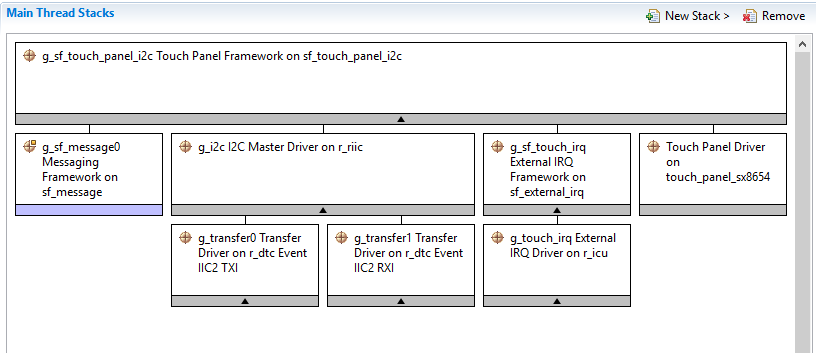
#### 9.1.10.1 ADC

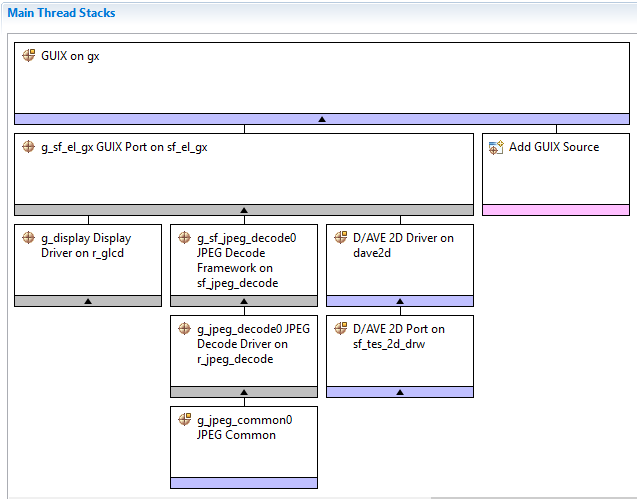


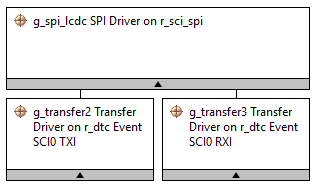
#### 9.1.10.2 PWM



#### 9.1.10.3 LCD







## Standars

Integrator project is going to be based on MISRA standar which provide world-leading best practice guidelines for the safe and secure application of both embedded control systems and standalone software.

### MISRA C:2012 (MISRA C3)

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.

### MISRA C 2012

**Table D.3. MISRA C 2012**

| **Rule/Directive** | **Summary** | **Description** | **Category** | **Related Coverity Checker** |
| --- | --- | --- | --- | --- |
| Directive 1.1 | Directive | Any implementation-defined behaviour on which the output of the program depends shall be documented and understood. | Required |  |
| Directive 2.1 | Directive | All source files shall compile without any compilation errors. | Required |  |
| Directive 3.1 | Directive | All code shall be traceable to documented requirements. | Required |  |
| Directive 4.1 | Directive | Run-time failures shall be minimized. | Required |  |
| Directive 4.2 | Directive | All usage of assembly language should be documented. | Advisory |  |
| Directive 4.3 | Directive | Assembly language shall be encapsulated and isolated. | Required | MISRA C-2012 Directive 4.3 |
| Directive 4.4 | Directive | Sections of code should not be "commented out". | Advisory | MISRA C-2012 Directive 4.4 |
| Directive 4.5 | Directive | Identifiers in the same name space with overlapping visibility should be typographically unambiguous. | Advisory | MISRA C-2012 Directive 4.5 |
| Directive 4.6 | Directive | Typedefs that indicate size and signedness should be used in place of the basic numerical types. | Advisory | MISRA C-2012 Directive 4.6 |
| Directive 4.7 | Directive | If a function returns error information, then that error information shall be tested. | Required | MISRA C-2012 Directive 4.7 |
| Directive 4.8 | Directive | If a pointer to a structure or union is never dereferenced within a Translation Unit, then the implementation of the object should be hidden. | Advisory | MISRA C-2012 Directive 4.8 |
| Directive 4.9 | Directive | A function should not be used in preference to a function-like macro where they are interchangeable. | Advisory | MISRA C-2012 Directive 4.9 |
| Directive 4.10 | Directive | Precautions shall be taken in order to prevent the contents of a header file being included more than once. | Required | MISRA C-2012 Directive 4.10 |
| Directive 4.11 | Directive | The validity of values passed to library functions shall be checked. | Required | MISRA C-2012 Directive 4.11 |
| Directive 4.12 | Directive | Dynamic memory allocation shall not be used. | Required | MISRA C-2012 Directive 4.12 |
| Directive 4.13 | Directive | Functions which are designed to provide operations on a resource should be called in an appropriate sequence. | Advisory | MISRA C-2012 Directive 4.13 |
| Directive 4.14 | Directive | The validity of values received from external sources shall be checked. | Required | MISRA C-2012 Directive 4.14 |
| Rule 1.1 | Rule | The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits. | Required | MISRA C-2012 Rule 1.1 |
| Rule 1.2 | Rule | Language extensions should not be used. | Advisory | MISRA C-2012 Rule 1.2 |
| Rule 1.3 | Rule | There shall be no occurrence of undefined or critical unspecified behaviour. | Required |  |
| Rule 2.1 | Rule | A project shall not contain unreachable code. | Required | MISRA C-2012 Rule 2.1 |
| Rule 2.2 | Rule | There shall be no dead code. | Required | MISRA C-2012 Rule 2.2 |
| Rule 2.3 | Rule | A project should not contain unused type declarations. | Advisory | MISRA C-2012 Rule 2.3 |
| Rule 2.4 | Rule | A project should not contain unused tag declarations. | Advisory | MISRA C-2012 Rule 2.4 |
| Rule 2.5 | Rule | A project should not contain unused macro declarations. | Advisory | MISRA C-2012 Rule 2.5 |
| Rule 2.6 | Rule | A function should not contain unused label declarations. | Advisory | MISRA C-2012 Rule 2.6 |
| Rule 2.7 | Rule | There should be no unused parameters in functions. | Advisory | MISRA C-2012 Rule 2.7 |
| Rule 3.1 | Rule | The character sequences /\* and // shall not be used within a comment. | Required | MISRA C-2012 Rule 3.1 |
| Rule 3.2 | Rule | Line-splicing shall not be used in // comments. | Required | MISRA C-2012 Rule 3.2 |
| Rule 4.1 | Rule | Octal and hexadecimal escape sequences shall be terminated. | Required | MISRA C-2012 Rule 4.1 |
| Rule 4.2 | Rule | Trigraphs should not be used. | Advisory | MISRA C-2012 Rule 4.2 |
| Rule 5.1 | Rule | External identifiers shall be distinct. | Required | MISRA C-2012 Rule 5.1 |
| Rule 5.2 | Rule | Identifiers declared in the same scope and name space shall be distinct. | Required | MISRA C-2012 Rule 5.2 |
| Rule 5.3 | Rule | An identifier declared in an inner scope shall not hide an identifier declared in an outer scope. | Required | MISRA C-2012 Rule 5.3 |
| Rule 5.4 | Rule | Macro identifiers shall be distinct. | Required | MISRA C-2012 Rule 5.4 |
| Rule 5.5 | Rule | Identifiers shall be distinct from macro names. | Required | MISRA C-2012 Rule 5.5 |
| Rule 5.6 | Rule | A typedef name shall be a unique identifier. | Required | MISRA C-2012 Rule 5.6 |
| Rule 5.7 | Rule | A tag name shall be a unique identifier. | Required | MISRA C-2012 Rule 5.7 |
| Rule 5.8 | Rule | Identifiers that define objects or functions with external linkage shall be unique. | Required | MISRA C-2012 Rule 5.8 |
| Rule 5.9 | Rule | Identifiers that define objects or functions with internal linkage should be unique. | Advisory | MISRA C-2012 Rule 5.9 |
| Rule 6.1 | Rule | Bit-fields shall only be declared with an appropriate type. | Required | MISRA C-2012 Rule 6.1 |
| Rule 6.2 | Rule | Single-bit named bit fields shall not be of a signed type. | Required | MISRA C-2012 Rule 6.2 |
| Rule 7.1 | Rule | Octal constants shall not be used. | Required | MISRA C-2012 Rule 7.1 |
| Rule 7.2 | Rule | A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type. | Required | MISRA C-2012 Rule 7.2 |
| Rule 7.3 | Rule | The lowercase character "l" shall not be used in a literal suffix. | Required | MISRA C-2012 Rule 7.3 |
| Rule 7.4 | Rule | A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char". | Required | MISRA C-2012 Rule 7.4 |
| Rule 8.1 | Rule | Types shall be explicitly specified. | Required | MISRA C-2012 Rule 8.1 |
| Rule 8.2 | Rule | Function types shall be in prototype form with named parameters. | Required | MISRA C-2012 Rule 8.2 |
| Rule 8.3 | Rule | All declarations of an object or function shall use the same names and type qualifiers. | Required | MISRA C-2012 Rule 8.3 |
| Rule 8.4 | Rule | A compatible declaration shall be visible when an object or function with external linkage is defined. | Required | MISRA C-2012 Rule 8.4 |
| Rule 8.5 | Rule | An external object or function shall be declared once in one and only one file. | Required | MISRA C-2012 Rule 8.5 |
| Rule 8.6 | Rule | An identifier with external linkage shall have exactly one external definition. | Required | MISRA C-2012 Rule 8.6 |
| Rule 8.7 | Rule | Functions and objects should not be defined with external linkage if they are referenced in only one translation unit. | Advisory | MISRA C-2012 Rule 8.7 |
| Rule 8.8 | Rule | The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage. | Required | MISRA C-2012 Rule 8.8 |
| Rule 8.9 | Rule | An object should be defined at block scope if its identifier only appears in a single function. | Advisory | MISRA C-2012 Rule 8.9 |
| Rule 8.10 | Rule | An inline function shall be declared with the static storage class. | Required | MISRA C-2012 Rule 8.10 |
| Rule 8.11 | Rule | When an array with external linkage is declared, its size should be explicitly specified. | Advisory | MISRA C-2012 Rule 8.11 |
| Rule 8.12 | Rule | Within a n enumerator list, the value of an implicitly-specified enumeration constant shall be unique. | Required | MISRA C-2012 Rule 8.12 |
| Rule 8.13 | Rule | A pointer should point to a const-qualified type whenever possible. | Advisory | MISRA C-2012 Rule 8.13 |
| Rule 8.14 | Rule | The restrict type qualifier shall not be used. | Required | MISRA C-2012 Rule 8.14 |
| Rule 9.1 | Rule | The value of an object with automatic storage duration shall not be read before it has been set. | Mandatory | MISRA C-2012 Rule 9.1 |
| Rule 9.2 | Rule | The initializer for an aggregate or union shall be enclosed in braces. | Required | MISRA C-2012 Rule 9.2 |
| Rule 9.3 | Rule | Arrays shall not be partially initialized. | Required | MISRA C-2012 Rule 9.3 |
| Rule 9.4 | Rule | An element of an object shall not be initialized more than once. | Required | MISRA C-2012 Rule 9.4 |
| Rule 9.5 | Rule | Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly. | Required | MISRA C-2012 Rule 9.5 |
| Rule 10.1 | Rule | Operands shall not be of an inappropriate essential type. | Required | MISRA C-2012 Rule 10.1 |
| Rule 10.2 | Rule | Expressions of essentially character type shall not be used inappropriately in addition and subtraction operation. | Required | MISRA C-2012 Rule 10.2 |
| Rule 10.3 | Rule | The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category. | Required | MISRA C-2012 Rule 10.3 |
| Rule 10.4 | Rule | Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category. | Required | MISRA C-2012 Rule 10.4 |
| Rule 10.5 | Rule | The value of an expression should not be cast to an inappropriate essential type. | Advisory | MISRA C-2012 Rule 10.5 |
| Rule 10.6 | Rule | The value of a composite expression shall not be assigned to an object with wider essential type. | Required | MISRA C-2012 Rule 10.6 |
| Rule 10.7 | Rule | If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type. | Required | MISRA C-2012 Rule 10.7 |
| Rule 10.8 | Rule | The value of a composite expression shall not be cast to a different essential type category or a wider essential type. | Required | MISRA C-2012 Rule 10.8 |
| Rule 11.1 | Rule | Conversions shall not be performed between a pointer to a function and any other type. | Required | MISRA C-2012 Rule 11.1 |
| Rule 11.2 | Rule | Conversions shall not be performed between a pointer to an incomplete type and any other type. | Required | MISRA C-2012 Rule 11.2 |
| Rule 11.3 | Rule | A cast shall not be performed between a pointer to object type and a pointer to a different object type. | Required | MISRA C-2012 Rule 11.3 |
| Rule 11.4 | Rule | A conversion should not be performed between a pointer to object and an integer type. | Advisory | MISRA C-2012 Rule 11.4 |
| Rule 11.5 | Rule | A conversion should not be performed from pointer to void into pointer to object. | Advisory | MISRA C-2012 Rule 11.5 |
| Rule 11.6 | Rule | A cast shall not be performed between pointer to void and an arithmetic type. | Required | MISRA C-2012 Rule 11.6 |
| Rule 11.7 | Rule | A cast shall not be performed between pointer to object and a non-integer arithmetic type. | Required | MISRA C-2012 Rule 11.7 |
| Rule 11.8 | Rule | A cast shall not remove any const or volatile qualification from the type pointed to by a pointer. | Required | MISRA C-2012 Rule 11.8 |
| Rule 11.9 | Rule | The macro NULL shall be the only permitted form of integer null pointer constant. | Required | MISRA C-2012 Rule 11.9 |
| Rule 12.1 | Rule | The precedence of operators within expressions should be made explicit. | Advisory | MISRA C-2012 Rule 12.1 |
| Rule 12.2 | Rule | The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand. | Required | MISRA C-2012 Rule 12.2 |
| Rule 12.3 | Rule | The comma operator should not be used. | Advisory | MISRA C-2012 Rule 12.3 |
| Rule 12.4 | Rule | Evaluation of constant expressions should not lead to unsigned integer wrap-around. | Advisory | MISRA C-2012 Rule 12.4 |
| Rule 12.5 | Rule | The sizeof operator shall not have an operand which is a function parameter declared as "array of type". | Mandatory | MISRA C-2012 Rule 12.5 |
| Rule 13.1 | Rule | Initializer lists shall not contain persistent side effects. | Required | MISRA C-2012 Rule 13.1 |
| Rule 13.2 | Rule | The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders. | Required | MISRA C-2012 Rule 13.2 |
| Rule 13.3 | Rule | A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator. | Advisory | MISRA C-2012 Rule 13.3 |
| Rule 13.4 | Rule | The result of an assignment operator should not be used. | Advisory | MISRA C-2012 Rule 13.4 |
| Rule 13.5 | Rule | The right hand operand of a logical && or || operator shall not contain persistent side effects. | Required | MISRA C-2012 Rule 13.5 |
| Rule 13.6 | Rule | The operand of the sizeof operator shall not contain any expression which has potential side effects. | Mandatory | MISRA C-2012 Rule 13.6 |
| Rule 14.1 | Rule | A loop counter shall not have essentially floating type. | Required | MISRA C-2012 Rule 14.1 |
| Rule 14.2 | Rule | A for loop shall be well-formed. | Required | MISRA C-2012 Rule 14.2 |
| Rule 14.3 | Rule | Controlling expressions shall not be invariant. | Required | MISRA C-2012 Rule 14.3 |
| Rule 14.4 | Rule | The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type. | Required | MISRA C-2012 Rule 14.4 |
| Rule 15.1 | Rule | The goto statement should not be used. | Advisory | MISRA C-2012 Rule 15.1 |
| Rule 15.2 | Rule | The goto statement shall jump to a label declared later in the same function. | Required | MISRA C-2012 Rule 15.2 |
| Rule 15.3 | Rule | Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement. | Required | MISRA C-2012 Rule 15.3 |
| Rule 15.4 | Rule | There should be no more than one break or goto statement used to terminate any iteration statement. | Advisory | MISRA C-2012 Rule 15.4 |
| Rule 15.5 | Rule | A function should have a single point of exit at the end. | Advisory | MISRA C-2012 Rule 15.5 |
| Rule 15.6 | Rule | The body of an iteration-statement or a selection-statement shall be a compound statement. | Required | MISRA C-2012 Rule 15.6 |
| Rule 15.7 | Rule | All if ... else if constructs shall be terminated with an else statement. | Required | MISRA C-2012 Rule 15.7 |
| Rule 16.1 | Rule | All switch statements shall be well formed. | Required | MISRA C-2012 Rule 16.1 |
| Rule 16.2 | Rule | A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement. | Required | MISRA C-2012 Rule 16.2 |
| Rule 16.3 | Rule | An unconditional break statement shall terminate every switch-clause. | Required | MISRA C-2012 Rule 16.3 |
| Rule 16.4 | Rule | Every switch statement shall have a default label. | Required | MISRA C-2012 Rule 16.4 |
| Rule 16.5 | Rule | A default label shall appear as either the first or the last switch label of a switch statement. | Required | MISRA C-2012 Rule 16.5 |
| Rule 16.6 | Rule | Every switch statement shall have at least two switch clauses. | Required | MISRA C-2012 Rule 16.6 |
| Rule 16.7 | Rule | A switch expression shall not have an essentially Boolean type. | Required | MISRA C-2012 Rule 16.7 |
| Rule 17.1 | Rule | The features of <stdarg.h> shall not be used. | Required | MISRA C-2012 Rule 17.1 |
| Rule 17.2 | Rule | Functions shall not call themselves, either directly or indirectly. | Required | MISRA C-2012 Rule 17.2 |
| Rule 17.3 | Rule | A function shall not be declared implicitly. | Mandatory | MISRA C-2012 Rule 17.3 |
| Rule 17.4 | Rule | All exit paths from a function with non-void return type shall have an explicit return statement with an expression. | Mandatory | MISRA C-2012 Rule 17.4 |
| Rule 17.5 | Rule | The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements. | Advisory | MISRA C-2012 Rule 17.5 |
| Rule 17.6 | Rule | The declaration of an array parameter shall not contain the static keyword between the []. | Mandatory | MISRA C-2012 Rule 17.6 |
| Rule 17.7 | Rule | The value returned by a function having non-void return type shall be used. | Required | MISRA C-2012 Rule 17.7 |
| Rule 17.8 | Rule | A function parameter should not be modified. | Advisory | MISRA C-2012 Rule 17.8 |
| Rule 18.1 | Rule | A pointer resulting from arithmetic on a pointer operand shall address an elements of the same array as that pointer operand. | Required | MISRA C-2012 Rule 18.1 |
| Rule 18.2 | Rule | Subtraction between pointers shall only be applied to pointers that address elements of the same array. | Required | MISRA C-2012 Rule 18.2 |
| Rule 18.3 | Rule | The relational operators >, >=, < and <= shall only be applied to pointers that point into the same object. | Required | MISRA C-2012 Rule 18.3 |
| Rule 18.4 | Rule | The +, -, += and -= operators should not be applied to an expression of pointer type. | Advisory | MISRA C-2012 Rule 18.4 |
| Rule 18.5 | Rule | Declarations should contain no more than two levels of pointer nesting. | Advisory | MISRA C-2012 Rule 18.5 |
| Rule 18.6 | Rule | The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist. | Required | MISRA C-2012 Rule 18.6 |
| Rule 18.7 | Rule | Flexible array members shall not be declared. | Required | MISRA C-2012 Rule 18.7 |
| Rule 18.8 | Rule | Variable-length array types shall not be used. | Required | MISRA C-2012 Rule 18.8 |
| Rule 19.1 | Rule | An object shall not be assigned or copied to an overlapping object. | Mandatory | MISRA C-2012 Rule 19.1 |
| Rule 19.2 | Rule | The union keyword should not be used. | Advisory | MISRA C-2012 Rule 19.2 |
| Rule 20.1 | Rule | #include directives should only be preceded by preprocessor directives or comments. | Advisory | MISRA C-2012 Rule 20.1 |
| Rule 20.2 | Rule | The ', " or \ characters and the /\* or // character sequences shall not occur in a header file name. | Required | MISRA C-2012 Rule 20.2 |
| Rule 20.3 |  | The #include directive shall be followed by either a <filename> or "filename" sequence. | Required | MISRA C-2012 Rule 20.3 |
| Rule 20.4 | Rule | A macro shall not be defined with the same name as a keyword. | Required | MISRA C-2012 Rule 20.4 |
| Rule 20.5 | Rule | #undef should not be used. | Advisory | MISRA C-2012 Rule 20.5 |
| Rule 20.6 | Rule | Tokens that look like a preprocessing directive shall not occur within a macro argument. | Required | MISRA C-2012 Rule 20.6 |
| Rule 20.7 | Rule | Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses. | Required | MISRA C-2012 Rule 20.7 |
| Rule 20.8 | Rule | The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1. | Required | MISRA C-2012 Rule 20.8 |
| Rule 20.9 | Rule | All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation. | Required | MISRA C-2012 Rule 20.9 |
| Rule 20.10 | Rule | The # and ## preprocessor operators should not be used. | Advisory | MISRA C-2012 Rule 20.10 |
| Rule 20.11 | Rule | A macro parameter immediately following a # operator shall not immediately be followed by a ## operator. | Required | MISRA C-2012 Rule 20.11 |
| Rule 20.12 | Rule | A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators. | Required | MISRA C-2012 Rule 20.12 |
| Rule 20.13 | Rule | A line whose first token is # shall be a valid preprocessing directive. | Required | MISRA C-2012 Rule 20.13 |
| Rule 20.14 | Rule | All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related. | Required | MISRA C-2012 Rule 20.14 |
| Rule 21.1 | Rule | #define and #undef shall not be used on a reserved identifier or reserved macro name. | Required | MISRA C-2012 Rule 21.1 |
| Rule 21.2 | Rule | A reserved identifier or macro name shall not be declared. | Required | MISRA C-2012 Rule 21.2 |
| Rule 21.3 | Rule | The memory allocation and deallocation functions of <stdlib.h> shall not be used. | Required | MISRA C-2012 Rule 21.3 |
| Rule 21.4 | Rule | The standard header file <setjmp.h> shall not be used. | Required | MISRA C-2012 Rule 21.4 |
| Rule 21.5 | Rule | The standard header file <signal.h> shall not be used. | Required | MISRA C-2012 Rule 21.5 |
| Rule 21.6 | Rule | The Standard Library input/output functions shall not be used. | Required | MISRA C-2012 Rule 21.6 |
| Rule 21.7 | Rule | The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used. | Required | MISRA C-2012 Rule 21.7 |
| Rule 21.8 | Rule | The library functions abort, exit and system of <stdlib.h> shall not be used. | Required | MISRA C-2012 Rule 21.8 |
| Rule 21.9 | Rule | The library functions bsearch and qsort of <stdlib.h> shall not be used. | Required | MISRA C-2012 Rule 21.9 |
| Rule 21.10 | Rule | The Standard Library time and date functions shall not be used. | Required | MISRA C-2012 Rule 21.10 |
| Rule 21.11 | Rule | The standard header file <tgmath.h> shall not be used. | Required | MISRA C-2012 Rule 21.11 |
| Rule 21.12 | Rule | The exception handling features of <fenv.h> should not be used. | Advisory | MISRA C-2012 Rule 21.12 |
| Rule 21.13 | Rule | Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF. | Mandatory | MISRA C-2012 Rule 21.13 |
| Rule 21.14 | Rule | The Standard Library function memcmp shall not be used to compare null terminated strings. | Required | MISRA C-2012 Rule 21.14 |
| Rule 21.15 | Rule | The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types. | Required | MISRA C-2012 Rule 21.15 |
| Rule 21.16 | Rule | The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type. | Required | MISRA C-2012 Rule 21.16 |
| Rule 21.17 | Rule | Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters. | Mandatory | MISRA C-2012 Rule 21.17 |
| Rule 21.18 | Rule | The size\_t argument passed to any function in <string.h> shall have an appropriate value. | Mandatory | MISRA C-2012 Rule 21.18 |
| Rule 21.19 | Rule | The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type. | Mandatory | MISRA C-2012 Rule 21.19 |
| Rule 21.20 | Rule | The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror shall not be used following a subsequent call to the same function. | Mandatory | MISRA C-2012 Rule 21.20 |
| Rule 22.1 | Rule | All resources obtained dynamically by means of Standard Library functions shall be explicitly released. | Required | MISRA C-2012 Rule 22.1 |
| Rule 22.2 | Rule | A block of memory shall only be freed if it was allocated by means of a Standard Library function. | Mandatory | MISRA C-2012 Rule 22.2 |
| Rule 22.3 | Rule | The same file shall not be open for read and write access at the same time on different streams. | Required | MISRA C-2012 Rule 22.3 |
| Rule 22.4 | Rule | There shall be no attempt to write to a stream which has been opened as read-only. | Mandatory | MISRA C-2012 Rule 22.4 |
| Rule 22.5 | Rule | A pointer to a FILE object shall not be dereferenced. | Mandatory | MISRA C-2012 Rule 22.5 |
| Rule 22.6 | Rule | The value of a pointer to a FILE shall not be used after the associated stream has been closed. | Mandatory | MISRA C-2012 Rule 22.6 |
| Rule 22.7 | Rule | The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF. | Required | MISRA C-2012 Rule 22.7 |
| Rule 22.8 | Rule | The value of errno shall be set to zero prior to a call to an errno-setting-function. | Required | MISRA C-2012 Rule 22.8 |
| Rule 22.9 | Rule | The value of errno shall be tested against zero after calling an errno-setting-function. | Required | MISRA C-2012 Rule 22.9 |
| Rule 22.10 | Rule | The value of errno shall only be tested when the last function to be called was an errno-setting-function. | Required | MISRA C-2012 Rule 22.10 |

### How to Achieve MISRA Compliance

Achieving MISRA compliance takes knowledge, skill, and the right tools.  
Here are some steps you can take to comply with MISRA.

1. **Know the Rules:** You need to know the MISRA coding rules pertinent to which version of C or C++ you’re using.
2. **Check Your Code Constantly:** Continuously inspecting your code for violations is the best way to improve quality.
3. **Set Baselines:** Embedded systems come with legacy codebases. By setting baselines, you can focus on making sure your new code is compliant.
4. **Prioritize Violations Based on Risk:** You could have hundreds or even thousands of violations in your code. That’s why it’s important to prioritize rule violations based on risk severity. Some static code analysis tools can do this for you.
5. **Document Your Deviations:** Sometimes there are exceptions to the rule. But when it comes to compliance, every rule deviation needs to be well-documented.
6. **Monitor Your MISRA Compliance:** Keep an eye on how MISRA compliant your code is. Using a static code analyzer makes this easier by automatically generating a compliance report.
7. **Choose the Right Static Code Analyzer:** Choosing the right static code analyzer makes everything else easy. It takes care of scanning your code — new and legacy — for violations. It prioritizes vulnerabilities based on risk.

## Naming Convention

The naming convention defined for this Integrator Project are defined in the next sections:

### Files Names

* All the file names shall contain DSE\_<File Name> as a prefix, for instance:
  + DSE\_<Module name>\_config.h
  + DSE\_< Module name >.h
  + DSE\_< Module name >.c

### 9.3.2 Definitions, Structures and Specials Cases

* All definitions and macros shall be defined in Capital case.
* All structures shall contain DSE\_<Struct Name> in toggle case.
* All enums shall contain DSE\_<STATE/FUNC>\_<STATE NAME> prefix.
* Enum must contain a DSE\_<STATE/FUNC>\_MAX\_<VALUE>
* Type defined shall contain the <TypeVariableName>\_T indicating this is a typedefined by user.}

### Threads names

* DSE\_<Name>\_Thread
* DSE\_ADC\_Thread
* DSE\_screen\_Thread

### 9.3.4 CallBacks

* dsw\_<Name>\_CallBack
* dsw\_timerControl\_CallBack
* dsw\_ControlSystem\_CallBack

### 9.3.4 Interrups

* dse\_<Name>\_Interrup
* dse\_ADC\_Interrup

### 9.3.5 Variable names

All variables shall :

* Set in lower case.
* Contain the prefix:  dse\_<variable name>, for instance:
  + uint8\_t dse\_variable1;
  + boolean dse\_variable2;
  + uint16\_t dse\_variable3;
  + uint32\_t dse\_variable4;

### 9.3.6 Function names

All functions  must contain as a prefix: Dse\_Func<function name> as follow:

* Dse\_Func <Action\_1>
* Dse\_Func <Action\_2>
* Dse\_Func <Action\_3>

#### 9.3.7 Comments in Files

Comments in File

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

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
TODO  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

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

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
BUG  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

* All the comments allowed shall be block comments.

### Header Files

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

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

/\* ======== GLOBAL CONSTANT MACROS =======\*/

/\* ======== Global Data Types & Structures ========\*/

/\* ======== GLOBAL FUNCTION PROTOTYPES ========\*/

### Source Files

Every file shall contain the following comment at the top:

/\* ======================================================== \*/  
/\*  
 \* file **<file name with extention>**  
 \*  
 \* description  **...**  
 \*  
 \* version version: **xx**   
 \* author  derived\_by:       **Lastname, Name.**  
 \* date    date\_modified:    **weekday Month dd, yyyy. hh:mm:ss**  
 \*  
 \*------------------------------------------------------------------------------

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

/\* ======== Includes ========\*/  
  
/\* ======== GLOBAL CONSTANT MACROS =======\*/

/\* ======== Global Data Types & Structures ========\*/

/\* ======== GLOBAL FUNCTION PROTOTYPES ========\*/

# Software Testing

This section is based Software Testing process from SPICE. The purpose of the Software Testing process is to confirm that the integrated software meets the defined software requirements.

### 10.1 Verification, Black Box Testing Strategy

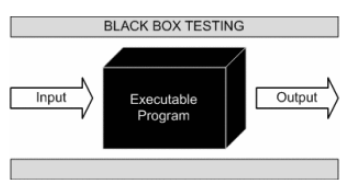
Verification team is responsible for creating test cases in order to find any issue that mismatches the desired functionality or established requirements, with the current behavior. Also Verification Team defines the different testing techniques used to ensure the product quality, that means that each system requirement is implemented correctly and is ready to be delivered.

Primary Objective for verification team is verifying that a software product meets the requirements by testing the product. Also verification team determine the software quality under test identifying faults and bugs.

Secondary Objective for verification process is firstly identify and expose all issues found and associated risks, secondly communicate all known issues, to Software team, and finally ensure that all issues are addressed in an appropriate matter before the software release. In order to reach this goal, verification team shall apply careful and methodical testing to ensure that each module of the product is scrutinized and, consequently, the issues (bugs) found during testing are dealt appropriately**.**

#### 10.1.1 Verification, Black Box Testing Process

1. Generate test cases that ensure compliance with the requirements.
2. Perform testing: integration testing, system testing, functional testing, regression testing and unit testing.
3. Find and report any behavior that mismatches the desired functionality or established requirements.
4. Generate bug ZenHub tickets that describes the issues found in the software and provide the support needed to reproduce and solve the issue.
5. Generate reports with the behavior of the device in each software version.



#### 10.1.2 Manual Testing

The Verification team perform the Manual testing, following these steps:

1. Read and understand the product requirements.
2. Draft Test cases that cover all the requirements mentioned in the documentation.
3. Review, correction and approved the test cases with Peer Review event, including all functional competencies for this revision.
4. Test cases execution
5. Report and create bugs tickets.
6. Report the results with program manager
7. Once bugs are fixed, again execute the failing test cases to verify they pass in the next software release.

### 10.2 White Box Testing Strategy

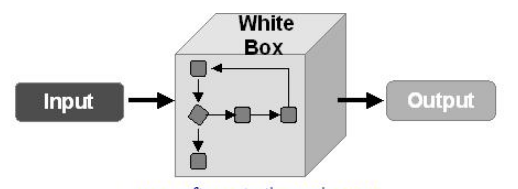
The strategy for Software Testing is to ensure that all software requirements have been tested and passed. Many of the software requirements are functional requirements and are tested as part of Systems Testing. If a software requirement is not a functional requirement and cannot be tested as part of the Systems testing, the SW team generates a test plan for these requirements and the testing is executed by the SW team.

Software Testing by the software team will be executed at each major delivery milestone as outlined in the detailed SW schedule in ZenHub, beginning with DV release. Software testing will be based upon Software Requirements as defined above.

#### 10.2.1 Verification, White Box Testing Process

1. Generate test cases that ensure compliance with the requirements.
2. Perform testing: integration testing, system testing, functional testing, regression testing and unit testing.
3. Find and report any behavior that mismatches the desired functionality or established requirements.
4. Generate bug ZenHub tickets that describes the issues found in the software and provide the support needed to reproduce and solve the issue.

Generate reports with the behavior of the device in each software version.



#### 10.2.2 Manual Testing

The Verification team perform the White Box testing, following these steps:

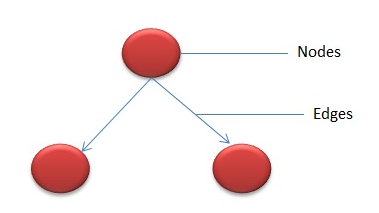
1. Read and understand the product requirements.
2. Draft Test cases that cover all the requirements mentioned in the documentation.
3. Review, correction and approved the test cases with Peer Review event, including all functional competencies for this revision.
4. Test cases execution using Gtest from Google
5. Report and create bugs tickets if some issue was found during the Test Sets.
6. Report the results with program manager
7. Once bugs are fixed, again execute the failing test cases to verify they pass in the next software release.

### 10.3 Cyclomatic Complexity Redundance index

Cyclomatic complexity is a software metric used to measure the complexity of a program. These metric, measures independent paths through program source code. Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths. Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

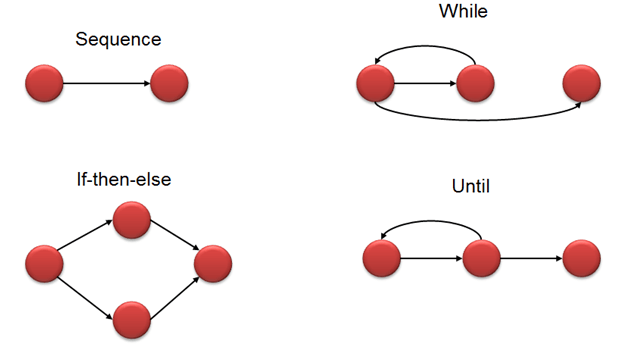
This metric was developed by Thomas J. McCabe in 1976 and it is based on a control flow representation of the program. Control flow depicts a program as a graph which consists of Nodes and Edges.

In the graph, Nodes represent processing tasks while edges represent control flow between the nodes.



#### Flow graph notation for a program:

Flow Graph notation for a program is defines. several nodes connected through the edges. Below are Flow diagrams for statements like if-else, While, until and normal sequence of flow.



#### How to Calculate Cyclomatic Complexity

##### Mathematical representation:

Mathematically, it is set of independent paths through the graph diagram. The Code complexity of the program can be defined using the formula -

V(G) = E - N + 2

Where,

E - Number of edges

N - Number of Nodes

V (G) = P + 1

Where P = Number of predicate nodes (node that contains condition)

Example:

i = 0;

n=4; //N-Number of nodes present in the graph

while (i<n-1) do

j = i + 1;

while (j<n) do

if A[i]<A[j] then

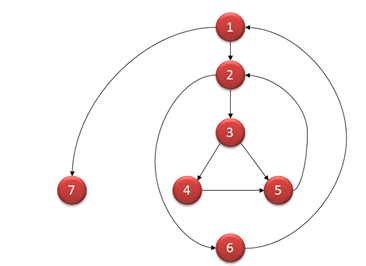
swap(A[i], A[j]);

end do;

i=i+1;

end do;

Flow graph for this program will be



Steps to be followed:

The following steps should be followed for computing Cyclomatic complexity and test cases design.

**Step 1** - Construction of graph with nodes and edges from the code  
**Step 2** - Identification of independent paths  
**Step 3**- Cyclomatic Complexity Calculation  
**Step 4**- Design of Test Cases

Once the basic set is formed, TEST CASES should be written to execute all the paths.

#### ****10.3.4 Complexity Table****

Following table gives overview on the complexity number and corresponding meaning of v (G):

|  |  |
| --- | --- |
| **Complexity Number** | Meaning |
| 1-10 | * Structured and well written code * High Testability * Cost and Effort is less |
| 10-20 | * Complex Code * Medium Testability * Cost and effort is Medium |
| 20-40 | * Very complex Code * Low Testability * Cost and Effort are high |
| >40 | * Not at all testable * Very high Cost and Effort |

#### Tools for Cyclomatic Complexity calculation:

Many tools are available for determining the complexity of the application. Some complexity calculation tools are used for specific technologies. Complexity can be found by the number of decision points in a program. The decision points are if, for, for-each, while, do, catch, case statements in a source code.

Examples of tools are

* [OCLint](https://github.com/oclint/oclint) - Static code analyzer for C and Related Languages
* [devMetrics](https://devmetrics.co/) - Analyzing metrics for[C#](https://www.guru99.com/c-tutorial.html)projects
* Reflector Add In - Code metrics for .NET assemblies
* [GMetrics](https://github.com/dx42/gmetrics) - Find metrics in[Java](https://www.guru99.com/java-tutorial.html)related applications
* [NDepends](https://www.ndepend.com/) - Metrics in Java applications

# 11 Release

The purpose of the Product Release process is to control the release of Software.

### 11.1 Strategy

The strategy for Product Release is to ensure that all items required for the product release have been completed, reviewed and approved prior to final release.

*Software Iteration*: After software engineer is finished with a phase of functional implementation, the software needs to be archived with a final baseline in GitHub.

*Software Issue*: After software engineer is finished with a phase of functional implementation (several iterations in conjunction), the software needs to be archived with a baseline in GitHub and made available for different groups to test and validate. The issue process makes an archive or image of the product software at a specific point in a project.

*Software Release*: After the different groups (Test and Systems) evaluate and approve the software and EEPROM map, the release analysts promote the software to a “released” state and produce a document called a Release Memo *(only in the case of Build and Production release*)

#### Release Checklists

Release checklists will be completed for every customer release.

The general content of the release checklists is specified below:

* Configuration Item Release List
  + Software Documentation and versions
  + Tool documentation and versions
* Integration Checks
  + ZenHub bugs tickets solved
  + Verification of checksums
  + Code compilation parameters verified
* Bench Checks
  + Verify customer tool functionality
  + Verify fault-free operation
  + Ensure acceptable stack operation
  + Verify hardware related parameters are OK
  + Measure task time/throughput

## 11.1 Software Development Folder

The Software Development Folder shall have the next structure:

* 1. Requirements
  2. Planning
  3. Desing
  4. Verification
  5. Quality Document
  6. Results
  7. Software Delopment Plan

## 11.2 Integration Test Strategy

The integration team shall assurance the release for the next competency for that reason they have to apply Basic Functionality Test as:

* Bootloader
* Reprograming Function
* Validate SW Versions
* Resets
* Communications are enables
* Transitions between Modules

## 11.3 Validation Test Strategy

#### 11.3.1 Validation Process

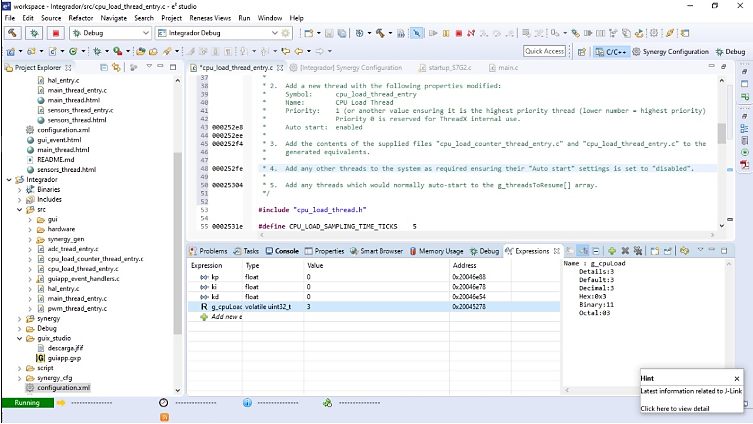
1. Generate test cases that ensure compliance with the requirements.
2. Perform testing: integration testing, system testing, functional testing, regression testing and unit testing.
3. Find and report any behavior that mismatches the desired functionality or established requirements.
4. Generate bug ZenHub tickets that describes the issues found in the software and provide the support needed to reproduce and solve the issue.
5. Generate reports with the behavior of the device in each software version.

#### 11.3.2 Manual Testing

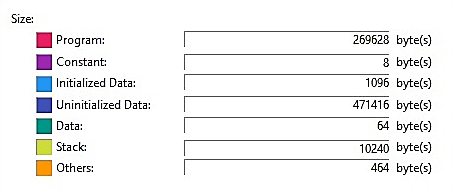
The Verification team perform the Manual testing, following these steps:

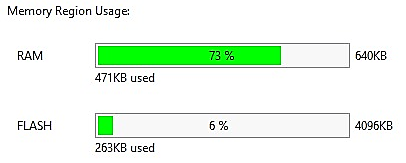
1. Read and understand the product requirements.
2. Draft Test cases that cover all the requirements mentioned in the documentation.
3. Review, correction and approved the test cases with Peer Review event, including all functional competencies for this revision.
4. Test cases execution
5. Report and create bugs tickets.
6. Report the results with program manager
7. Once bugs are fixed, again execute the failing test cases to verify they pass in the next software release.

## 11.4 Throughput and Flash RAM measurement



**Flash Memory**





# 12 Results

The implemented control gives us a variation of 100 Rpm when it is trying to stabilize by itself. Below is an example how control is trying to stabilizing:



As you can see, the graphics are according to requirements given by customer. Duty Cycle is changing to comply the set point given by user, in fact, the duty cycle variation can be, between 0% to 100%, depends on PID Control; Due to driver given by customer when duty cycle is 0% the motor tries to comply the maximum velocity also when duty cycle is 100% the motor stops.

The diagnostics could not be implemented due to development time because the documentation was so extensive by the time given by costumer; the diagnostics were included into Black box (Test cases) regarding Software Development Plan.

The black box test was executed; all the test cases were designed according requirements.

# 13 Lessons Learned

* + 1. The team must have more members to develop this project or extend the delivery time to complete it.
    2. Organize the time in a better way to finish all the works assigned.
    3. The method to implement a good project based on process documentation.
    4. Develop test cases to evaluate the functionality according requirements.
    5. Add comments to code is a good practice to do more easily the understanding for external engineers.
    6. Work in team is really important when you are going to develop a long project.

References:

Mccabe's Cyclomatic Complexity: Calculate with Flow Graph (2019)

<https://www.guru99.com/cyclomatic-complexity.html>