TEST WRITING HANDBOOK

CODING RULES, TEST ARCHITECTURE, IMPLEMENTATION PROCESS

# Version History

|  |  |  |  |
| --- | --- | --- | --- |
| **Version Number** | **Version Date** | **Author** | **Description of Changes** |
| 1.0 | 03.AUG.2017 |  | Draft revision |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Table of Contents

[Version History 2](#_Toc490218377)

[Table of Contents 3](#_Toc490218378)

[Reference Documents 5](#_Toc490218379)

[Definitions, Acronyms, and Abbreviations 6](#_Toc490218380)

[1. Objectives 7](#_Toc490218381)

[2. Target audience 7](#_Toc490218382)

[3. Coding Rules 8](#_Toc490218383)

[3.1. ProveTech 8](#_Toc490218384)

[3.1.1. Self-documenting code convention 8](#_Toc490218385)

[3.1.2. Naming Conventions 11](#_Toc490218386)

[3.1.3. Functions and Sub procedures 13](#_Toc490218387)

[3.1.4. Structuring the files 15](#_Toc490218388)

[3.1.5. Files and Folder Organization 18](#_Toc490218389)

[3.1.6. Visual Basic 19](#_Toc490218390)

[3.1.7. General Formatting 22](#_Toc490218391)

[3.1.8. Editor Configuration 24](#_Toc490218392)

[4. Test Strategy 25](#_Toc490218393)

[4.1. Test Strategies Definitions <TO\_DO> 25](#_Toc490218394)

[4.2. Test Types <TO\_DO> 25](#_Toc490218395)

[4.3. Test Techniques <TO\_DO> 25](#_Toc490218396)

[4.3.1. Static 25](#_Toc490218397)

[4.3.2. Dynamic 25](#_Toc490218398)

[4.4. Test Tools <TO\_DO> 25](#_Toc490218399)

[4.5. Robustness Tests <TO\_DO> 25](#_Toc490218400)

[4.6. Cyber Security Tests 26](#_Toc490218401)

[4.6.1. Fuzz Testing 26](#_Toc490218402)

[4.6.2. <TO\_DO> 27](#_Toc490218403)

[4.7. Test Architecture 28](#_Toc490218404)

[4.7.1. Script architecture (ProveTech) 28](#_Toc490218405)

[4.7.2. Models architecture (Matelo) <TO\_DO> 33](#_Toc490218406)

[4.8. Test Specification 34](#_Toc490218407)

[4.8.1. Idea 34](#_Toc490218408)

[4.8.2. Description 34](#_Toc490218409)

[4.8.3. Output Format 35](#_Toc490218410)

[4.8.4. Rules 36](#_Toc490218411)

[4.9. Test Reusability 37](#_Toc490218412)

[5. Test Implementation Process 38](#_Toc490218413)

[5.1. Test scripts 38](#_Toc490218414)

[5.1.1. Test script states 38](#_Toc490218415)

[5.1.2. Test design 38](#_Toc490218416)

[5.2. Test models <TO\_DO> 38](#_Toc490218417)

[5.3. Test implementation 38](#_Toc490218418)

[5.4. Reviews 39](#_Toc490218419)

[5.4.1. Peer review 39](#_Toc490218420)

[5.4.2. Skill review 39](#_Toc490218421)

[5.5. Traceability 40](#_Toc490218422)

[5.5.1. Requirements 40](#_Toc490218423)

[5.5.2. Defects 40](#_Toc490218424)

[6. Appendix A 41](#_Toc490218425)

[6.1. Test Specification Guideline 41](#_Toc490218426)

[6.2. Test Specification Word Template 41](#_Toc490218427)

[6.3. Testing With Matelo 41](#_Toc490218428)

# Reference Documents

|  |  |
| --- | --- |
| **Reference Identification** | **Document Name** |
| [[R1]](https://msdn.microsoft.com/en-us/library/ektke1b0(v=vs.71).aspx) | VBScript Coding Conventions (Microsoft) |
| [[R2]](http://vfile.visteon.com/livelink/livelink.exe?func=ll&objId=10933116&objAction=browse&viewType=1) | Global\_Framework\_Architecture |
| [[R3]](http://vistway.visteon.com/CtrlWebIsapi.dll/?__id=docDetails.showDoc&doc=AB60F5CE82374A5DA57F35FC00563782&dpt=1) | Validation Types of Tests |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Definitions, Acronyms, and Abbreviations

|  |  |  |
| --- | --- | --- |
| **Acronym or Abbreviation** | **Word or Phrase** | **Definition** |
| RIO | Risk Issue Opportunity | ) |
| TC | Test Case |  |
| DUT | Device Under Test |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Objectives

The objectives of this document are to define rules for test strategy, test coding and implementation which will result in:

* readibility and understanding of tests
* maintanance and transfer of test projects
* tests reuse

# Target audience

Test engineers, senior test engineers, lead test engineers and technical professionals.

It is expected the reader of this document to be familiar with:

* global test process inside VISTWAY
* Global\_Framework\_Architecture document [R2]
* ProveTech and Matelo

# Coding Rules

## ProveTech

### Self-documenting code convention

**Writing understandable tests is a must!**

Adding comments in our scripts is mandatory but often we end up in a situation where comments are obsolete, fuzzy and misleading, too long, sometimes even more than the code itself. However, there are ways to reduce the need for comments and improve the description of a test case. We can make use of certain coding techniques to clarify our code, simply by using the programming language’s features to our advantage. Not only does this help make our code easier to understand, it can also help improve the design of the test case overall.

This is often called self-documenting. A three broad categories can be defined when describing the techniques for self-documenting code:

* Structure – the structure of code or directories is used to clarify the purpose
* Naming – functions and variables naming
* Syntax – we make use of (or avoid using) features of the language to clarify code

#### Structure

##### Move code into a function

Put specific actions in functions. The function’s name is descriptive of what it does, so the code no longer needs clarification. As an additional benefit, we now have a useful helper function that can be used elsewhere, so this method also helps reduce duplication.

Sub GoToSleep (Optional wait\_ms As Integer = SLEEP\_TIME\_MS, \_

Optional sleep\_type As String = "CUT\_COM")

Select Case UCase(sleep\_type)

Case "CUT\_COM"

System.SetSignal("CAN\_GlobalA\_LS\_Tx.TX\_enable",0) ' Stop CAN communication

Case "STOP\_VNS"

SetVNs("ALL","ALL",DEACTIVATE)

Case Else

Report.Error("GoToSleep() - unknown sleep type : " & sleep\_type)

End Select

SetAmbientLight(0) '0%

If (wait\_ms > 0) Then WaitTime(wait\_ms, time\_ms)

End Sub

##### Replace conditional expression with function

In order to make complex conditional expressions easier to use, they can be encapsulated in function with meaningful name.

Function IsAutomaticBench() As Boolean

Dim tbName As String

tbName = System.GetReplacementVar("TestBench")

tbName = LCase(tbName)

Return CBool(InStr(LCase(AUTOMATIC\_BENCHES), tbName))

End Function

##### Replace expression with variable

For complex calculations with multiple operands, additional variables can be used. Their names should correspond to the operations they represent.

multiplier = a \* b

divisor = c / d

factor = 13.54

value = (multiplier + divisor) \* factor

##### Code grouping

This can be used to indicate a relationship between the different parts of the code, so that anyone changing it in the future has an easier time finding which parts they may also need to touch.

Set\_Units(METRIC)

speedoVal = Util.rand(0, SPEEDO\_MAX)

tachoVal = Util.rand(0, TACH\_RPM\_END)

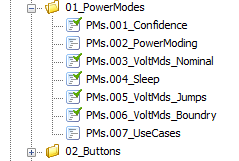
SetValueODI("Speedometer\_Value", speedoVal)

SetValueODI("Tachometer\_Value", tachoVal)

WaitTime(500, time\_ms)

Verify\_Gauges("ALL", "ON", speedoVal, METRIC, tachoVal)

##### Directory and file structure

When creating test cases inside ProveTech, follow a naming convention used in the project. For more details refer to chapter “Files and Folder Organization” in this document.

#### Naming

Follow established naming conventions. If you follow the same conventions as elsewhere in the tests, anyone reading it can make safe assumptions about the meanings of things based on what it means elsewhere.

For variable naming please refer to chapter “Variables” in this document.

For functions naming please refer to chapter “Procedures (Function / Sub)”in this document.

#### Syntax

##### Don’t use syntax tricks

Don’t use strange tricks that may confuse people. Always prefer simple expressions. Syntax tricks are not going to do anyone any favors.

Return ((a And b) Or CBool(5 Mod 2)) Xor c

##### Use named constants, avoid magic values

If you have special values in your code — such as numbers or string values — consider using a constants instead. Even if it seems clear now, when coming back to it in a month or two, nobody will have any idea why that particular number was put there.

#Region "SPEED\_SIGN"

Public Const TSM\_BASED = False

Public Const NAV\_BASED = True

Public Const NONE = 0

Public Const RECTANGULAR = 1

Public Const CIRCULAR = 2

#End Region

##### Avoid Boolean flags

Boolean flags can make for hard-to-understand code. For example:

Enable\_Speed\_Sign(ACTIVATE, True)

What is the meaning of True? Instead use the named constant NAV\_BASED from example in 3.2.2.

Enable\_Speed\_Sign(ACTIVATE, NAV\_BASED)

### Naming Conventions

#### Prefixes

Prefixes should always be in lower case letters. It is possible to combine prefixes by concatenating them – scope prefix then data type prefix.

##### Scope Prefixes

Global variables and constants should start with small letter “g”.

gintSVPowerOnOffState

##### Data Types Prefixes

For purposes of readability and consistency, use the following prefixes with descriptive names for variables in your code. Based on Microsoft VBScript Coding Conventions [R1].

|  |  |  |
| --- | --- | --- |
| **Subtype** | **Prefix** | **Example** |
| Boolean | bln | blnFound |
| Byte | byt | bytRasterData |
| Currency | cur | curValue |
| Date (Time) | dtm | dtmStart |
| Decimal | dec | decNumber |
| Double | dbl | dblTolerance |
| Huge | hug | hugNumber |
| Integer | int | intQuantity |
| Long | lng | lngDistance |
| Object | obj | objCurrent |
| SByte | sby | sbyResponse |
| Single | sng | sngAverage |
| String | str | strFirstName |
| Uhuge | uhg | uhgNumber |
| UInteger | uin | uinNumber |
| ULong | uln | ulnNumber |
| Variant | var | varCode |
| Error | err | errOrderNum |

#### Constants

Names of the constants should be meaningful noun or noun phrases with all letters capitalized.

Multiple words should be separated using the underscore “\_” character.

Public Const VM\_NOT\_OPR\_W\_RST As Integer = 0

Public Const VM\_NOT\_OPR\_LOW As Integer = 1

Public Const VM\_PART\_OPR\_LOW As Integer = 2

Public Const VM\_FULLY\_OPR As Integer = 3

Public Const VM\_PART\_OPR\_HIGH As Integer = 4

Public Const VM\_NOT\_OPR\_HIGH As Integer = 5

For example:

#### Variables

Variable and parameter names should be meaningful noun or noun phrases, without spaces, with the first letter lowercase and the first letter of any subsequent words capitalized. The first few letters of the variable name define the type of the variable. The remainder of the name should describes the role that the variable plays. If the variable has a global scope, start its name with global scope prefix.

Declare variables and constants at the beginning of procedures (functions or subs). VisualBasic allows you to declare variables anywhere within procedure, but this is not a good practice.

Indicate units – if you will represent numeric parameters, you can include the expected unit.

Don’t use shortcuts – a, b, i or j are not acceptable names.

Dim dtmTotalExecTime\_sec As Date

Refer to examples in chapter 4.1.1 inside this document.

#### Array Names

Use plural form when naming arrays. This will help understand the nature of these variables. Note that Variants are often used to hold arrays when they are passed as return values from functions. In this case, the Variant should be named according to the array naming convention.

Dim varPages As Variant

Dim strPagesNames() As String

#### Procedures (Function / Sub)

##### General Rules

Procedure names should be meaningful verbs or verb phrases that describe the functional abstraction of the procedure. The first word should be a verb. Procedure names should be without spaces, in mixed case, with the first letter uppercase, and the first letter of each subsequent word capitalized. Names should allow the reader to visualize the purpose of the procedure.

Avoid using vague words like “handle” or “manage”: HandleLinks(), ManageObjects(). What do either of these do?

Indicate return value: ReadParameter(), GetProductPowerMode (), GetProductState(). This is not something you can always do, but it’s helpful where it is possible.

##### Boolean Function Names

Procedures that return a Boolean results of checking a condition (or conditions) should indicate this by starting with Is, Has or similar.

Function IsAutomaticBench() As Boolean

##### Conversion Method Names

Procedures that converting format or type should indicate this by starting with To.

Function ToString(intValue As Integer) As String

Sub SetPowerMode(intPM As Integer)

##### State Modifying Method Names

Procedures that modify a state or mode should indicate this by starting with Set.

Sub SetVoltageValue(dblVolts As Double, Optional intWait\_ms As Integer = 500)

### Functions and Sub procedures

#### Size

Keep procedures to a manageable size. Each procedure should be concise enough that its entire purpose can be easily expressed and understood. One rule of thumb is that the well-constructed procedure should fit on a printed page. A more relevant rule when editing the code is that it fits on one screen in the development environment.

#### Parameters

Avoid long list of procedure parameters. If your function requires more than 5 to 6 parameters than it is bad designed. Either separate the function and its actions to simpler functions, or reconsider if all these parameters are needed or correctly defined. Functions with “heavy” list of parameters are difficult to be used and are prerequisites for mistakes.

Example of bad function design is shown below. All eight bits can be combined in a single byte parameter. In case it is needed bits to be separately managed a single array parameter can also be used:

Function ReadDTCStatusMask( ByVal bit0 As Integer, \_

ByVal bit1 As Integer, \_

ByVal bit2 As Integer, \_

ByVal bit3 As Integer, \_

ByVal bit4 As Integer, \_

bit5 As Integer,\_

bit6 As Integer, \_

ByVal bit7 As Integer, \_

ByVal DTCNumber1 As Byte, \_

ByVal DTCNumber2 As Byte, \_

ByVal DTCNumber3 As Byte, \_

Optional bIsSet As Boolean = True, \_

Optional ByVal bMsgReport = True) As Boolean

Solution 1:

Function ReadDTCStatusMask( bytBitsDTC As Byte, \_

bytDTCNumber1 As Byte, \_

bytDTCNumber2 As Byte, \_

bytDTCNumber3 As Byte, \_

Optional blnIsSet As Boolean = True, \_

Optional ByVal blnMsgReport = True) As Boolean

Solution 2:

Function ReadDTCStatusMask( intBitsDTC() As Integer, \_

bytDTCNumber1 As Byte, \_

bytDTCNumber2 As Byte, \_

bytDTCNumber3 As Byte, \_

Optional blnIsSet As Boolean = True, \_

Optional ByVal blnMsgReport = True) As Boolean

#### Return value

Always specify a return value when working with functions. This will protect you from undefined or erroneous test case behavior due to strange returned result. Don’t forget to specify return value before exiting the function, no matter if it is after all the actions inside are completed successfully or on error exit. Check carefully all the exits of the function.

Function GoToPage(intPageToGo As Integer) As Integer

…

Select Case intPageToGo

Case PAGE\_SPEED\_1 : strA2LPageName = "P\_PAGE1\_ENABLE"

Case PAGE\_SPEED\_2 : strA2LPageName = "P\_PAGE2\_ENABLE"

Case PAGE\_PERFORMANCE : strA2LPageName = "P\_PAGE3\_ENABLE"

Case PAGE\_LIST\_1 : strA2LPageName = "P\_PAGE4\_ENABLE"

Case PAGE\_LIST\_2 : strA2LPageName = "P\_PAGE5\_ENABLE"

Case PAGE\_URL : strA2LPageName = "P\_PAGE6\_ENABLE"

Case Else

Report.Error("GoToPage() - unknown page requested : " & CStr(intPageToGo))

Return gCurrentPage

End Select

If (0 = ReadPARAM(strA2LPageName,,,"INTEGER")) Then

Report.Error("GoToPage() - cannot go to DISABLED page " & strA2LPageName)

Return gCurrentPage

End If

…

Return gCurrentPage

End Function

### Structuring the files

#### Code Commenting

The following comment standards provide a consistent approach to documenting code.

##### Comment Formatting

Begin comments with an apostrophe (‘). All Visual Basic comments start with an apostrophe and end with a line break. This is also the default comment formatting of the ProveTech tool.

##### Procedure Comments

Begin each procedure with a comment block. The comment block shall precede the definition of the procedure and contain a description of the procedure – its author, date of the creation/last change, goal, inputs and outputs. Well-commented procedures make the code easier to understand.

'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

'\* Author : Vasil Kanev

'\* Date : 26.10.2015

'\*---------------------------------------------------------------------------

'\* NAME : CalcLagFilt

'\* GOAL : Function to calculate First Order Lag Filer

'\*---------------------------------------------------------------------------

'\* INPUT : dblStartVal - Start Value

'\* INPUT : dblK - P\_XX\_FIRST\_ORDER\_K

'\* INPUT : dblEndVal - End Value

'\* INPUT : dblCalcTime - How often to recalculate filter lag value (ex. 20ms)

'\* INPUT : dblEndTime - After this time filter lag value will be in CalcVal

'\*---------------------------------------------------------------------------

'\* Output : dblCalcVal - Calculated Value of signal XX

'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function CalcLagFilt (dblStartVal As Double, \_

dblK As Double, \_

dblEndVal As Double, \_

dblCalcTime As Double, \_

dblEndTime As Double) As Double

##### Declaration Comments

Describe the abstraction of all declared constants and variables, the meanings of which are not obvious from the name. Also describe any constraints of the variables.

Dim strV\_Cal\_File\_Part\_Number\_1 As String 'read by DID C2, range[0-0xFFFFF]

Dim strV\_Cal\_File\_Part\_Number\_Ac\_1 As String 'read by DID D2, range[0-0xFFFFF]

##### Over commenting

Over commenting the code is as bad as putting no comments at all. Too long comments or too much comments practically may “hide” the code and to impact badly the readability and understanding of the code.

* Try keeping the comments to their minimum and use them whenever the self-documenting code is not enough.
* Avoid obvious comments. Do not comment obvious code. If the code is not obvious, then first see if you can rewrite the code so that the use of comments will be unnecessary.

Dim intConnectionsNumber As Long 'number of connections

* Use short and concise comments.
* Use proper spelling and punctuation. Poorly written comments are distracting and can obscure the intended meaning.

#### Consistent Indentation

Indentation improves readability. Using the standard indentation improves maintainability, allowing the indent level of blocks of code to be easily changed.

Use four spaces for each indent level. The default tab spacing of in VisualBasic is four spaces, and this should be maintained.

Four spaces is enough to make the indentation level obvious, without taking up too much horizontal space.

For intIndPM = PM\_OFF To PM\_CRANK

If (intIndPM = PM\_RUN) And (intIndPM <> gintCurrent\_PM) Then

PrepareSubscribeODI()

SetValueSignal(pSysPwrMd, intIndPM)

SubscribeODI()

WaitTime(500, time\_ms)

Else

SetValueSignal(pSysPwrMd, intIndPM)

WaitTime(500, time\_ms)

End If

gintCurrent\_PM = intIndPM

SetValueSignal(pSysBkUpPwrMd, RandomWithExclude(PM\_OFF, PM\_CRANK, Array(indPM)))

WaitTime(500, time\_ms)

If (intIndPM = PM\_RUN) Then

Verify\_PowerMode(intIndPM, ON\_STATE)

Else

Verify\_PowerMode(intIndPM, OFF\_STATE)

End If

Next

#### Vertical alignment

Always try align vertically your code. This improves significantly readability of the code.

Function CalcLagFilt (dblStartVal As Double, \_

dblK As Double, \_

dblEndVal As Double, \_

dblCalcTime As Double, \_

dblEndTime As Double) As Double

Dim intInd As Integer

Dim strPageEnblVal As String

Dim strPageState As String

Dim strPageCal As String

Dim strPageName As String

Dim strCalString As String

SetPARAM("P\_ETHERNET\_ENABLE" , "1" , , INIT\_ONLY) 'Enable

SetPARAM("P\_INFO3\_RESULT\_TIME" , "250", , INIT\_ONLY) '250 ms

SetPARAM("P\_INFO3\_PROCESSING\_TIME", "200", , INIT\_ONLY) '200 ms

SetPARAM("P\_INFO3\_SEARCH\_INTERVAL", "200", , INIT\_ONLY) '200 ms

bytParamData(0) = CByte(Left (SW\_VER,2 ))

bytParamData(1) = CByte(Mid (SW\_VER,3,2))

bytParamData(2) = CByte(Mid (SW\_VER,5,2))

bytParamData(3) = CByte(Right(SW\_VER,2 ))

#### Code Grouping

Put a blank line between logical sections of a procedure. This improves readability.

Report\_step("MANUAL\_2")

Report.AddRequirements("SysFS\_DI-525.R0")

Report.AddRequirements("SysFS\_DI-562.R3")

SetPARAM("P\_DISPLAY\_MANUAL\_GEAR\_ENABLE", "1")

SetValueODI("Gear\_Position\_Display\_Manual", MANUAL)

GoToPage(PAGE\_PERFORMANCE)

VerifyGears(R)

#### DRY (Don't Repeat Yourself) Principle

It is a [principle](https://en.wikipedia.org/wiki/Principle#Principle_as_axiom_or_logical_fundament) aimed at reducing repetition of all kinds. The principle states:

*"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."*

When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically unrelated elements. Additionally, elements that are logically related all change predictably and uniformly, and are thus kept in [sync](https://en.wikipedia.org/wiki/Synchronization). The principle is based mostly on using [functions](https://en.wikipedia.org/wiki/Method_%28computer_science%29) and [subroutines](https://en.wikipedia.org/wiki/Subroutine). Use basic functions and subroutines to define basic actions. Use this basic elements to build more complex actions and so on.

#### Avoid Deep Nesting

It is hard to understand beyond three levels of nesting. This is known as “Dangerously Deep Nesting”. The code is hard to read and prone to logical errors.

Bad example: Good example:

If (cond1) Then

If (cond2) Then

' do something

ElseIf (cond3) Then

If (cond4) Then

' do something

Else

If (cond5) Then

If (cond6) Then

If (cond7) Then

' do something

End If

End If

End If

End If

End If

Else

' do something

End If

If (cond1) Then

Select Case condition

Case cond2

' do something

Case cond3

ExecuteOnCombinedCond(cond4,cond5,cond6,cond7)

Case Else

' unknown condition

' Report error

End Select

Else

' do something

End If

Sub ExecuteOnCombinedCond(cond4, cond5, cond6, cond7)

If cond4 Then

' do something

Else

If (cond5) And (cond6) And (cond7) Then

' do something

End If

End If

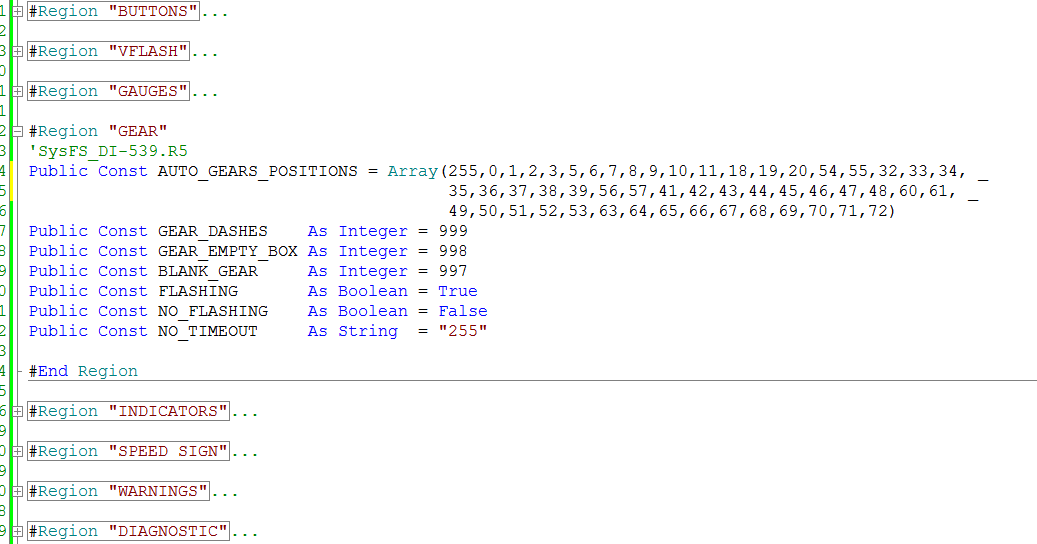
End Sub

#### Limit Line Length

In order to give a comprehensive look of your test cases try fit your code in certain vertical limit. This will improve the readability as there will be no too long lines going outside of the editor’s visible area. If external editor is used refer to chapter 4.7.2 of this document.

#### Regions

You can use code regions to organize related pieces of code. You can click the + or - sign to the left of a region's start to expand or collapse its code. This makes the code easier to read (you can hide things you are not interested in) and easier to manage (search functions only search code that is expanded so you can restrict your search to relevant code). Region directive is useful when you have lot of code and you need to add formatting to it.



### Files and Folder Organization

It is a must to organize test cases logically in test folders and in certain well defined order. This will provide a good understanding for the purpose of each test, its connection to particular feature and test case’s uniqueness. For details about ProveTech project structure refer to chapter 4.3 Project Structure in [R2].

Since requirements and defects by their nature are unique, the same shall be applied to the tests. A unique ID is used for each test created in ProveTech. This ID is unique for the project, but it also gives information what the test is about.

Example for test case ID definition is FFF.XXX, where:

* ‘FFF’ is the feature abbreviation

|  |  |  |
| --- | --- | --- |
| **Feature** | **Abbreviation** | **Short description** |
| Power Modes | PMs | Tests PM confidence, backup power mode and general sleep / wake up sequences. |
| Diagnostics | DIA | Tests HUD’s diagnostic services |
| Warnings | WRN | Tests the behavior of each applicable driver alert (warning). Functional behavior impacted |
| Indicators | IND | Tests the HUD’s telltales |
| Buttons (Electrical interfaces) | BTN | Test the correct interpretation of the inputs from the electrical interfaces for the HUD. Navigation through pages is also included |
| Gauges | GAU | Tests the HUD’s speedometer and tachometer |
| Human Machine Interface | HMI | Tests pages structure and user interaction with HUD |
| Dimming | DIM | Tests HUD’s back lighting (Dimming) |
| Robustness | RBT | Stress tests |

* ‘XXX’ is the number of the test

Feature abbreviation can be defined as in the table below:

### Visual Basic

#### If-ElseIf-Else

Format the If-ElseIf-Else structure like the example below. Always put constants on the left.

Always use Else statement reporting error – refer to chapter “Report.Error()” in this document.

If (ST\_ON = intIndState) Then

If (Indicator(SHIFT\_UP).clr(GREEN) = strColor) Or ("ALL" = strColor) Then

SetValueODI("DDH\_109\_UP\_DOWN\_SHIFT", 0) ' 0 = Gear Shift Up Active

Else

Report.Error("Set\_Indicator() : wrong Gear Shift Up color - " & strColor)

End If

ElseIf (ST\_OFF = intIndState) Then

If (Indicator(SHIFT\_UP).clr(GREEN) = strColor) Or ("ALL" = strColor) Then

SetValueODI("DDH\_109\_UP\_DOWN\_SHIFT", 2) ' 2 = No Gear Shift Indicators

Else

Report.Error("Set\_Indicator() : wrong Gear Shift Up color - " & strColor)

End If

Else

Report.Error("Set\_Indicator() : wrong Gear Shift Up state")

End If

#### Select Case

Format the Select-Case structure like the example below.

When the Select Case expression is string type use UCase() or LCase() functions to make all letters the same case. This will make the test script more robust to mistakes with characters’ case.

Always use Case Else statement reporting error – refer to chapter 4.5.4.

Select Case UCase(strColor)

Case "ALL"

SetValueODI("Acc\_Tracking\_Lead\_Vehicle\_Solid\_On", intIndState)

SetValueODI("TrackingVehicle", intIndState)

Case "GREEN"

SetValueODI("Acc\_Tracking\_Lead\_Vehicle\_Solid\_On", intIndState)

Case "AMBER"

SetValueODI("TrackingVehicle", intIndState)

Case Else

Report.Error("Set\_Indicator() : wrong VA color - " & strColor)

End Select

#### Loops

##### While

Format the While loop structure like the example below.

Always initialize the escape condition right before entering into the loop (e.g. intIndex = 0)

When incrementation index is used, avoid escape condition containing check for particular value – for example intIndex = 100. If for some reason intIndex never sets to 100, loop will never be terminated.

Always use check for ‘less than’ or ‘bigger than’ – for example intIndex < 100.

intIndex = 0

While (intIndex < 100)

'do something

If (intIndex < 10) Then

intIndex = intIndex + 1

ElseIf (intIndex < 90) Then

intIndex = intIndex + 5

ElseIf (intIndex < 100) Then

intIndex = intIndex + 1

Else

Report.Error("FuncName() : intIndex beyond limits - " & intIndex)

End If

Wend

##### For

Format the For loop structure like the example below.

For intIndex = 0 To 100

'do something

If (7 <> intIndex) Then

'do something else

Else

Report.Error("FuncName() : intIndex beyond limits - " & intIndex)

End If

Next

##### Do

Format the Do loop structure like the examples below.

Always initialize the escape condition right before entering into the loop (e.g. intIndex = 0)

When incrementation index is used, avoid escape condition containing check for particular value – for example intIndex = 100. If for some reason intIndex never sets to 100, loop will never be terminated.

Always use check for ‘less than’ or ‘bigger than’ – for example intIndex < 100.

Please be careful with Loop Until () and Loop While() – the two loops in the below examples are doing exactly the same.

intIndex = 0

Do

'do something

If (intIndex < 10) Then

intIndex = intIndex + 1

ElseIf (intIndex < 90) Then

intIndex = intIndex + 5

ElseIf (intIndex < 100) Then

intIndex = intIndex + 1

Else

Report.Error("FuncName() : intIndex beyond limits - " & intIndex)

End If

Loop Until (intIndex > 99)

intIndex = 0

Do

'do something

If (intIndex < 10) Then

intIndex = intIndex + 1

ElseIf (intIndex < 90) Then

intIndex = intIndex + 5

ElseIf (intIndex < 100) Then

intIndex = intIndex + 1

Else

Report.Error("FuncName() : intIndex beyond limits - " & intIndex)

End If

Loop While (intIndex < 100)

#### Report.Error()

It is very important to use error reporting whenever possible inside condition structures like Select-Case, If-ElseIf-Else. Put Report.Error() in the Else condition as a notification mechanism if unknown or wrong condition is used. This way, if a mistake is made it will be marked as the test step will fail with NOK result and the reason will be reported inside. Otherwise there is a risk the mistake to be missed and this to result in wrong test sequence, hence wrong testing, hence a defect can be missed.

Use the below formatting

Report.Error(“<Function name> : <reason for the error> - ” & <value of variable/parameter>)

* <Function name> – name of the function where the report error message is put
* <reason for the error> – wrong or unknown conditions/value
* <value of variable/parameter>) – print the condition/value itself, this will help to understand the mistake

In the below example is shown how a small mistake in the function parameter may stay unnoticed, so the product not to be set in the desired mode and a whole specific behavior not to be checked. This on the other hand to be a prerequisite for customer issues with consequent complications. Fortunately, adding the Else statement with error reporting will notify the error.

Sub Main

SetProductMode("DEGRADE-MODE")

End Sub

Function SetProductMode(strMode As String) As Integer

Select Case UCase(strMode)

Case "NORMAL\_MODE"

'set normal mode

Return 0

Case "DEGRADE\_MODE"

'set degrade mode

Return 1

Case "TRANSPORT\_MODE"

'set transport mode

Return 2

Case Else

Report.Error("SetProductMode() : unknown mode - " & CStr(strMode))

Return -1

End Select

End Function

### General Formatting

#### Max number of lines

It is recommended a test case to be no longer than 1000 lines.

Use common functions to combine sequence of actions so to keep your test case neat and tidy.

#### Option Statements

Options “UsesAmbiguousError On” and “Explicit” should be included at the beginning of each module.

Option Explicit will enforce explicit declarations for all variables prior their use.

Option UsesAmbiguousError On will enforce check for “When an unqualified symbol not defined in the current macro/module and it is defined in more than one '[#Uses](mk:@MSITStore:C:\Program%20Files%20(x86)\MBtech\PROVEtechTA_2017\bin\ww10_000.CHM::/ww10_000htm/doc_uses_def.htm) module an "Ambiguous name" error occurs”

Option UsesAmbiguousError On

Option Explicit

Sub Main

…

#### Preventing blocked execution

A bad example of poorly designed test is when it contains structures which may block the execution. This results not only of stopped test sequence but may have even bigger impact when such test is executed in a row with others. Then the whole list can be blocked and a good amount of test time to be wasted.

##### Infinite loops

Be careful not to create endless loops. Check carefully all exit conditions. Avoid logical forks (branching) if not especially needed. **Think simple!**

One way to avoid infinite loops is described in chapter “Loops” in this document.

Another way is to set execution time. If still in the loop after some time, then terminate it.

dtmStartTimer\_sec = Timer

dtmStopTime\_sec = 300

While(some\_condition)

'do something

If ((Timer - dtmStartTimer\_sec) > dtmStopTime\_sec) Then

'report error

'exit loop

End If

Wend

##### Time message box – auto expire

Often, a message box with some notes about test actions or checks is used inside manual tests. The message box will remain active until the tester closes it. If such manual test is set as automatic by mistake or wrong filtering is applied then it will be executed as automatic in a row with other automatic tests. This will result in blocked execution sequence.

This can be avoided by using a message box with timeout. Use everywhere such type of message box. During test execution the box will appear and after defined time will automatically close. This will ensure not blocking of the test execution even if a manual test is run by mistake along with automatic ones. Just set long enough timeout in order requested test actions to be done before box to close automatically.

Declare Function MessageBoxTimeout Lib "user32.dll" Alias "MessageBoxTimeoutA" ( \_

ByVal hwnd As Long, \_

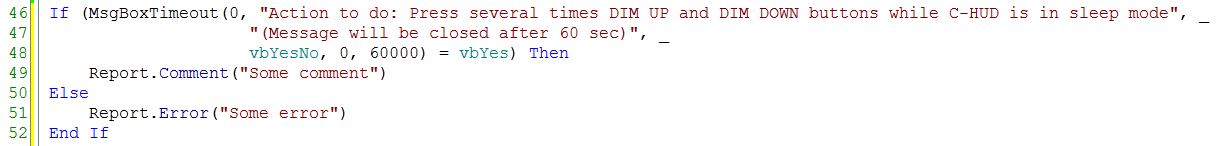
ByVal lpText As String, \_

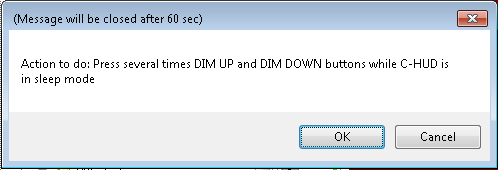
ByVal lpCaption As String, \_

ByVal uType As Long, \_

ByVal wLanguageID As Long, \_

ByVal lngMilliseconds As Long) As Long





#### “Common Constants” and “Common Variables“ files

It is a good practice to put all project’s global constants inside a single file and refer to the file wherever needed. This will improve management of the global common constants and will help to avoid duplications.

The same applies also for project’s global variables.

'#Uses "\*<Common\_Constants>"

'#Uses "\*<Common\_Variables>"

Use regions for logical structuring of the file. Refer to chapter 4.3.8 inside this document.

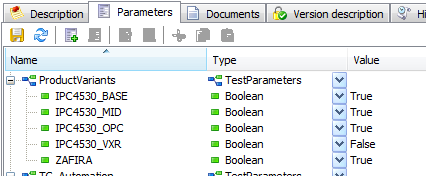
#### Variant Management

A ProveTech project can tests more than one product variants. This means that a variant management is needed so the right test actions to be applied to the right product variant.

There are two major elements of the adopted variant management inside ProveTech:

* Using test parameters to define the applicability of a test.

This determines if the whole test should be executed on particular product variant.

In the below example, the test will be skipped for IPC4530\_VXR variant.

* Test execution control inside test code.

If (VariantMgr.IsApplicable(IPC4530\_MID) Or VariantMgr.IsApplicable(ZAFIRA)) Then

SetPowerMode(PM\_RUN)

strLvlResult\_ON = "TTOFF"

ElseIf (VariantMgr.IsApplicable(IPC4530\_BASE)) Then

SetPowerMode(PM\_ACC)

strLvlResult\_ON = "TTON"

End If

This is direct management of the test actions according the tested variant.

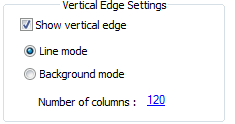
**NB!**

Refer to [R2] for detailed explanation of how to use variant management.

### Editor Configuration

If an external editor is used to create test cases, rather than built in ProveTech, there are few things that should be considered in order the same code indentation and formatting to be achieved.

#### Spaces instead of tabs

Replace tab with 4 spaces.

#### Maximum Line Length

Avoid lines longer than 120 characters. Longer lines are more difficult to read. Set vertical line as right margin so to keep the same visibility in different editors.

# Test Strategy

## Test Strategies Definitions <TO\_DO>

* Strategy 1
* Strategy 2
* …

Master Test Strategy document and its importance to the project.

## Test Types <TO\_DO>

* Validation Types of Tests [R3]

## Test Techniques <TO\_DO>

### Static

#### Reviews

#### Analysis

### Dynamic

#### Specification Based

##### Equivalent Partitioning

##### Boundary Value Analysis

##### State Transition Testing

##### Decision Tables

##### Use Case Testing

#### Experience Based

##### Error Guessing

##### Exploratory Testing

## Test Tools <TO\_DO>

Right tools for the right tests

## Robustness Tests <TO\_DO>

## Cyber Security Tests

### Fuzz Testing

#### Idea

Fuzz testing is a software testing technique that involves injecting the tested product with a large number of inputs in an effort to uncover system anomalies and security vulnerabilities. The inputs may be invalid for the software component or protocol, valid but unexpected input (in timing, values or order), or completely random. Best technique to find unknown vulnerabilities such as buffer overflows or error handling.

During fuzzing device is checked for:

* Security breaches:
  + Security mechanisms are still working correctly
  + Security mechanisms cannot be bypassed
* Normal functioning
* That stays in its functional normal state – no behavioral anomalies
* In case of anomalies or unexpected resets:
  + Check recovering to normal state
  + Check the recovery conditions

Additional actions that may be executed during fuzzing:

* Security check at random moment:
  + Security access
  + Send security locked services without unlocking the CHUD
* Parallel fuzzing:
  + Parallel CAN communication
  + Parallel hardware inputs interactions (buttons, sensors, etc.)

#### Mechanisms to generate fuzz data

##### Random

* + Generating random input of messages, files, commands to the software being tested
  + The simplest form of fuzzing technique, but may be the least effective
  + It can be used at any stage of the development cycle
  + It requires however, a large number of test cases and test time

##### Mutation-based generation

* + Works on valid data and applies random changes to it
  + Modification of protocol packets length
  + Changing bits
  + Injecting special characters
  + Modifying structure of file format

##### Model-driven method

* + Requires deep knowledge and understanding of the tested product
  + Also called “Smart Fuzzing”
  + Uses behavioral models, specifications or protocol requirements as a basis
  + Modifications include changing data fields content, changes in logical behavior of the product, injecting specially prepared inputs based on lessons learned of similar products

##### Examples

The below list contains a few examples that depending on the device, application, or the protocol being tested are likely to trigger an issue:

* + Null characters
  + Long strings
  + Blank strings
  + Values such as: 1, 0, 1, and 2
  + Semi colons
  + String values such as (%n, %s, etc.)
  + maximum/minimum values for integers

#### Fuzzing Subjects

##### Interfaces and Protocols

* + Wi-Fi
  + Bluetooth
  + Ethernet
  + USB
  + Flex ray
  + CAN

##### Media Formats

* + PNG
  + JPEG
  + MP3
  + MP4

#### Fuzz Testing Process

##### Entry Criteria

All security and functional requirements for the specific fuzzing subject shall be implemented and activated.

##### Exit Criteria

* Planned tests are completed
* A critical security defect is found, which affects all tests

### <TO\_DO>

## Test Architecture

### Script architecture (ProveTech)

A well-organized test architecture (test structure) should be built on hierarchy. Strictly defining test levels simplifies significantly the test creation and test maintenance activities. A DRY principle should be used as a basis.

A test architecture should be built on three levels – LOGICAL, COMMON, EXECUTION.

Example1 - LOGICAL level

'#Language "WWB-COM"

'#Uses "\*<Synch>"

'#Uses "\*<PowerSupply>"

'#Uses "\*<DutCheck>"

'#Uses "\*<Global\_Parameters>"

'#Uses "\*<Global\_Variables>"

'#Uses "\*<DutSet>"

'#Uses "\*<CanSleep>"

'#Uses "\*<Report>"

'#Uses "\*<Diagnostic>"

'#Uses "\*<Util>"

'#Uses "\*<Common>"

'#Uses "\*<Common\_Constants>"

'#Uses "\*<Common\_Variables>"

'#Uses "\*<Indicators\_Common>"

'#Uses "\*<Indicators\_Check>"

'#Uses "\*<VariantMgr>"

'#Uses "\*<tc\_PreInit>"

'#Uses "\*<BusSignal>"

Option UsesAmbiguousError On

Option Explicit

Sub Main

If TC\_NOT\_Applicable() Then Return

TC\_PreConditions("IND\_INIT")

Report\_step("IND\_CONFIDENCE\_CHECK\_ALL\_OFF")

INDs\_TurnOFF\_All()

Verify\_IND("ALL", ST\_OFF)

GoToPage(PAGE\_SPEED\_1)

SetValueODI("SPEEDOMETER\_VALUE", 10) ' Set 10

Test\_IND\_ACC("CONFIDENCE")

Test\_IND\_PedestrianDetected("CONFIDENCE")

Test\_IND\_LKA("CONFIDENCE")

Test\_IND\_VehicleAhead("CONFIDENCE")

Test\_IND\_LDW("CONFIDENCE")

Test\_IND\_RSA("CONFIDENCE")

GoToPage(PAGE\_PERFORMANCE)

Test\_IND\_GSI("CONFIDENCE")

Test\_IND\_AST("CONFIDENCE")

Report\_step("IND\_CONFIDENCE\_ALL\_OFF\_end")

Verify\_IND("ALL", ST\_OFF)

TC\_PostConditions()

End Sub

Example2 - LOGICAL level

Example 2 - COMMON level

Sub Main

If TC\_NOT\_Applicable() Then Return

TC\_PreConditions("PM\_VOLT\_MODES")

Dim intFromVM As Integer

Dim intToVM As Integer

Dim strRepFromVM As String

Dim strRepToVM As String

intFromVM = 0

intToVM = 0

Report.AddRequirements("SysFS\_PowerMgntGM-1290.R3,SysFS\_PowerMgntGM-1289.R2")

Report.Comment("RTC enh - 760175")

Verify\_Gauges("SPEEDO", "ON", 100)

For intFromVM = VM\_NOT\_OPR\_W\_RST To (VM\_UBOUND - 1)

Select Case intFromVM

Case VM\_NOT\_OPR\_W\_RST : strRepFromVM = "VM\_NOT\_OPR\_W\_RST"

Case VM\_NOT\_OPR\_LOW : strRepFromVM = "VM\_NOT\_OPR\_LOW"

Case VM\_PART\_OPR\_LOW : strRepFromVM = "VM\_PART\_OPR\_LOW"

Case VM\_FULLY\_OPR : strRepFromVM = "VM\_FULLY\_OPR"

Case VM\_PART\_OPR\_HIGH : strRepFromVM = "VM\_PART\_OPR\_HIGH"

Case VM\_NOT\_OPR\_HIGH : strRepFromVM = "VM\_NOT\_OPR\_HIGH"

Case Else : Report.Error("Unknown voltage mode - " & CStr(intFromVM))

End Select

If intFromVM = VM\_NOT\_OPR\_W\_RST Then

Report\_step("Test CHUD transition from " & strRepFromVM & " to " & strRepFromVM)

SetVoltageMode(intFromVM, 5) 'wait 5sec

Verify\_VoltageMode(intFromVM)

ElseIf Not (intFromVM = VM\_NOT\_OPR\_HIGH) And (intToVM = VM\_NOT\_OPR\_HIGH) Then

Report\_step("Test CHUD transition from " & strRepToVM & " to " & strRepFromVM)

SetVoltageMode(intFromVM, 5) 'wait 5sec

Verify\_VoltageMode(intFromVM)

End If

For intToVM = VM\_NOT\_OPR\_W\_RST To (VM\_UBOUND - 1)

If intFromVM <> intToVM Then

Select Case intToVM

Case VM\_NOT\_OPR\_W\_RST : strRepToVM = "VM\_NOT\_OPR\_W\_RST"

Case VM\_NOT\_OPR\_LOW : strRepToVM = "VM\_NOT\_OPR\_LOW"

Case VM\_PART\_OPR\_LOW : strRepToVM = "VM\_PART\_OPR\_LOW"

Case VM\_FULLY\_OPR : strRepToVM = "VM\_FULLY\_OPR"

Case VM\_PART\_OPR\_HIGH : strRepToVM = "VM\_PART\_OPR\_HIGH"

Case VM\_NOT\_OPR\_HIGH : strRepToVM = "VM\_NOT\_OPR\_HIGH"

Case Else : Report.Error("Unknown voltage mode - " & CStr(intToVM))

End Select

Report\_step("Test CHUD transition from " & strRepFromVM & " to " & strRepToVM)

SetVoltageMode(intToVM, 5) 'wait 5sec

Verify\_VoltageMode(intToVM)

If Not (intToVM = VM\_NOT\_OPR\_HIGH) Then

Report\_step("Test CHUD transition from " & strRepToVM & " to " & strRepFromVM)

SetVoltageMode(intFromVM, 5) 'wait 5sec

Verify\_VoltageMode(intFromVM)

End If

End If

Next

Next

Report\_step("Set fully operational voltage mode")

SetVoltageMode(VM\_FULLY\_OPR, 5) 'wait 5sec

Set\_Power\_On\_OFF\_STATE(STATE\_ON,True)

Verify\_Gauges("SPEEDO", "ON", 100)

SetPARAM\_multi("P\_DTC\_B1325\_03\_VOLTS=90,P\_DTC\_B1325\_07\_VOLTS=160")

TC\_PostConditions()

End Sub

Sub SetVoltageMode( intVoltageMode As Integer, \_

intWaitSeconds As Integer, \_

Optional blnPressON As Boolean = False, \_

Optional blnWaitForMotion As Boolean = True, \_

Optional blnRecoverPosition As Boolean = True)

Dim varDetectedPosition As Variant

Select Case intVoltageMode

Case VM\_NOT\_OPR\_HIGH

SetVoltageValue(RandomDouble(P\_VOLTAGE\_DOWN\_HIGH\_18V , 19.0))

Case VM\_PART\_OPR\_HIGH

SetVoltageValue(RandomDouble(P\_VOLTAGE\_PARTIAL\_HIGH\_16V , P\_VOLTAGE\_DOWN\_HIGH\_18V))

Case VM\_FULLY\_OPR

SetVoltageValue(RandomDouble(P\_VOLTAGE\_PARTIAL\_LOW\_9V , P\_VOLTAGE\_PARTIAL\_HIGH\_16V))

Case VM\_PART\_OPR\_LOW

SetVoltageValue(RandomDouble(P\_VOLTAGE\_DOWN\_LOW\_8V , P\_VOLTAGE\_PARTIAL\_LOW\_9V))

Case VM\_NOT\_OPR\_LOW

SetVoltageValue(RandomDouble(P\_VOLTAGE\_NO\_OPE\_7V , P\_VOLTAGE\_DOWN\_LOW\_8V))

Case VM\_NOT\_OPR\_W\_RST

SetVoltageValue(RandomDouble(6.0 , P\_VOLTAGE\_NO\_OPE\_7V))

Case Else

Report.Error("SetVoltageMode() - unknown voltage mode")

Exit Sub

End Select

WaitTime(intWaitSeconds, time\_s)

'SysFS\_PowerMgntGM-6502.R2

If (VM\_FULLY\_OPR = intVoltageMode) Then

If (blnPressON) Then

PressButton(BTN\_ON\_PAGE,STATE\_PRESS)

SV\_Power\_On\_Off\_State = ON\_STATE

SetAmbientLight(DEF\_AMB\_LIGHT) '50%

End If

Else

SV\_Power\_On\_Off\_State = OFF\_STATE

SetAmbientLight(0) '0%

End If

If blnWaitForMotion Then

' wait for motion - 6 seconds and 1 additional

WaitTime(7000, time\_ms)

'--------------------------------------

End If

If blnRecoverPosition And (VM\_FULLY\_OPR = intVoltageMode) Then

Report.Comment("872302 - Combiner position is not stored correctly")

ReadPID(COMBINER\_MIRROR\_POSITION,varDetectedPosition)

If varDetectedPosition <> COMBINER\_VISION\_POSITION Then

PressButton(BTN\_OFF, STATE\_PRESS)

SetAmbientLight(0) '0%

WaitTime(5000, time\_ms)

PressButton(BTN\_ON\_PAGE, STATE\_PRESS)

SetAmbientLight(DEF\_AMB\_LIGHT) '50%

WaitTime(5000, time\_ms)

PressButton(BTN\_UP, STATE\_PRESS,,,15000) ' hold button for 15 seconds

Else

Report.Comment("After BatteryReset() CHUD is as expected: " & CStr(varDetectedPosition))

End If

End If

End Sub

Example 3 – EXECUTION level

Sub SetVoltageValue(dblVolts As Double, Optional intWaitTime\_ms As Integer = 500)

If VariantMgr.IsApplicable\_Sample\_Descr(".\*TC\_DESIGN\_REPORT.\*") Then Exit Sub

Dim s As String

If (dblVolts < 0) Then

Report.Warning("Voltage " & dblVolts & " is lower than 0 volts, limiting to 0")

dblVolts = 0

ElseIf (dblVolts > 20) Then

Report.Warning("Voltage " & dblVolts & " is higher than 20 volts, limiting to 20")

dblVolts = 20

Else

s = Format(dblVolts, "00.00")

s = Replace(s,",",".")

Debug.Print(s)

End If

If IsAutomaticBench() Then

PowerSupply.Value = dblVolts

Report.DefectID("671216") ' Closed

If dblVolts < P\_VOLTAGE\_NO\_OPE\_7V Then

gPRVS\_VOLT\_MODE = gCRNT\_VOLT\_MODE

gCRNT\_VOLT\_MODE = gVM\_NOT\_OPR\_W\_RST

ElseIf (dblVolts >= P\_VOLTAGE\_NO\_OPE\_7V) And (dblVolts < P\_VOLTAGE\_DOWN\_LOW\_8V) Then

gPRVS\_VOLT\_MODE = gCRNT\_VOLT\_MODE

gCRNT\_VOLT\_MODE = gVM\_NOT\_OPR\_LOW

ElseIf (dblVolts >= P\_VOLTAGE\_DOWN\_LOW\_8V) And (dblVolts < P\_VOLTAGE\_PARTIAL\_LOW\_9V) Then

PRVS\_VOLT\_MODE = CRNT\_VOLT\_MODE

CRNT\_VOLT\_MODE = VM\_PART\_OPR\_LOW

ElseIf (dblVolts >= P\_VOLTAGE\_PARTIAL\_LOW\_9V) And (dblVolts <= P\_VOLTAGE\_PARTIAL\_HIGH\_16V) Then

gPRVS\_VOLT\_MODE = gCRNT\_VOLT\_MODE

gCRNT\_VOLT\_MODE = gVM\_FULLY\_OPR

ElseIf (dblVolts > P\_VOLTAGE\_PARTIAL\_HIGH\_16V) And (dblVolts <= P\_VOLTAGE\_DOWN\_HIGH\_18V) Then

gPRVS\_VOLT\_MODE = gCRNT\_VOLT\_MODE

gCRNT\_VOLT\_MODE = gVM\_PART\_OPR\_HIGH

ElseIf dblVolts > P\_VOLTAGE\_DOWN\_HIGH\_18V Then

gPRVS\_VOLT\_MODE = gCRNT\_VOLT\_MODE

gCRNT\_VOLT\_MODE = gVM\_NOT\_OPR\_HIGH

Else

Report.Error("SetVoltageValue() - unknown voltage mode ")

End If

Else

If (MessageBoxTimeout(0,"Set Voltage to : " & CStr(dblVolts), \_

"( Message will be closed after 60 sec )",vbOK,0,60000) <> vbOK) Then

Report.Error("Voltage is NOT set to : " & CStr(dblVolts))

End If

End If

Report.Comment("Voltage is set to : " & CStr(dblVolts) & "V")

If (intWaitTime\_ms > 0) Then WaitTime(intWaitTime\_ms)

End Sub

### Models architecture (Matelo) <TO\_DO>

For details how to use Matelo please refer to Appendix A, “Testing With Matelo” in this document

#### Interface to ProveTech

#### Basic elements

#### Structure organization

## Test Specification

### Idea

Test specification is high level textual description of the test algorithm - the scenario and sequence of test actions that will verify particular list of requirements and validate product behavior. It is used:

* + to create tests – transformation of high level test design description into particular test actions (test scripts or models)
  + input for test reviews

Creating test specification is actually helping to understand the requirements and think over the best way they can be tested. However test specification should not only be focused on requirements. It also should take into account the general product behavior. Or in other words not only to verify the requirements, but to validate the product and all this to be put in understandable and readable format.

Main target of creating test specification should be to think logically about sequences of actions and checks, stepping on the knowledge about test techniques, test types and lessons learned. Working on logical level is very important as it helps to expose logical errors in testing often detected as defects in end-user operation.

**NB!**

Test and test specification has to be maintained together.

Test content (script or model) should always be synchronized with test specification and vice versa.

### Description

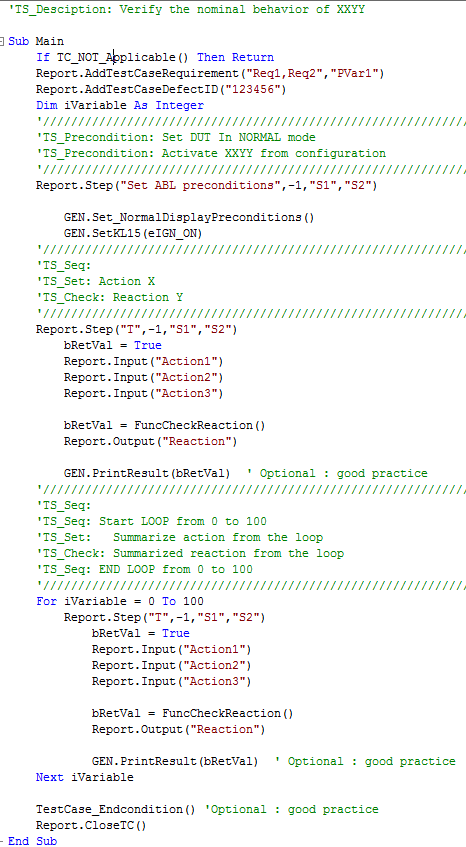
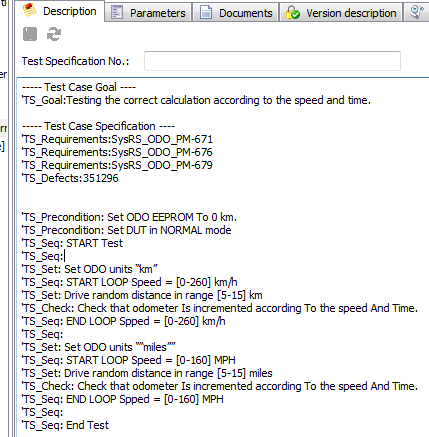
The description of test specification is done by using a key words to indicate the data types. Test specification is extracted and generated afterwards according to these key words.

#### Key Words

* **TS\_Desctiption** – The main purpose of the test
* **TS\_Precondition** – Only specific conditions explicitly needed for the test
* **TS\_Seq** – Used for separation of different sequences (e.g. TS\_Seq: START LOOP or END LOOP)
* **TS\_Set** – Tag for performed actions.
* **TS\_Check** – Tag for expected results checks.
* **TS\_Requirements** – Tag for requirements used to create the test specification.
* **TS\_Defects** – Used to list defects’ IDs relevant to the current test specification.

#### Area

These key words can be placed in two areas:

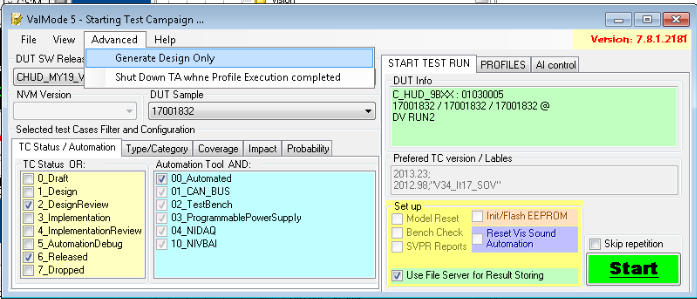
* + Directly inside test script
  + In “Description” tab in ProveTech

### Output Format

Output files with test specification content can be html based (.html or .xml) or Office based (word, excel).

There are several ways to generate test specification:

* Using Generate Design option of the execution tool



* Using test specification generation tool – refer to “Test Specification Guideline“ in the Appendix A in this document
* Using VPAS system - <http://vpas.visteon.com/>

### Rules

* Text in TS\_Desctiption should be maximum one sentence.
* Check carefully the list of requirements and defects. They have to be the relevant to the specific description. A wrong requirement or defect ID can be very misleading.
* For better understanding and readability of the test specification a unique test step name can be added. It should be the same as used inside the test script. This helps a lot to trace certain description with particular test actions.
* Make clear check description – what is expected and why. This is very important in order understand if the right testing is done.
* TS\_Requirements can be used along with every test step description. This will improve understanding the reason for the actions in certain test step

## Test Reusability

**Reusability is one of the most important characteristic of our tests!**

This should be a main target when creating tests. Using what is already implemented and proven as successful test design is a major advantage in test development process. It is:

* already debugged when used in the past
* saving time - directly integrated in the new project
* improving knowledge transfer – a good ideas are not encapsulated inside a project.

In order to achieve a high level of reusability of our tests there are rules which should be strictly followed:

* Use architecture levels described in this document. It is very important to base your test project structure on hierarchy. Using elements (functions or models) to encapsulate particular actions and build test scenarios with these elements is the easiest way to be possible to reuse certain parts of the tests.
* Use variant management techniques described in this document. It is impossible to test project variants without clearly marking which test actions are applicable for particular variant. Once having this profiling inside tests, later it will be possible to identify which parts of the tests to be reused in order to fit perfectly to the new project.
* Test specification – a good description of test scenario (test actions) is helping a lot to understand the test logic and to filter what can be reused
* This document – it is the perfect guideline to reusability. Practically all the information inside is linked to reusability. Well structured, readable, cleverly created and well described test is the essence of reusability.

# Test Implementation Process

## Test scripts

### Test script states

It is very important to use a test case states during its life cycle:

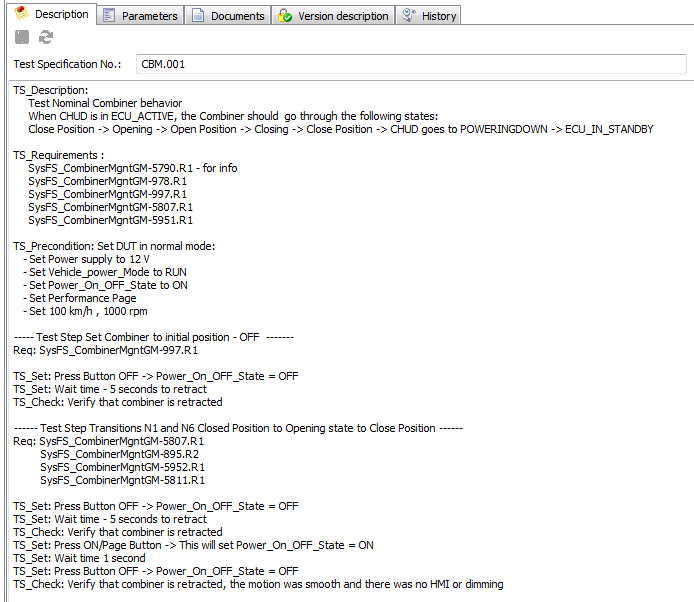
* this provides information about the status of the test, for example is it ready to be executed
* may indicate “test case ready for review”
* helps avoid executing dropped test cases or still in implementation.

Always use correct test case state (refer to chapter 5.1 Life cycle of Test Case in [R2]).

### Test design

It is made inside ProveTech tool as initial step in test case creation (details about test case creation in chapter 5 of [R2]). The output of test design is the test specification - for details refer to “Test Specification” in this document. Test design is under review control as part of peer and skill reviews.

Test design example:



## Test models <TO\_DO>

## Test implementation

For details about preparation of test cases, please refer to [1.32.2, Prepare Test Procedures](http://vistway.visteon.com/CtrlWebIsapi.dll/?__id=webDiagram.show&map=0%3A8464C7967899424AAE107306F1C25BB7&dgm=7675C3F697D84DB9A0AC841AF17F35FF) (VISTWAY).

Pay special attention to [1.32.2.2, Test Case Writing](http://vistway.visteon.com/CtrlWebIsapi.dll/?__id=webDiagram.show&map=0%3A8464C7967899424AAE107306F1C25BB7&dgm=8EF111CA10C4449491928BC4E0D57AB3) (VISTWAY).

## Reviews

Reviews are very important part of proving the right test actions are chosen. They should be a mandatory step in test implementation process. No matter the type of the review, if a MAJOR issue is not closed before validation start, the issue has to be reported as RTC defect with validation skill responsible to fix it. Risk assessment has to be done and communicated to the project via risk in RIO section in enterProj.

### Peer review

Done by test team members it is some kind of “internal” review for the validation team. The general rule is that the author should not review its own tests.

* Focus

Checks for correct understanding of the requirements and their correct transformation in test actions. Checks test case formatting, following of the test design rules, test algorithm, test implementation and test design description vs test implementation.

* Scope

All existing test cases with state DesignReview (ProveTech parameter TC\_State, subparameter 2\_DesignReview=True)

* Trigger condition

Peer review should be triggered on major test case changes with impact on test scenario.

* Trace

All found issues should be documented in issue log form – for details refer to VISTWAY [1.32.2, Prepare Test Procedures](http://vistway.visteon.com/CtrlWebIsapi.dll/?__id=webDiagram.show&map=0%3A8464C7967899424AAE107306F1C25BB7&dgm=7675C3F697D84DB9A0AC841AF17F35FF).

### Skill review

One step further in improving test cases is performing the so called ‘skill review’. Test cases are reviewed by other skills (development or system). Requirement engineers and development engineers make review with test engineers on the features that they all are working on. Skill review checks test algorithm and uses test design as input.

* Focus

Checks for correct understanding of the requirements and their correct transformation in test actions. Checks test algorithm and test design description vs requirements.

* Scope

All test cases with state ImplementationReview (ProveTech parameter TC\_State, subparameter 4\_ImplementationReview=True)

* Trigger condition

A skill review for a test case should start right after a peer review is finished (all issues are analyzed and closed).

* Trace

All found issues should be documented in issue log form. Although there are not specific rules about documenting the skill review outputs, it is recommended to use the same issue log form as for peer review.

## Traceability

Keeping an interconnection between requirements and tests, defects and tests, requirements and defects is extremely important.

### Requirements

In order to have traceability between requirements and tests, the requirements’ IDs are reported in tests. It is recommended to add requirements’ IDs inside test steps which are testing them. This will give the best understanding about the logic of the tests and their focus.

Report\_step("GEARS\_NO\_AUTOMATIC")

Report.AddRequirements("SysFS\_DI-535.R2") 'Gea-0100

Report.AddRequirements("SysFS\_DI-541.R2") 'cond b)

Report.AddRequirements("SysFS\_HMI-108.R1,SysFS\_HMI-96.R2,SysFS\_HMI-97.R0")

**NB :**

*If reporting multiple requirements within single AddRequirements function, keep in mind to remove any spaces between them!*

### Defects

* Report defect

The same principle as for reporting requirements is used also for reporting defects.

Report\_step("GAUGES\_TACHO\_min-max-min in Performance view")

Report.AddRequirements("SysFS\_DI-266.R1,SysFS\_DI-264.R1,SysFS\_DI-270.R0")

Report.DefectID("785717")

Report.DefectID("794785")

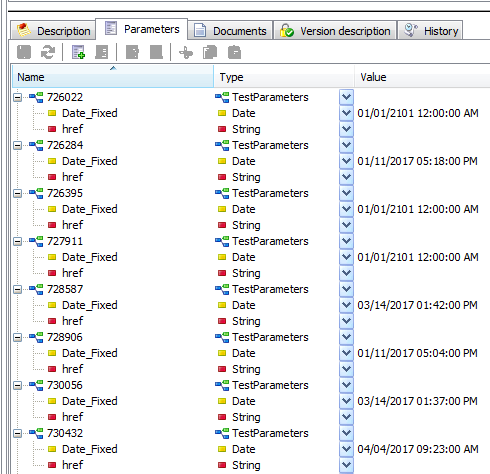
* Close defect

The full list of defects reported inside a test project is contained in object “\_IssueList”. This object can be found in *“\*\\<Database Name>\<Project Name>\PrjCfg\\_Coverage\\_IssueList*”.

(e.g. *\*\\Global\_Sofia\GM\_HUD\PrjCfg\\_Coverage\\_IssueList*). Each defect inside this list has a parameter “Date\_Fixed” which contains the date when the defect is closed in RTC. Open/active defect has default date 01/01/2101 12:00:00 AM. In order to mark a defect as “Closed”, the actual close date from RTC should be added **manually.**

**NB:**

*Default date looks pretty much to the real close date, so check carefully the defect ID and its “Date\_Fixed” content!*

For example:

# Appendix A

Documents listed below are added for convenience. They will be part of the initial versions of this document. Later will be added in the “Reference Documents” list.

## Test Specification Guideline



## Test Specification Word Template



## Testing With Matelo

