Test pattern creation guide

System C Verification Environment : Test pattern creation guide (ver 1.4)

Summary

The purpose of this designer manual is to describe the method to create the test pattern for SystemC Verification Environment

Relative Document:
INT-SLD-08009.odt
Simulation guide
Gen Env guide

Table of Contents

Index of Tables

Index of Figures

1. Introduction

Model verification is very important task in model development. The quality of each model is based on the quality of verification. Hence, to improve the model quality is to improve the verification task. In the past, each projects had its own style to verify. Each styles have advantages and disadvantages. By the way, verification task, in fact, received least supported tools and didn't sufficient guide to help designer to complete verification task with great quality.

In the conclusion, to unify verification method and improve verification quality are reasons to create this guide.

This guide has main purposes :

- Describe common test patterns creation methodology.
- Describe usage of supporting scripts.

The position of this guide in verification task is described in below :

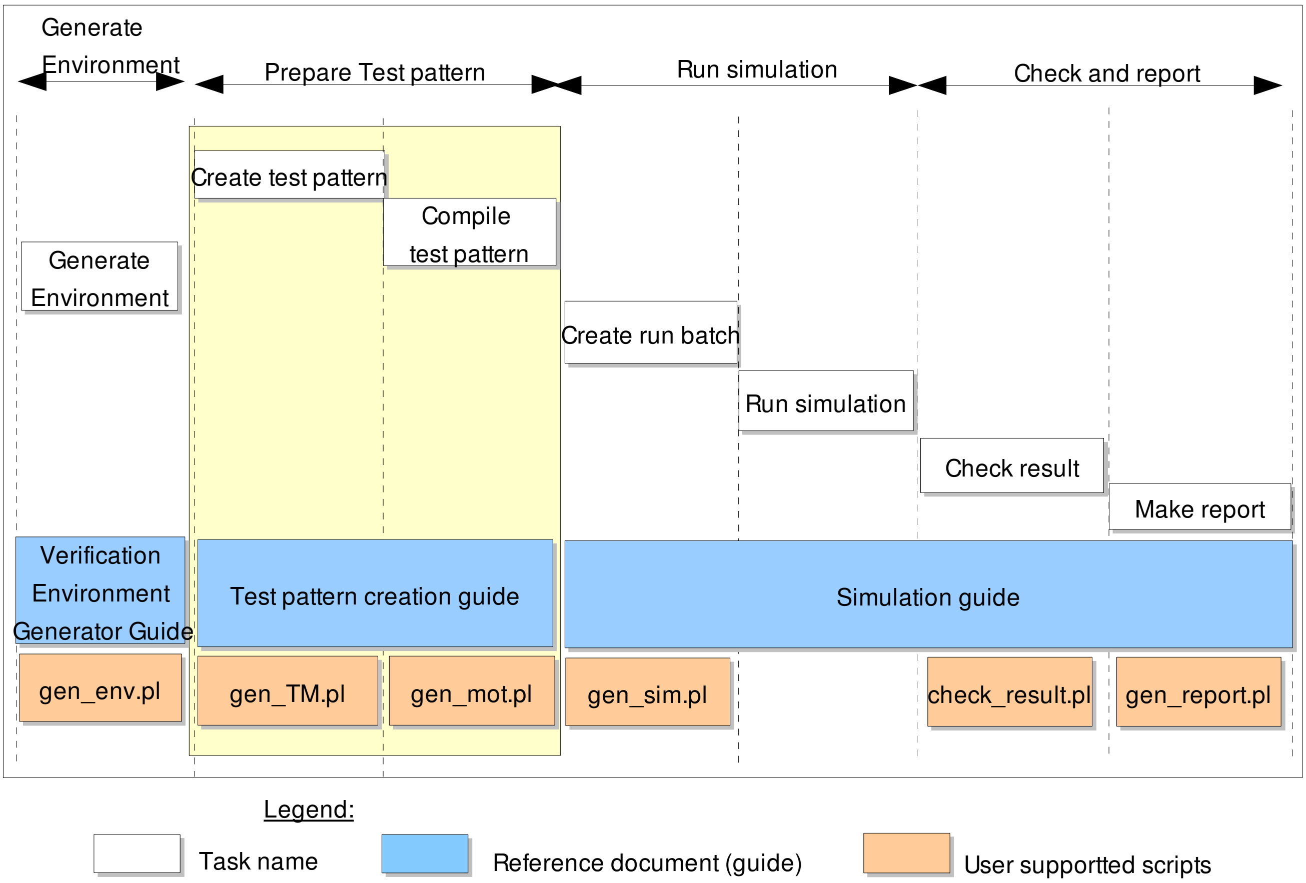


Figure 1.1: Scope of this guide in model verification task

2. Test pattern creation methodology

To create a test pattern, designer usually follows below steps ;

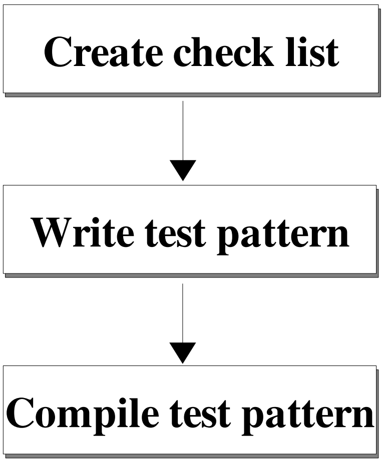


Figure 2.1: Test pattern creation flow

Explanation :

Create check list :

After investigating model specification, designer must decide model to verify under which particular test cases. He/she must list all test cases into the document called *"check list"* (in model development flow version 1.0, this document is named with suffix VRF-SLD-xxxxx-01). By the way, designer also write down below information for each test case in check list :

- Name of test case
- Name of test pattern for test case
- Schedule plan for finish test case
- Actual plan for each test case
- How to check this test case
- Other information if necessary

Write test pattern :

When designer finished creating check list, he/she can start to write test pattern for each test case. Test pattern for verifying model can be C or ASM source file or text file (.txt).

Compile test pattern :

For the projects use .mot file to verify, after finishing test pattern, designer must compile test pattern into executable file. Executable file is different from each project. In this scope of this document, executable file will be MOT file because environment is used here will be Forest (ISS environment). To compile MOT file, designer can use SHC compile which can be invoked by using HEW or command line.

For the projects use text file (.txt) to verify, designer doesn't need to compile the test pattern.

2.1.Create check list

2.1.1.Check list creation

Check list is document that describes all test cases which are used to verify one model. According to model development flow, check list is created by designer in internal specification phase. The content of check list is list of test case.

Test case in check list is defined as a circumstance in which model is used. The circumstance is based on the requirement document, HW specification or usages that are imagine by designer. The quality of designed model is based on not only design skills of designer but also the number and quality of test cases,so the more test cases model can run, the more bugs can be eliminated.

In the conclusion, designer must imagine all corner cases or common cases of model and verify model in all circumstances. Common case is common usage circumstance of model. In the general,common cases are described in HW manual of designed model. Some notable common cases are :

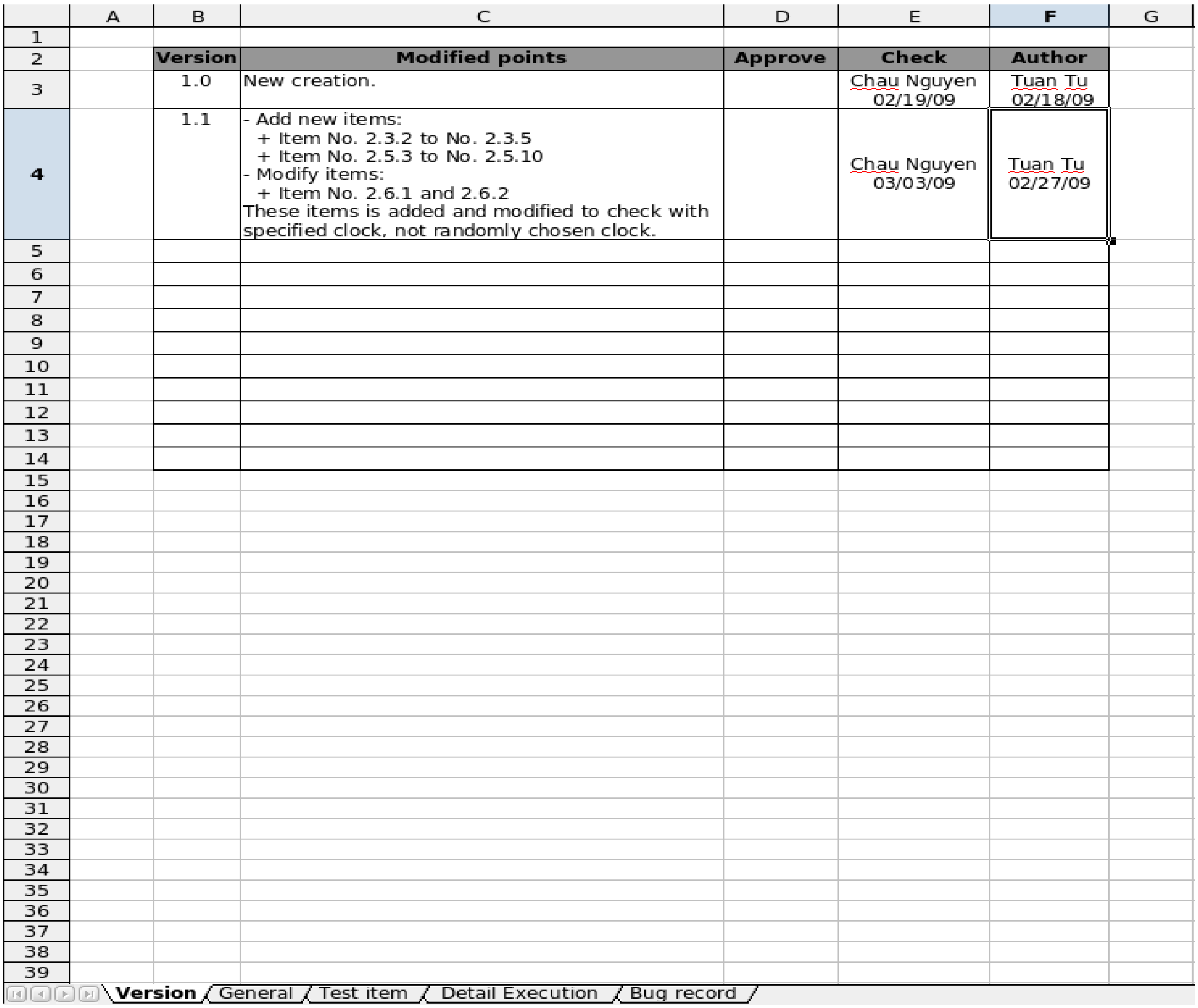
- Interface (Register setting).
- Functions (operations, run with/without interrupt occurrence,...).
- Co-operation (Combination operation of verified model with other model such as verified model with DMAC...).
- Other (Multi channel operation...).

Corner cases or rare occurrence cases are cases that describe usage of model under combination of multi functions or difficult or impossible usage such as usage of model with lower powers or process model with large data .... Usually, corner cases are vast however designer must imagine as much as possible corner cases to eliminate all potential bugs of model.

2.1.2.Check list content

Check list usually is Excel file and, according to model development flow, named with format VRF-SLD-xxxxx-01. Check list consists several required sheets such as below :

Sheet 1 - History : This sheet is used to control version for modified points.



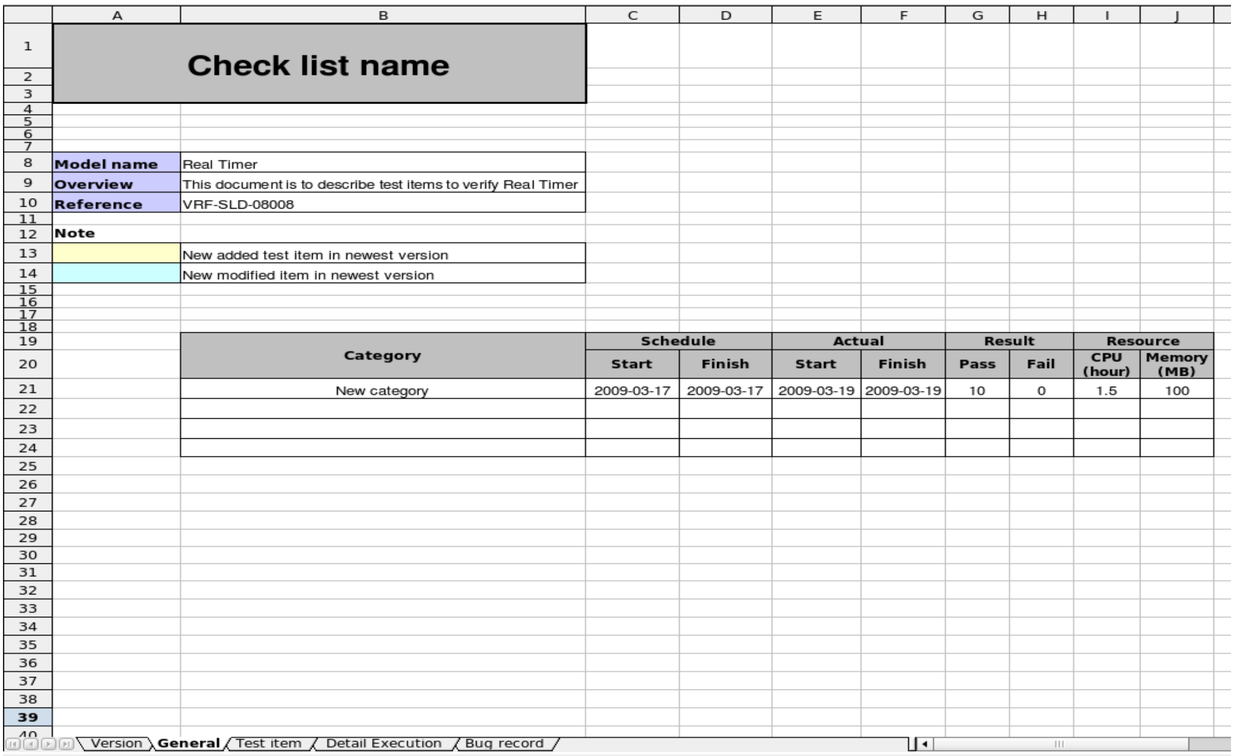
Version	Modified points	Approve	Check	Author
1.0	New creation.		Chau Nguyen 02/19/09	Tuan Tu 02/18/09
1.1	- Add new items: + Item No. 2.3.2 to No. 2.3.5 + Item No. 2.5.3 to No. 2.5.10 - Modify items: + Item No. 2.6.1 and 2.6.2 These items is added and modified to check with specified clock, not randomly chosen clock.		Chau Nguyen 03/03/09	Tuan Tu 02/27/09

Figure 2.2: Example of version sheet

Explanation :

- *Version* : Version of check list
- *Modified points* : Updated point for each version
- *Approve* : Who and when each version of check list is approved.
- *Check* : Who and when each version of check list is checked.
- *Author* : Who and when each version of check list is created.

Sheet 2 - General : This sheet is used to describe general information about check list



Check list name

Model name	Real Timer
Overview	This document is to describe test items to verify Real Timer
Reference	VRF-SLD-08008

Note

🟨	New added test item in newest version
🟦	New modified item in newest version

Category	Schedule		Actual		Result		Resource	
	Start	Finish	Start	Finish	Pass	Fail	CPU (hour)	Memory (MB)
New category	2009-03-17	2009-03-17	2009-03-19	2009-03-19	10	0	1.5	100

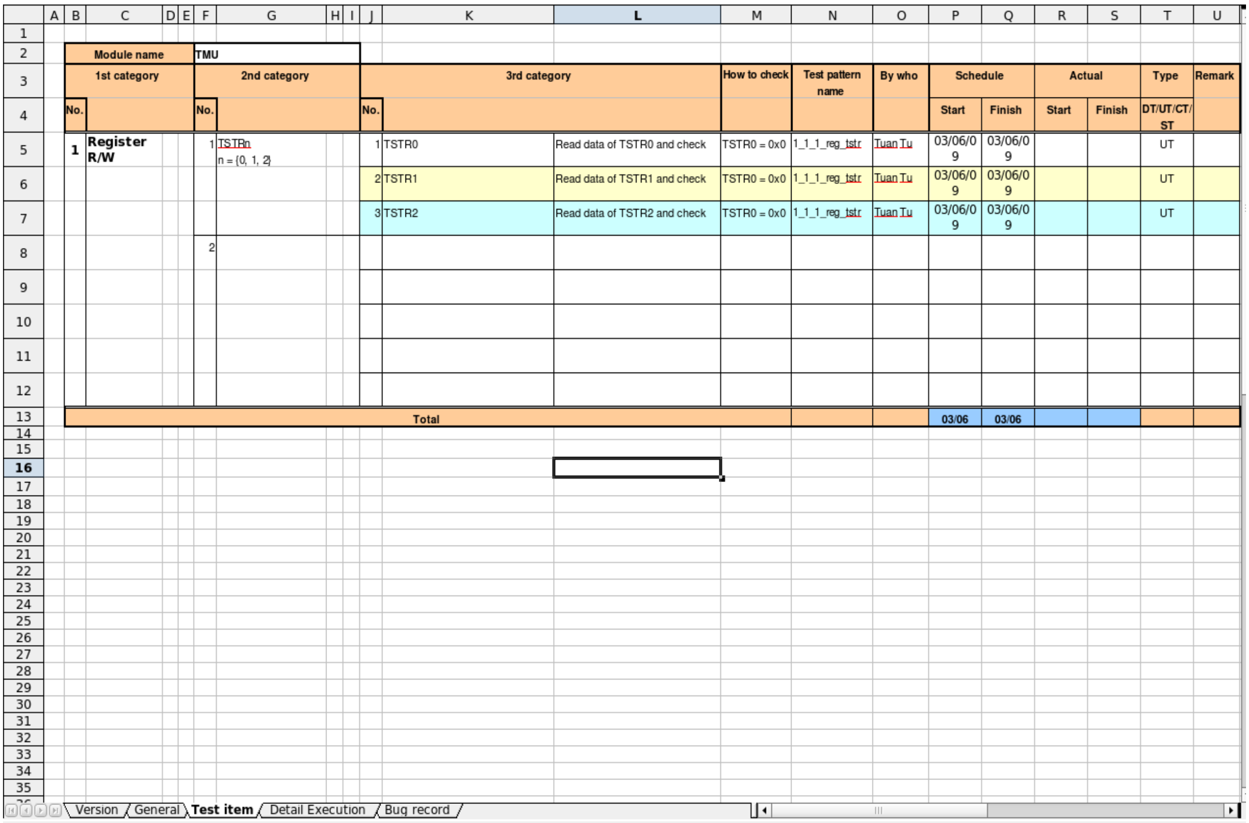
Figure 2.3: Example of general sheet

Explanation :

- *Check list name* : Name of check list file. Ex : VRF-SLD-08001-01
- *Model name* : name of verified model.
- *Overview* : Overview information about model verification mission
- *Reference* : The relative documents. Ex : Verification specification or manual ....
- *Note* : Stipulate for using color for updated or new added test case in each version.
- *Category* : Name of first category
- *Schedule* : Schedule plan to complete corresponding category. Complete means when model can pass all test cases.
- *Actual* : Actual plan to complete corresponding category. Complete means when model can pass all test cases.
- *Category* : Name of first category of test cases.
- *Result* : Result of corresponding category. Pass column indicates number of passed items in this category. Fail column indicates number of failed items in this category.
- *Resource* : Hardware resource for running corresponding category. CPU (hour) : total CPU hour for finishing this category. Memory (MB) : total memory usage for running this category.

Note : After finishing verifying one category or all,the designer can add information for table by collecting information in *"Test Item"* sheet or by using *check_result.pl* script. For *"Test Item"* please refer next page. For *"check_result.pl"* please refer *"Simulation Guide".*

Sheet 3 – Test item : This sheet is used to describe information of each test case



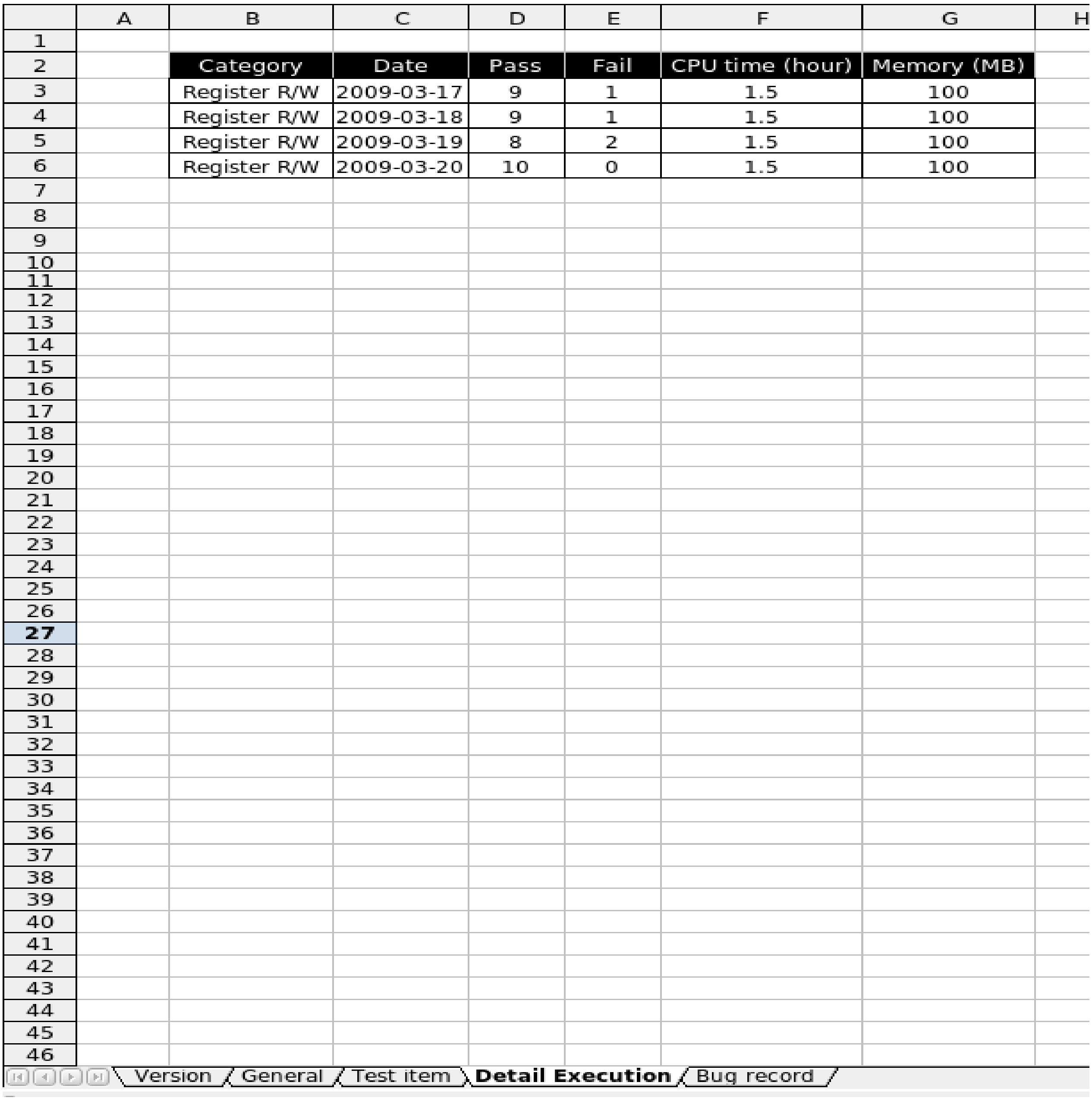
Module name	TMU														
1st category		2nd category		3rd category			How to check	Test pattern name	By who	Schedule		Actual		Type	Remark
No.		No.		No.						Start	Finish	Start	Finish	DT/UT/CT/ST	
1	Register R/W	1	TSTRn n = {0, 1, 2}	1	TSTR0	Read data of TSTR0 and check	TSTR0 = 0x0	1_1_1_reg_tstr	Tuan Tu	03/06/09	03/06/09			UT	
				2	TSTR1	Read data of TSTR1 and check	TSTR0 = 0x0	1_1_1_reg_tstr	Tuan Tu	03/06/09	03/06/09			UT	
				3	TSTR2	Read data of TSTR2 and check	TSTR0 = 0x0	1_1_1_reg_tstr	Tuan Tu	03/06/09	03/06/09			UT	
		2													
Total										03/06	03/06				

Figure 2.4: Example of Test Item sheet

Explanation :

- *Model name* : Name of verified model
- *1st category* : Consists two columns. First column : number of each first category and second column : name of each first category.
- *2nd category* : Consists two columns. First column : number of each second category and second column : name of each second category.
- *3rd category* : Consists three columns. First column : number of each test case, second column : name of each test case and final column : description about this test case.
- *How to check* : describes how to judge the result of this test case is pass. The column is important so designer must write information in detail. Example : Designer must write representative input condition, and concrete expected value.
- *By who* : Author progresses test case.
- *Schedule* : Schedule plan to complete corresponding test case. Complete means when model can pass test case.
- *Actual* : Actual plan to complete corresponding test case. Complete means when model can pass test case.
- *Type* : DT/UT/CT/ST : test case is desk test, unit test, combine test or system test.
- *Remark* : Some notes for corresponding test case

Sheet 4 – Detail Execution : This sheet is used to describe detail information of executed category



Category	Date	Pass	Fail	CPU time (hour)	Memory (MB)
Register R/W	2009-03-17	9	1	1.5	100
Register R/W	2009-03-18	9	1	1.5	100
Register R/W	2009-03-19	8	2	1.5	100
Register R/W	2009-03-20	10	0	1.5	100

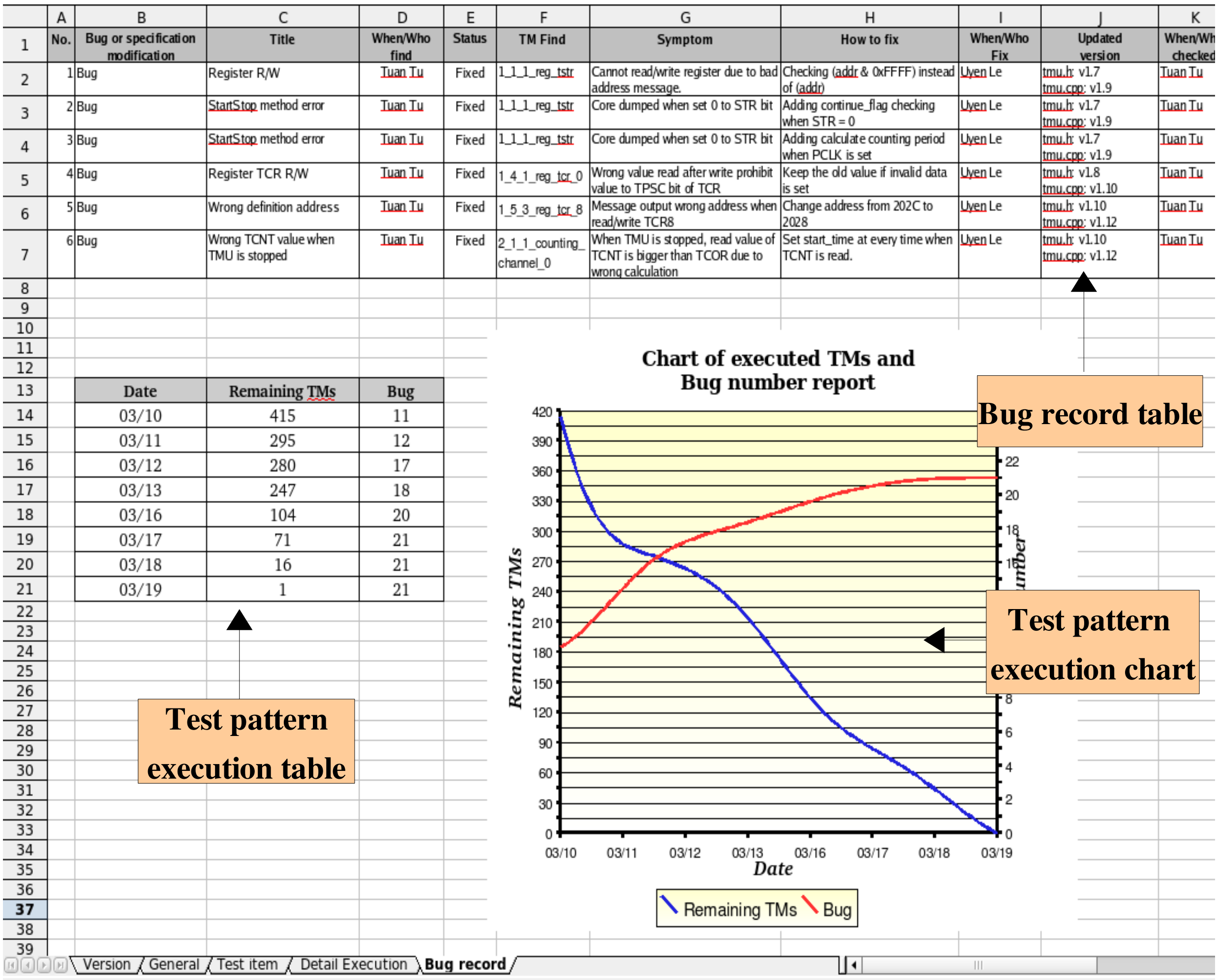
Figure 2.5: Example of detail execution sheet

Explanation :

- *Category* : Name of executed category
- *Date* : Execution date
- *Pass* : Number of passed test cases.
- *Fail* : Number of failed test cases
- *CPU time* : CPU time to execute corresponding category
- *Memory* : Usage memory of executed category

Note : The table will be updated by using *check_result.pl* script. For *"Test Item"* please refer next page because this sheet manages the detail execution of each category for each running time. For *"check_result.pl"* please refer *"Simulation Guide".*

Sheet 5 – Bug record : This sheet is used to manage bugs in verification task.



No.	Bug or specification modification	Title	When/Who find	Status	TM Find	Symptom	How to fix	When/Who Fix	Updated version	When/Wh checked
1	Bug	Register R/W	Tuan Tu	Fixed	1_1_1_reg_tstr	Cannot read/write register due to bad address message.	Checking (addr & 0xFFFF) instead of (addr)	Uyen Le	tmu.h: v1.7 tmu.cpp: v1.9	Tuan Tu
2	Bug	StartStop method error	Tuan Tu	Fixed	1_1_1_reg_tstr	Core dumped when set 0 to STR bit	Adding continue_flag checking when STR = 0	Uyen Le	tmu.h: v1.7 tmu.cpp: v1.9	Tuan Tu
3	Bug	StartStop method error	Tuan Tu	Fixed	1_1_1_reg_tstr	Core dumped when set 0 to STR bit	Adding calculate counting period when PCLK is set	Uyen Le	tmu.h: v1.7 tmu.cpp: v1.9	Tuan Tu
4	Bug	Register TCR R/W	Tuan Tu	Fixed	1_4_1_reg_tcr_0	Wrong value read after write prohibit value to TPSC bit of TCR	Keep the old value if invalid data is set	Uyen Le	tmu.h: v1.8 tmu.cpp: v1.10	Tuan Tu
5	Bug	Wrong definition address	Tuan Tu	Fixed	1_5_3_reg_tcr_8	Message output wrong address when read/write TCR8	Change address from 202C to 2028	Uyen Le	tmu.h: v1.10 tmu.cpp: v1.12	Tuan Tu
6	Bug	Wrong TCNT value when TMU is stopped	Tuan Tu	Fixed	2_1_1_counting_channel_0	When TMU is stopped, read value of TCNT is bigger than TCOR due to wrong calculation	Set start_time at every time when TCNT is read.	Uyen Le	tmu.h: v1.10 tmu.cpp: v1.12	Tuan Tu

Date	Remaining TMs	Bug
03/10	415	11
03/11	295	12
03/12	280	17
03/13	247	18
03/16	104	20
03/17	71	21
03/18	16	21
03/19	1	21

Figure 2.6: Example of bug record sheet

Explanation :

- *Bug record table* : this table manages bugs that are found during verification task.
- *Test pattern execution table* : this table manages verification progress.
- *Test pattern execution chart* : this chart describes the relationship between executed test patterns and found bugs.

Note : The chart can't be generated automatically by *"Test pattern execution table"* designer should use Office tool to make a chart *"Test pattern execution chart"*

2.2.Write test pattern

2.2.1.Test pattern creation

Test pattern is text file such as normal text file, C or ASM source file. For model development with Forest environment, test pattern is C or ASM source file.

Test pattern creation is the progress that transfers each test case in check list into C or ASM source file. According to model development, this step will be operated in design and verification phase.

Like check list creation, test pattern creation requires a lot manually workload, so, in generally, this step takes a lot of time to finish. However unlike check list creation, test pattern creation can be supported by using automation scripts to boost creation progress because a lot of component used in the step are common, for example test pattern template or setting model ... According to tangible situation, designer is encouraged to create supported tools to complete this task with highest quality and fastest time.

2.2.2.Test pattern content

After having check list, designer will create test pattern for each test case. For the content of test pattern,mainly, it will consist four parts : ***header***,***register address definition***, ***value setting of registers of verified model*** and ***result judgment***.

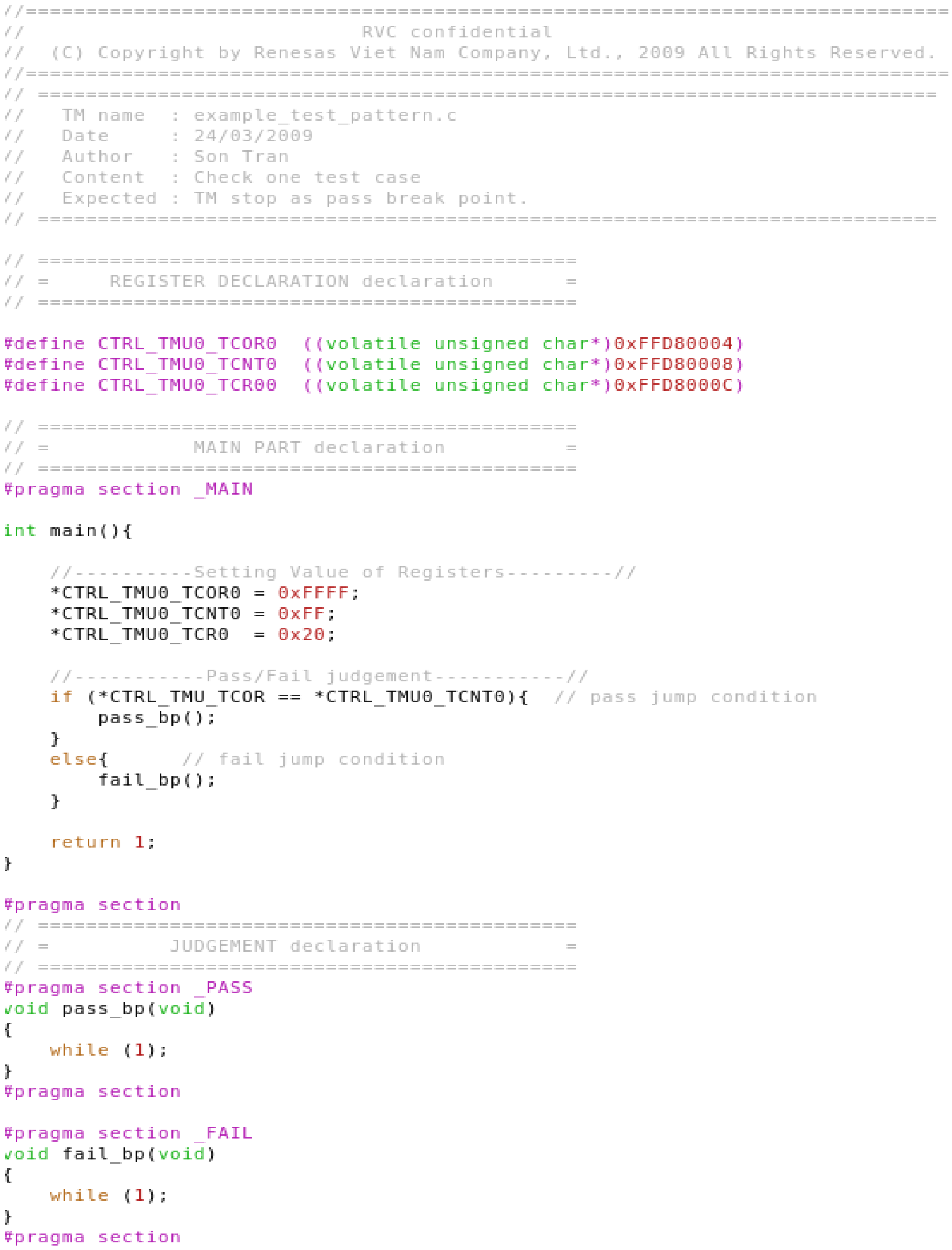
- ***Header*** is used to describe information about test pattern.
- ***Register address definition*** defines register address of verified model or other relevant models.

- ***Value setting of registers of verified model*** is combination value of model registers to describe one or more functions according to specification, model must provide for designer.

- ***The result judgment*** can help designer in two purposes :
  - To judge the test pattern is pass or fail
  - To help designer quickly to debug the fail reason

  In general, judgment result creation can be divided into two styles :
  - Compare with expected data. If the result data match with expected data. The test pattern is considered pass otherwise it will be fail.
  - Move to breakpoint when expected function occurs. If the expected function occurs, test pattern can jump into a particular break point to stop executing test pattern otherwise it will jump to another particular break point to indicate that expected functions doesn't occurs within running time.

Example : To check counting function of TMU model, designer must set value for relevant registers such as TCOR, TCNT and TCR. By the way, to judge result, designer expect final result of register TCNT will equal with value of TCOR.



```
//=============================================================================
//                            RVC confidential
//  (C) Copyright by Renesas Viet Nam Company, Ltd., 2009 All Rights Reserved.
//=============================================================================
// ===========================================================================
//   TM name  : example_test_pattern.c
//   Date     : 24/03/2009
//   Author   : Son Tran
//   Content  : Check one test case
//   Expected : TM stop as pass break point.
// ===========================================================================

// =============================================
// =     REGISTER DECLARATION declaration      =
// =============================================

#define CTRL_TMU0_TCOR0  ((volatile unsigned char*)0xFFD80004)
#define CTRL_TMU0_TCNT0  ((volatile unsigned char*)0xFFD80008)
#define CTRL_TMU0_TCR00  ((volatile unsigned char*)0xFFD8000C)

// =============================================
// =            MAIN PART declaration          =
// =============================================
#pragma section _MAIN

int main(){

    //----------Setting Value of Registers---------//
    *CTRL_TMU0_TCOR0 = 0xFFFF;
    *CTRL_TMU0_TCNT0 = 0xFF;
    *CTRL_TMU0_TCR0  = 0x20;

    //-----------Pass/Fail judgement-----------//
    if (*CTRL_TMU_TCOR == *CTRL_TMU0_TCNT0){  // pass jump condition
        pass_bp();
    }
    else{      // fail jump condition
        fail_bp();
    }

    return 1;
}

#pragma section
// =============================================
// =          JUDGEMENT declaration            =
// =============================================
#pragma section _PASS
void pass_bp(void)
{
    while (1);
}
#pragma section

#pragma section _FAIL
void fail_bp(void)
{
    while (1);
}
#pragma section
```

Figure 2.7: Example of TMU test pattern

Explanation :

Header :

```
//================================================================================
//                          RVC confidential
//  (C) Copyright by Renesas Viet Nam Company, Ltd., 2009 All Rights Reserved.
//================================================================================
// ================================================================================
//   TM name   : example_test_pattern.c
//   Date      : 24/03/2009
//   Author    : Son Tran
//   Content   : Check one test case
//   Expected : TM stop as pass break point.
// ================================================================================
```

Header is used to control information about test pattern. Some common information such as author, date creation, short description or something else will be written in beginning of test pattern. Version of test pattern can be managed here too.

Register address definition :

```
#define CTRL_TMU0_TCOR0  ((volatile unsigned char*)0xFFD80004)
#define CTRL_TMU0_TCNT0  ((volatile unsigned char*)0xFFD80008)
#define CTRL_TMU0_TCR00  ((volatile unsigned char*)0xFFD8000C)
```

The above code defines readable name for three registers in TMU. In one model, register is particular address in memory so that designer can be hard to remember which address is belong to which register. To define one register,designer can use format :

```
#define register_name ((*volatile unsigned {C type}){address})
```

#define : C keyword for defining a value

***unsigned {C type}** : can be ***unsigned char,*unsigned short int,*unsigned int.*unsigned long**
***unsigned char :** for 8 bits register
***unsigned short int :** for 16 bits register
***unsigned int** or ***unsigned long :** for 32 bits register

volatile : C keyword to indicate compile to not optimize this address because this address will be changed within running time by *external source*. During result judgment creation, some addresses (registers) will be checked continuously until they satisfy one value. If designer doesn't use *volatile* keyword, the compile will ignore to read again value of used address (register) from internal memory because it considers this address like static variable that can be changed *inside source code only not by any external sources*. However, within running time, the address (register) will be updated value according to status of model. Thus, if compiler makes test pattern to ignore to read again from internal memory whenever designer get value from this address, the test pattern can't run correctly.

{address} : address of register. The value of address match with that of actual address model register. “Actual” term means that, sometime, the memory of Forest can't match with memory of specification so that designer will use another memory area to replace temporally to design and verify model.

The replacement won't affect to model function because the register address consist two part [***Segment***] [***Offset***], for example : 0XFFFF0008. The segment is FFFF and the offset is 0008.The segment is memory that allocates verified model so that the bus can transfer the data to correct model but when data go inside model, *model use offset only* to judge register which will contain data. In the conclusion, model, in general, doesn't care about the segment address to operate its functions. To write test pattern, designer must pay attention to this point to use the correct address of model.

Value setting of registers of verified model

```
#pragma section _MAIN

int main(){

   //----------Setting Value of Registers---------//
   *CTRL_TMU0_TCOR0 = 0xFFFF;
   *CTRL_TMU0_TCNT0 = 0xFF;
   *CTRL_TMU0_TCR0  = 0x20;

   //-----------Pass/Fail judgment-----------//
   if (*CTRL_TMU_TCOR == *CTRL_TMU0_TCNT0){  // pass jump condition
      pass_bp();
   }
   else{      // fail jump condition
      fail_bp();
   }

   return 1;
}

#pragma section
```

Value setting of registers will be placed inside one *main function* that describes content of test pattern. After compiling test pattern, designer will receive a MOT file. The MOT file consists the machine code of test pattern and is loaded into memory of chip. However, sometime this file will be loaded into illegal memory such ROM or model memory so that the CPU can't run the test pattern. To avoid this situation, the *main function* must be placed between two *pragma section* keyword. The first *pragma section* will go with label. Here, the label is *_MAIN*. In fact, *_MAIN* is memory where *main function* will be loaded in. In Forest environment, *_MAIN* usually is 0X0C400000. To configure label _MAIN with number 0x0C400000, designer can use HEW or SHC compiler command.

The result judgment

```
#pragma section _MAIN

int main(){


    .......


    //-----------Pass/Fail judgment-----------//
    if (*CTRL_TMU_TCOR == *CTRL_TMU0_TCNT0){  // pass jump condition
        pass_bp();
    }
    else{       // fail jump condition
        fail_bp();
    }

    return 1;
}

#pragma section

// ============================================
// =          JUDGEMENT declaration           =
// ============================================
#pragma section _PASS
void pass_bp(void)
{
    while (1);
}
#pragma section

#pragma section _FAIL
void fail_bp(void)
{
    while (1);
}
#pragma section
```

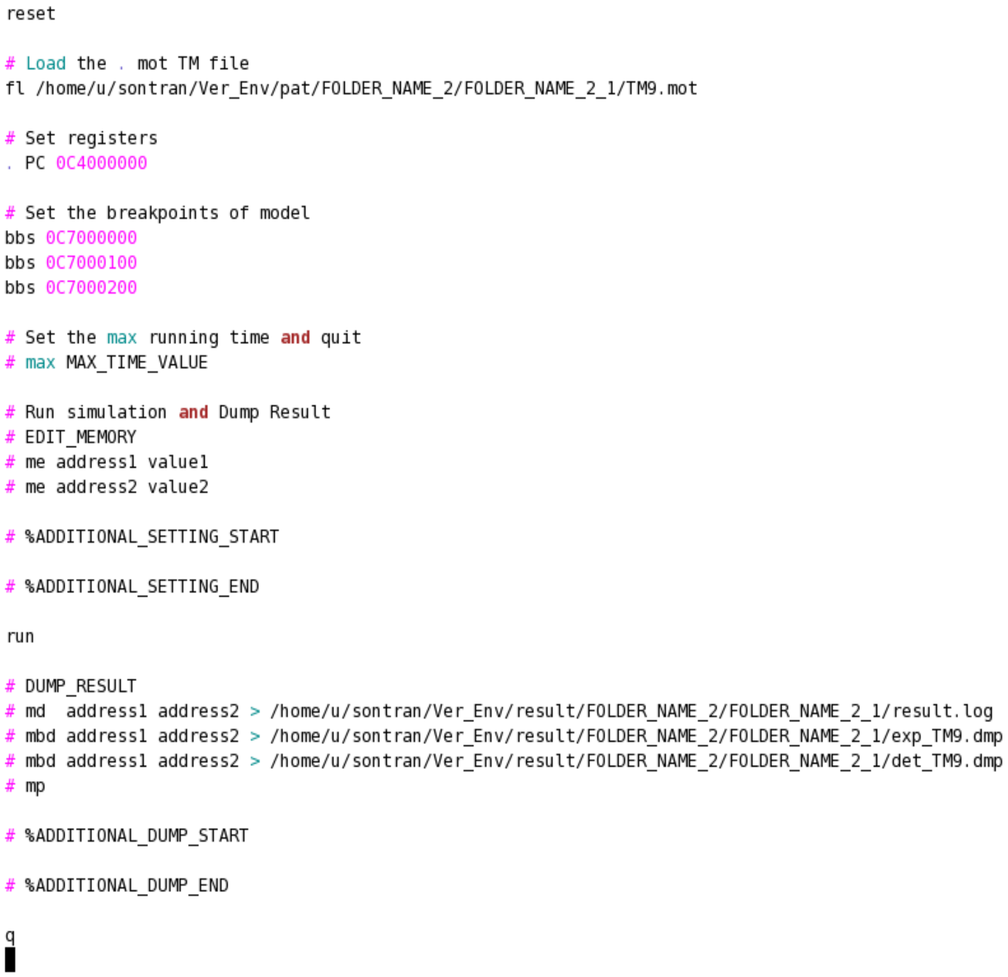
The result judgment can be written inside *main function* or *other place*. There are several techniques to judge the result of test pattern but in model development, the common technique is break point technique. The break point technique is technique that uses some memory addresses like break points to halt, break or stop running progress of test pattern. According to break points, designer can know status of running progress of test pattern. In common, designer uses two points are PASS and FAIL to judge and debug test pattern result.

To use break point technique, designer should prepare :

- Break points and break functions
- Use *bbs* command in ssc file

Break point address can be declared by using two *pragma section* keyword with indicated address. In common in Forest, PASS point is 0x0C700000 and FAIL point is 0x0C700100. Break functions , in the simplest way, are written with infinite loop (*while (1)*) to make sure CPU will halt here until the simulation progress will be broken by *bbs* command.

In this below is the example of one ssc file with break points :



```
reset

# Load the . mot TM file
fl /home/u/sontran/Ver_Env/pat/FOLDER_NAME_2/FOLDER_NAME_2_1/TM9.mot

# Set registers
. PC 0C4000000

# Set the breakpoints of model
bbs 0C7000000
bbs 0C7000100
bbs 0C7000200

# Set the max running time and quit
# max MAX_TIME_VALUE

# Run simulation and Dump Result
# EDIT_MEMORY
# me address1 value1
# me address2 value2

# %ADDITIONAL_SETTING_START

# %ADDITIONAL_SETTING_END

run

# DUMP_RESULT
# md  address1 address2 > /home/u/sontran/Ver_Env/result/FOLDER_NAME_2/FOLDER_NAME_2_1/result.log
# mbd address1 address2 > /home/u/sontran/Ver_Env/result/FOLDER_NAME_2/FOLDER_NAME_2_1/exp_TM9.dmp
# mbd address1 address2 > /home/u/sontran/Ver_Env/result/FOLDER_NAME_2/FOLDER_NAME_2_1/det_TM9.dmp
# mp

# %ADDITIONAL_DUMP_START

# %ADDITIONAL_DUMP_END

q
```

Figure 2.8: Example of ssc file with breakpoint technique

For ssc commands, please refer document *“CEDAR&CEDAR2_simple_manual_050330.pdf”*

2.2.3.Test pattern structure

In the simplest way, designer can create only one file which contains all necessary fields for one test pattern to verify one test case. However, this method is not good because it will reduce reuse ability and increase the memory usage to store test patterns.

To avoid these problems, designer should divide test pattern into two main type files :

- Main file : describes test case or value setting of register and result judgment only. In conclusion, any parts that are unique for each test pattern will be descried in this file. One test pattern will have one main file.
- Supporting files : describe common function, variables,register address, or interrupt ..... The common file can be shared among test patterns by using *include* directive.

To organize test patterns efficiently, designer should seek common parts among test pattern then divide them into several files so that these files can be shared not only among all test patterns but also among other projects. The notable common parts in test pattern are :

1. Model register addresses
2. CPU setup
3. Break point address and functions
4. Interrupt setting
5. Model register initialization.
6. Common variables and functions

For (1), model register addresses will be saved into *regdef.h*. The file can be reused in several projects. For (2),(3),(5) and (6), designer can create *common.h* to store all of them. *common.h* can be shared among test patterns in one project. For (5) in case it differs from each test pattern, designer can write it in main file.

For (4), four files - *intc_handler.h,intc_handler.src,set_int_factor.h,set_intc_factor.cpp* - are used to set interrupt for one model. These file, originally, are created by *gen_intc.pl* in Forest environment. In the conclusion, designer can use these file to build one test pattern :

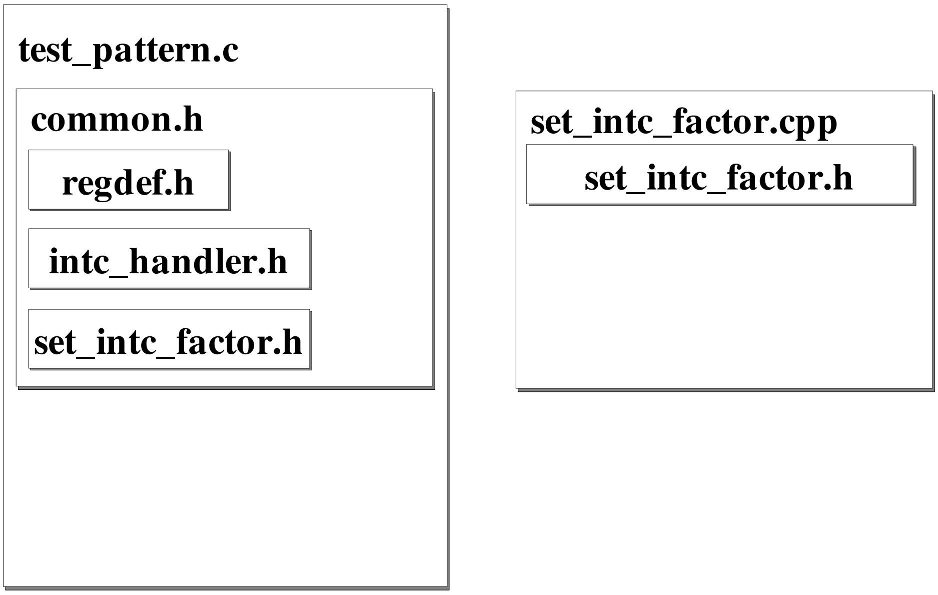


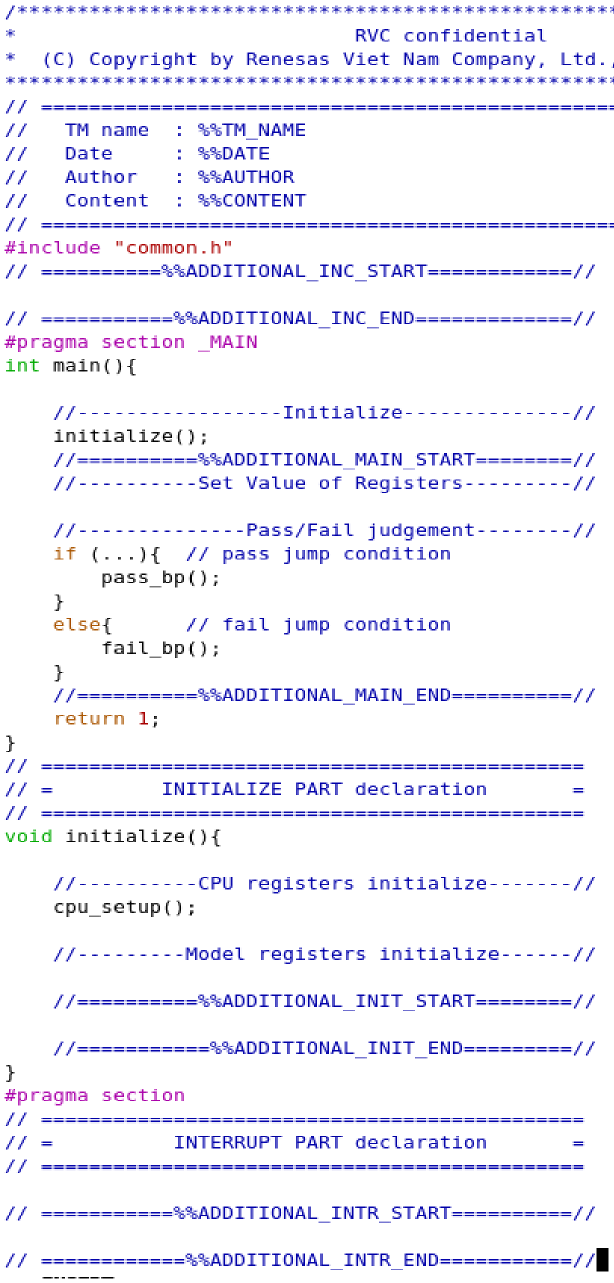
Figure 2.9: Test pattern structure

Explanation:

In the test_pattern.c (main file), this file will include *common.h* that includes *regdef.h, intc_handler.h* and *set_intc_*factor.h. In the *set_intc_factor.cpp*, this file includes *set_intc_factor.h.* File *intc_handler.src* will be used in compilation progress. Designer can refer below example to have clear picture about test pattern structure.

In checking register or function without interrupt, four files (i*ntc_handler.h,intc_handler.src,set_int_factor.h,set_intc_factor.cpp*) can't be ignored. The *common.h,regdef.h, intc_handler.h,set_intc_factor.h, set_intc_factor.cpp,*and *intc_handler.src* are supporting files.

The structure in this guide isn't required standard for all projects, thus designer can change the structure of test pattern to make most convenient in the verification task.



```
/*****************************************************************************
*                            RVC confidential
*  (C) Copyright by Renesas Viet Nam Company, Ltd., %%YEAR All Rights Reserved.
******************************************************************************/
// ===========================================================================
//   TM name  : %%TM_NAME
//   Date     : %%DATE
//   Author   : %%AUTHOR
//   Content  : %%CONTENT
// ===========================================================================
#include "common.h"
// ==========%%ADDITIONAL_INC_START============//

// ===========%%ADDITIONAL_INC_END=============//
#pragma section _MAIN
int main(){

    //-----------------Initialize--------------//
    initialize();
    //==========%%ADDITIONAL_MAIN_START========//
    //----------Set Value of Registers---------//

    //--------------Pass/Fail judgement--------//
    if (...){  // pass jump condition
        pass_bp();
    }
    else{      // fail jump condition
        fail_bp();
    }
    //==========%%ADDITIONAL_MAIN_END==========//
    return 1;
}
// =============================================
// =         INITIALIZE PART declaration       =
// =============================================
void initialize(){

    //----------CPU registers initialize-------//
    cpu_setup();

    //---------Model registers initialize------//

    //==========%%ADDITIONAL_INIT_START========//

    //============%%ADDITIONAL_INIT_END=========//
}
#pragma section
// =============================================
// =          INTERRUPT PART declaration       =
// =============================================

// ===========%%ADDITIONAL_INTR_START==========//

// ============%%ADDITIONAL_INTR_END===========//
```

Figure 2.10: Example of main file of test pattern with structure

Figure 2.11: Example of supporting file of test pattern with structure

Legend :

The rest_bp is used in debug mode to halt the program.

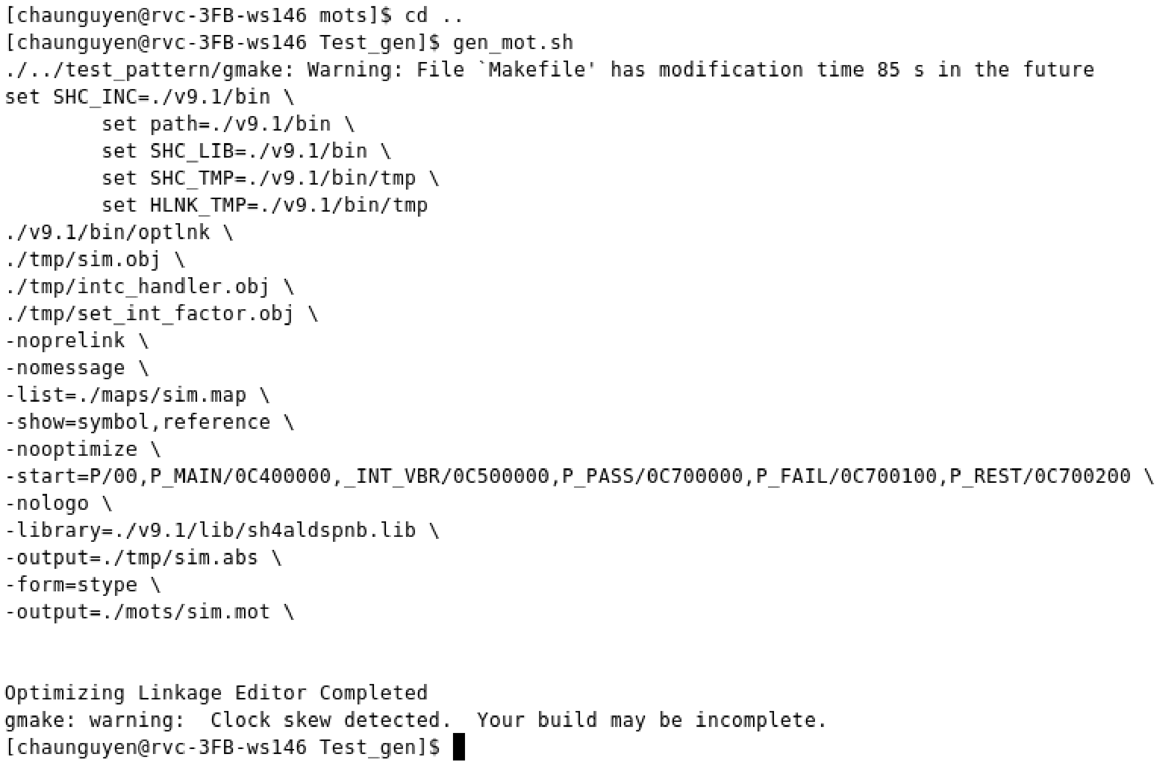
2.3.Compile test pattern

Compile test pattern is the process that transfers from C or ASM file into executable file. According to each environment, designer will have different compilation process and executable file. In the scope of this guide, environment will be Forest (Instruction Set Simulation environment) and executable file will MOT file.

Mainly, to transfer from C or ASM file into MOT file, designer must use SH compiler. This compiler can be invoked by GUI environment (HEW) or command line. Based on the characteristic of model development, HEW is not suitable. Normally, designer must write from hundreds of test patterns to thousands of test patterns, so if used HEW, the compilation process cost much time. The recommended method is command line usage.

Unlike check list and test pattern creation, compilation can be pure automatically process, so designer is encouraged to use script to reduce time to compile test pattern.

After compiling successfully , designer can load MOT file into Forest by using *"fl"* command.



```
[chaunguyen@rvc-3FB-ws146 mots]$ cd ..
[chaunguyen@rvc-3FB-ws146 Test_gen]$ gen_mot.sh
./../test_pattern/gmake: Warning: File `Makefile' has modification time 85 s in the future
set SHC_INC=./v9.1/bin \
        set path=./v9.1/bin \
        set SHC_LIB=./v9.1/bin \
        set SHC_TMP=./v9.1/bin/tmp \
        set HLNK_TMP=./v9.1/bin/tmp
./v9.1/bin/optlnk \
./tmp/sim.obj \
./tmp/intc_handler.obj \
./tmp/set_int_factor.obj \
-noprelink \
-nomessage \
-list=./maps/sim.map \
-show=symbol,reference \
-nooptimize \
-start=P/00,P_MAIN/0C400000,_INT_VBR/0C500000,P_PASS/0C700000,P_FAIL/0C700100,P_REST/0C700200 \
-nologo \
-library=./v9.1/lib/sh4aldspnb.lib \
-output=./tmp/sim.abs \
-form=stype \
-output=./mots/sim.mot \


Optimizing Linkage Editor Completed
gmake: warning:  Clock skew detected.  Your build may be incomplete.
[chaunguyen@rvc-3FB-ws146 Test_gen]$ █
```

Figure 2.12: Example of SH compilation process

3. Supported tools in test pattern creation process

3.1.gen_tm.pl

3.1.1.Introduction

If the test patterns are organized into a structure, designer can create some script to boost the teat pattern creation progress. In this guide, *gen_tm.pl* is introduced to apply in writing test pattern step.

Using this script, designer can generate list of test pattern files with corresponding test case in the check list, by the way, designer also can get some supporting files that are used along with test pattern.

Some notable functions of *gen_tm.pl* :

- Generate test pattern template from check list with some features :
  - *Generate new test pattern source file if it doesn't exist*
  - *Generate test pattern with updating parts if it exists.*
- Generate common.h and regdef.h
- Generate ssc file for each test pattern
- Store generated file in suitable folder
- Support *additional user script* to generate additional content of test pattern.

Requirement :

- Folder structure must be model environment standard folder. Please refer chapter 4.1 for more information.
- Check list must be Excel format (.xls) with version from 2003 or early.
- Skeleton (.skl) files must be stores in required folders.

Note :

- The generated test pattern file is just the template. Thus to complete the test pattern, designer must ,at least,write *value setting of registers of verified model* section.
- According to specific project, designer can modify skeleton file. (.skl) to boost the verification task and to receive expected test pattern template.

3.1.2.Block diagram

The block chart of *gen_tm.pl* will describe below :

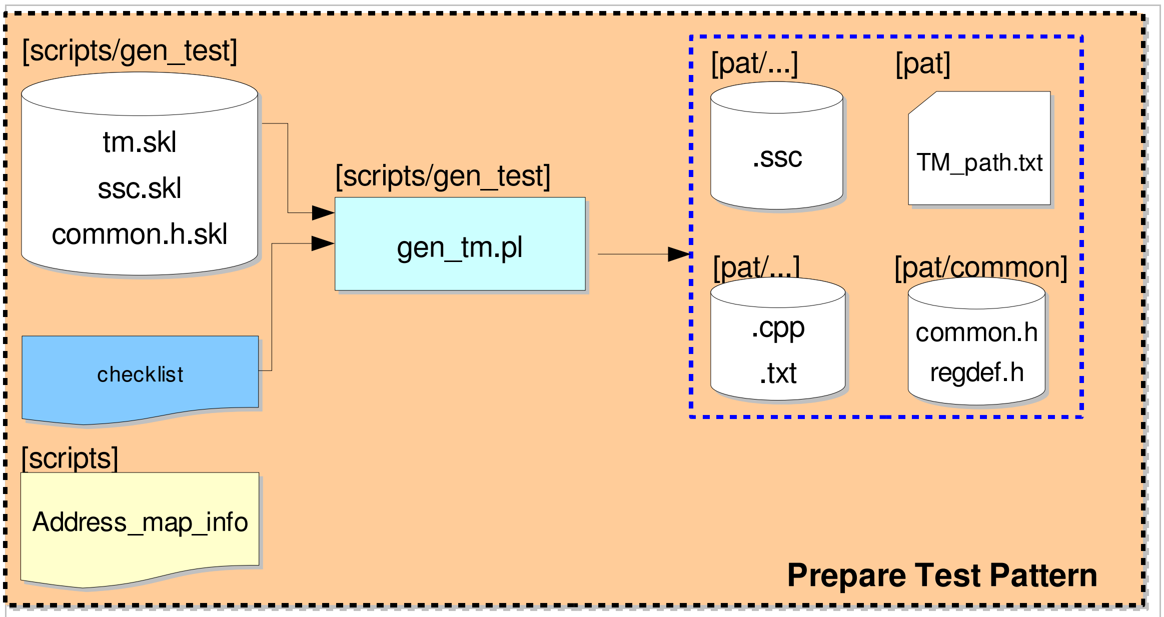


Figure 3.1: Block chart of test pattern creation

Explanation:

The main operation of *gen_tm.pl* is based on the input verification model checklist and some skeleton files to create list of test patterns source code template and store them in the suitable folders.

From the *check list* and *tm.skl*, *gen_tm.pl* will create list of .cpp files or .txt files (test pattern files) that are named according to name of test patter in check list and store result in **[pat/[1st category]/[2nd category]/[3rd category]]** folder.

From the *check list*, *ssc.skl* and *Address_map_info, gen_tm.pl* will create ssc file and store them in **[pat/[1st category]/[2nd category]/[3rd category]]**

From the *common.h.skl* and *Address_map_info* file, *common.h* will be created and stored in **[pat/common].**

From *Address_map_info,* regdef.h will be created and stored in **[pat/common]**

During generation, *TM_path.txt* will be created to help other script in next step. *TM_path.txt* consists path location of each test pattern. Its stored location is **[pat]**

The **[pat/[1st category]/[2nd category]/[3rd category]]** is not fixed. It can be modified by using script's option.

The script run with the standard structure folder. Pleas refer appendix 4.1 to get more information.

3.1.3.Supported functions

Type	Content
Script name	gen_tm.pl
Language	Perl version 5.8.5 or later
Purpose	This script is used to generate test pattern template for designer to help designer writing test pattern more faster, this also support designer to create ssc file using for control simulation.
Input file	checklist, tm.skl,ssc.skl, common.h.skl, Address_map_info.txt, these file is automatically read by script except the checklist.
Output file	**common.h**: contains common functions for all test pattern such as CPU initialize, pass/fail jumping function or etc. **regdef.h**: contains register address definition of model. common.h and regdef.h are stored in [common] folder in pat directory [pat/common]. **Test pattern template files**: contain the standard template of a test pattern, this guides designer where should to put necessary code. **ssc files**: contains the simulation control instruction of each test pattern. Test pattern file and ssc file are stored follows category folder structure defined in the checklist. For example [pat/Initialize] contain the test pattern source code and ssc file for checking the Initialize value of CMT registers. **TM_path.txt**: contains each TM path info to help designer easier to create run simulation batch file based on this info. This file is stored under [pat] directory.
Usage	**perl gen_tm.pl <checklist> -[option] -p <path> -r <path> -i <path> -sname <name> -snum <index>**
Argument	**checklist_name** : path filename to input the checklist e.g: ../abc/VRF-SLD-08010-01.xls (Excel 97/2000/Xp format)
Option	“**-p**” **<path>** – the path of test pattern storing folder. – Default value : ../../pat “**-r**” **<path>** – the path of result storing folder. – Default value : ../../results “**-i**” **<path>** – the path to folder contains common.h. – Default value : ../../common “**-sname**” **<name>** – the name of beginning sheet in checklist. – Default value : “testitem”**.** “**-snum**” **<index>** – number of total checking sheets. – Default value : 1 “**-l**” **<level>** – Description: specify the category level in the checklist to create category folder in pat/model directory to store test pattern. Valid **<level>** value : 1, 2, 3. – Default value: 1. “**-h**” **<header_column>** – Description: specify the column number in the checklist to get the goal of test pattern to write into content tag of test pattern header. -Default value: 10 (third category in checklist) “**-s**” – Description: indicating that script will create ssc file or not – Default value: none “**-help**” - Description: option for help – Default value: none

Table 3.1: gen_tm.pl functions

3.1.4.Running message

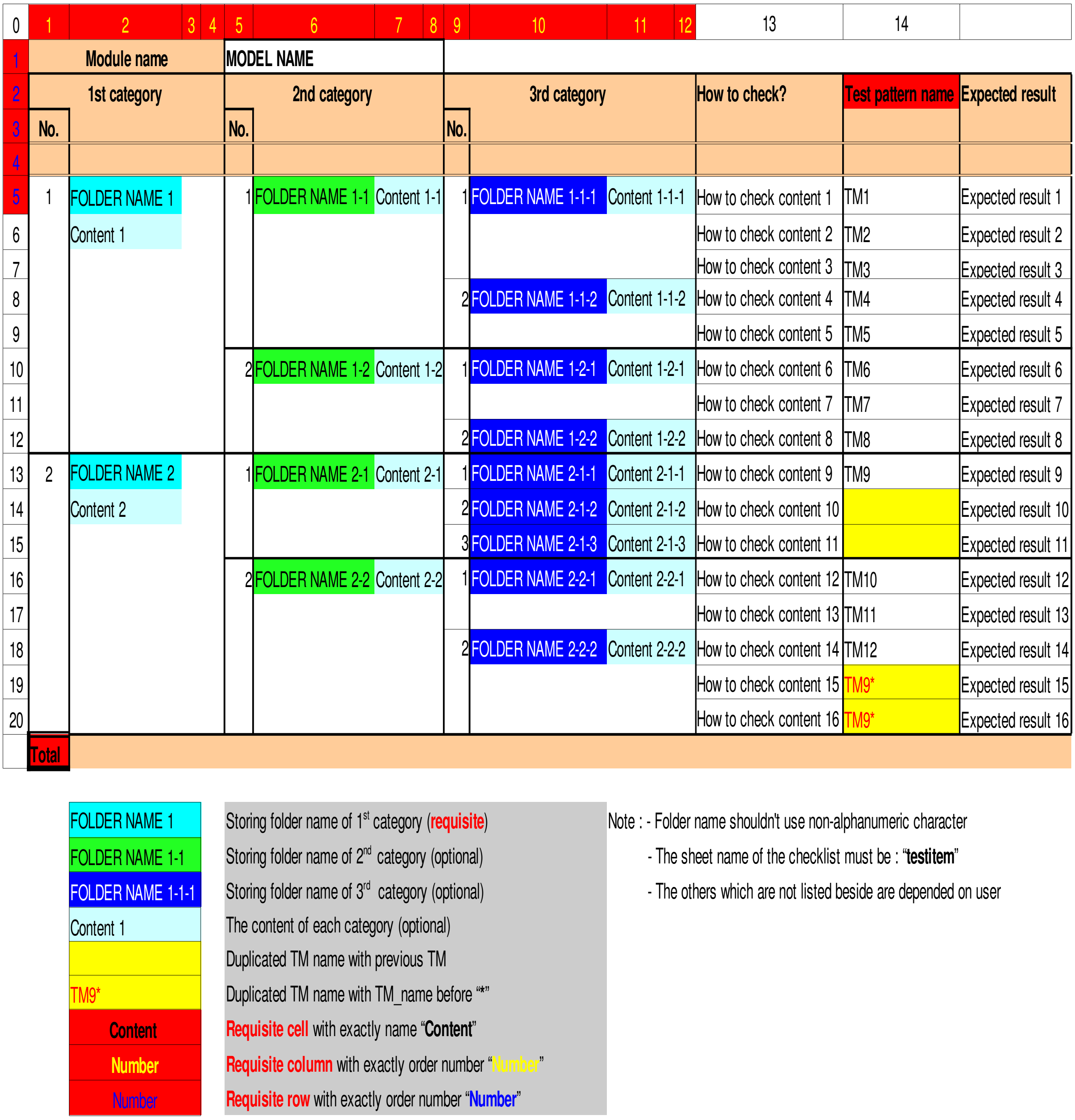
Severity	Message	Description
Error	Can't open checklist file : ....xls	Can not open checklist file
Error	Lack of information of pattern folder path	User not define the path of storing pattern folder
Error	Lack of information of result folder path	User not define the path of storing result folder
Error	FIRST CATEGORY ORDER NOT CORRECT AT ROW:	The order number of 1st category of checklist is not correct 1,2,3,4 (right) --> 1,2,4,5 or 1,3,2,4(wrong)
Error	SECOND CATEGORY ORDER NOT CORRECT AT ROW:	The order number of 2nd category of checklist is not correct 1,2,3,4 (right) --> 1,2,4,5 or 1,3,2,4(wrong)
Error	THIRD CATEGORY ORDER NOT CORRECT AT ROW:	The order number of 3rd category of checklist is not correct 1,2,3,4 (right) --> 1,2,4,5 or 1,3,2,4(wrong)
Info	Checklist : exam_checklist.xls	Name of input checklist
Info	Sheet: testitem	Name of checklist sheet
Info	TM descriptions: Column[13] --> [How to check?]	TM description column
Info	Hierarchy of storing folder : 2	The hierachy level of storing test patterns
Info	Gen test pattern successful!	Generating test patterns finish.
Info	Gen .ssc files successful!	Generating .ssc files finish.
Info	Don't gen .ssc files! (Add "-s" option for gen .ssc file)	Don't generate .ssc files due to user
Info	Gen common.h sucessful!	Generating common.h finish.
Info	Gen regdef.h sucessful!	Generating regdef.h finish.

Table 3.2: Running message of gen_tm.pl

3.1.5.Input files

3.1.5.1. Check list

Some specified point that designer must follow for the correct generated result.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
1	Module name				MODEL NAME										
2	1st category				2nd category				3rd category				How to check?	Test pattern name	Expected result
3	No.				No.				No.						
4															
5	1	FOLDER NAME 1			1	FOLDER NAME 1-1	Content 1-1		1	FOLDER NAME 1-1-1	Content 1-1-1		How to check content 1	TM1	Expected result 1
6		Content 1											How to check content 2	TM2	Expected result 2
7													How to check content 3	TM3	Expected result 3
8									2	FOLDER NAME 1-1-2	Content 1-1-2		How to check content 4	TM4	Expected result 4
9													How to check content 5	TM5	Expected result 5
10					2	FOLDER NAME 1-2	Content 1-2		1	FOLDER NAME 1-2-1	Content 1-2-1		How to check content 6	TM6	Expected result 6
11													How to check content 7	TM7	Expected result 7
12									2	FOLDER NAME 1-2-2	Content 1-2-2		How to check content 8	TM8	Expected result 8
13	2	FOLDER NAME 2			1	FOLDER NAME 2-1	Content 2-1		1	FOLDER NAME 2-1-1	Content 2-1-1		How to check content 9	TM9	Expected result 9
14		Content 2							2	FOLDER NAME 2-1-2	Content 2-1-2		How to check content 10		Expected result 10
15									3	FOLDER NAME 2-1-3	Content 2-1-3		How to check content 11		Expected result 11
16					2	FOLDER NAME 2-2	Content 2-2		1	FOLDER NAME 2-2-1	Content 2-2-1		How to check content 12	TM10	Expected result 12
17													How to check content 13	TM11	Expected result 13
18									2	FOLDER NAME 2-2-2	Content 2-2-2		How to check content 14	TM12	Expected result 14
19													How to check content 15	TM9*	Expected result 15
20													How to check content 16	TM9*	Expected result 16
	Total														

FOLDER NAME 1	Storing folder name of 1st category (requisite)
FOLDER NAME 1-1	Storing folder name of 2nd category (optional)
FOLDER NAME 1-1-1	Storing folder name of 3rd category (optional)
Content 1	The content of each category (optional)
	Duplicated TM name with previous TM
TM9*	Duplicated TM name with TM_name before "*"
Content	Requisite cell with exactly name "Content"
Number	Requisite column with exactly order number "Number"
Number	Requisite row with exactly order number "Number"

Note : - Folder name shouldn't use non-alphanumeric character

- The sheet name of the checklist must be : "**testitem**"
- The others which are not listed beside are depended on user

Figure 3.2: Checklist format

Some obligation of checklist :

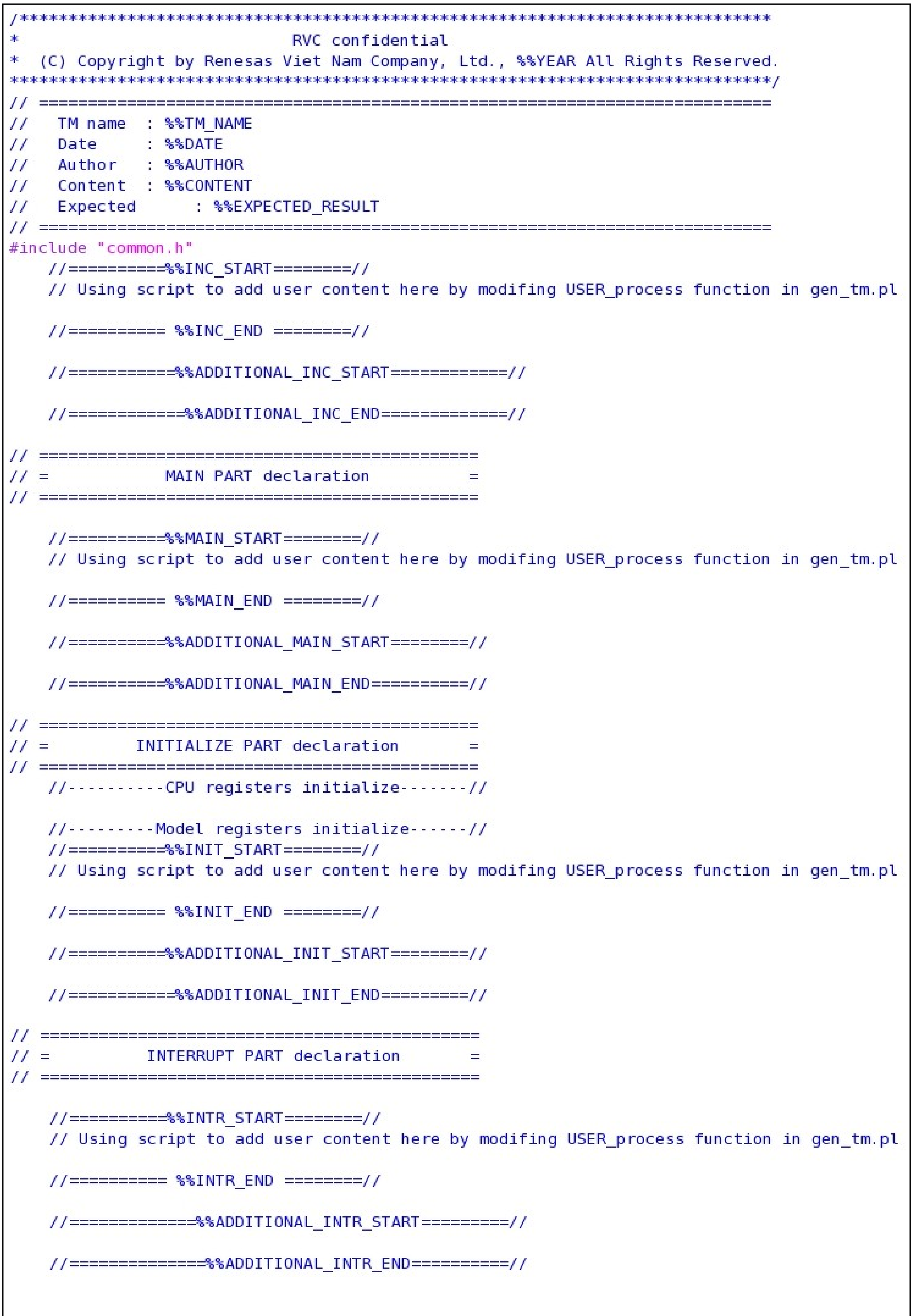
- Checklist sheet has to have name : "**testitem**"
- Storing folder of 1st category have to have at least 1 character and legal (all illegal character will be convert to "_")
- Name of sub-folder shouldn't use non-alphanumeric character.
- When ending of subcategory of 1st category, designer has to add cell "**Total**".
- The column contains test patterns' name has to have name "**Test pattern name**"
- From column 0 -> 12 , designer has to follow exactly structure format for the right result.
- From row 0 -> 5 , designer has to follow exactly structure format for the right result.
- User can using multi sheet. These sheets must be continuous series

Other parts :

- Designer can modify other parts freely.
- For duplicated test patterns, designer can let blank or fill the name of duplicated test pattern with mark : "*" (star). For example : TM9***.**
- If column "**Expected result**" exists, test pattern's header will have more information about expected result.
- Test pattern name extension can be **.txt** or **.cpp**. The default extension is **.cpp** .

3.1.5.2. *tm.skl*

Test pattern skeleton stored in **[scripts/gen_test]**, the content is described below:



```
/****************************************************************************
*                            RVC confidential
*  (C) Copyright by Renesas Viet Nam Company, Ltd., %%YEAR All Rights Reserved.
****************************************************************************/
// ===========================================================================
//   TM name  : %%TM_NAME
//   Date     : %%DATE
//   Author   : %%AUTHOR
//   Content  : %%CONTENT
//   Expected      : %%EXPECTED_RESULT
// ===========================================================================
#include "common.h"
    //==========%%INC_START========//
    // Using script to add user content here by modifing USER_process function in gen_tm.pl

    //========== %%INC_END ========//

    //===========%%ADDITIONAL_INC_START============//

    //============%%ADDITIONAL_INC_END============//

// =============================================
// =            MAIN PART declaration          =
// =============================================

    //==========%%MAIN_START========//
    // Using script to add user content here by modifing USER_process function in gen_tm.pl

    //========== %%MAIN_END ========//

    //==========%%ADDITIONAL_MAIN_START========//

    //==========%%ADDITIONAL_MAIN_END==========//

// =============================================
// =         INITIALIZE PART declaration       =
// =============================================
    //----------CPU registers initialize-------//

    //---------Model registers initialize------//
    //==========%%INIT_START========//
    // Using script to add user content here by modifing USER_process function in gen_tm.pl

    //========== %%INIT_END ========//

    //==========%%ADDITIONAL_INIT_START========//

    //===========%%ADDITIONAL_INIT_END=========//

// =============================================
// =          INTERRUPT PART declaration       =
// =============================================

    //==========%%INTR_START========//
    // Using script to add user content here by modifing USER_process function in gen_tm.pl

    //========== %%INTR_END ========//

    //============%%ADDITIONAL_INTR_START=========//

    //==============%%ADDITIONAL_INTR_END==========//
```

Figure 3.3: tm.skl content

- **Explanation:**

%%YEAR: this tag is replaced by the current year get from operating system

%%TM_NAME: this tag is be replaced by test pattern name get from checklist

%%DATE: this tag is replaced by the current date get from operation system

%%AUTHOR: this tag is replaced by designer who create this test pattern

%%CONTENT: this tag is replaced by the content in the "What to check" column in the checklist.

At the beginning when test pattern file does not exist, a new pure test pattern will be created based on tm.skl file. If test pattern exists a new will be created but the content between below tags in the old test pattern will be kept for the new test pattern.

%%ADDTIONAL_INC_START and **%%ADDITIONAL_INC_END**,
%%ADDITIONAL_MAIN_START and **%%ADDITIONAL_MAIN_END**,
%%ADDITIONAL_INIT_START and **%%ADDITIONAL_INIT_END**,
%%ADDITIONAL_INTR_START and **%%ADDITIONAL_INTR_END**

User can modify user script (**user_script.pm**) to insert user content into below tags.

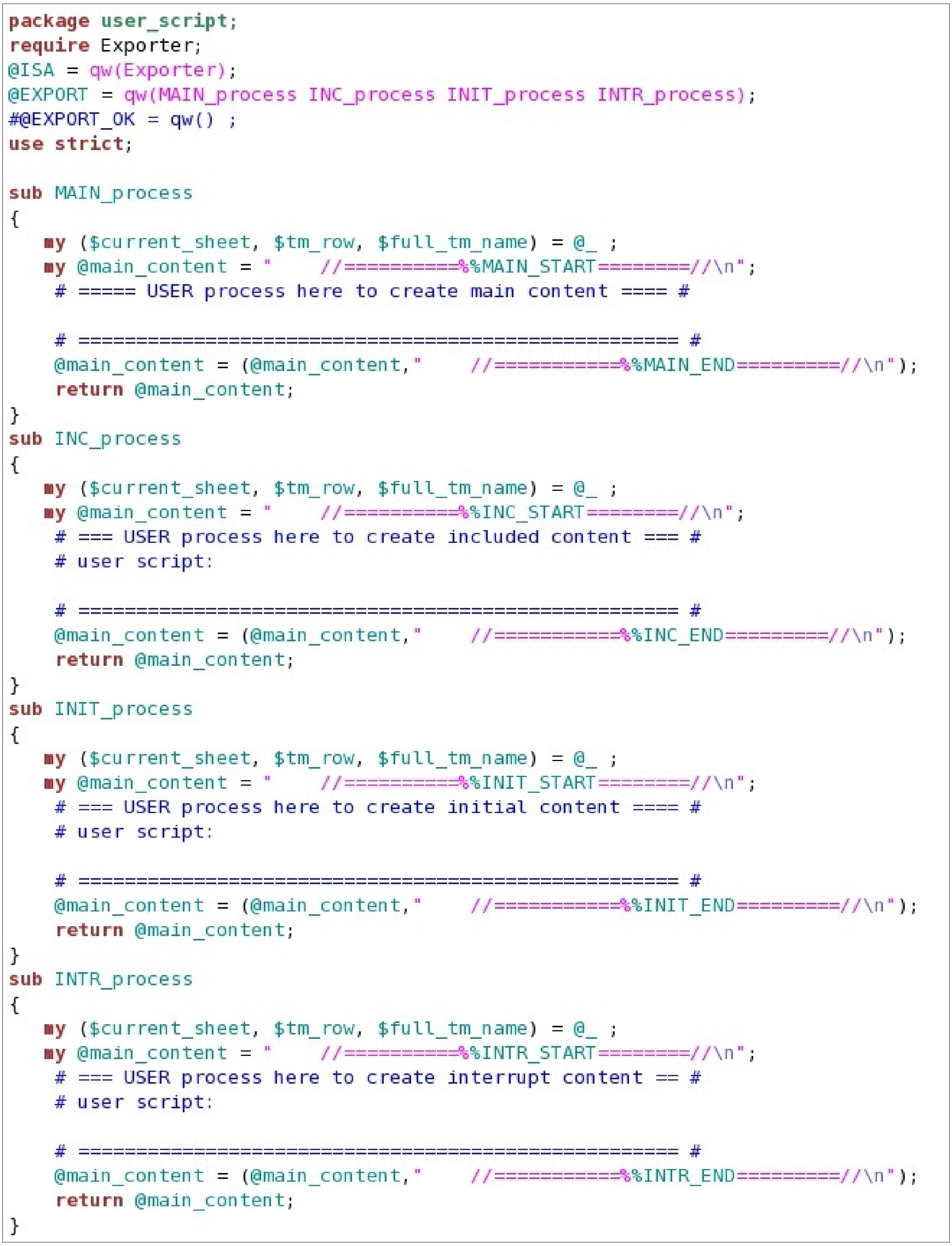
%%INC_START and **%%INC_END**,
%%MAIN_START and **%%MAIN_END**,
%%INIT_START and **%%INIT_END**,
%%INTR_START and **%%NTR_END**

- **Note:**
  - Designer can customize the content of this file to adjust to specified model.
  - The below contents are optional :

    %%ADDITIONAL_MAIN_START and **%%ADDITIONAL_MAIN_END**
    %%ADDITIONAL_INC_START and **%%ADDITIONAL_INC_END**
    %%ADDITIONAL_INTR_START and **%%ADDITIONAL_INTR_END**
    %%ADDITIONAL_INT_START and **%%ADDITIONAL_INT_END**
  - The content between or **%MAIN_START** and **%%MAIN_END** is obligatory.

3.1.5.3. *user_script.pm*

Designer can design user script to generate content of test pattern.



```
package user_script;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(MAIN_process INC_process INIT_process INTR_process);
#@EXPORT_OK = qw() ;
use strict;

sub MAIN_process
{
   my ($current_sheet, $tm_row, $full_tm_name) = @_ ;
   my @main_content = "    //==========%%MAIN_START========//\n";
    # ===== USER process here to create main content ==== #

    # ==================================================== #
    @main_content = (@main_content,"    //===========%%MAIN_END=========//\n");
    return @main_content;
}
sub INC_process
{
   my ($current_sheet, $tm_row, $full_tm_name) = @_ ;
   my @main_content = "    //==========%%INC_START========//\n";
    # === USER process here to create included content === #
    # user script:

    # ==================================================== #
    @main_content = (@main_content,"    //===========%%INC_END=========//\n");
    return @main_content;
}
sub INIT_process
{
   my ($current_sheet, $tm_row, $full_tm_name) = @_ ;
   my @main_content = "    //==========%%INIT_START========//\n";
    # === USER process here to create initial content ==== #
    # user script:

    # ==================================================== #
    @main_content = (@main_content,"    //===========%%INIT_END=========//\n");
    return @main_content;
}
sub INTR_process
{
   my ($current_sheet, $tm_row, $full_tm_name) = @_ ;
   my @main_content = "    //==========%%INTR_START========//\n";
    # === USER process here to create interrupt content == #
    # user script:

    # ==================================================== #
    @main_content = (@main_content,"    //===========%%INTR_END=========//\n");
    return @main_content;
}
```

Figure 3.4: user_script.pm content

- **Explanation**:

A package *user_script* contains four *blank functions* that are defined in an external file *user_script.pm*. *gen_tm.pl* will include this file. In these functions, users will define their scripts. Four blank functions are :

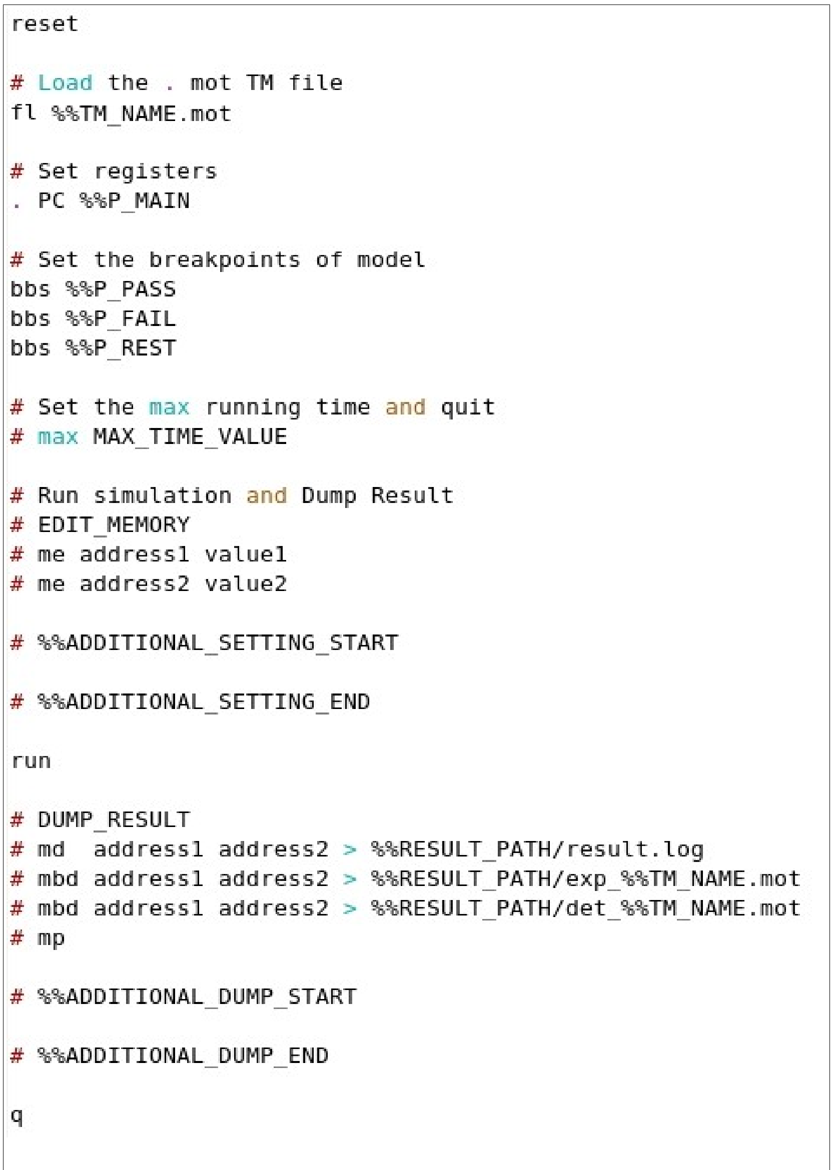
- *INC_process {}* : users define their script to create included content
- *MAIN_process {}* : users define their script to create main content
- *INTC_process {}* : users define their script to create initial content
- *INTR_process {}* : users define their script to create interrupt content

The input of these functions is the information of position of current test pattern in checklist (sheet, row number, test pattern name). User can base on the position of current test pattern to get all necessary information to generate the main content, included content, initial content and interrupt content of test pattern. The outputs of these functions are the content that will be inserted into test pattern.

- Block **%%INC_START – %%INC_END** is defined in *tm.skl* to insert included content which is generated by user script.
- Block **%%MAIN_START – %%MAIN_END** is defined in *tm.skl* to insert main content which is generated by user script.
- Block **%%INIT_START – %%INIT_END** is defined in *tm.skl* to insert initial content which is generated by user script.
- Block **%%INTR_START – %%INTR_END** is defined in *tm.skl* to insert interrupt content which is generated by user script.

3.1.5.4. ssc.skl

Designer can customize ssc skeleton file 's content according to specified model, this file is stored in **[.../scripts/gen_test]** folder, the content of this file is described below :



```
reset

# Load the . mot TM file
fl %%TM_NAME.mot

# Set registers
. PC %%P_MAIN

# Set the breakpoints of model
bbs %%P_PASS
bbs %%P_FAIL
bbs %%P_REST

# Set the max running time and quit
# max MAX_TIME_VALUE

# Run simulation and Dump Result
# EDIT_MEMORY
# me address1 value1
# me address2 value2

# %%ADDITIONAL_SETTING_START

# %%ADDITIONAL_SETTING_END

run

# DUMP_RESULT
# md  address1 address2 > %%RESULT_PATH/result.log
# mbd address1 address2 > %%RESULT_PATH/exp_%%TM_NAME.mot
# mbd address1 address2 > %%RESULT_PATH/det_%%TM_NAME.mot
# mp

# %%ADDITIONAL_DUMP_START

# %%ADDITIONAL_DUMP_END

q
```

Figure 3.5: ssc.skl content

- **Explanation:**

%%TM_NAME: this tag is replaced by test pattern name get from checklist

%%P_MAIN: this tag is replaced by test pattern P_MAIN program section get from Address_map_info file

%%P_PASS: this tag is replaced by test pattern P_PASS program section get from Address_map_info file

%%P_FAIL: this tag is replaced by test pattern P_FAIL program section get from Address_map_info file

%%P_REST: this tag is replaced by test pattern P_REST program section get from Address_map_info file

%%RESULT_PATH: this tag is replace by the path where to store the result of test pattern.

At the beginning when ssc file does not exist, a new pure ssc will be created based on tm.skl file.

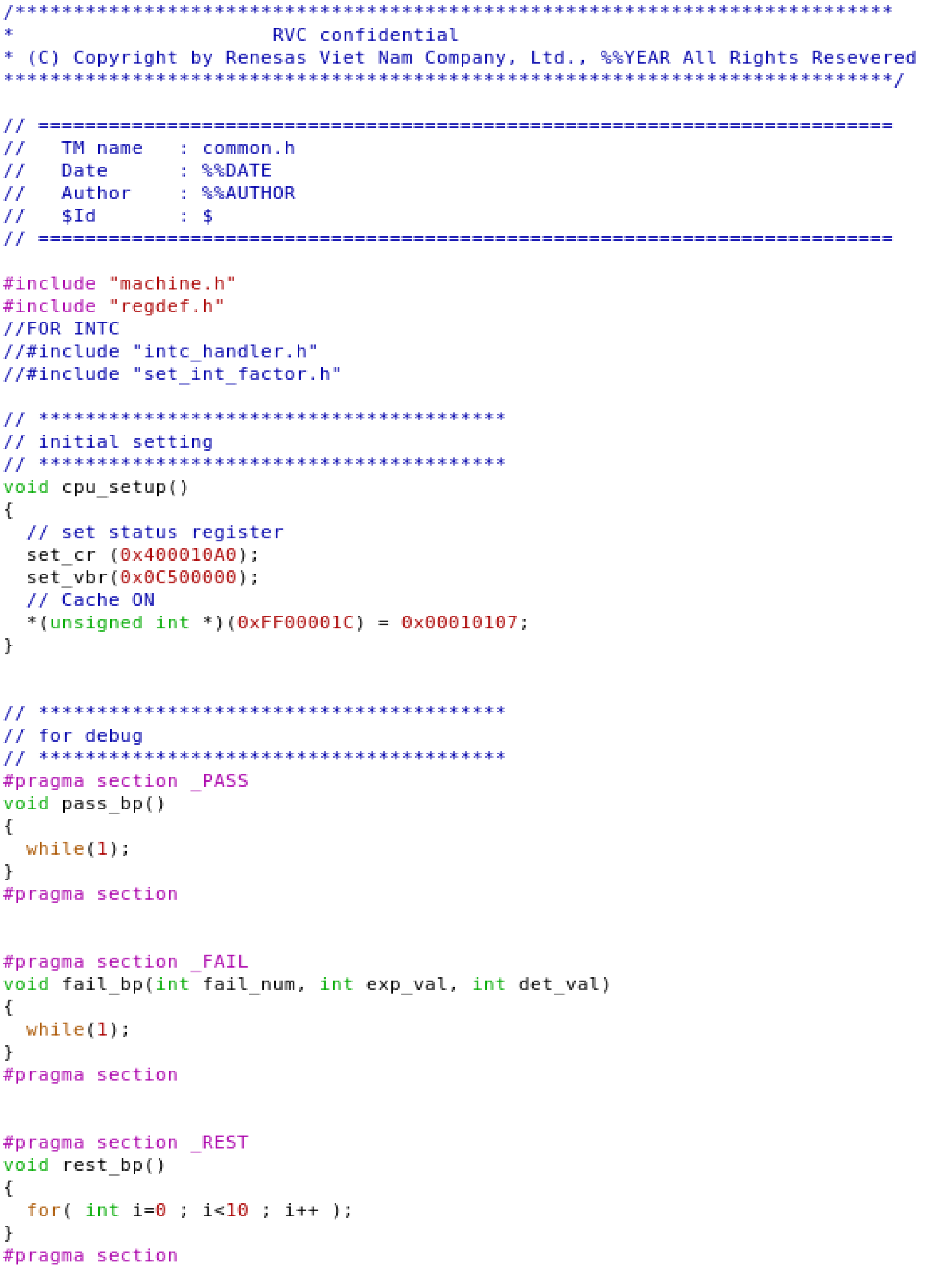
The content in below tags will be kept for new ssc file in case ssc file existed.

%%ADDITIONAL_SETTING_START and **%%ADDITIONAL_SETTING_END**,
%%ADDITIONAL_DUMP_START and **%%ADDITIONAL_DUMP_END**

- **Note:**
  - Designer can customize the content of ssc file according to specified model, both the content between **%%ADDITIONAL_SETTING_START** and **%%ADDITIONAL_SETTING_END**, **%%ADDITIONAL_DUMP_START** and **%%ADDITIONAL_DUMP_END** are optional part that designer don't need to fill if unnecessary.

3.1.5.5. common.h.skl

common.h skeleton file, this file is stored in **[.../script/gen_test]** folder, the content of this file is described below :



```
/***************************************************************************
*                         RVC confidential
* (C) Copyright by Renesas Viet Nam Company, Ltd., %%YEAR All Rights Resevered
***************************************************************************/

// ========================================================================
//   TM name   : common.h
//   Date      : %%DATE
//   Author    : %%AUTHOR
//   $Id       : $
// ========================================================================

#include "machine.h"
#include "regdef.h"
//FOR INTC
//#include "intc_handler.h"
//#include "set_int_factor.h"

// ****************************************
// initial setting
// ****************************************
void cpu_setup()
{
  // set status register
  set_cr (0x400010A0);
  set_vbr(0x0C500000);
  // Cache ON
  *(unsigned int *)(0xFF00001C) = 0x00010107;
}


// ****************************************
// for debug
// ****************************************
#pragma section _PASS
void pass_bp()
{
  while(1);
}
#pragma section


#pragma section _FAIL
void fail_bp(int fail_num, int exp_val, int det_val)
{
  while(1);
}
#pragma section


#pragma section _REST
void rest_bp()
{
  for( int i=0 ; i<10 ; i++ );
}
#pragma section
```

Figure 3.6: common.h.skl content

- **Explanation:**

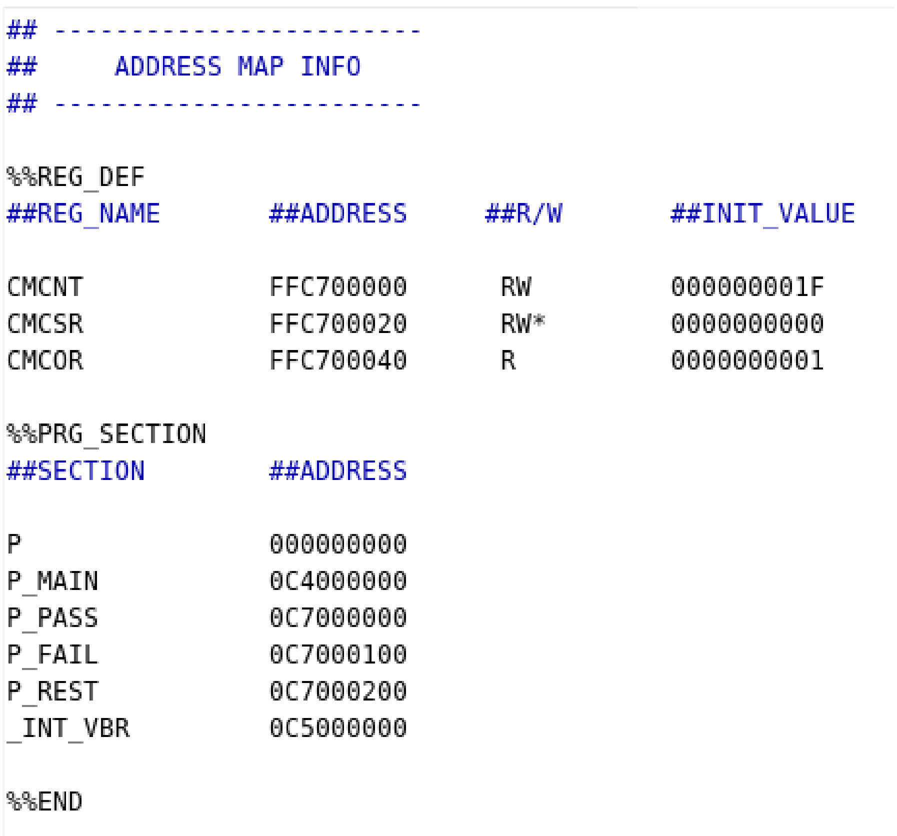
  %%YEAR, %%DATE: these tag are replaced by current date get from operation system

  %%AUTHOR: this tag is replaced by designer who create this file

- Note: Designer can customize the content of this file according to specified model

3.1.5.6. Address_map_info.txt

Address_map_info.txt contains information of model register address and program section info,this file is stored in **[…./scripts]** folder, the content of this file is described below



```
## ------------------------
##     ADDRESS MAP INFO
## ------------------------

%%REG_DEF
##REG_NAME        ##ADDRESS     ##R/W       ##INIT_VALUE

CMCNT             FFC700000      RW         000000001F
CMCSR             FFC700020      RW*        0000000000
CMCOR             FFC700040      R          0000000001

%%PRG_SECTION
##SECTION         ##ADDRESS

P                 000000000
P_MAIN            0C4000000
P_PASS            0C7000000
P_FAIL            0C7000100
P_REST            0C7000200
_INT_VBR          0C5000000

%%END
```

Figure 3.7: Address Map Info content

- **Explanation:**

%%REG_DEF: indicates that this is register address definition area, the register information need to be filled in this section is

- REG_NAME: register name
- ADDRESS: register address,
- R/W: register access (R: can read, W: can write , W*: just can write “0” value, W+: just can write “1” value)
- INIT_VALUE: register initialize value.

Column **##INIT_VALUE** is the initialize value of register. However, *gen_tm.pl* only use the 1st and 2nd column to generate *regdef.h* file. Other columns are planed to use in future.

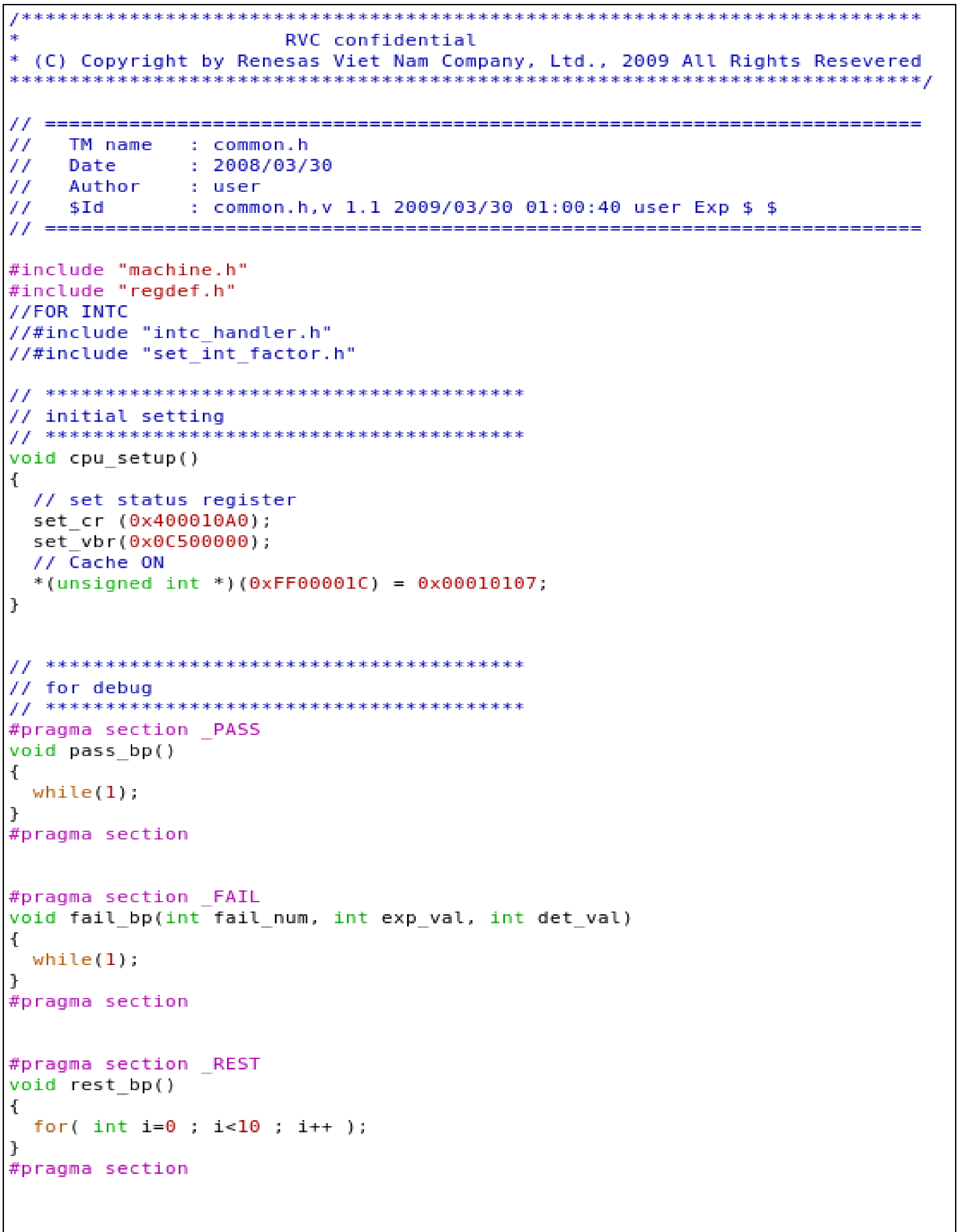
%%PRG_SECTION: indicates that this is program section definition area, the info that need to be declared in this section is

- SECTION: section name
- ADDRESS: section address.

3.1.6.Output file description

3.1.6.1. common.h

common.h contains common things among test patterns such as CPU initialize, PASS/FAILl jump or etc functions, usually it is almost the same with common.h.skl file except interrupt inclusion. This file is stored under **[pat**/**common]** directory.



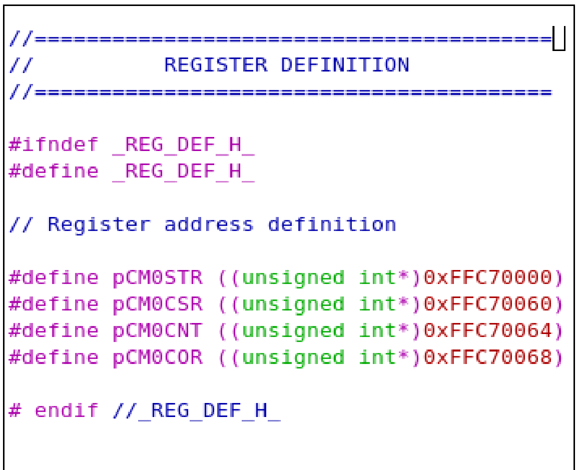
```
/***************************************************************************
*                         RVC confidential
* (C) Copyright by Renesas Viet Nam Company, Ltd., 2009 All Rights Resevered
****************************************************************************/

// ========================================================================
//   TM name    : common.h
//   Date       : 2008/03/30
//   Author     : user
//   $Id        : common.h,v 1.1 2009/03/30 01:00:40 user Exp $ $
// ========================================================================

#include "machine.h"
#include "regdef.h"
//FOR INTC
//#include "intc_handler.h"
//#include "set_int_factor.h"

// ****************************************
// initial setting
// ****************************************
void cpu_setup()
{
  // set status register
  set_cr (0x400010A0);
  set_vbr(0x0C500000);
  // Cache ON
  *(unsigned int *)(0xFF00001C) = 0x00010107;
}


// ****************************************
// for debug
// ****************************************
#pragma section _PASS
void pass_bp()
{
  while(1);
}
#pragma section


#pragma section _FAIL
void fail_bp(int fail_num, int exp_val, int det_val)
{
  while(1);
}
#pragma section


#pragma section _REST
void rest_bp()
{
  for( int i=0 ; i<10 ; i++ );
}
#pragma section
```

Figure 3.8: common.h example

In this example the **%%YEAR**, **%%DATE**, **%%AUTHOR** are replaced by the real value get from the operation system, comment of interrupt include is remove for usage.

3.1.6.2. regdef.h

regdef.h contains register address definition of model, it is created based on register address definition info get from *Address_map_info.txt* file. *regdef.h* is stored in **[pat/common]** folder is included in file *common.h*. The content of this file is described below

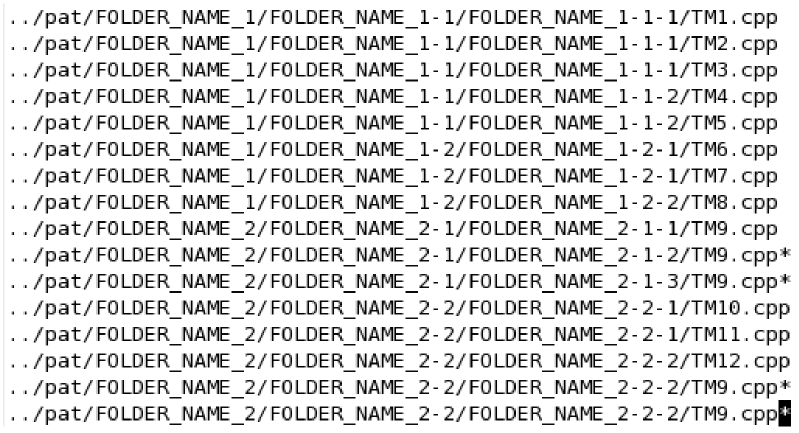


```
//========================================
//          REGISTER DEFINITION
//========================================

#ifndef _REG_DEF_H_
#define _REG_DEF_H_

// Register address definition

#define pCM0STR ((unsigned int*)0xFFC70000)
#define pCM0CSR ((unsigned int*)0xFFC70060)
#define pCM0CNT ((unsigned int*)0xFFC70064)
#define pCM0COR ((unsigned int*)0xFFC70068)

# endif //_REG_DEF_H_
```

Figure 3.9: regdef.h example

3.1.6.3. TM_path.txt

TM_path.txt contains the path where stores of each test pattern, this file info help designer for easy to create script to build test pattern to mot file or create running simulation batch file or etc, this file is stored under [pat] directory. The content of this file is described as below



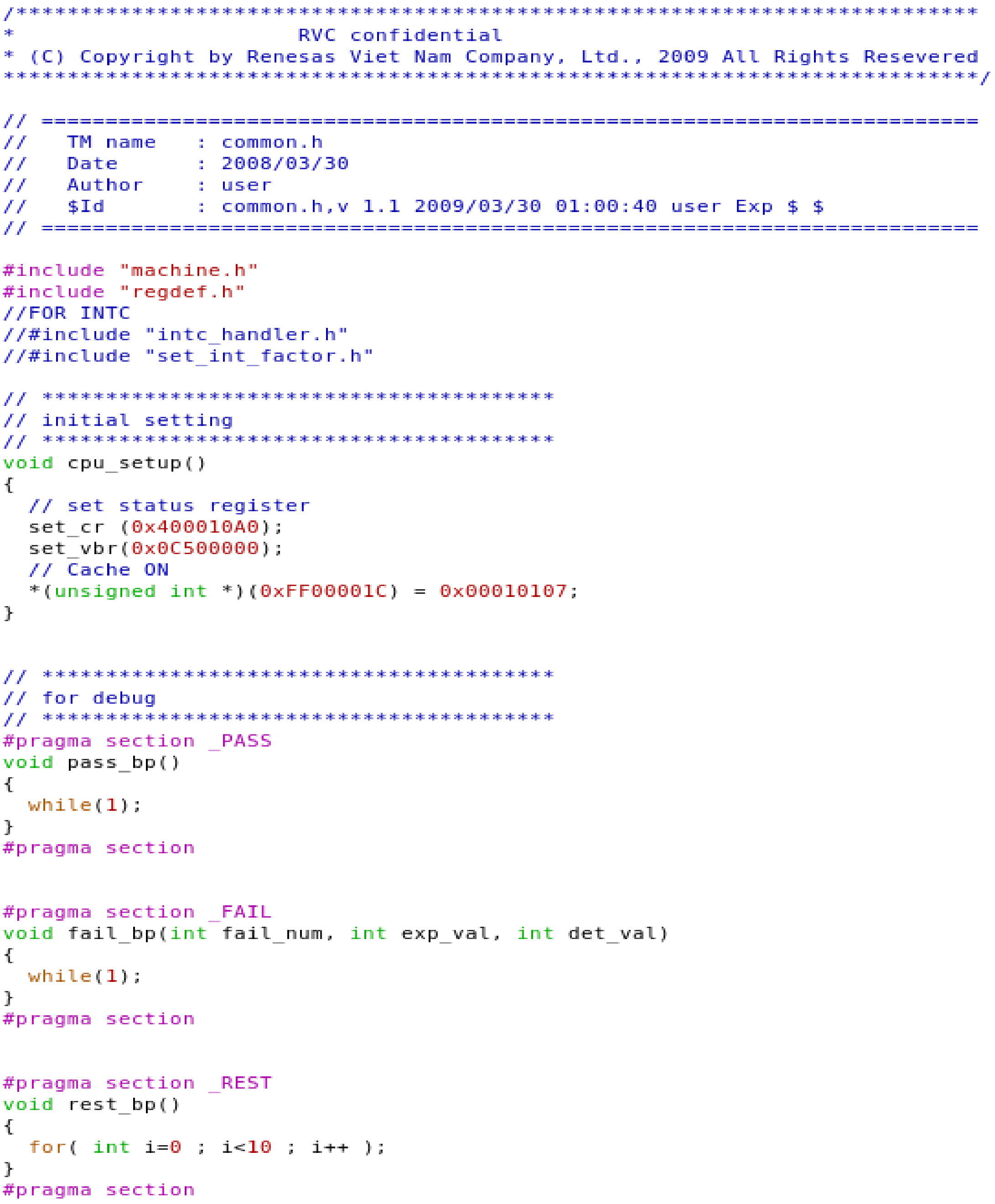
```
../pat/FOLDER_NAME_1/FOLDER_NAME_1-1/FOLDER_NAME_1-1-1/TM1.cpp
../pat/FOLDER_NAME_1/FOLDER_NAME_1-1/FOLDER_NAME_1-1-1/TM2.cpp
../pat/FOLDER_NAME_1/FOLDER_NAME_1-1/FOLDER_NAME_1-1-1/TM3.cpp
../pat/FOLDER_NAME_1/FOLDER_NAME_1-1/FOLDER_NAME_1-1-2/TM4.cpp
../pat/FOLDER_NAME_1/FOLDER_NAME_1-1/FOLDER_NAME_1-1-2/TM5.cpp
../pat/FOLDER_NAME_1/FOLDER_NAME_1-2/FOLDER_NAME_1-2-1/TM6.cpp
../pat/FOLDER_NAME_1/FOLDER_NAME_1-2/FOLDER_NAME_1-2-1/TM7.cpp
../pat/FOLDER_NAME_1/FOLDER_NAME_1-2/FOLDER_NAME_1-2-2/TM8.cpp
../pat/FOLDER_NAME_2/FOLDER_NAME_2-1/FOLDER_NAME_2-1-1/TM9.cpp
../pat/FOLDER_NAME_2/FOLDER_NAME_2-1/FOLDER_NAME_2-1-2/TM9.cpp*
../pat/FOLDER_NAME_2/FOLDER_NAME_2-1/FOLDER_NAME_2-1-3/TM9.cpp*
../pat/FOLDER_NAME_2/FOLDER_NAME_2-2/FOLDER_NAME_2-2-1/TM10.cpp
../pat/FOLDER_NAME_2/FOLDER_NAME_2-2/FOLDER_NAME_2-2-1/TM11.cpp
../pat/FOLDER_NAME_2/FOLDER_NAME_2-2/FOLDER_NAME_2-2-2/TM12.cpp
../pat/FOLDER_NAME_2/FOLDER_NAME_2-2/FOLDER_NAME_2-2-2/TM9.cpp*
../pat/FOLDER_NAME_2/FOLDER_NAME_2-2/FOLDER_NAME_2-2-2/TM9.cpp*
```

Figure 3.10: TM_path.txt example

3.1.6.4. .cpp, .ssc and .txt

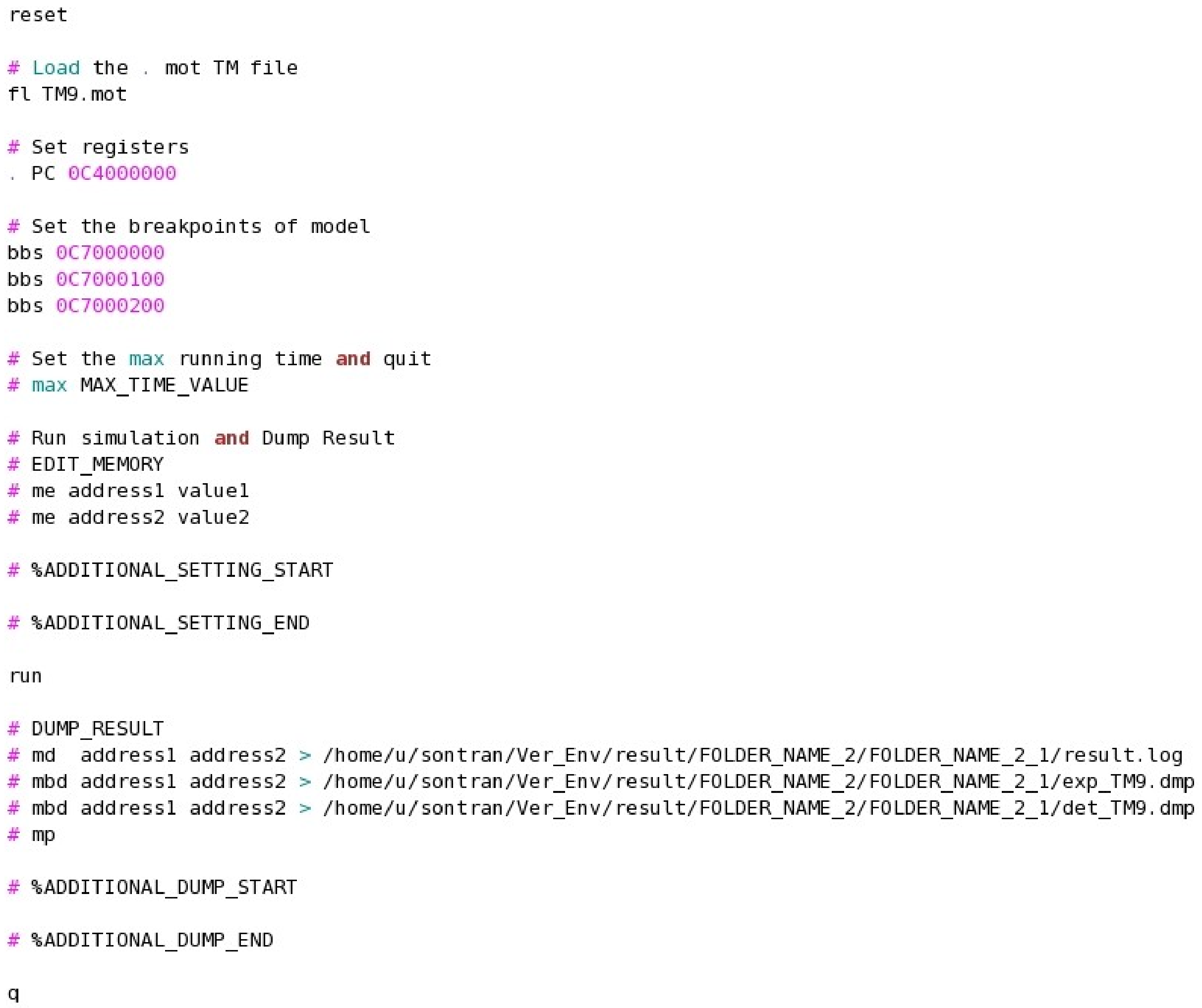
Test pattern source file and ssc file will be stored under **[pat/1st category/2nd category/3rd category].** The folder can be different due to value of opint "**-l**" in *gen_tm.pl.*

Designer can put additional source code into the **%%ADDITIONAL BLOCK**. For the further purpose of designer, designer can modify the skeleton file then regenerate the test patterns. New test patterns and .ssc files still keeping the additional codes.



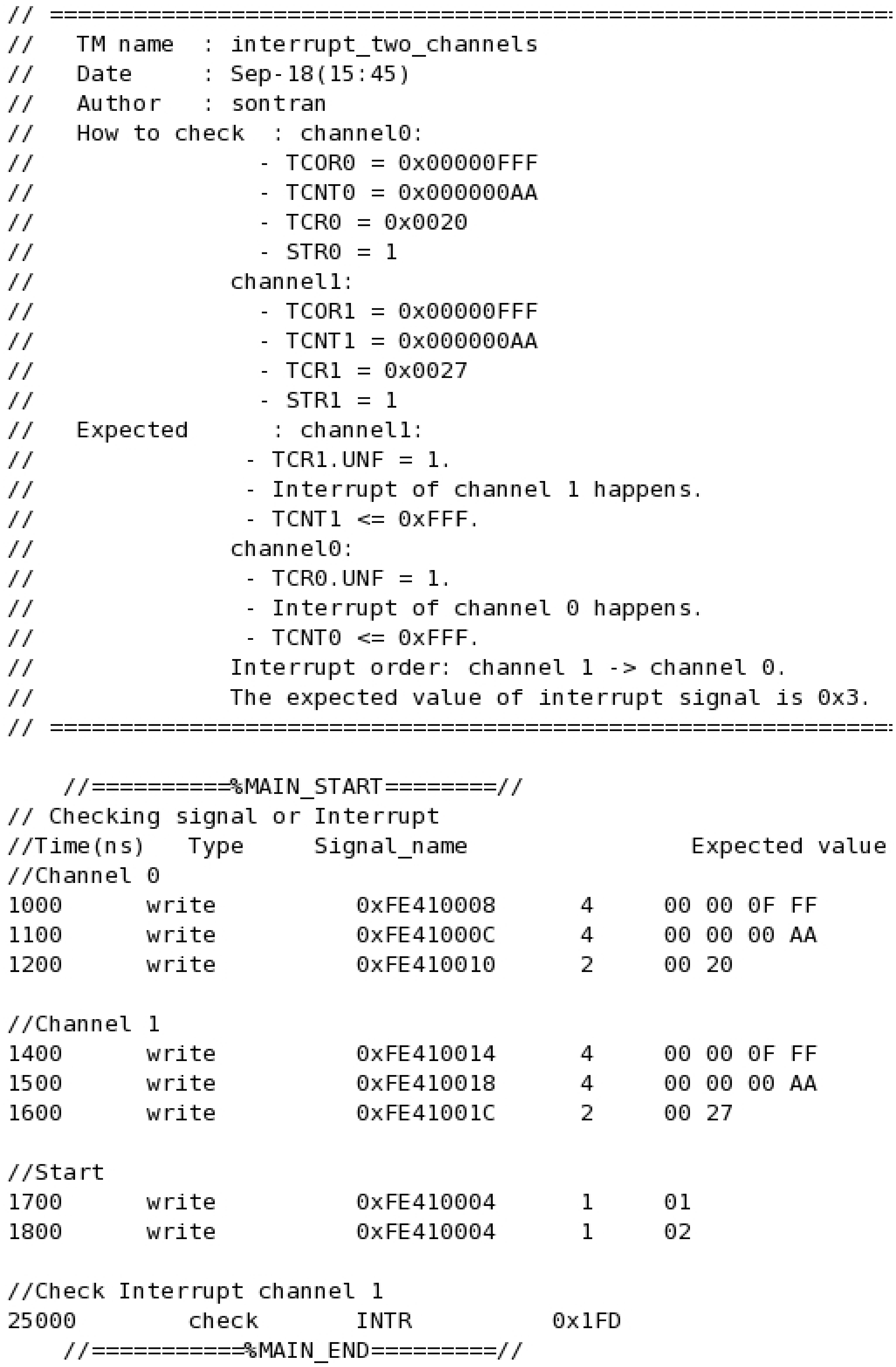
```
/****************************************************************************
*                      RVC confidential
* (C) Copyright by Renesas Viet Nam Company, Ltd., 2009 All Rights Resevered
*****************************************************************************/

// ==========================================================================
//   TM name   : common.h
//   Date      : 2008/03/30
//   Author    : user
//   $Id       : common.h,v 1.1 2009/03/30 01:00:40 user Exp $ $
// ==========================================================================

#include "machine.h"
#include "regdef.h"
//FOR INTC
//#include "intc_handler.h"
//#include "set_int_factor.h"

// ****************************************
// initial setting
// ****************************************
void cpu_setup()
{
  // set status register
  set_cr (0x400010A0);
  set_vbr(0x0C500000);
  // Cache ON
  *(unsigned int *)(0xFF00001C) = 0x00010107;
}


// ****************************************
// for debug
// ****************************************
#pragma section _PASS
void pass_bp()
{
  while(1);
}
#pragma section


#pragma section _FAIL
void fail_bp(int fail_num, int exp_val, int det_val)
{
  while(1);
}
#pragma section


#pragma section _REST
void rest_bp()
{
  for( int i=0 ; i<10 ; i++ );
}
#pragma section
```

Figure 3.11: Example generated test pattern file



```
reset

# Load the . mot TM file
fl TM9.mot

# Set registers
. PC 0C4000000

# Set the breakpoints of model
bbs 0C7000000
bbs 0C7000100
bbs 0C7000200

# Set the max running time and quit
# max MAX_TIME_VALUE

# Run simulation and Dump Result
# EDIT_MEMORY
# me address1 value1
# me address2 value2

# %ADDITIONAL_SETTING_START

# %ADDITIONAL_SETTING_END

run

# DUMP_RESULT
# md  address1 address2 > /home/u/sontran/Ver_Env/result/FOLDER_NAME_2/FOLDER_NAME_2_1/result.log
# mbd address1 address2 > /home/u/sontran/Ver_Env/result/FOLDER_NAME_2/FOLDER_NAME_2_1/exp_TM9.dmp
# mbd address1 address2 > /home/u/sontran/Ver_Env/result/FOLDER_NAME_2/FOLDER_NAME_2_1/det_TM9.dmp
# mp

# %ADDITIONAL_DUMP_START

# %ADDITIONAL_DUMP_END

q
```

Figure 3.12: Example of .ssc file



```
// ============================================================
//   TM name  : interrupt_two_channels
//   Date     : Sep-18(15:45)
//   Author   : sontran
//   How to check  : channel0:
//                 - TCOR0 = 0x00000FFF
//                 - TCNT0 = 0x000000AA
//                 - TCR0 = 0x0020
//                 - STR0 = 1
//               channel1:
//                 - TCOR1 = 0x00000FFF
//                 - TCNT1 = 0x000000AA
//                 - TCR1 = 0x0027
//                 - STR1 = 1
//   Expected      : channel1:
//                - TCR1.UNF = 1.
//                - Interrupt of channel 1 happens.
//                - TCNT1 <= 0xFFF.
//               channel0:
//                - TCR0.UNF = 1.
//                - Interrupt of channel 0 happens.
//                - TCNT0 <= 0xFFF.
//               Interrupt order: channel 1 -> channel 0.
//               The expected value of interrupt signal is 0x3.
// ============================================================

    //==========%MAIN_START========//
// Checking signal or Interrupt
//Time(ns)   Type     Signal_name                Expected value
//Channel 0
1000      write          0xFE410008      4     00 00 0F FF
1100      write          0xFE41000C      4     00 00 00 AA
1200      write          0xFE410010      2     00 20

//Channel 1
1400      write          0xFE410014      4     00 00 0F FF
1500      write          0xFE410018      4     00 00 00 AA
1600      write          0xFE41001C      2     00 27

//Start
1700      write          0xFE410004      1     01
1800      write          0xFE410004      1     02

//Check Interrupt channel 1
25000        check       INTR          0x1FD
    //===========%MAIN_END=========//
```

Figure 3.13: Example of generated .txt test pattern file

3.1.7.Usage

3.1.7.1. Running with normal structure folder and skeleton file

This chapter will guide designer to run a simple example with all the input stored in folder **[home/u/designer/Ver_Env/script]**.

Command :

```
perl gen_tm.pl <checklist name> [option] -p <path of pattern folder>
  -r <path of result folder> -i <path of included folder>
```

Example :

Base on checklist exam_checklist.xls (Figure 3.2.1), create test pattern template with hierarchy level is 2, content column of TM is 13 (*"What to check?"* column), pat folder and result folder are stored at *"/home/u/designer/Ver_Env/" :*

```
% perl gen_tm.pl exam_checklist.xls -l 2 -h 13 -s -p /home/u/user/Ver_Env/pat/
  -r /home/u/user/Ver_Env/result/ -i /home/u/user/Ver_Env/pat/common
```

exam_checklist.xls : example checklist. designer can input the checklist in another folder for further purpose. Ex: */home/u/designer/Output/Verification/Document/VRF-SLD-08009.xls*

-l 2 : hierarchy level is 2 (default is 1) (Optional)

-h 13 : header column 13 (example_checklist.xls --> sheet "testitem" —>column "What to check?"). designer can specify the exactly column in checklist for the content of TM description. (Optional)

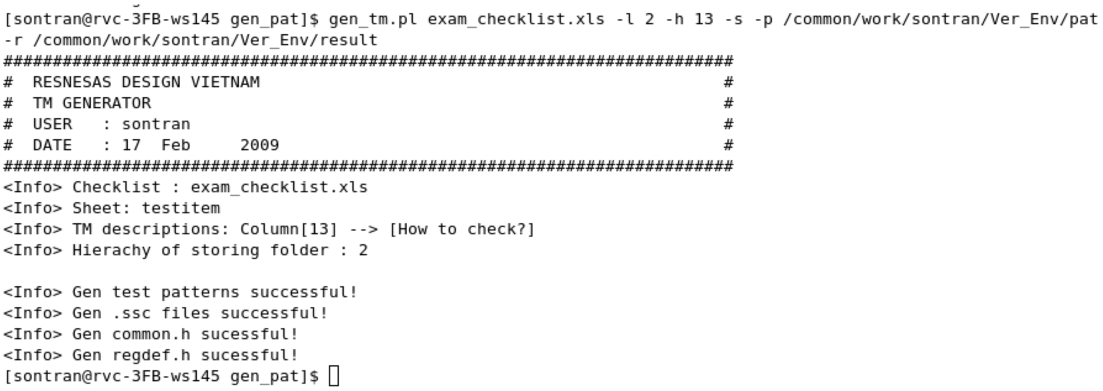
-s : enable creating .ssc files or not. (Optional)

-p /home/u/designer/Ver_Env/pat/ : path to pat folder (Optional with folder structure). When folder structure is changed, this option is **requisite** with **absolute path**.

-r /home/u/designer/Ver_Env/results/ : path to results folder (Optional with folder structure). When folder structure is changed, this option is **requisite** with **absolute path**.

-i /home/u/designer/Ver_Env/pat/common : path to folder contains common.h file (Optional with folder structure). When folder structure is changed, this option is **requisite** with **absolute path**.

Running :

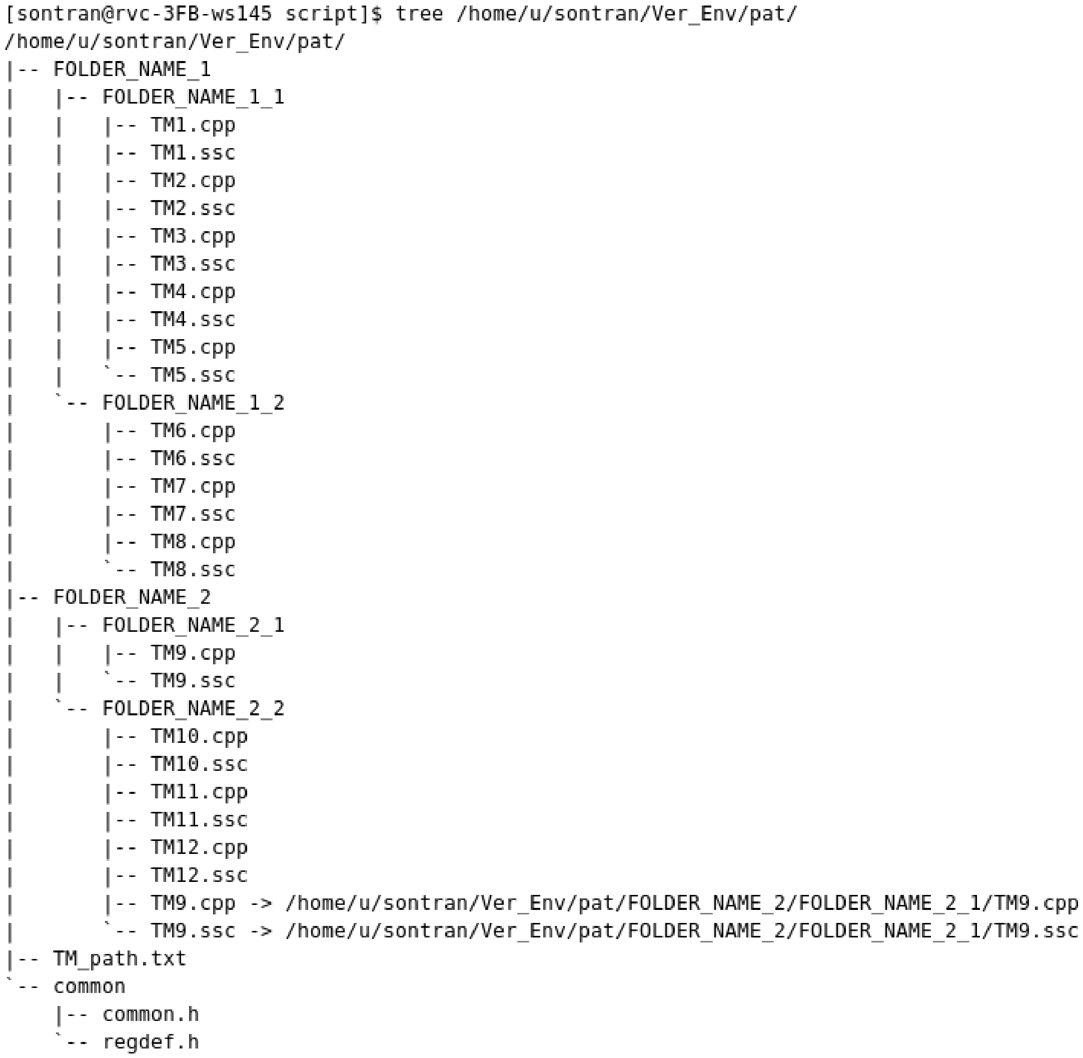


```
[sontran@rvc-3FB-ws145 gen_pat]$ gen_tm.pl exam_checklist.xls -l 2 -h 13 -s -p /common/work/sontran/Ver_Env/pat
-r /common/work/sontran/Ver_Env/result
##########################################################################
#  RESNESAS DESIGN VIETNAM                                               #
#  TM GENERATOR                                                          #
#  USER   : sontran                                                      #
#  DATE   : 17  Feb     2009                                             #
##########################################################################
<Info> Checklist : exam_checklist.xls
<Info> Sheet: testitem
<Info> TM descriptions: Column[13] --> [How to check?]
<Info> Hierachy of storing folder : 2

<Info> Gen test patterns successful!
<Info> Gen .ssc files successful!
<Info> Gen common.h sucessful!
<Info> Gen regdef.h sucessful!
[sontran@rvc-3FB-ws145 gen_pat]$ 
```

Figure 3.14: Example of running gen_tm.pl script

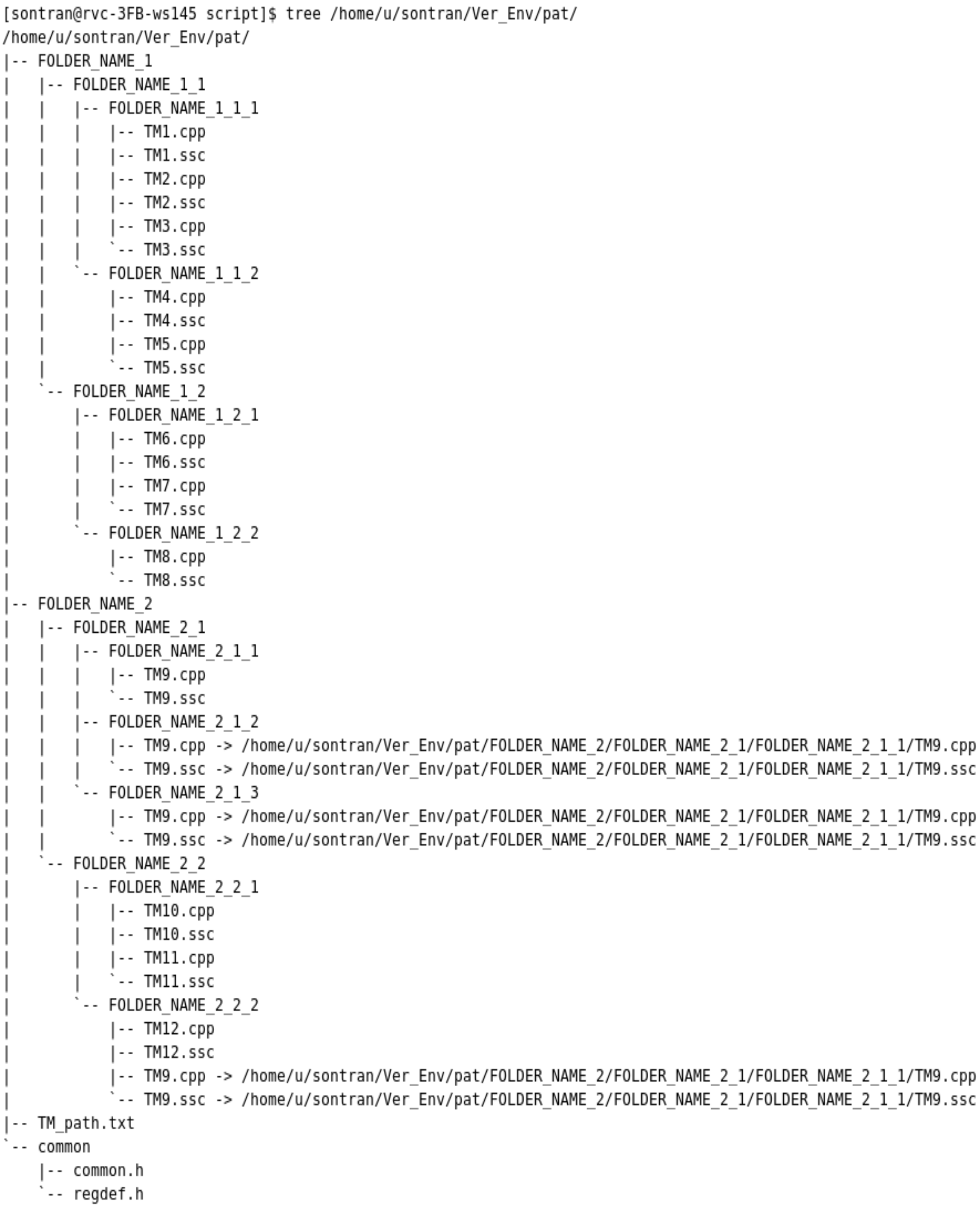
Result :

After running the script, script generated the test patterns and .ssc files then stored as below :



```
[sontran@rvc-3FB-ws145 script]$ tree /home/u/sontran/Ver_Env/pat/
/home/u/sontran/Ver_Env/pat/
|-- FOLDER_NAME_1
|   |-- FOLDER_NAME_1_1
|   |   |-- TM1.cpp
|   |   |-- TM1.ssc
|   |   |-- TM2.cpp
|   |   |-- TM2.ssc
|   |   |-- TM3.cpp
|   |   |-- TM3.ssc
|   |   |-- TM4.cpp
|   |   |-- TM4.ssc
|   |   |-- TM5.cpp
|   |   `-- TM5.ssc
|   `-- FOLDER_NAME_1_2
|       |-- TM6.cpp
|       |-- TM6.ssc
|       |-- TM7.cpp
|       |-- TM7.ssc
|       |-- TM8.cpp
|       `-- TM8.ssc
|-- FOLDER_NAME_2
|   |-- FOLDER_NAME_2_1
|   |   |-- TM9.cpp
|   |   `-- TM9.ssc
|   `-- FOLDER_NAME_2_2
|       |-- TM10.cpp
|       |-- TM10.ssc
|       |-- TM11.cpp
|       |-- TM11.ssc
|       |-- TM12.cpp
|       |-- TM12.ssc
|       |-- TM9.cpp -> /home/u/sontran/Ver_Env/pat/FOLDER_NAME_2/FOLDER_NAME_2_1/TM9.cpp
|       `-- TM9.ssc -> /home/u/sontran/Ver_Env/pat/FOLDER_NAME_2/FOLDER_NAME_2_1/TM9.ssc
|-- TM_path.txt
`-- common
    |-- common.h
    `-- regdef.h
```

Figure 3.15: Example of Folder structure (hierarchy level = 2)



```
[sontran@rvc-3FB-ws145 script]$ tree /home/u/sontran/Ver_Env/pat/
/home/u/sontran/Ver_Env/pat/
|-- FOLDER_NAME_1
|   |-- FOLDER_NAME_1_1
|   |   |-- FOLDER_NAME_1_1_1
|   |   |   |-- TM1.cpp
|   |   |   |-- TM1.ssc
|   |   |   |-- TM2.cpp
|   |   |   |-- TM2.ssc
|   |   |   |-- TM3.cpp
|   |   |   `-- TM3.ssc
|   |   `-- FOLDER_NAME_1_1_2
|   |       |-- TM4.cpp
|   |       |-- TM4.ssc
|   |       |-- TM5.cpp
|   |       `-- TM5.ssc
|   `-- FOLDER_NAME_1_2
|       |-- FOLDER_NAME_1_2_1
|       |   |-- TM6.cpp
|       |   |-- TM6.ssc
|       |   |-- TM7.cpp
|       |   `-- TM7.ssc
|       `-- FOLDER_NAME_1_2_2
|           |-- TM8.cpp
|           `-- TM8.ssc
|-- FOLDER_NAME_2
|   |-- FOLDER_NAME_2_1
|   |   |-- FOLDER_NAME_2_1_1
|   |   |   |-- TM9.cpp
|   |   |   `-- TM9.ssc
|   |   |-- FOLDER_NAME_2_1_2
|   |   |   |-- TM9.cpp -> /home/u/sontran/Ver_Env/pat/FOLDER_NAME_2/FOLDER_NAME_2_1/FOLDER_NAME_2_1_1/TM9.cpp
|   |   |   `-- TM9.ssc -> /home/u/sontran/Ver_Env/pat/FOLDER_NAME_2/FOLDER_NAME_2_1/FOLDER_NAME_2_1_1/TM9.ssc
|   |   `-- FOLDER_NAME_2_1_3
|   |       |-- TM9.cpp -> /home/u/sontran/Ver_Env/pat/FOLDER_NAME_2/FOLDER_NAME_2_1/FOLDER_NAME_2_1_1/TM9.cpp
|   |       `-- TM9.ssc -> /home/u/sontran/Ver_Env/pat/FOLDER_NAME_2/FOLDER_NAME_2_1/FOLDER_NAME_2_1_1/TM9.ssc
|   `-- FOLDER_NAME_2_2
|       |-- FOLDER_NAME_2_2_1
|       |   |-- TM10.cpp
|       |   |-- TM10.ssc
|       |   |-- TM11.cpp
|       |   `-- TM11.ssc
|       `-- FOLDER_NAME_2_2_2
|           |-- TM12.cpp
|           |-- TM12.ssc
|           |-- TM9.cpp -> /home/u/sontran/Ver_Env/pat/FOLDER_NAME_2/FOLDER_NAME_2_1/FOLDER_NAME_2_1_1/TM9.cpp
|           `-- TM9.ssc -> /home/u/sontran/Ver_Env/pat/FOLDER_NAME_2/FOLDER_NAME_2_1/FOLDER_NAME_2_1_1/TM9.ssc
|-- TM_path.txt
`-- common
    |-- common.h
    `-- regdef.h
```

Figure 3.16: Example of Folder structure (hierarchy level = 3)

The detail of each output files were described in previous chapter.

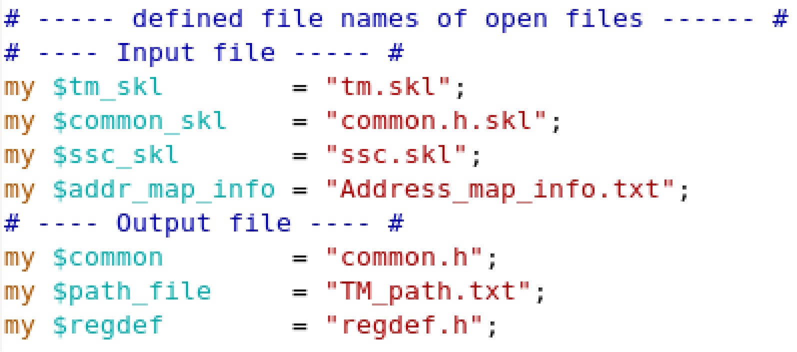
For the duplicated test patterns, they will be automatically linked to the first test pattern.

3.1.7.2. Running with specified structure folder or skeleton file

- To prevent the changing of structure folder for storing test pattern, designer must to define the **-p** and **-r** option to specify the path of pattern folder and result folder.
- For checklist stored in another folder, designer can define the path pf checklist when using command to generate test patterns.
- For the changing of storing folder of input file, designer have to redefine it in gen_tm.pl script.

  For example :

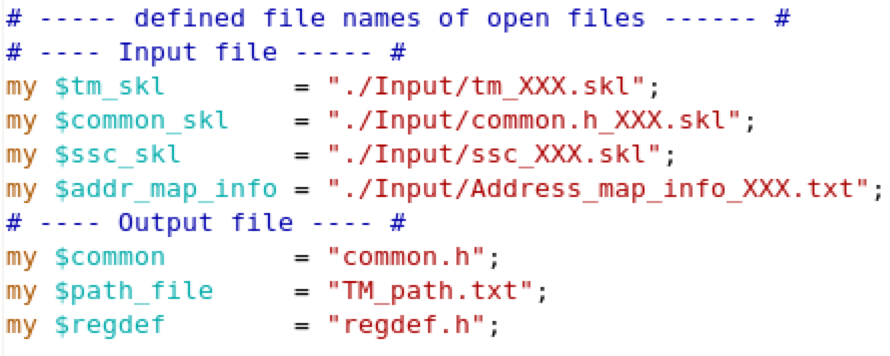
  - Skeleton files, address map info file store in the same folder with gen_tm.pl in **..../script**/ (default)



```
# ----- defined file names of open files ------ #
# ---- Input file ----- #
my $tm_skl        = "tm.skl";
my $common_skl    = "common.h.skl";
my $ssc_skl       = "ssc.skl";
my $addr_map_info = "Address_map_info.txt";
# ---- Output file ---- #
my $common        = "common.h";
my $path_file     = "TM_path.txt";
my $regdef        = "regdef.h";
```

Figure 3.17: Defined path of skeleton file in gen_tm.pl

- Change storing folder to : **..../script/Input**/



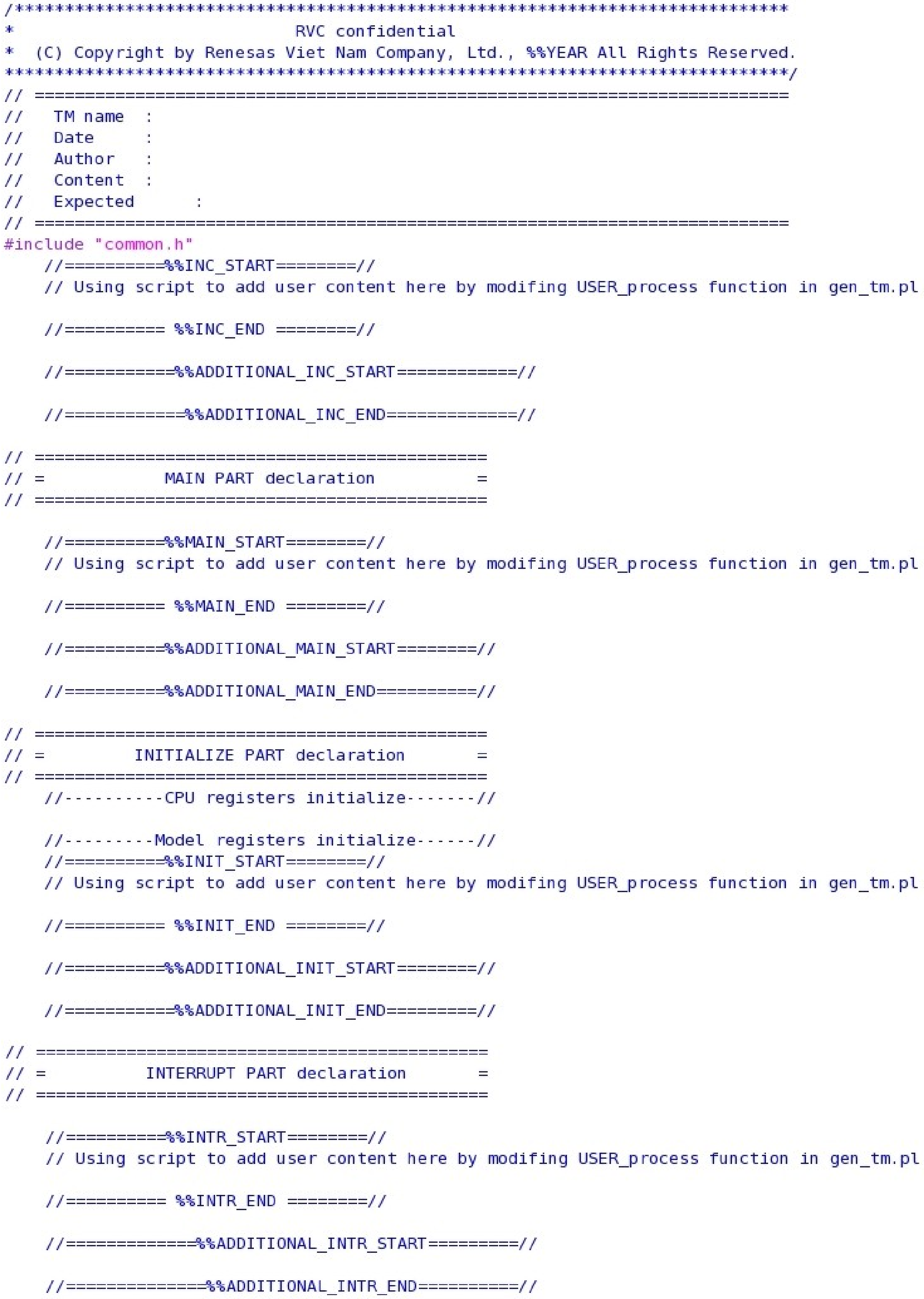
```
# ----- defined file names of open files ------ #
# ---- Input file ----- #
my $tm_skl        = "./Input/tm_XXX.skl";
my $common_skl    = "./Input/common.h_XXX.skl";
my $ssc_skl       = "./Input/ssc_XXX.skl";
my $addr_map_info = "./Input/Address_map_info_XXX.txt";
# ---- Output file ---- #
my $common        = "common.h";
my $path_file     = "TM_path.txt";
my $regdef        = "regdef.h";
```

Figure 3.18: New defined path of skeleton file in gen_tm.pl

3.1.8.Test pattern template usage

3.1.8.1. Test pattern template

After generating based on checklist, designer will have the test patten template as below. The test pattern template has 5 main parts : Header template, Included part, Main part, Initialize part and Interrupt routine part. These part will be described more detail below.



```
/*************************************************************************
*                          RVC confidential
*  (C) Copyright by Renesas Viet Nam Company, Ltd., %%YEAR All Rights Reserved.
*************************************************************************/
// ======================================================================
//   TM name  :
//   Date     :
//   Author   :
//   Content  :
//   Expected      :
// ======================================================================
#include "common.h"
    //==========%%INC_START========//
    // Using script to add user content here by modifing USER_process function in gen_tm.pl

    //========== %%INC_END ========//

    //===========%%ADDITIONAL_INC_START============//

    //============%%ADDITIONAL_INC_END============//

// ============================================
// =            MAIN PART declaration          =
// ============================================

    //==========%%MAIN_START========//
    // Using script to add user content here by modifing USER_process function in gen_tm.pl

    //========== %%MAIN_END ========//

    //==========%%ADDITIONAL_MAIN_START========//

    //==========%%ADDITIONAL_MAIN_END==========//

// ============================================
// =         INITIALIZE PART declaration       =
// ============================================
    //----------CPU registers initialize-------//

    //---------Model registers initialize------//
    //==========%%INIT_START========//
    // Using script to add user content here by modifing USER_process function in gen_tm.pl

    //========== %%INIT_END ========//

    //==========%%ADDITIONAL_INIT_START========//

    //===========%%ADDITIONAL_INIT_END=========//

// ============================================
// =          INTERRUPT PART declaration       =
// ============================================

    //==========%%INTR_START========//
    // Using script to add user content here by modifing USER_process function in gen_tm.pl

    //========== %%INTR_END ========//

    //=============%%ADDITIONAL_INTR_START=========//

    //==============%%ADDITIONAL_INTR_END==========//
```

Figure 3.19: Example of test pattern template

In this standard test pattern, there are 3 main area for designer to modify:

- Inside the **%%BLOCK**.
- Inside the **%%ADDITIONAL_BLOCK**
- Outside the **%%ADDITIONAL_BLOCK**

Inside the %%BLOCK: All the codes inside will be **regenerated** by user script after designer regenerates the test pattern.
Inside the %%ADDITIONAL_BLOCK: All the codes inside will be **kept** after designer regenerates the test pattern. Designer will add code here for each pattern.
Outside the %%ADDITIONAL_BLOCK: All the codes outside will be **deleted** and replaced by template contents after designer regenerates the test pattern. Designer can modify in **tm.skl** then regenerate to apply the modifications to all of test patterns.

3.1.8.2. Generated part by user script

Designer can design 4 function of ***user_script.pm*** (*INC_process {}, MAIN_process {}, INIT_process {}, INTR_process {}*) to generate the content of each part (Included, main, initialize and interrupt part). The content will be **automatically** inserted into :

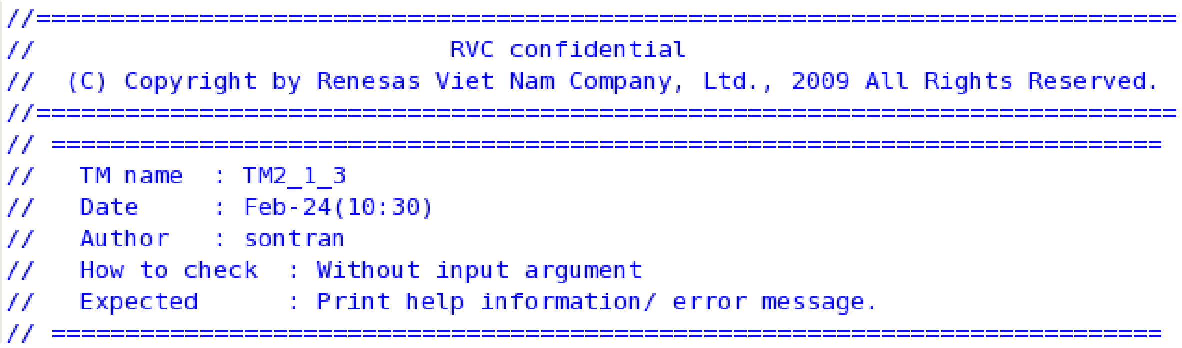
- Block **%%INC_START – %%INC_END** contains included content which is generated by user script.
- Block **%%MAIN_START – %%MAIN_END** contains main content which is generated by user script.
- Block **%%INIT_START – %%INIT_END** contains initial content which is generated by user script
- Block **%%INTR_START – %%INTR_END** contains interrupt content which is generated by user script.

3.1.8.3. Header Template

In this part, designer will declare the test pattern name, Date, Author and the content of TM (Tm description). By using gen_tm.pl, this part was automatically created by collecting

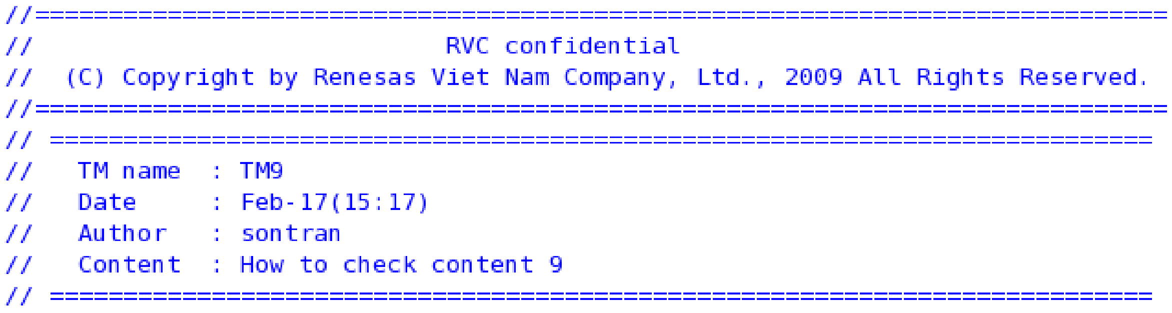
these information from checklist. If in checklist, there is "**Expected result**" column, the header

template will be liked below :



```
//============================================================================
//                            RVC confidential
//  (C) Copyright by Renesas Viet Nam Company, Ltd., 2009 All Rights Reserved.
//============================================================================
// ==========================================================================
//   TM name  : TM2_1_3
//   Date     : Feb-24(10:30)
//   Author   : sontran
//   How to check  : Without input argument
//   Expected      : Print help information/ error message.
// ==========================================================================
```

Figure 3.20: Example of Header Part with expected column

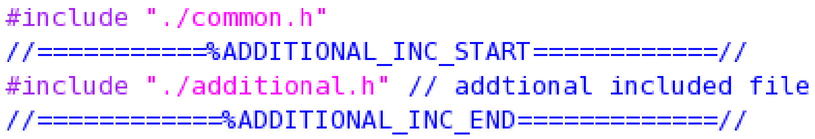


```
//=============================================================================
//                            RVC confidential
//  (C) Copyright by Renesas Viet Nam Company, Ltd., 2009 All Rights Reserved.
//=============================================================================
// ===========================================================================
//   TM name  : TM9
//   Date     : Feb-17(15:17)
//   Author   : sontran
//   Content  : How to check content 9
// ===========================================================================
```

Figure 3.21: Example of Header Part without expected column

3.1.8.4. Included part

In this part, designer will define the include files if necessary. Designer can predefine in tm.skl file for generating all test patterns. By the way, designer can define specified include for each test pattern by adding the definition between block **%%ADDITION_INC_START** and **%%ADDITIONAL_INC_END** and between block **%%ADDITION_INC_START** and **%%ADDITIONAL_INC_END**.



```
#include "./common.h"
//===========%ADDITIONAL_INC_START============//
#include "./additional.h" // addtional included file
//============%ADDITIONAL_INC_END=============//
```

Figure 3.22: Example of Included part

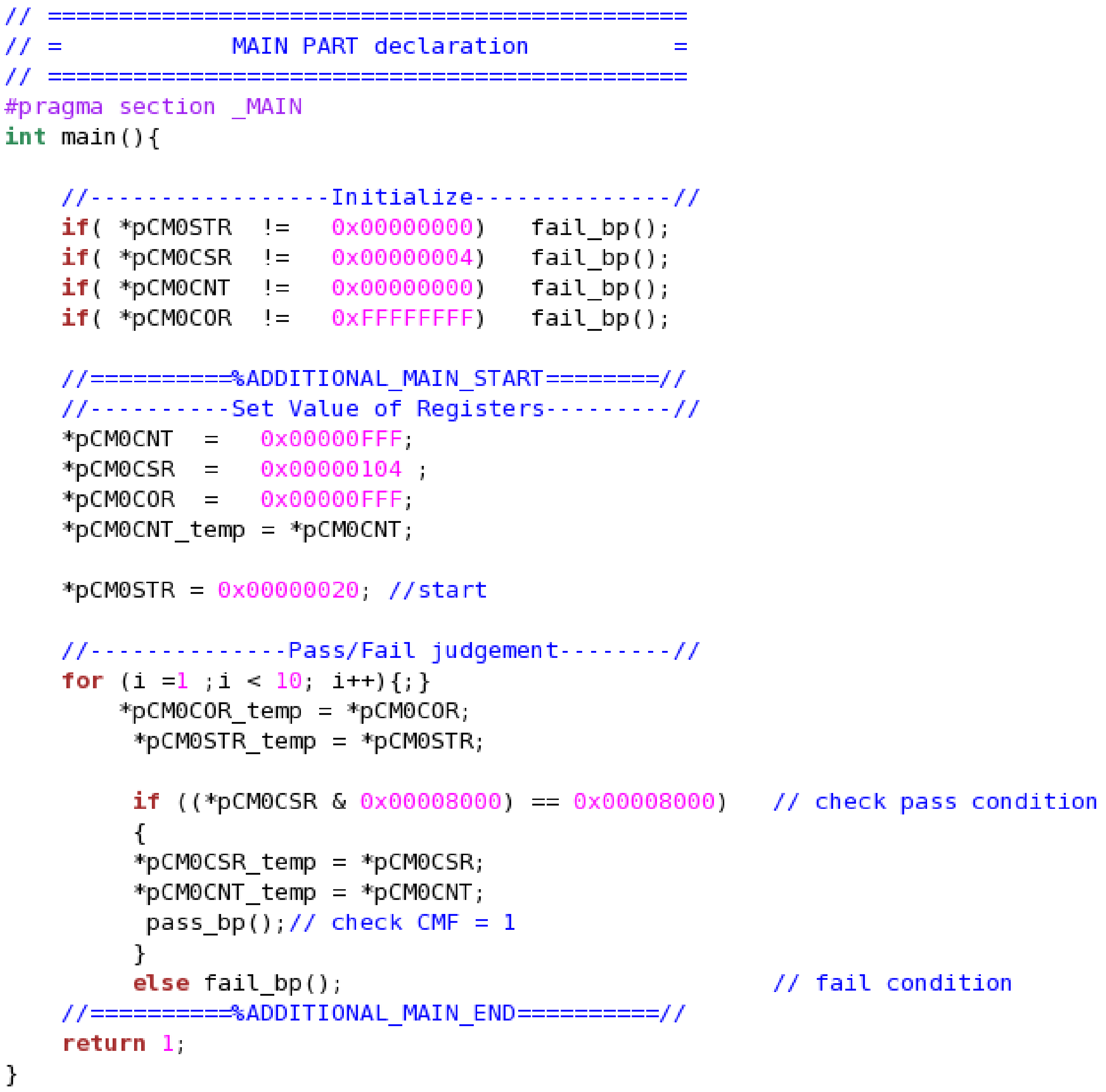
3.1.8.5. Main part

In this part, designer will modify the Initialize part, setting register part for main operation and Pass/Fail judgment part of test pattern.

- **Initialize part :** This part is used for initialize the registers of the model. Designer will define the initialize value for each register.
- **Setting register part:** This part is used for setting the value of each register for main operation of test pattern. This part should be modify by designer for each test pattern base on the main operation of each test pattern.
- **Pass/Fail judgment part :** This part is used for checking the PASS/FAIL condition. When model is running, if one of the conditions occurs, model will jump to PASS/FAILl section. This part should be modify by designer for each test pattern base on the main

operation of each test pattern.

Designer can predefine any part for generating all of the test patterns or adding between block **%%ADDITION_MAIN_START** and **%%ADDITIONAL_MAIN_END** for each test patterns. For Main part, designer defined for each patterns is recommendation.



```
// ============================================
// =            MAIN PART declaration         =
// ============================================
#pragma section _MAIN
int main(){

    //----------------Initialize-------------//
    if( *pCM0STR  !=   0x00000000)   fail_bp();
    if( *pCM0CSR  !=   0x00000004)   fail_bp();
    if( *pCM0CNT  !=   0x00000000)   fail_bp();
    if( *pCM0COR  !=   0xFFFFFFFF)   fail_bp();

    //=========%ADDITIONAL_MAIN_START=======//
    //---------Set Value of Registers---------//
    *pCM0CNT  =   0x00000FFF;
    *pCM0CSR  =   0x00000104 ;
    *pCM0COR  =   0x00000FFF;
    *pCM0CNT_temp = *pCM0CNT;

    *pCM0STR = 0x00000020; //start

    //-------------Pass/Fail judgement--------//
    for (i =1 ;i < 10; i++){;}
        *pCM0COR_temp = *pCM0COR;
         *pCM0STR_temp = *pCM0STR;

         if ((*pCM0CSR & 0x00008000) == 0x00008000)   // check pass condition
         {
         *pCM0CSR_temp = *pCM0CSR;
         *pCM0CNT_temp = *pCM0CNT;
          pass_bp();// check CMF = 1
         }
         else fail_bp();                               // fail condition
    //=========%ADDITIONAL_MAIN_END=========//
    return 1;
}
```

Figure 3.23: Example of Main part

3.1.8.6. Initialize part

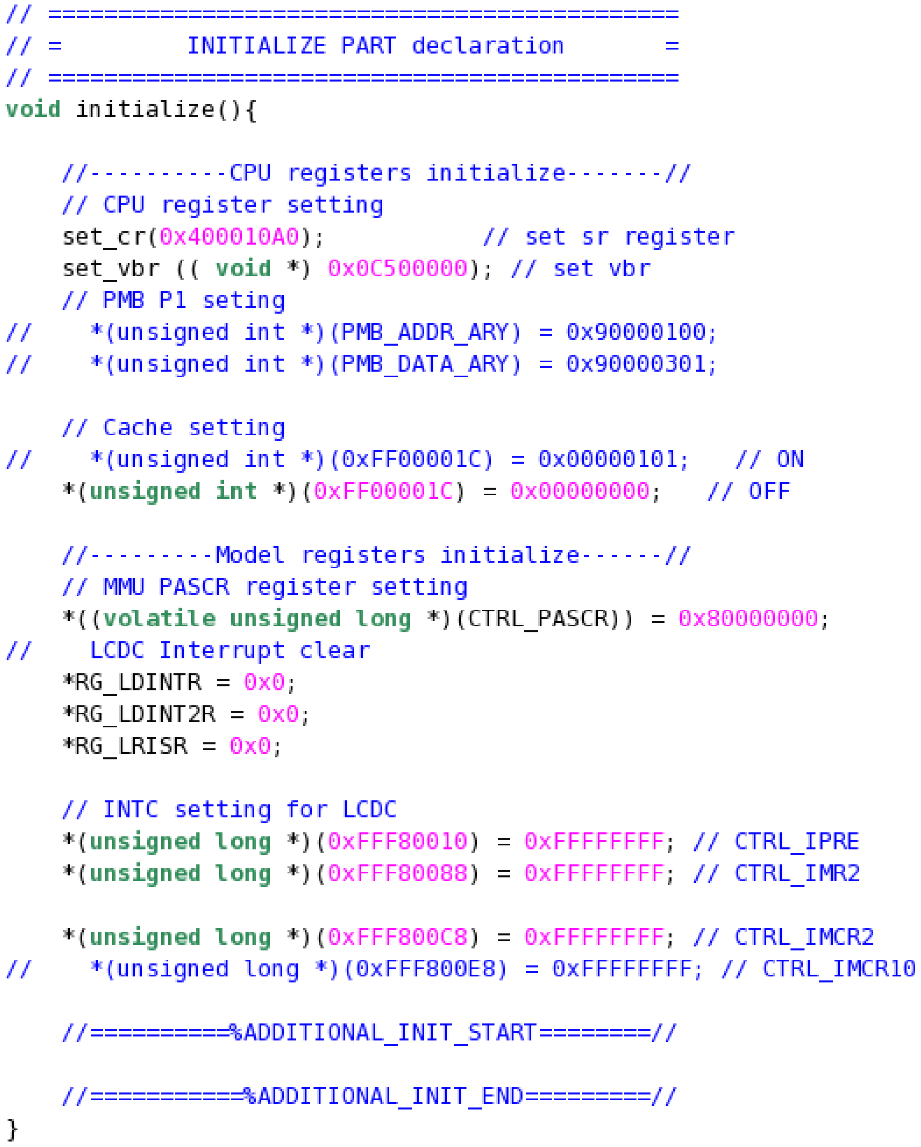
In this part, designer will declare the Initialize function, this function consist of all initialize part such as : CPU setting, Model register initialize, Cache setting...

- **CPU setting :** setting SR register, setting VBR ... this part designer can predefine in *tm.skl* for adding to all test patterns.

- **Model register initialize :** set initialize value of model register for initialize function.
- **Cache setting :** setting cache for CPU.

This part is usually added to all test patterns, designer should predefine in tm.skl. For other purpose of each test pattern, designer can modify additional part into between block :

%%ADDITIONAL_INIT_START and **%%ADDITIONAL_INIT_END**.

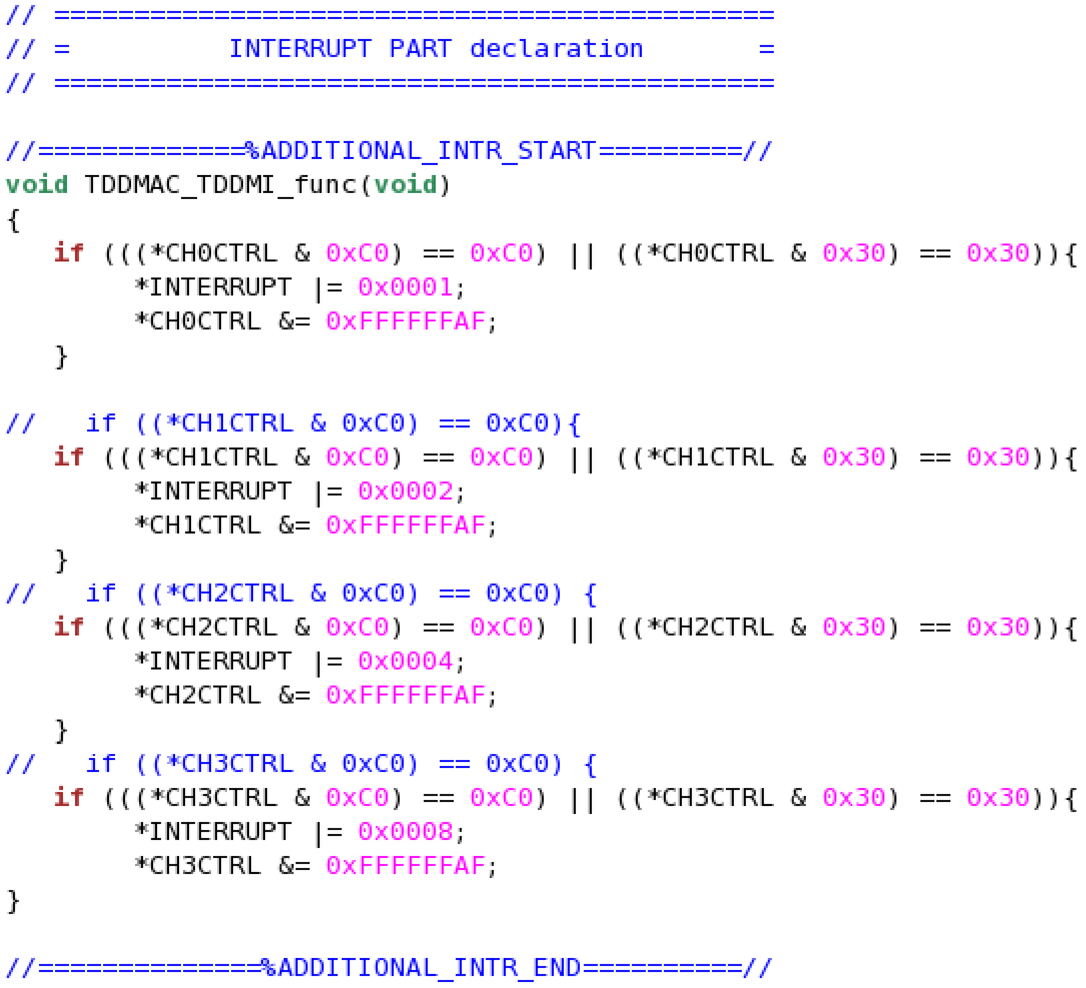


```
// =============================================
// =        INITIALIZE PART declaration       =
// =============================================
void initialize(){

    //----------CPU registers initialize-------//
    // CPU register setting
    set_cr(0x400010A0);           // set sr register
    set_vbr (( void *) 0x0C500000); // set vbr
    // PMB P1 seting
//    *(unsigned int *)(PMB_ADDR_ARY) = 0x90000100;
//    *(unsigned int *)(PMB_DATA_ARY) = 0x90000301;

    // Cache setting
//    *(unsigned int *)(0xFF00001C) = 0x00000101;   // ON
    *(unsigned int *)(0xFF00001C) = 0x00000000;   // OFF

    //---------Model registers initialize------//
    // MMU PASCR register setting
    *((volatile unsigned long *)(CTRL_PASCR)) = 0x80000000;
//    LCDC Interrupt clear
    *RG_LDINTR = 0x0;
    *RG_LDINT2R = 0x0;
    *RG_LRISR = 0x0;

    // INTC setting for LCDC
    *(unsigned long *)(0xFFF80010) = 0xFFFFFFFF; // CTRL_IPRE
    *(unsigned long *)(0xFFF80088) = 0xFFFFFFFF; // CTRL_IMR2

    *(unsigned long *)(0xFFF800C8) = 0xFFFFFFFF; // CTRL_IMCR2
//    *(unsigned long *)(0xFFF800E8) = 0xFFFFFFFF; // CTRL_IMCR10

    //==========%ADDITIONAL_INIT_START========//

    //===========%ADDITIONAL_INIT_END=========//
}
```

Figure 3.24: Example of initialization part

3.1.8.7. Interrupt routine part

In this part, designer will modify the interrupt function for model. Because there are only a part of test patterns using interrupt so this part should be written for each test pattern. designer can add into between : **%%ADDITIONAL_INTR_START** and **%%ADDITIONAL_INTR_END**.



```
// =============================================
// =          INTERRUPT PART declaration       =
// =============================================

//=============%ADDITIONAL_INTR_START=========//
void TDDMAC_TDDMI_func(void)
{
   if (((*CH0CTRL & 0xC0) == 0xC0) || ((*CH0CTRL & 0x30) == 0x30)){
        *INTERRUPT  |= 0x0001;
        *CH0CTRL &= 0xFFFFFFAF;
   }

//   if ((*CH1CTRL & 0xC0) == 0xC0){
   if (((*CH1CTRL & 0xC0) == 0xC0) || ((*CH1CTRL & 0x30) == 0x30)){
        *INTERRUPT  |= 0x0002;
        *CH1CTRL &= 0xFFFFFFAF;
   }
//   if ((*CH2CTRL & 0xC0) == 0xC0) {
   if (((*CH2CTRL & 0xC0) == 0xC0) || ((*CH2CTRL & 0x30) == 0x30)){
        *INTERRUPT  |= 0x0004;
        *CH2CTRL &= 0xFFFFFFAF;
   }
//   if ((*CH3CTRL & 0xC0) == 0xC0) {
   if (((*CH3CTRL & 0xC0) == 0xC0) || ((*CH3CTRL & 0x30) == 0x30)){
        *INTERRUPT  |= 0x0008;
        *CH3CTRL &= 0xFFFFFFAF;
}

//==============%ADDITIONAL_INTR_END==========//
```

Figure 3.25: Example of Interrupt part

3.2.gen_mot.pl

3.2.1.Introduction

To automatic the compilation progress by using SH compiler, *gen_mot.pl* is created to take this mission. *gen_mot.pl* will help designer to compile list of test pattern into MOT file.

Some notable feature :

– Compile serial test patterns into MOT file

– Specify output results

Requirement :

– SH compiler. SH compiler is the Renesas in-house tool.

3.2.2.Block chart

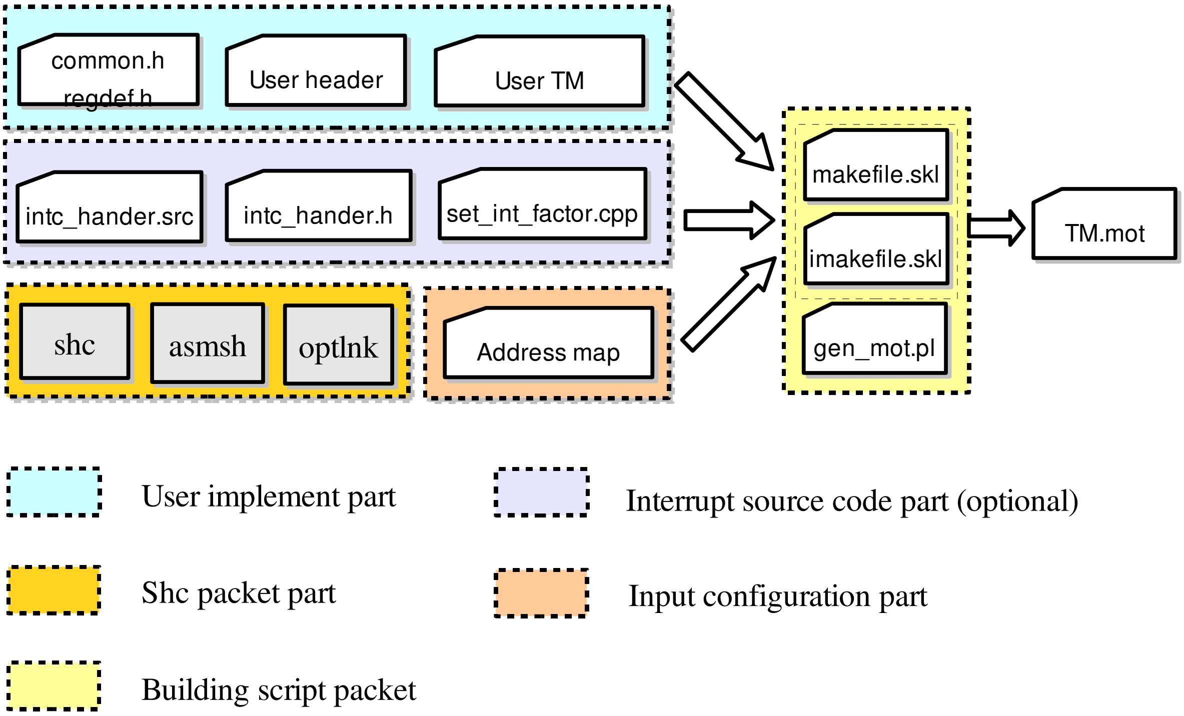


Figure 3.26: Test pattern building script flow

•**Explanation:** There are four necessary basic parts to the building script to generate mot file as below:

○ **Designer implement part:** this includes test patterns and other related included file, common.h is the place to define common functions such as cpu initialize, pass and fail loop function, regdef.h is the place for designer to defined register address of models.

○ **Interrupt part:** this part includes source files for interrupt execution and it is an optional part which belong to test pattern has interrupt or not.

○ **SH-compiler packet part:** this includes necessary tools to build C source file to mot file. This packet is created by Renesas

○ **Input configuration part:** this includes some input info for building script such as address map info or etc...

3.2.3.Supported function

Type	Content
Script name	gen_mot.pl
Language	Perl version 5.8.5 or later
Purpose	This script is used to convert test pattern source file into mot format by creating makefile for test pattern source code
Input file	+ Test pattern source file or folder that contains a set of source file. + Makefile skeleton: makefile.skl for source file without interrupt, imakefile.skl for source file contains interrupt, these file are put at the same directory with gen_mot.pl script + Address_map_info.txt contains program section info, this file is put in [scripts] directory.
Output file	Test pattern mot file.
Usage	*perl gen_mot.pl* **[input_option] <input_path> [option]**
Argument	**[input_option]** **"-r" <input_path>** : indicating that <input_path> is a path to directory – Default value: none **"-f" <input_path>**: indicating that <input_path> is a path file name – Default value: none **<input_path>** : path to file or directory based on input_option
Option	**-o <output>** : specify path for output mot file - default value is the same directory that contains input test pattern. **-t <tool_chain>** : specify path for SH compiler tool chain - default value is "./" **-i <include>** : specify path for included file - default value is the same directory contains test pattern **"-help"** : help information

Table 3.3: gen_mot.pl functions

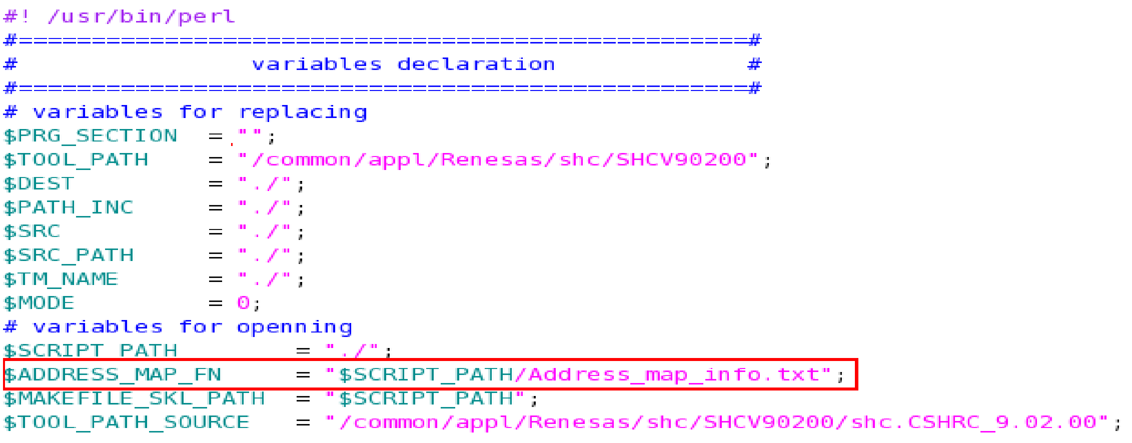
3.2.4.Running message

Severity	Error Message	Content
Error	Lacking argument for option !	User lacking of declaring argument for option that needs an argument along with.
Error	Option is not legal !	User declare an illegal option that does support in script
Error	Can not open Address_map_info.txt file !	Addres_map_info.txt file can not be opened due to wrong path or not exist.
Error	PROGRAM SECTION in Address_map_info.txt is illegal !	Program section tag in Address_map_info.txt is illegal not the same with "%%PRG_SECTION". May be "%%PRGORAM_SECTION" or etc.
Error	Can not open common.h file !	Can not open file common.h that included in test pattern due to wrong path or not exist.
Error	Can not open makefile.skl/imakefile.skl file !	Can not open file makefile.skl or imakefile.skl file due to wrong path or not exist.
Error	Can not open $dir directory !	Can not specify the input directory that user needs to build.
Error	Can not open $file.cpp file !	Can not specify the input file that user needs to build.
Error	C3321 (F) Illegal environment variable	This error from SHC tool, notify that user does not source the "source file" for this tool yet.

Table 3.4: Running message of gen_mot.pl

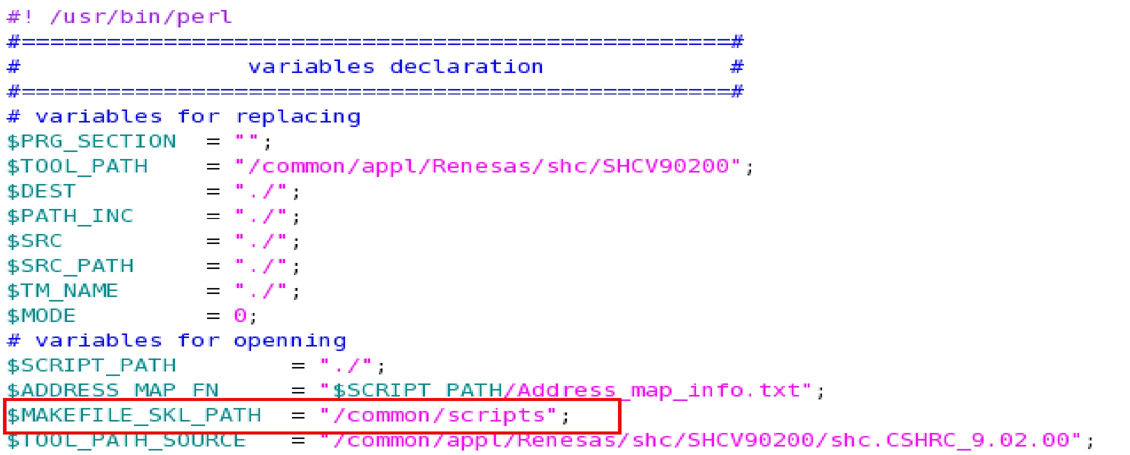
3.2.5.Input file

Before running *gen_mot.pl* script, designer needs to ensure that the input files that requires for the script is enough. These input files are *makefile.skl*, *imakefile.skl* and *Address_map_info.txt*. *makefile.skl* and *imakefile.skl* are put in the same directory with gen_mot script and *Address_map_info.txt* is put in **[scripts]** directory. When designer wants to change the location of these file,designer must modified directly to the scripts. The location of this info the in the script is described as below:



```
#! /usr/bin/perl
#==================================================#
#                 variables declaration            #
#==================================================#
# variables for replacing
$PRG_SECTION   = "";
$TOOL_PATH     = "/common/appl/Renesas/shc/SHCV90200";
$DEST          = "./";
$PATH_INC      = "./";
$SRC           = "./";
$SRC_PATH      = "./";
$TM_NAME       = "./";
$MODE          = 0;
# variables for openning
$SCRIPT_PATH         = "./";
$ADDRESS_MAP_FN      = "$SCRIPT_PATH/Address_map_info.txt";
$MAKEFILE_SKL_PATH   = "$SCRIPT_PATH";
$TOOL_PATH_SOURCE    = "/common/appl/Renesas/shc/SHCV90200/shc.CSHRC_9.02.00";
```

Figure 3.27: address map info path change location



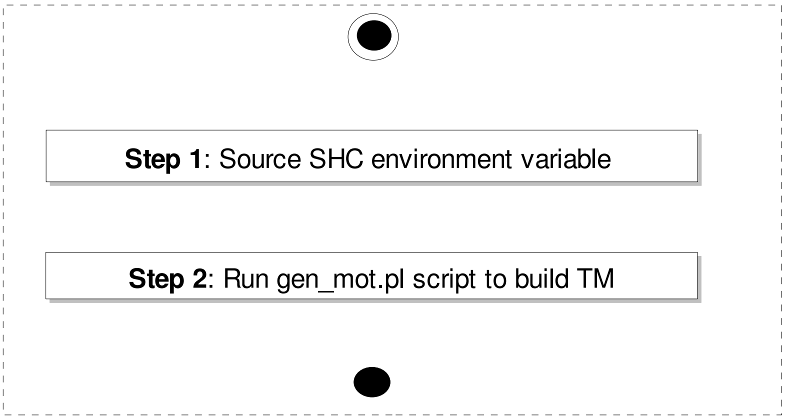
```
#! /usr/bin/perl
#==================================================#
#                 variables declaration            #
#==================================================#
# variables for replacing
$PRG_SECTION   = "";
$TOOL_PATH     = "/common/appl/Renesas/shc/SHCV90200";
$DEST          = "./";
$PATH_INC      = "./";
$SRC           = "./";
$SRC_PATH      = "./";
$TM_NAME       = "./";
$MODE          = 0;
# variables for openning
$SCRIPT_PATH         = "./";
$ADDRESS_MAP_FN      = "$SCRIPT_PATH/Address_map_info.txt";
$MAKEFILE_SKL_PATH   = "/common/scripts";
$TOOL_PATH_SOURCE    = "/common/appl/Renesas/shc/SHCV90200/shc.CSHRC_9.02.00";
```

Figure 3.28: makefile skeleton path change location

3.2.6.Output file

The output file will be MOT file. The format of MOT file won't be describe in the scope of this guide.Please refer “Motorola S-record Format.pdf” document.

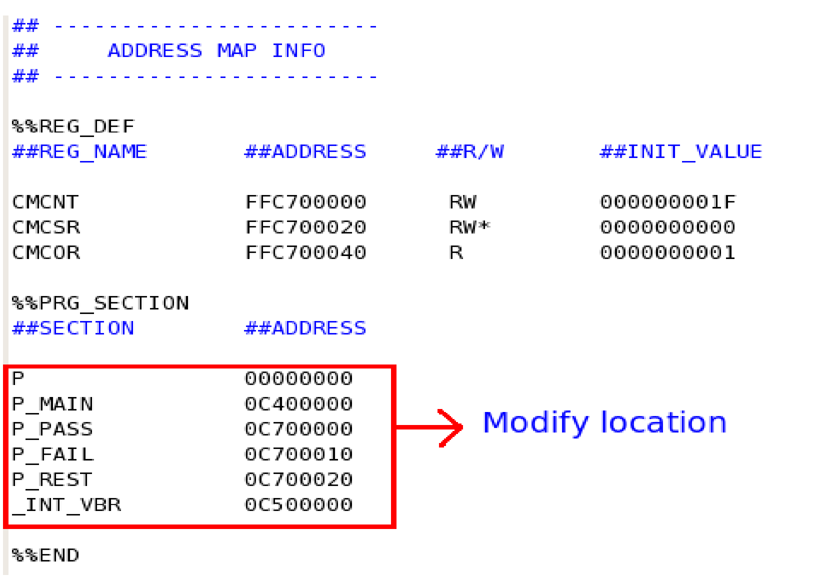
3.2.7.Usage



Step 1: Source SHC environment variable
Step 2: Run gen_mot.pl script to build TM

Figure 3.29: Building test pattern operation flow

__Note :__ In some case if designer wants to change program section info for test pattern, designer can change this info in *Address_map_info.txt file*. The location that designer needs to modified is as follow before compiling test pattern :

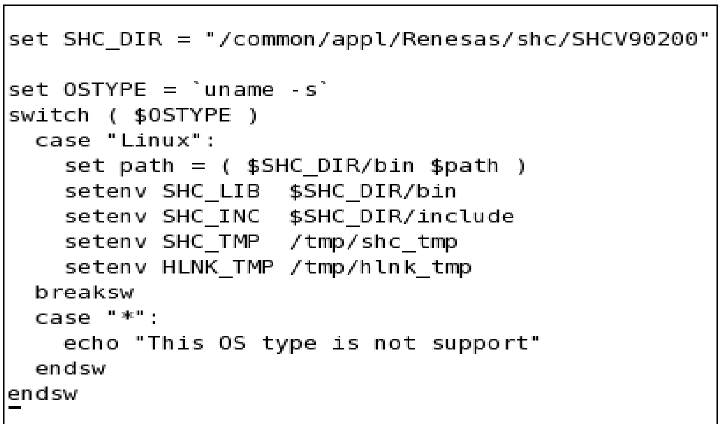


```
## ------------------------
##     ADDRESS MAP INFO
## ------------------------

%%REG_DEF
##REG_NAME        ##ADDRESS      ##R/W       ##INIT_VALUE

CMCNT             FFC700000       RW         000000001F
CMCSR             FFC700020       RW*        0000000000
CMCOR             FFC700040       R          0000000001

%%PRG_SECTION
##SECTION         ##ADDRESS

P                 00000000
P_MAIN            0C400000
P_PASS            0C700000
P_FAIL            0C700010
P_REST            0C700020
_INT_VBR          0C500000

%%END
```

Figure 3.30: Address_map_info.txt modifying location

Explanation:

Step 1: Before running *gen_mot.pl* script, designer needs to confirm that SHC tool is installed and sources the "source file" of this tool to set environment variable for building. For example if SHC tool is installed at: /common/appl/Renesas/shc/SHCV90200/, designer needs to find the ".cshrc" file in this installed directory and source that file . The content of this source file is usually as below:



```
set SHC_DIR = "/common/appl/Renesas/shc/SHCV90200"

set OSTYPE = `uname -s`
switch ( $OSTYPE )
  case "Linux":
    set path = ( $SHC_DIR/bin $path )
    setenv SHC_LIB  $SHC_DIR/bin
    setenv SHC_INC  $SHC_DIR/include
    setenv SHC_TMP  /tmp/shc_tmp
    setenv HLNK_TMP /tmp/hlnk_tmp
  breaksw
  case "*":
    echo "This OS type is not support"
  endsw
endsw
```

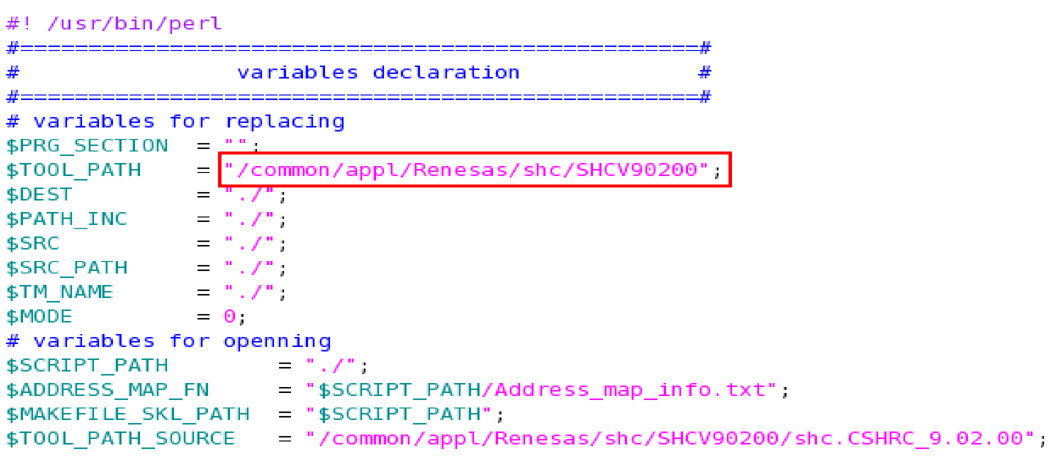
Figure 3.31: shc source file content

Example:

Designer needs to source "SHC source file" to set environment variables, this source is usually put at the same directory with SHC tool setup directory.

source /common/appl/Renesas/shc/SHCV90200/**shc.CSHRC_9.02.00**

For some reasons that cause inconvenience for designer to set the tool chain path because designer needs to declare tool chain path option *"-t <tool_chain>"* in each time designer builds test pattern, there has another way to resolve by changing directly this path in source script. The location that designer needs to change is described as below:



```
#! /usr/bin/perl
#==================================================#
#                variables declaration             #
#==================================================#
# variables for replacing
$PRG_SECTION   = "";
$TOOL_PATH     = "/common/appl/Renesas/shc/SHCV90200";
$DEST          = "./";
$PATH_INC      = "./";
$SRC           = "./";
$SRC_PATH      = "./";
$TM_NAME       = "./";
$MODE          = 0;
# variables for openning
$SCRIPT_PATH         = "./";
$ADDRESS_MAP_FN      = "$SCRIPT_PATH/Address_map_info.txt";
$MAKEFILE_SKL_PATH   = "$SCRIPT_PATH";
$TOOL_PATH_SOURCE    = "/common/appl/Renesas/shc/SHCV90200/shc.CSHRC_9.02.00";
```

Figure 3.32: tool chain path change location

Step 2: In this step,designer runs *gen_mot.pl* scripts to build test pattern from source file to mot file. Then run script file to build test pattern file with the below syntax:

```
perl gen_mot.pl <input option> <input> [option]
```

Example:

```
perl gen_mot.pl -f /common/work/input_test/test.cpp  -t /common/appl/Renesas/shc/SHCV90200/
```

Example above shows that designer wants to build test.cpp file to mot file, the output and included file is the same with input path : /common/work/input_test/ and the tool chain is specified at : /common/appl/Renesas/shc/SHCV90200/

If designer wants to put output mot file to another place that is not the same with the input he needs to declare -o <output> option to specify this and the input argument can be as below:

```
perl gen_mot.pl -f /common/work/input_test/test.cpp -o /common/work/output_test\

                -t /common/appl/Renesas/shc/SHCV90200/
```

If designer wants to declare an include path he needs to declare -i <include> option to specify this and the input argument is like example as below:

```
perl gen_mot.pl -f /common/work/input_test/test.cpp -o /common/work/output_test\

                -i /common/include -t /common/appl/Renesas/shc/SHCV90200/
```

Note: -i is just support one include, designer can not declare multiple -i option in this version.

If designer wants to build a set of test pattern in a directory, he needs to declare -r <input_option> to specify this and the input argument is like example as below:

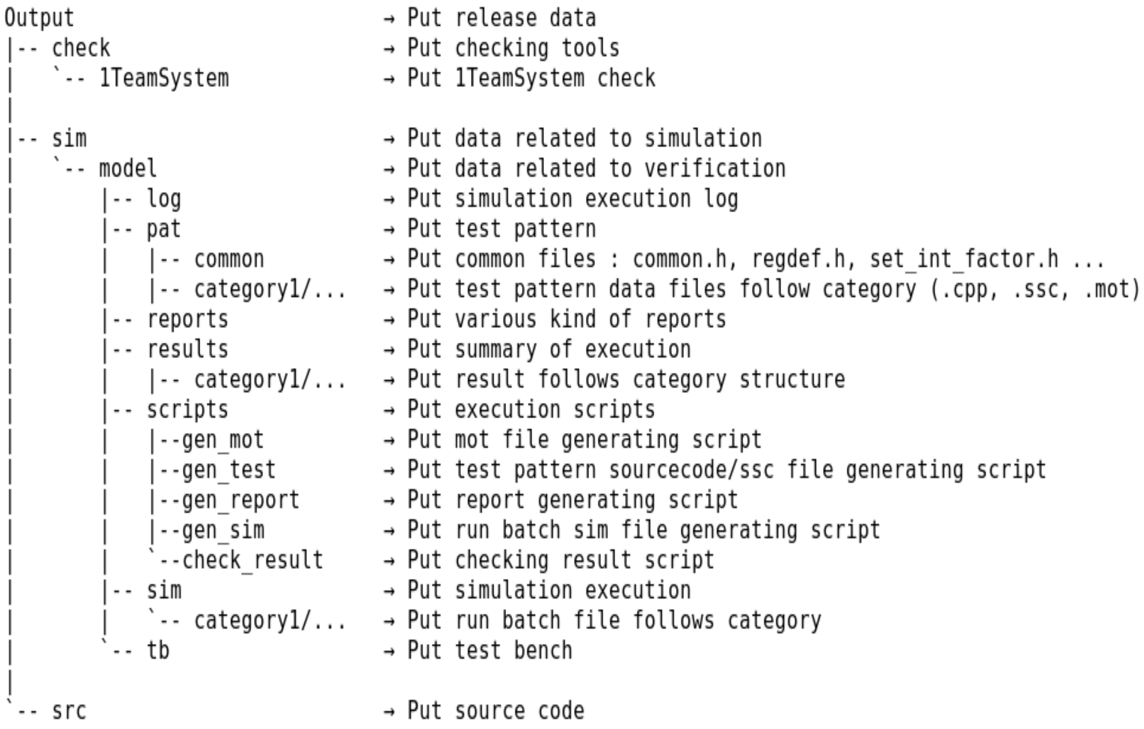
```
perl gen_mot.pl -r /common/work/input_test/  -t /common/appl/Renesas/shc/SHCV90200/
```

In the above example designer wants to build all test pattern in /common/work/test directory and puts all output mot file in the same directory with input (input_test).

4. Appendix

4.1.Standard directory structure for model environment

The new folder structure is based on IP folder structure which is use commonly among projects :



```
Output                        → Put release data
|-- check                     → Put checking tools
|   `-- 1TeamSystem           → Put 1TeamSystem check
|
|-- sim                       → Put data related to simulation
|   `-- model                 → Put data related to verification
|       |-- log               → Put simulation execution log
|       |-- pat               → Put test pattern
|       |   |-- common        → Put common files : common.h, regdef.h, set_int_factor.h ...
|       |   |-- category1/... → Put test pattern data files follow category (.cpp, .ssc, .mot)
|       |-- reports           → Put various kind of reports
|       |-- results           → Put summary of execution
|       |   |-- category1/... → Put result follows category structure
|       |-- scripts           → Put execution scripts
|       |   |--gen_mot        → Put mot file generating script
|       |   |--gen_test       → Put test pattern sourcecode/ssc file generating script
|       |   |--gen_report     → Put report generating script
|       |   |--gen_sim        → Put run batch sim file generating script
|       |   `--check_result   → Put checking result script
|       |-- sim               → Put simulation execution
|       |   `-- category1/... → Put run batch file follows category
|       `-- tb                → Put test bench
|
`-- src                       → Put source code
```

Figure 4.1: Model development folder structure

Explanation:

- All released data are put in **[Output]**.
- Environment now is stored under **[Output/sim/model/tb]** directory
- Test patterns are stored under **[Output/sim/model/pat]** directory, test patterns are created follow category folder structure. In each sub **[category]** folder, there are three sub folders **[src]**, **[ssc]**, **[mot]** to store source file, ssc file and mot file respectively. Included files (interrupt included file, common.h, regdef.h) are stored in **[.../pat/common]** folder. **[pat]** folder also stores designer's additional scripts.
- Work directories are organized follows category structure hierarchy, **[Output/sim/model/sim]** will store run all test pattern batch file, each sub category has own run batch file.
- Dump data and check result of test pattern are stored under **[Output/sim/model/results]**.
- Simulation log files are stored under **[Output/sim/model/log]** directory.
- Generated reports are store under **[Output/sim/model/reports]** directory.

4.2.Simulation controlled command file (ssc file)

4.2.1.Standard format

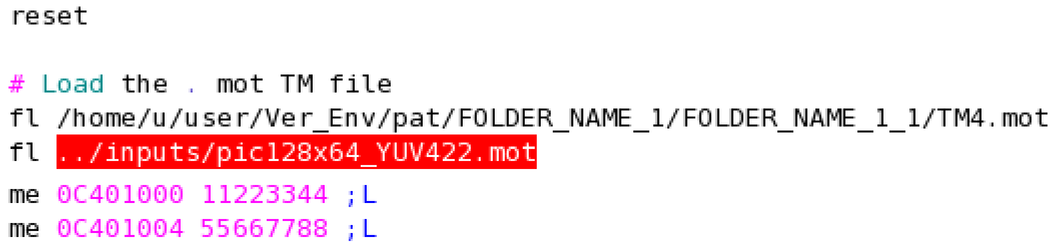
Simulation controlled command file (*.ssc*) is the file using to control simulation progress. ssc file usually has 5 parts :

- Memory initialize setting (MOT file loading)
- Register initialize
- End condition (break points, max instruction setting)
- Start/Run
- Memory dump

For ssc command, please refer document *"CEDAR&CEDAR2_simple_manual_050330.pdf"*

4.2.2.Memory initialize setting

In this part, CPU will load MOT file to memory for initialize setting by using command *"fl".* MOT file can be test pattern file and input or output data for environment such as input picture to verify image processing model. Besides that, designer can also write directly to memory by using *"me"* command.

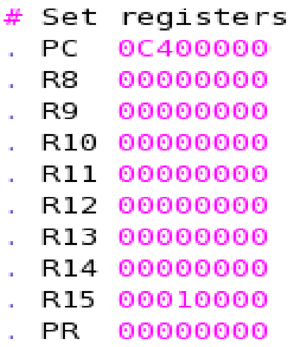


```
reset

# Load the . mot TM file
fl /home/u/user/Ver_Env/pat/FOLDER_NAME_1/FOLDER_NAME_1_1/TM4.mot
fl ../inputs/pic128x64_YUV422.mot
me 0C401000 11223344 ;L
me 0C401004 55667788 ;L
```

Figure 4.2: Example of memory initialization setting

4.2.3.Register Initialize

In this part, designer will initialize value common and CPU registers. Designer should set PC register with address where test pattern will be loaded. In common, this address is 0x0C400000. CPU will start to run program (test pattern) from address which store in PC register.

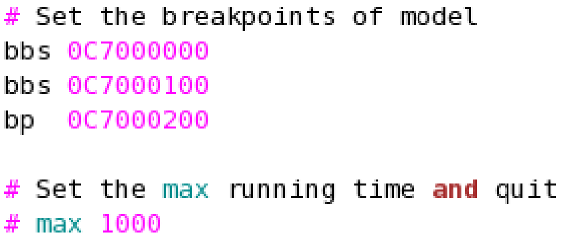


```
# Set registers
. PC   0C400000
. R8   00000000
. R9   00000000
. R10  00000000
. R11  00000000
. R12  00000000
. R13  00000000
. R14  00000000
. R15  00010000
. PR   00000000
```

Figure 4.3: Example of register initialization

4.2.4.End condition

In this part, designer will set the breakpoints, max instruction for CPU. When PC pointer reaches to these addresses, CPU will halt, break or stop simulation according to used commands such as *"bp"*, *"bbs"*.Sometime, designer can't know the break points so that to limit CPU instructions, designer can set max instruction to bound the simulation by using *"max"* or *"maxi"* command.

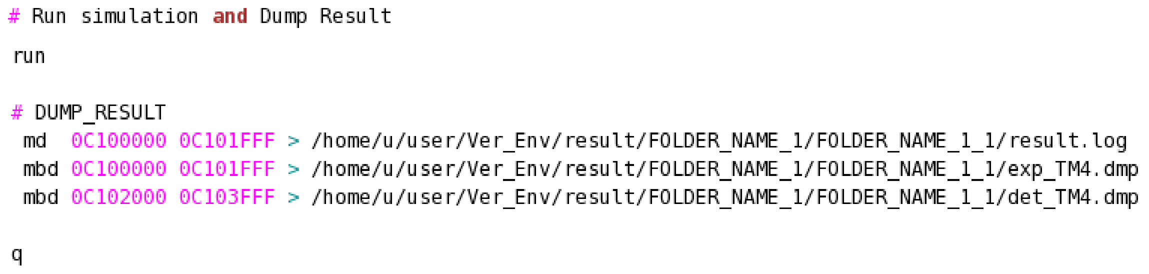


```
# Set the breakpoints of model
bbs 0C7000000
bbs 0C7000100
bp  0C7000200

# Set the max running time and quit
# max 1000
```

Figure 4.4: Example of End condition part

4.2.5.Start/Run and Dump Memory

In this part, designer will permit CPU to start simulating by using *"run"* command. After finishing, CPU will dump memory by using *"md"* or *"mbd"* for checking the result and quit the simulation progress by using *"q"* command.



```
# Run simulation and Dump Result
run

# DUMP_RESULT
 md  0C100000 0C101FFF > /home/u/user/Ver_Env/result/FOLDER_NAME_1/FOLDER_NAME_1_1/result.log
 mbd 0C100000 0C101FFF > /home/u/user/Ver_Env/result/FOLDER_NAME_1/FOLDER_NAME_1_1/exp_TM4.dmp
 mbd 0C102000 0C103FFF > /home/u/user/Ver_Env/result/FOLDER_NAME_1/FOLDER_NAME_1_1/det_TM4.dmp

q
```

Figure 4.5: Example of Start/Run and Dump memory part

4.3.Binary to mot file

Some image processing models need input image data for their processing and this input data are usually stored in memory under mot file format, that is the reason why designer needs to convert his input data from binary file to mot file format for his verification. There is a tool that can help designer about that, this tool is described as below

Type	Content
Script name	bin2mot
Language	C++
Purpose	Convert file from binary to mot format
Input file	Binary file
Output file	Mot file
Usage	*bin2mot* **<input_file> <ram_addr> [size]**
Argument	**<input_file>** : input binary file **<ram_addr>** : specified address in memory that needs to load **[size]** : specify the number of byte that needs to convert in input file. This is optional argument, if it is omitted, the converted size = input file size.

Table 4.1: bin2mot tool description

Example:

```
bin2mot image_01.dat 0C500000 1024 > image_01.mot
```

4.4.Output message from test pattern

In some case, designer wants to output debug messages into terminal from test pattern for monitoring execution process. There are several ways to solve this problem : by using a debug module or by using SCIF module. This guide focuses on the former method that uses debug module to output debug message and operation is shown as below.

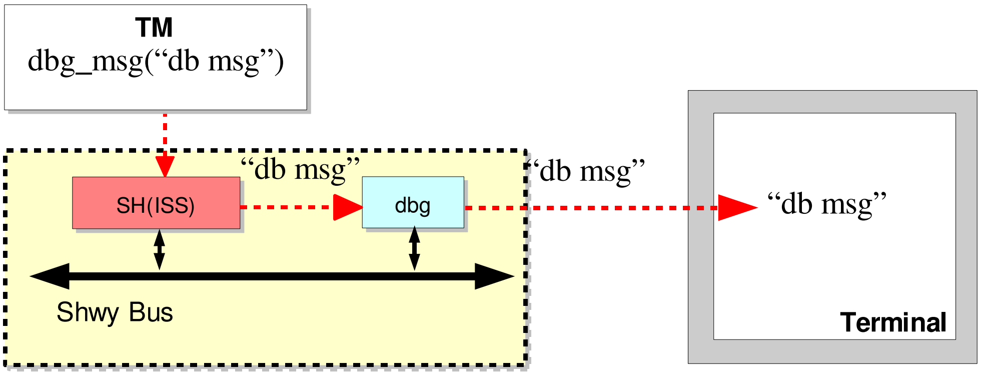
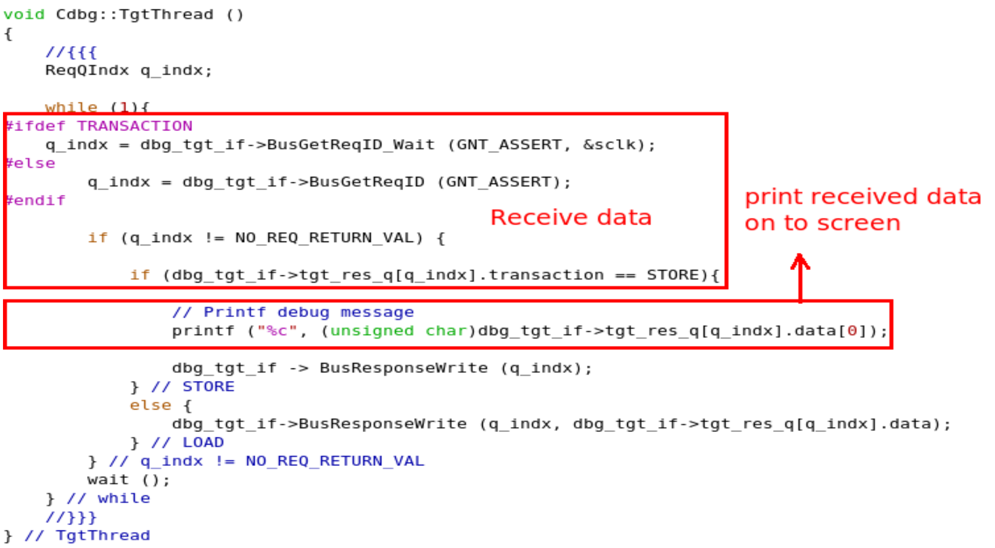


Figure 4.6: dbg module operation flow

Explanation: Test pattern call **dgb_msg** function with input string “db msg”, this data string will be sent to debug module by CPU via Shwy Bus. Debug module calls “printf” function with received data string “db msg” to display this debug message on the screen of terminal.

Debug module is an simple module that plays a role as a target in Shwy Bus. Instead of receiving data from Shwy Bus and write this data to internal registers like other target modules, debug module will print this received data to terminal by using “printf” function. The implementation of this module can be described simply by declaring a thread that receives data from Shwy Bus and print this data onto the screen as below:



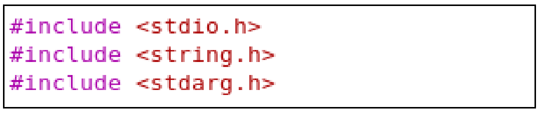
```
void Cdbg::TgtThread ()
{
    //{{{
    ReqQIndx q_indx;

    while (1){
#ifdef TRANSACTION
    q_indx = dbg_tgt_if->BusGetReqID_Wait (GNT_ASSERT, &sclk);
#else
        q_indx = dbg_tgt_if->BusGetReqID (GNT_ASSERT);
#endif

        if (q_indx != NO_REQ_RETURN_VAL) {

            if (dbg_tgt_if->tgt_res_q[q_indx].transaction == STORE){
                // Printf debug message
                printf ("%c", (unsigned char)dbg_tgt_if->tgt_res_q[q_indx].data[0]);

                dbg_tgt_if -> BusResponseWrite (q_indx);
            } // STORE
            else {
                dbg_tgt_if->BusResponseWrite (q_indx, dbg_tgt_if->tgt_res_q[q_indx].data);
            } // LOAD
        } // q_indx != NO_REQ_RETURN_VAL
        wait ();
    } // while
    //}}}
} // TgtThread
```

Figure 4.7: Debug module target thread source code

Designer needs to create and connect this debug module to his verification environment via Shwy Bus.

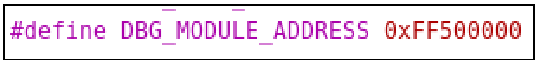
Next step, designer needs to implement the **dbg_msg** function in his test pattern follows below steps

- Include more library for dbg_msg usage



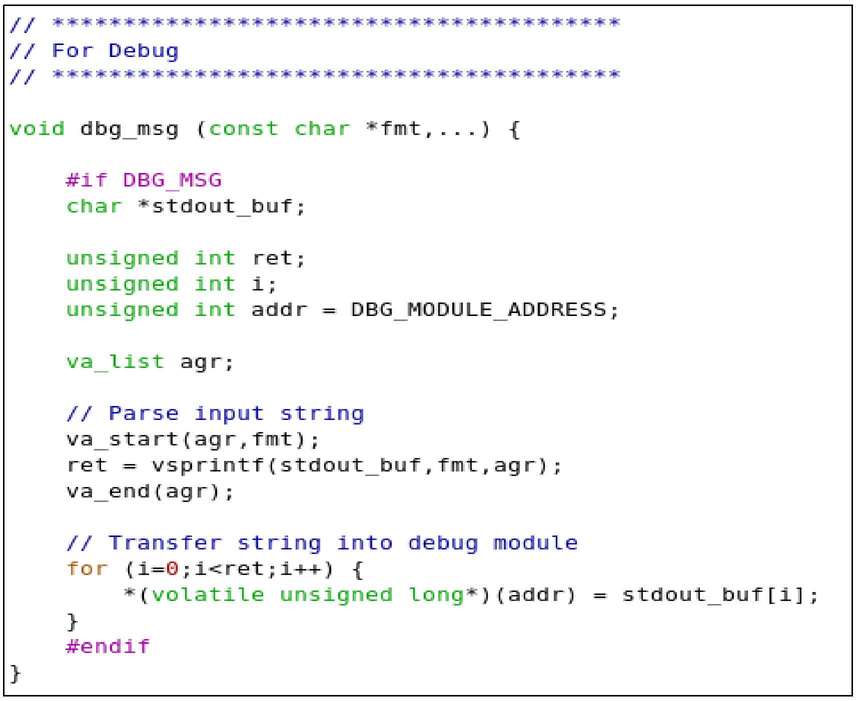
```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
```

- Define debug message module address, for ex: debug module address is FF500000



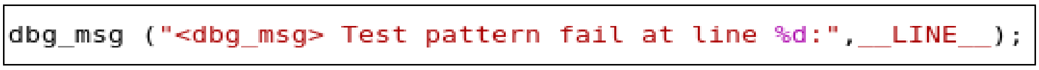
```
#define DBG_MODULE_ADDRESS 0xFF500000
```

- Implement **dbg_msg** function



```
// ****************************************
// For Debug
// ****************************************

void dbg_msg (const char *fmt,...) {

    #if DBG_MSG
    char *stdout_buf;

    unsigned int ret;
    unsigned int i;
    unsigned int addr = DBG_MODULE_ADDRESS;

    va_list agr;

    // Parse input string
    va_start(agr,fmt);
    ret = vsprintf(stdout_buf,fmt,agr);
    va_end(agr);

    // Transfer string into debug module
    for (i=0;i<ret;i++) {
        *(volatile unsigned long*)(addr) = stdout_buf[i];
    }
    #endif
}
```

Whenever designer wants to output debug message, he can use dbg_msg function in his test pattern like example as below:



```
dbg_msg ("<dbg_msg> Test pattern fail at line %d:",__LINE__);
```

- To use C standard library with SH cpu, user must recompile SH library by using SH compiler tool. Following example illustrates standard library compilation with SH compiler :

```
lbgsh -cpu=sh4a -fpu=single -output=sh4a.lib -endian=little -head=stdlib
```

For the detail information, please refer in *ESHCUM4B.PDF* document.

4.5. Interrupt routine implementation in test pattern

Designer can implement interrupt routine in test pattern in several ways. However, this document has small ambition to introduce one common way which is used with Forest environment. The following table describe related files for implementing interrupt in Forest environment.

No	File	Description
1	intc.h	Register verified model and its interrupt factor with INTC.
2	intc.cpp	Register detail information of each interrupt factors.
3	forest.h	Connect verified model with INTC if verified model use Shwy bus.
4	hpbc.h	Connect verified model with INTC if verified model use HPB bus only.
5	set_intc_factor.h	Register verified model and its interrupt factors.
6	set_intc_factor.cpp	Implement INTC setting for each interrupt factors.
7	intc_handler.h	Register interrupt routine (function) for each interrupt factor.
8	intc_handler.src	Register interrupt routine callback and even code when interrupt happen
9	common.h	Implement content of each interrupt routine.

Table 4.2: Related files for using interrupt in text pattern

For Forest environment, design must do following step :

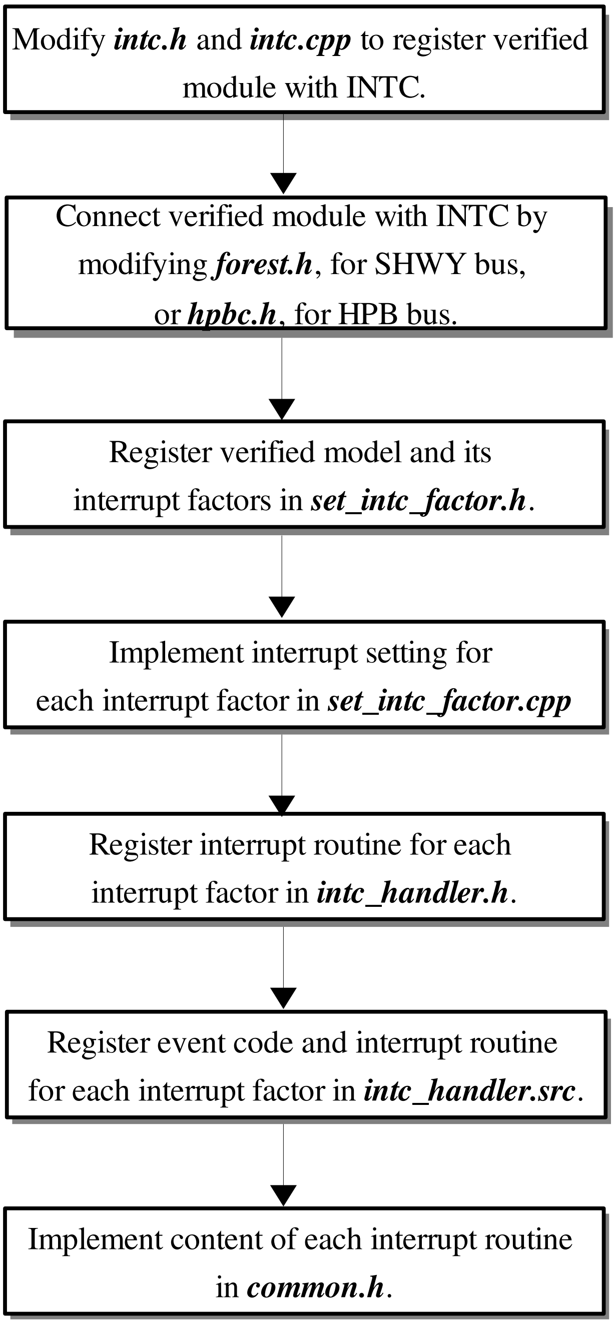
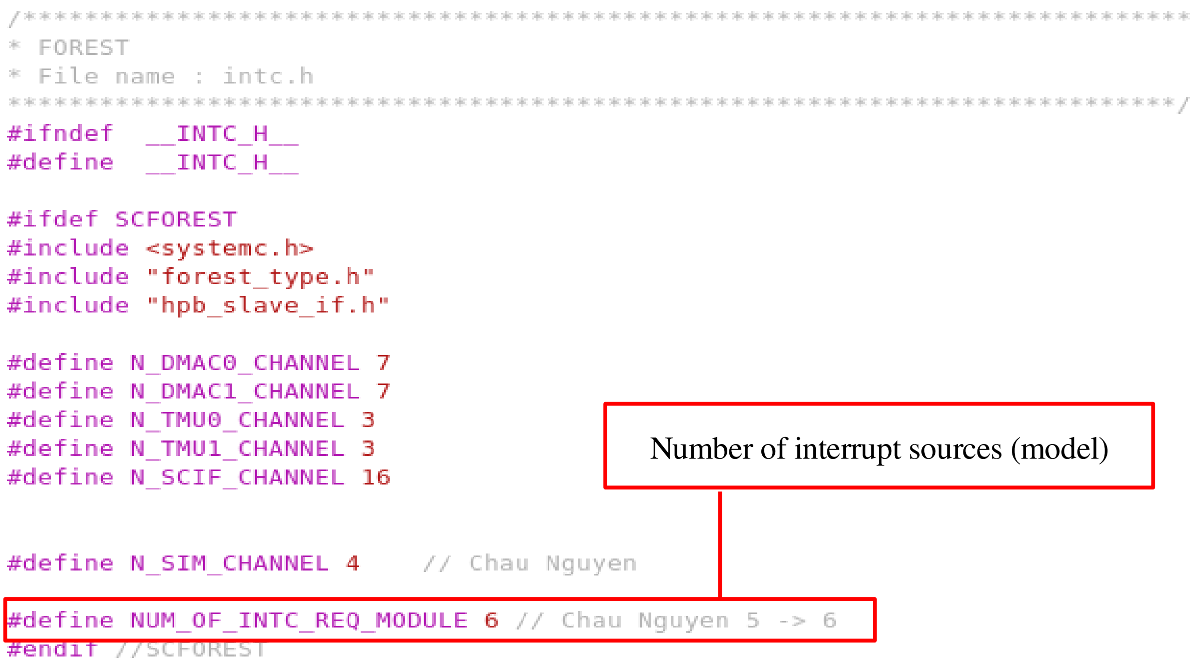

Modify ***intc.h*** and ***intc.cpp*** to register verified module with INTC.
Connect verified module with INTC by modifying ***forest.h***, for SHWY bus, or ***hpbc.h***, for HPB bus.
Register verified model and its interrupt factors in ***set_intc_factor.h***.
Implement interrupt setting for each interrupt factor in ***set_intc_factor.cpp***
Register interrupt routine for each interrupt factor in ***intc_handler.h***.
Register event code and interrupt routine for each interrupt factor in ***intc_handler.src***.
Implement content of each interrupt routine in ***common.h***.

Figure 4.8: Interrupt implementation steps

Explanation :

To implement test pattern with interrupt, designer must register interrupt information by using ***intc.h*** and ***intc.cpp***.

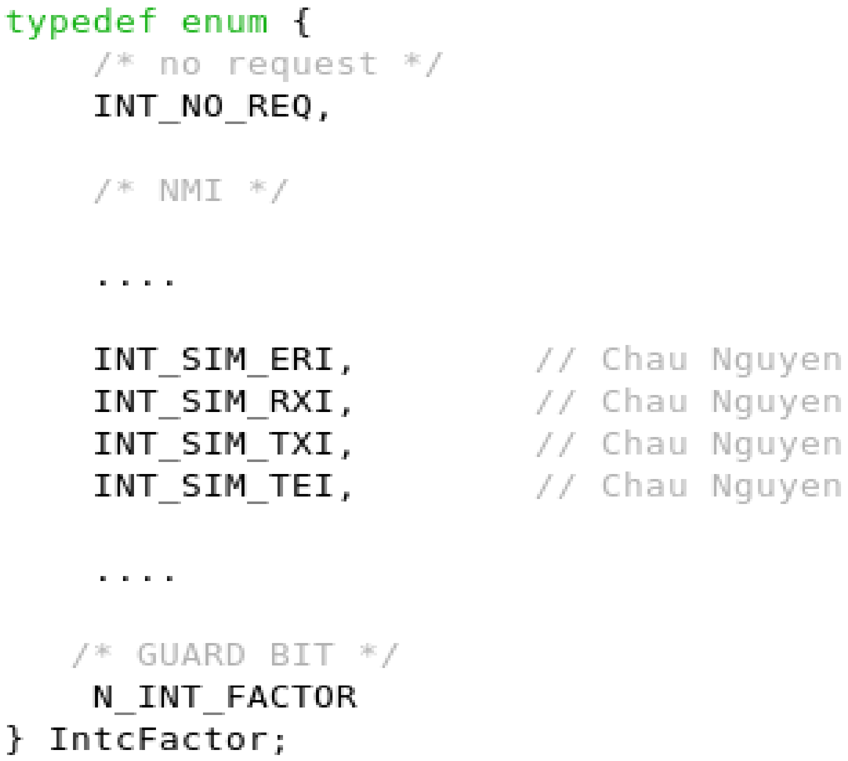
For ***intc.h***, designer will inform INTC how many interrupt source (model) INTC must control and total interrupt factors of each interrupt source. Moreover, ***intc.h*** also help designer register order and interrupt table that contains detail information about each interrupt factors. Designer should notice that order of each interrupt factor in ***intc.h*** must consistent with order of information of each interrupt factor in ***intc.cpp.*** If order is not consistence, INTC will operate malfunction as a consequence of getting wrong information about interrupt factor.



```
/***************************************************************************
* FOREST
* File name : intc.h
***************************************************************************/
#ifndef   __INTC_H__
#define   __INTC_H__

#ifdef SCFOREST
#include <systemc.h>
#include "forest_type.h"
#include "hpb_slave_if.h"

#define N_DMAC0_CHANNEL 7
#define N_DMAC1_CHANNEL 7
#define N_TMU0_CHANNEL 3
#define N_TMU1_CHANNEL 3
#define N_SCIF_CHANNEL 16


#define N_SIM_CHANNEL 4     // Chau Nguyen

#define NUM_OF_INTC_REQ_MODULE 6 // Chau Nguyen 5 -> 6
#endif //SCFOREST
```

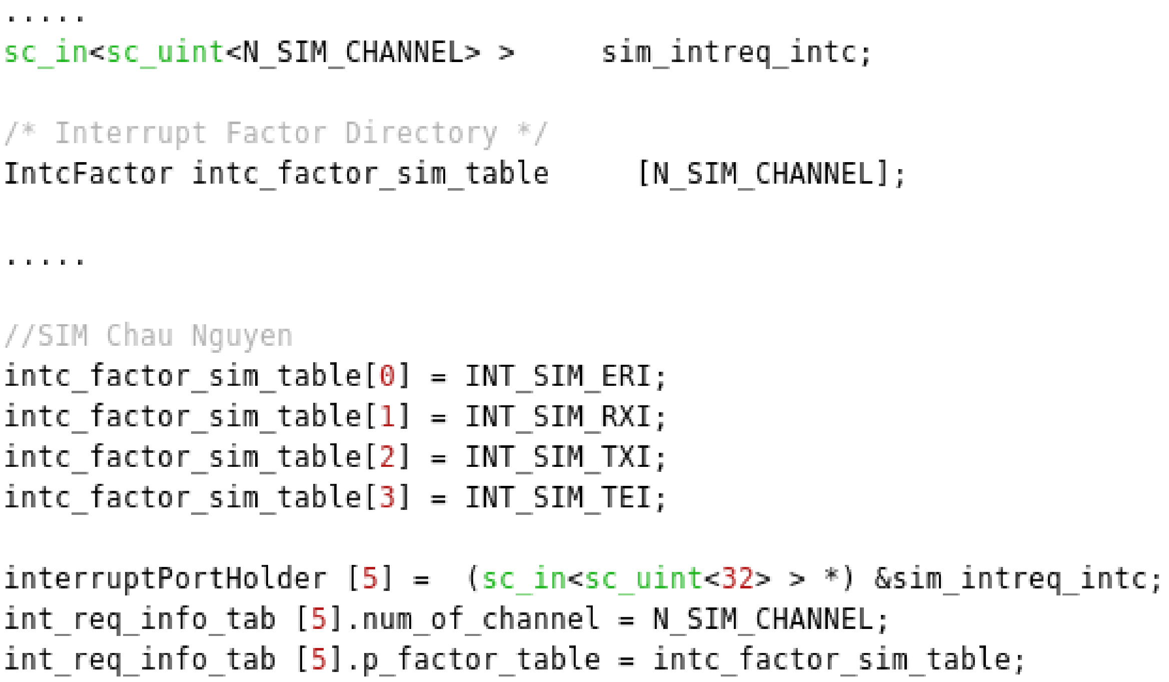
Number of interrupt sources (model)

Figure 4.9: intc.h : Total interrupt sources



```
typedef enum {
    /* no request */
    INT_NO_REQ,

    /* NMI */

    ....

    INT_SIM_ERI,        // Chau Nguyen
    INT_SIM_RXI,        // Chau Nguyen
    INT_SIM_TXI,        // Chau Nguyen
    INT_SIM_TEI,        // Chau Nguyen

    ....

   /* GUARD BIT */
    N_INT_FACTOR
} IntcFactor;
```

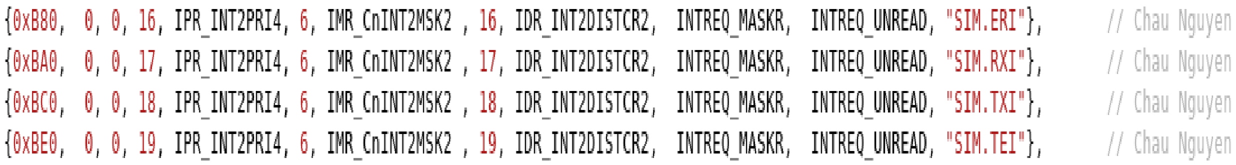
Figure 4.10: intc.h : Interrupt factor order registration



```
.....
sc_in<sc_uint<N_SIM_CHANNEL> >     sim_intreq_intc;

/* Interrupt Factor Directory */
IntcFactor intc_factor_sim_table     [N_SIM_CHANNEL];

.....

//SIM Chau Nguyen
intc_factor_sim_table[0] = INT_SIM_ERI;
intc_factor_sim_table[1] = INT_SIM_RXI;
intc_factor_sim_table[2] = INT_SIM_TXI;
intc_factor_sim_table[3] = INT_SIM_TEI;

interruptPortHolder [5] =  (sc_in<sc_uint<32> > *) &sim_intreq_intc;
int_req_info_tab [5].num_of_channel = N_SIM_CHANNEL;
int_req_info_tab [5].p_factor_table = intc_factor_sim_table;
```

Figure 4.11: intc.h : Interrupt factor information registration

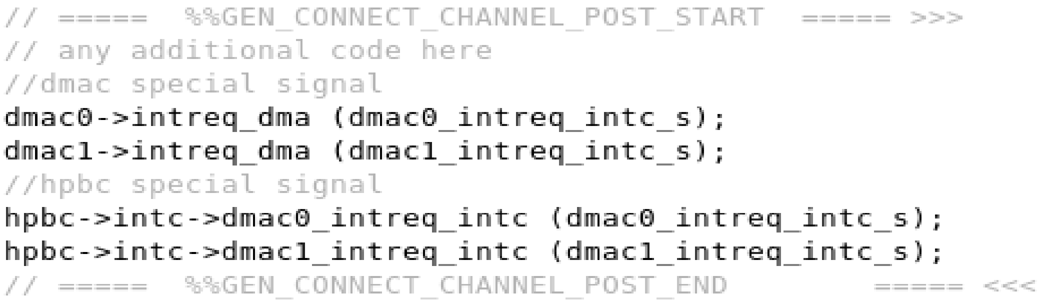
For ***intc.cpp***, designer will register detail information for each interrupt factor. Some notable information such as : event code, priority position, mask bit, clear bit, CPU distribution information ....



```
{0xB80,  0, 0, 16, IPR_INT2PRI4, 6, IMR_CnINT2MSK2 , 16, IDR_INT2DISTCR2,  INTREQ_MASKR,  INTREQ_UNREAD, "SIM.ERI"},      // Chau Nguyen
{0xBA0,  0, 0, 17, IPR_INT2PRI4, 6, IMR_CnINT2MSK2 , 17, IDR_INT2DISTCR2,  INTREQ_MASKR,  INTREQ_UNREAD, "SIM.RXI"},      // Chau Nguyen
{0xBC0,  0, 0, 18, IPR_INT2PRI4, 6, IMR_CnINT2MSK2 , 18, IDR_INT2DISTCR2,  INTREQ_MASKR,  INTREQ_UNREAD, "SIM.TXI"},      // Chau Nguyen
{0xBE0,  0, 0, 19, IPR_INT2PRI4, 6, IMR_CnINT2MSK2 , 19, IDR_INT2DISTCR2,  INTREQ_MASKR,  INTREQ_UNREAD, "SIM.TEI"},      // Chau Nguyen
```

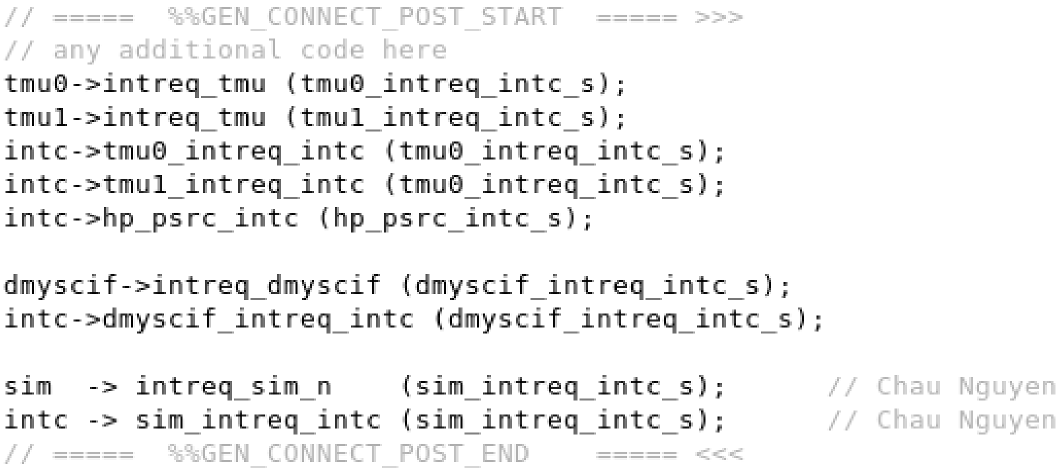
Figure 4.12: intc.cpp : Interrupt factor's detail information

To connect between verified model and INTC, designer must use ***forest.h*** or ***hpbc.h*** file according to which bus model uses.



```
// =====  %%GEN_CONNECT_CHANNEL_POST_START  ===== >>>
// any additional code here
//dmac special signal
dmac0->intreq_dma (dmac0_intreq_intc_s);
dmac1->intreq_dma (dmac1_intreq_intc_s);
//hpbc special signal
hpbc->intc->dmac0_intreq_intc (dmac0_intreq_intc_s);
hpbc->intc->dmac1_intreq_intc (dmac1_intreq_intc_s);
// =====  %%GEN_CONNECT_CHANNEL_POST_END        ===== <<<
```

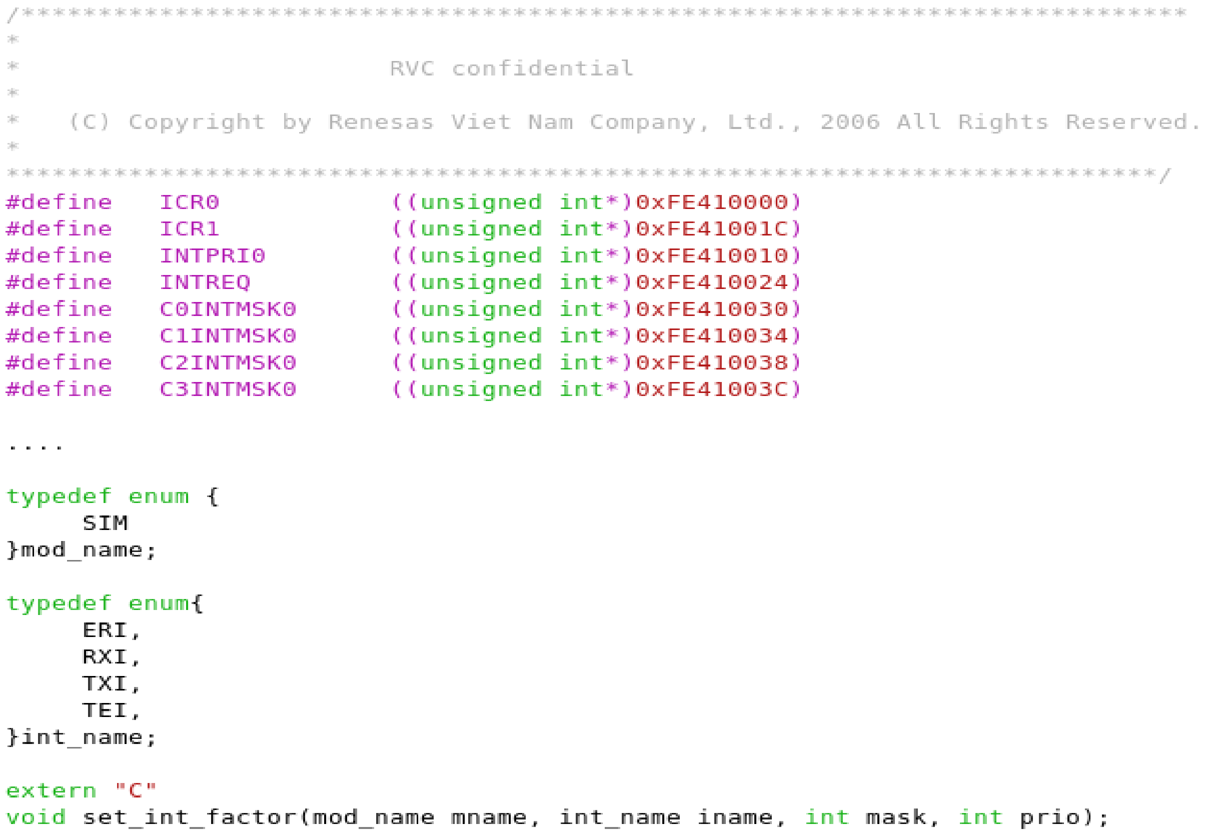
Figure 4.13: forest.h : Interrupt connection for SHwy model



```
// =====  %%GEN_CONNECT_POST_START  ===== >>>
// any additional code here
tmu0->intreq_tmu (tmu0_intreq_intc_s);
tmu1->intreq_tmu (tmu1_intreq_intc_s);
intc->tmu0_intreq_intc (tmu0_intreq_intc_s);
intc->tmu1_intreq_intc (tmu0_intreq_intc_s);
intc->hp_psrc_intc (hp_psrc_intc_s);

dmyscif->intreq_dmyscif (dmyscif_intreq_intc_s);
intc->dmyscif_intreq_intc (dmyscif_intreq_intc_s);

sim  -> intreq_sim_n    (sim_intreq_intc_s);      // Chau Nguyen
intc -> sim_intreq_intc (sim_intreq_intc_s);      // Chau Nguyen
// =====  %%GEN_CONNECT_POST_END    ===== <<<
```

Figure 4.14: hpbc.h : Interrupt connection for HBC model

After finishing registering and connecting verified model, designer can start to write test pattern with interrupt. First, designer must add information about INTC register, name and interrupt factors of verified model in ***set_intc_factor.h***. Unlike registration in INTC, the registration in ***set_intc_factor.h*** does not care about order because all of register information is used to make design more convenient when writing test pattern rather than configuring interrupt information.

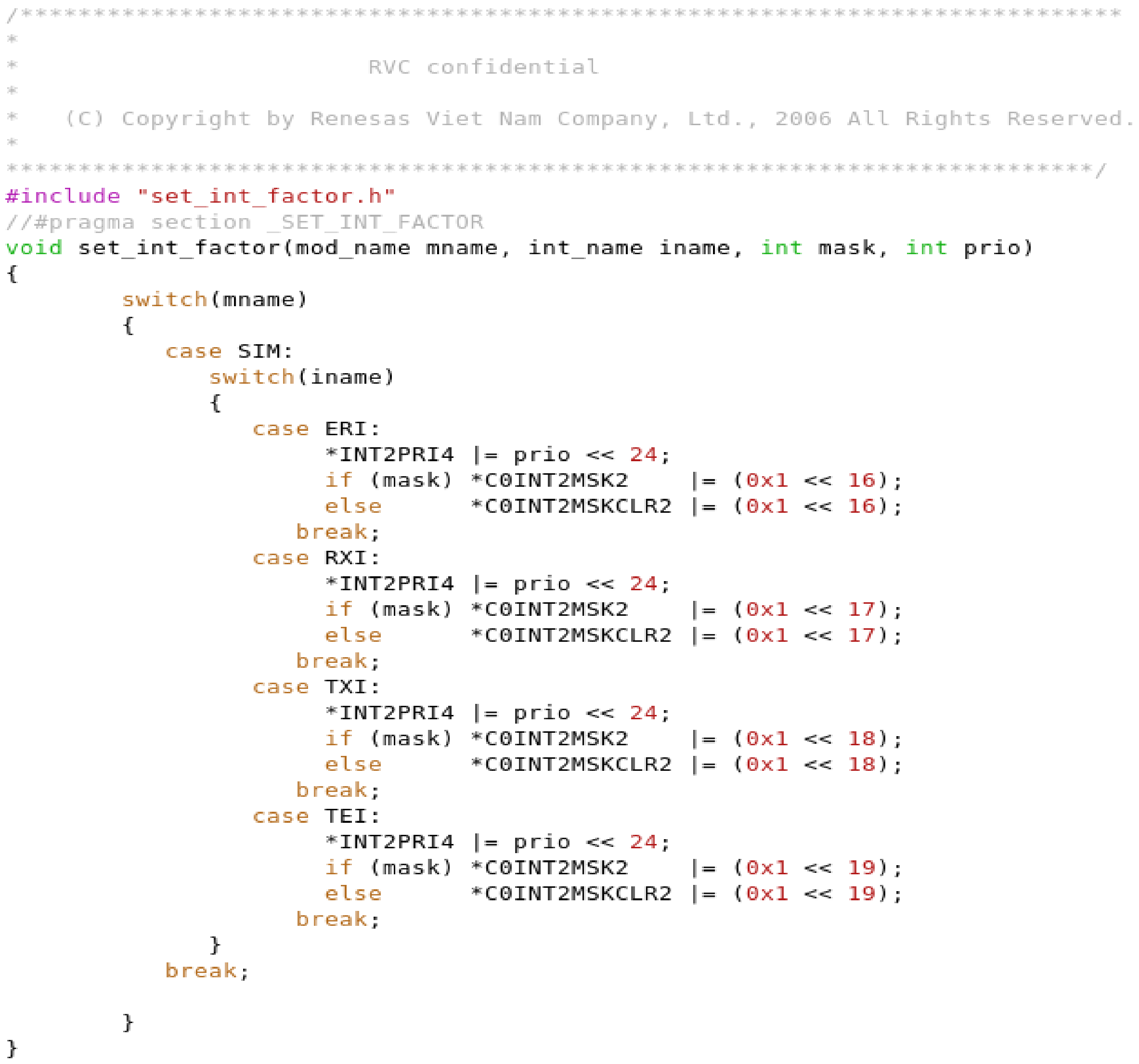


```
/***************************************************************************
*
*                         RVC confidential
*
*    (C) Copyright by Renesas Viet Nam Company, Ltd., 2006 All Rights Reserved.
*
***************************************************************************/
#define   ICR0           ((unsigned int*)0xFE410000)
#define   ICR1           ((unsigned int*)0xFE41001C)
#define   INTPRI0        ((unsigned int*)0xFE410010)
#define   INTREQ         ((unsigned int*)0xFE410024)
#define   C0INTMSK0      ((unsigned int*)0xFE410030)
#define   C1INTMSK0      ((unsigned int*)0xFE410034)
#define   C2INTMSK0      ((unsigned int*)0xFE410038)
#define   C3INTMSK0      ((unsigned int*)0xFE41003C)

....

typedef enum {
     SIM
}mod_name;

typedef enum{
     ERI,
     RXI,
     TXI,
     TEI,
}int_name;

extern "C"
void set_int_factor(mod_name mname, int_name iname, int mask, int prio);
```

Figure 4.15: set_intc_factor.h : Interrupt register information

Second, designer must setting interrupt register for each interrupt factor in ***set_intc_factor.cpp***

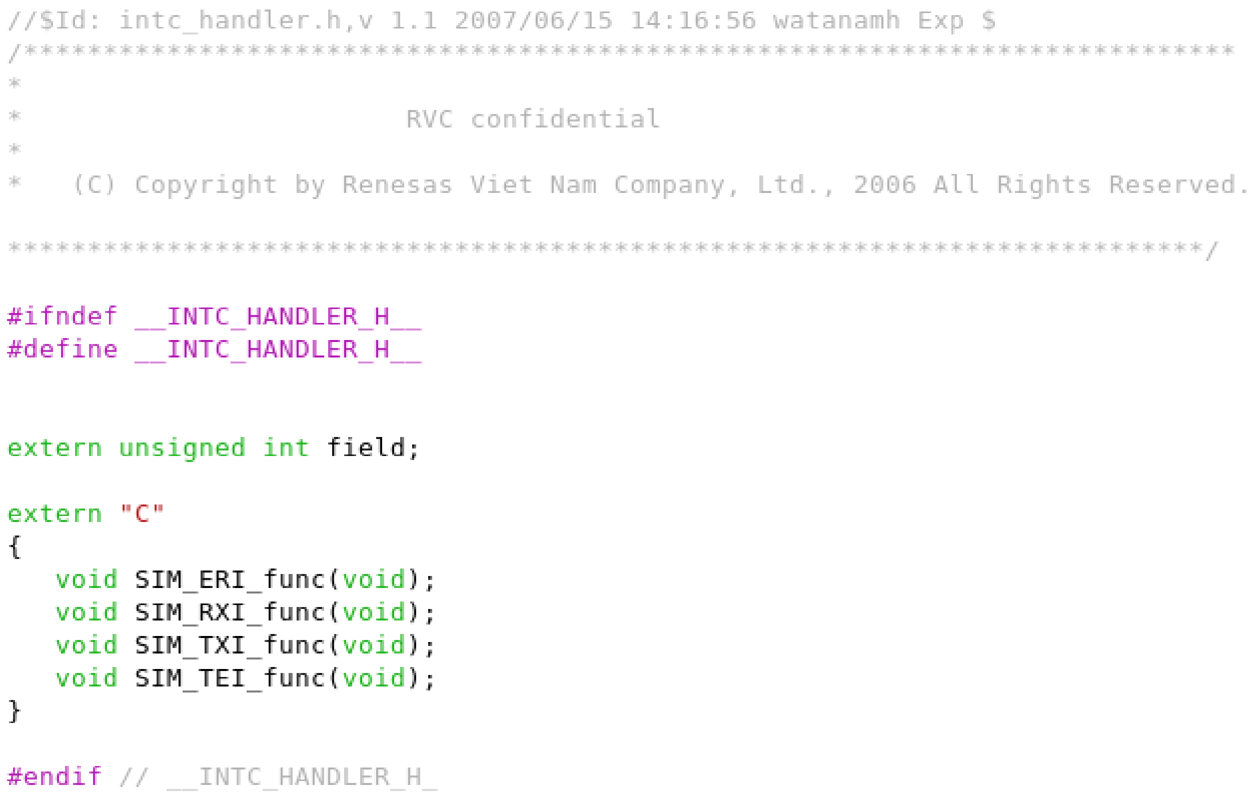
Some notable settings are clear, mask, priority .... Designer must be attention to one thing that Forest is highly customize environment, sometime interrupt information is not correct according to specification, therefore to use interrupt, designer must configure or setting interrupt registers according to interrupt factor information in intc.cpp. Example : INTC in Forest is INTC of RP1, all its register information and peripheral models allow to RP1 specification, sometime to verify one peripheral model from new product, designer must remove some exist models in Forest to get space for verified model's connection. The remove normally is done manually, information of each interrupt factors must be set corresponding to environment. The setting lead verified model have wrong information according to specification however the INTC in Forest still work well with modification information.



```
/****************************************************************************
*
*                        RVC confidential
*
*   (C) Copyright by Renesas Viet Nam Company, Ltd., 2006 All Rights Reserved.
*
***************************************************************************/
#include "set_int_factor.h"
//#pragma section _SET_INT_FACTOR
void set_int_factor(mod_name mname, int_name iname, int mask, int prio)
{
        switch(mname)
        {
           case SIM:
              switch(iname)
              {
                 case ERI:
                      *INT2PRI4 |= prio << 24;
                      if (mask) *C0INT2MSK2    |= (0x1 << 16);
                      else      *C0INT2MSKCLR2 |= (0x1 << 16);
                    break;
                 case RXI:
                      *INT2PRI4 |= prio << 24;
                      if (mask) *C0INT2MSK2    |= (0x1 << 17);
                      else      *C0INT2MSKCLR2 |= (0x1 << 17);
                    break;
                 case TXI:
                      *INT2PRI4 |= prio << 24;
                      if (mask) *C0INT2MSK2    |= (0x1 << 18);
                      else      *C0INT2MSKCLR2 |= (0x1 << 18);
                    break;
                 case TEI:
                      *INT2PRI4 |= prio << 24;
                      if (mask) *C0INT2MSK2    |= (0x1 << 19);
                      else      *C0INT2MSKCLR2 |= (0x1 << 19);
                    break;
              }
           break;

        }
}
```

Figure 4.16: set_intc_factor.cpp : Register configuration of interrupt factor

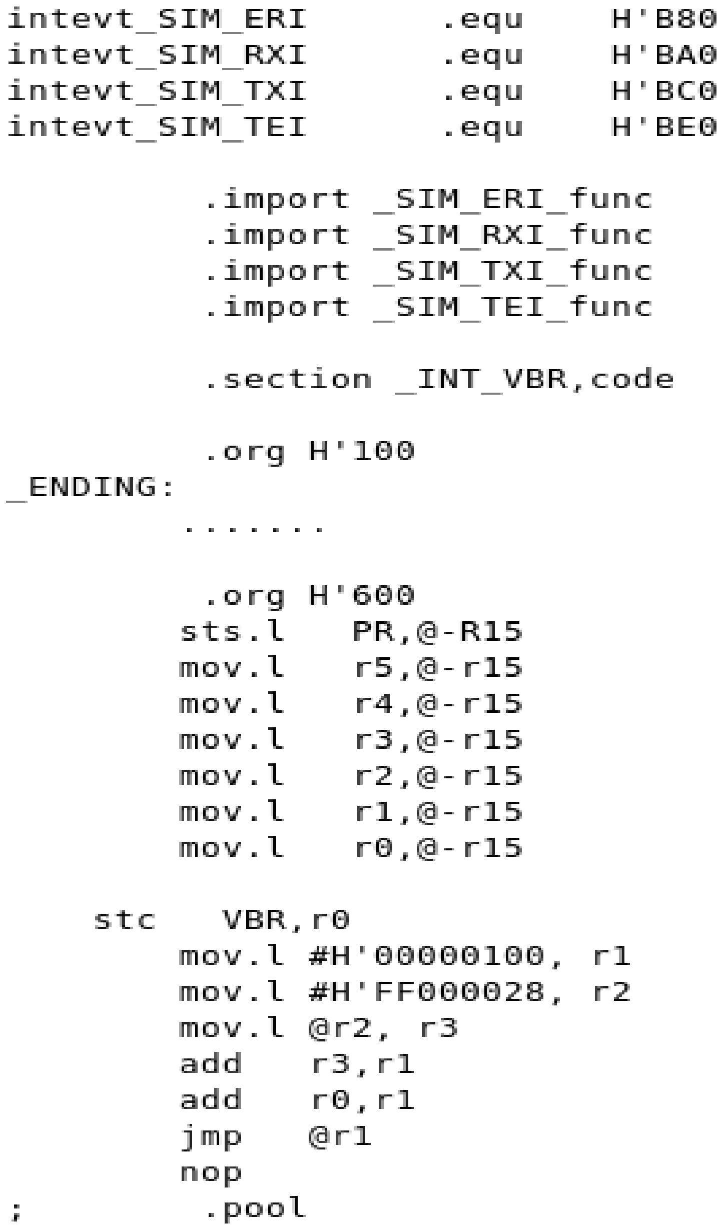
Third, designer must declare interrupt routine for each interrupt factor in ***intc_handle.h***. When interrupt occur, CPU will jump into interrupt routine to operate next statement. The content of interrupt can be implemented in ***intc_handle.h***, however, designer should do so in ***common.h*** because to write effective test pattern, sometime designer will use some common functions which are declare in ***common.h***. Thus, to avoid error during compilation, designer should implement interrupt routine functions in ***common.h*** file.



```
//$Id: intc_handler.h,v 1.1 2007/06/15 14:16:56 watanamh Exp $
/****************************************************************************
*
*                        RVC confidential
*
*   (C) Copyright by Renesas Viet Nam Company, Ltd., 2006 All Rights Reserved.

***************************************************************************/

#ifndef __INTC_HANDLER_H__
#define __INTC_HANDLER_H__


extern unsigned int field;

extern "C"
{
   void SIM_ERI_func(void);
   void SIM_RXI_func(void);
   void SIM_TXI_func(void);
   void SIM_TEI_func(void);
}

#endif // __INTC_HANDLER_H_
```

Figure 4.17: intc_handler.h : Interrupt routine declaration

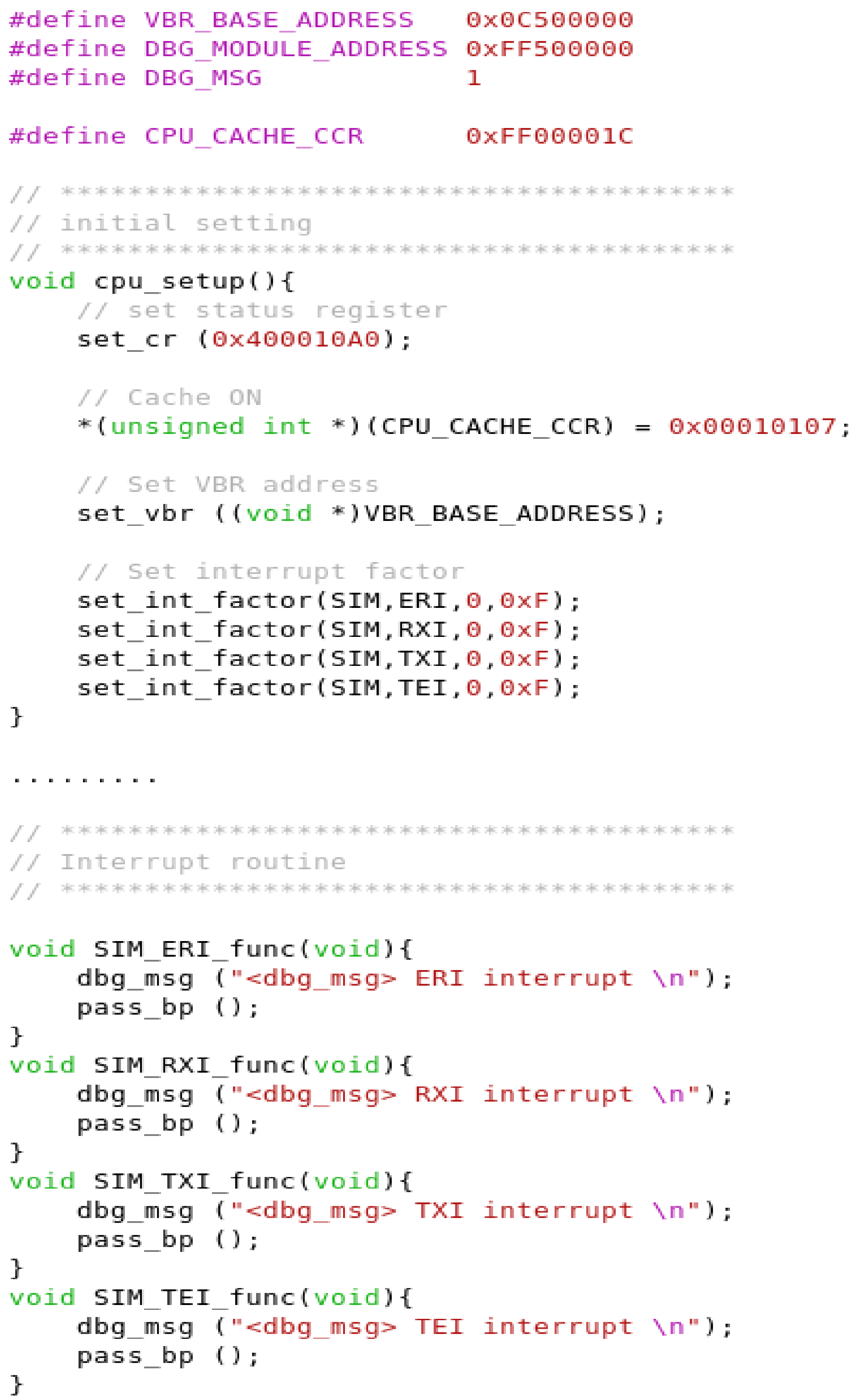
After declaring interrupt routine functions, designer must register them which even code of each interrupt factor in ***intc_handler.src.*** One small attention, this file is written by ASM language rather than C language.



```
intevt_SIM_ERI      .equ    H'B80
intevt_SIM_RXI      .equ    H'BA0
intevt_SIM_TXI      .equ    H'BC0
intevt_SIM_TEI      .equ    H'BE0

         .import _SIM_ERI_func
         .import _SIM_RXI_func
         .import _SIM_TXI_func
         .import _SIM_TEI_func

         .section _INT_VBR,code

         .org H'100
_ENDING:
        .......

         .org H'600
        sts.l   PR,@-R15
        mov.l   r5,@-r15
        mov.l   r4,@-r15
        mov.l   r3,@-r15
        mov.l   r2,@-r15
        mov.l   r1,@-r15
        mov.l   r0,@-r15

    stc    VBR,r0
        mov.l #H'00000100, r1
        mov.l #H'FF000028, r2
        mov.l @r2, r3
        add   r3,r1
        add   r0,r1
        jmp   @r1
        nop
;        .pool
```

Figure 4.18: intc_handler.src : Register interrupt routine with even code

Finally, designer must call interrupt setting register function (set_intc_factor) for each interrupt factor, set VBR (set_vbr) and implement content for each interrupt routine in ***common.h*** file.



```
#define VBR_BASE_ADDRESS   0x0C500000
#define DBG_MODULE_ADDRESS 0xFF500000
#define DBG_MSG            1

#define CPU_CACHE_CCR      0xFF00001C

// ****************************************
// initial setting
// ****************************************
void cpu_setup(){
    // set status register
    set_cr (0x400010A0);

    // Cache ON
    *(unsigned int *)(CPU_CACHE_CCR) = 0x00010107;

    // Set VBR address
    set_vbr ((void *)VBR_BASE_ADDRESS);

    // Set interrupt factor
    set_int_factor(SIM,ERI,0,0xF);
    set_int_factor(SIM,RXI,0,0xF);
    set_int_factor(SIM,TXI,0,0xF);
    set_int_factor(SIM,TEI,0,0xF);
}

.........

// ****************************************
// Interrupt routine
// ****************************************

void SIM_ERI_func(void){
    dbg_msg ("<dbg_msg> ERI interrupt \n");
    pass_bp ();
}
void SIM_RXI_func(void){
    dbg_msg ("<dbg_msg> RXI interrupt \n");
    pass_bp ();
}
void SIM_TXI_func(void){
    dbg_msg ("<dbg_msg> TXI interrupt \n");
    pass_bp ();
}
void SIM_TEI_func(void){
    dbg_msg ("<dbg_msg> TEI interrupt \n");
    pass_bp ();
}
```

Figure 4.19: common.h : Implementation for test pattern use interrupt

Revision History

Rev.No	Contents	Approval	Checked	Created
1.0	Create new.			Son Tran,Vu Pham,Chau Nguyen 03/27/09
1.1	- Fix figure 1.1 : Built -> Compile – Add more information about when designer fill table in sheet number 2 – Add more information about “How to check” in sheet number 3 – Add more information about when sheet 4 “Detail execution” is updated. – Add more information about “Test pattern execution table“ – Add legend for picture 2.11 to explain briefly about rest_bp function – Add Id tag in figure 3.3,3.5,3.7.,3.10 – Add set_vbr in figure 3.5 and 3.7 – Remove information about the support scrips in figure 4.1 – Add 4.3 Binary to mot file appendix – Add mode detail explanation for figure 2.9 – Add 4.4 Output debug message		Chau Nguyen 04/09/09	Chau Nguyen Vu Pham Son Tran 04/02/2009
1.2	– Add chapter 4.5 test pattern with interrupt			Chau Nguyen 04/23/2009
1.3	– Modify Figure 1.1 : Add Generate Environment part – Update table 3.1 : Update Option of gen_tm.pl – Update 3.1.5.1 : Add explanation about multi sheet and extension of test pattern name. – Updated 3.1.5.2 : Add explanation of generated part by user script. – Add 3.1.5.3 : Explanation about user_script.pm – Update 3.1.5.4 : remove %%MOT_PATH – Update 3.1.8.1, Figure 3.19 and add 3.1.8.2 ; Explanation about generated part by user script.		Chau Nguyen 10/06/2009	Son Tran 09/22/2009
1.4	– Add version of Perl – Modify Figure 1.1 : Change name of reference document of Generate Environment phase. – Add relative documents	A.Imoto 10/14/09	Chau Nguyen 10/13/2009	Son Tran 10/07/2009