Renesas Synergy™ Platform

S3 Series MCU Diagnostic Software User Guide

r11an0188eu0130
Rev. 1.3
Sep 27, 2018

Introduction

IEC 61508 is an international standard governing a range of electrical, electromechanical and electronic safety related systems. It defines the requirements needed to ensure that systems are designed, implemented, operated and maintained at the required Safety Integrity Level (SIL). Four SIL levels have been defined to indicate the risks involved in any particular system, with SIL4 being the highest risk level.

At the heart of the majority of safety related systems nowadays is a sophisticated and often highly integrated Microcontroller (MCU). An integral part of meeting the requirements of IEC61508 is the ability to verify the correct operation of critical areas of the MCU.

The Renesas Diagnostics Software is designed for use with the Synergy S3 Microcontroller Family. Tests are provided for coverage of the following critical areas of the MCU's operation: The Central Processing Unit (CPU), the Embedded Flash ROM memory, the Embedded RAM memory, the main clock structure (Main clock oscillator, PLL, MUX generating ICLK), and Vcc power supply.

The code was developed using the functional safety version 8.23.1.17132 of the IAR Embedded Workbench for ARM, which is certified by the TÜV SÜD certification body, and in accordance with IEC61508:2010 for use in safety related applications up to SIL3 level. This is also the systematic capability for the Renesas Diagnostics Software described in this document.

Please note that in the code some pragmas have been added in the shape of comments (e.g. "/*LDRA_INSPECTED 90 S Basic type declaration used. */") which have been used to mark code lines flagged to potentially violate a specific MISRA rule but judged as safe. See Annex C for details about the pragmas inserted.

Target Device

Synergy S3 Series MCU

Contents

1. Common Terminology

This section defines some common terms and acronyms used throughout the document and provides references to other relevant Renesas documentation.

1.1 Acronyms

Table 1.1 Terminology and acronyms

Acronym	Description
CRC	Cyclic Redundancy Check
LUT	Look Up Table
TS	Test Segment
TS_ID	Test Segment Identifier
WD	Watch Dog

2. Compiler Environment

The Diagnostic Software code was developed using the functional safety version 8.23.1.17132 IAR Embedded Workbench for ARM, which is certified by the TÜV SÜD certification body, for use in safety related applications up to level SIL3.

2.1 C Type Implementation

Integer C variables are assumed to be 32-bit implemented. Please, make sure that int type has to be represented in 32-bit format on the target environment.

2.2 IAR Environment Settings

The IAR environment should be set up as specified in Table 2.1.

Table 2.1 IAR project options

ID	Category	Sub-category	Setting description	Comment
1	General Options	Target	• Device := Renesas R7FS3A77C (S3A7) Renesas R7FS3A678 (S3A6) Renesas R7FS3A37A (S3A3) Renesas R7FS3A17C (S3A1) • Floating-point, Size of type ‘double’ := 32bits • Subnormal numbers := Treat as zero • Int, Size of type ‘int‘ := 32bits • Data model := Far	
2	General Options	Library Configuration	• Library := Normal DLIB	
3	General Options	Stack/Heap	• Privileged mode stack size := 0x1000	Consider this setting as typical. The stack size has to be greater than the one specified in the Resources Usage section.
4	C/C++ Compiler	Language1	• Language := C • C dialect := C99 • Language conformance := Standard with IAR extensions	
5	C/C++ Compiler	Language2	• Floating-point semantics := Strict conformance	
6	C/C++ Compiler	Code	• Align functions := 1 no alignment	
7	C/C++ Compiler	Optimizations	• Level := None	
8	Assembler	Language	• User symbols are case sensitive	
13	Linker	Library	• Automatic runtime library selection	
14	Linker	Others Sub-category	For RAM test specific testing see Section 4 For ROM test specific testing See section 5.	
15	Build	For RAM test specific testing see Section 4		

	Actions	For ROM test specific testing See section 5	

3. CPU Software Test

3.1 Test Objectives

The objective of the CPU Software Test is to verify the correct functionality of the CPU by adopting a mainly instruction based diagnosis, with the aim to detect permanent hardware failures of the CPU Core.

All instructions, with the only exceptions being the BKPT, SEV, WFE, WFI and DMB instructions, are used in the CPU core testing scheme.

Please see Document Reference [1] for the complete list of instructions. However, please note the primary aim is not to test individual instructions but to detect a hardware failure of the CPU core.

3.2 Software Structure

The software structure provides for two different levels of functions calls

a. The first level is the user interface function named *coreTest*.
b. The second lower level functions are named *testSegment* that are called by *coreTest*.

The *testSegment* functions execute the actual diagnostic of the core, whilst the *coreTest* allows the user to select and run of one or more of the *testSegment* functions in sequence and to collect the diagnostic results.

Up to 20 *testSegment* functions are available; from *testSegment0* to *testSegment19*. Table 3.1 below provides an overview of the *testSegment* functions.

Two types of *testSegment* functions are defined.

- *testSegment* of type “Fixed”:
  - operand data necessary to stimulate the core and run these functions is embedded in the code.
- *testSegment* of type “LUT”:
  - these functions can be called with different operand data taken from a Look Up Table.

Table 3.1 Test Segment Overview

TS_ID	Function Name	Objective of the Test	Test Segment Type
TS00	testSegment00	Testing of Jump instructions (using control flow)	Fixed
TS01	testSegment01	Logical instructions as AND, EOR, NOT, BIC	Fixed
TS02	testSegment02	Bit-level manipulation and test instructions as REVERSE, TEQ	Fixed
TS03	testSegment03	Floating point multiply instructions	LUT
TS04	testSegment04	Floating point addition/subtractions instructions plus additional floating points conversion instructions as VCVT and VCVTB	LUT
TS05	testSegment05	Floating point division instructions plus additional floating point instruction as VABS, VNEG and VCVT	LUT
TS06	testSegment06	Saturating instructions plus additional floating points conversion instructions as VCVT	Fixed
TS07	testSegment07	CPU Control Registers	Fixed
TS08	testSegment08	Integer multiply instructions using LUT data with MULS. (32bit results)	LUT
TS09	testSegment09	Divide instructions	LUT
TS10	testSegment10	Load and store using GPR only	Fixed
TS11	testSegment11	Floating point normalize and denormalized tests	Fixed
TS12	testSegment12	Load and store using floating point data registers plus floating point read port 0 and 1 tests	Fixed
TS13	testSegment13	Integer multiply using LUT data with UMUL and SMUL instruction. (64bit result)	LUT
TS14	testSegment14	FPU control register plus FPU extension registers and VSUB and conversion instruction	Fixed

TS_ID	Function Name	Objective of the Test	Test Segment Type
TS15	testSegment15	Shift and rotate instructions	Fixed
TS16	testSegment16	Integer addition and subtract instructions	LUT
TS17	testSegment17	Bit field instructions plus internal core register tests	Fixed
TS18	testSegment18	Packing and unpacking instructions	Fixed
TS19	testSegment19	Floating point square root plus internal core register tests.	LUT

Table 3.2 reports the association of the execution progress versus the *testSegment* to be executed and the related data set for LUT *testSegment*.

The execution order of the Test Segments (TSs) follows the order defined in Table 3.2 and the *coreTestInit* function is used to initialize the sequence.

The concept is to allow the user to select how many steps shall be performed by the *coreTest* function, so that the user is able to control the execution progress of the CPU core test. In this way, in case the user has specific execution time constraints, he can decide how many steps execute in order to fulfil the execution time constraints

Table 3.2 Execution steps association w.r.t. *testSegment*

Execution progress	Test Segment	Dataset (if applicable)
0	testSegment00	NA
1	testSegment01	NA
2	testSegment02	NA
3	testSegment03	Float32_MUL_set0
4	testSegment04	Float32_ADD_set0
5	testSegment05	Float32_DIV_set0
6	testSegment06	NA
7	testSegment07	NA
8	testSegment08	Int32_MUL_set0
9	testSegment09	Int32_DIV_set0
10	testSegment10	NA
11	testSegment11	NA
12	testSegment12	NA
13	testSegment13	Int32_UMUL_set0
14	testSegment14	NA
15	testSegment15	NA
16	testSegment16	Int32_ADD_set0
17	testSegment17	NA
18	testSegment18	NA
19	testSegment19	Float32_SQRT_set0
20	testSegment08	Int32_MUL_set1
21	testSegment08	Int32_MUL_set2
22	testSegment09	Int32_DIV_set1
23	testSegment09	Int32_DIV_set2
24	testSegment16	Int32_ADD_set1
25	testSegment16	Int32_ADD_set2
26	testSegment03	Int32_MUL_set0
27	testSegment03	Int32_MUL_set1

28	testSegment03	Int32_MUL_set2
29	testSegment04	Int32_ADD_set0
30	testSegment04	Int32_ADD_set1
31	testSegment04	Int32_ADD_set2

3.2.1 API and CPU Test Environment

All the *testSegment* functions are called through a main interface function named *coreTest*.

The *coreTest* function signature is defined as follows:

*void coreTest(uint8_t steps, const uint8_t forceFail, uint32_t *result);*

Table 4 describes in more detail the input and output of each function.

Please note by using the *forceFail* input it is possible to force the function to fail that is to return an error value. This type of software fault injection feature allows for testing of higher level fault handling mechanisms, specified at the application level.

Table 3.3 coreTest Interface

Table ID	Parameter type	C type	Name	Description
1	Input	unsigned int 8 bit	steps	Specify how many execution progresses have to be executed. Note that each execution of a LUT TS with a specific dataset count for 1 step (see Table 3.2 for details about association of *testSegment* to execution progress). Valid range of steps parameter is: 0 < steps < TOT_TESTSEGMENTS, where TOT_TESTSEGMENTS is the maximum number of execution progresses that could be performed in one run.
2	Input	const unsigned int 8 bit	forceFail	When set to 0 forces the function to fail generating a failure signature that is the inverted value of the correct expected signature. All other values do not have any effect on the function behavior.
3	Output	*unsigned int 32 bit	result	Global pass/fail result of all executed TSs: - 0 If at least one executed testSegment failed - 1 If all executed testSegments passed. - 2 If steps input parameter is out-of-range (see Table 3.2 for details about the valid range).

In order to correctly use *coreTest* function two other functions are given: "*coreTestInit*" function and "*getcoreTestStatus*" function.

The first one is the initialization function, written in C programming language, whose signature is defined as follows.

void coreTestInit(void)

The function has no input or output parameters, since it just initializes the different data structures needed for the correct execution of *coreTest*; in particular it resets the pointer to the next execution progress to be executed. As a consequence, after *coreTestInit* is called, the next TS to be executed will be the testSegment00 (see Table 3.2).

The second function offers to the user the possibility to get the next execution progress which will be executed in the next call of *coreTest* function.

The function is written in C programming language and its signature is defined as follows.

uint8_t getcoreTestStatus(void)

Table 3.4 describes in more details the output of the function.

Table 3.4 *getcoreTestStatus* Interface

Table ID	Parameter type	C type	Name	Description

Table ID	Parameter type	C type	Name	Description
1	Output	*unsigned int 8 bit	n.a.	It indicates the next execution step that will be executed.

test Segments functions are implemented with ARM Cortex-M4 assembly code with a C code interface.

Note that the need for an HW low level control makes the use of assembler necessary, for instance when calling specific assembly instructions with specific parameters.

Since it is possible to have two types of *testSegments* (Fixed or LUT) then we have the two following types of function signatures:

a. "Fixed"
   - *void testSegment**y** (const uint8_t forceFail, uint32_t *result) with y=00, 01, 02, 06, 07, 10, 11, 12, 14, 15, 17, 18.*

b. "LUT"
   - *void testSegment**x** (const uint8_t forceFail, uint32_t *result, const uint32_t *StartDataSet, const uint32_t GoldSign) with x= 03, 04, 05, 08, 09, 13, 16, 19.*

Table 3.5 describes in more details input and output of the functions.

Table 3.5 testSegment Interface

Table ID	testSegment type	Parameter type	C type	Name	Description
1	LUT or Fixed	input	const unsigned int 8 bit	forceFail	When set to 0 force the TS to fail generating a failure signature that is a NOT-inverted value of the proper signature. All other values do not have any effect on the function behavior.
2	LUT	input	const unsigned int 32 bit *	StartDataSet	Start address of the Look Up Table for the selected dataSet.
3	LUT or Fixed	output	const unsigned int 32 bit	GoldSign	Result of signature value.
4	LUT or Fixed	output	unsigned int 32 bit *	result	Pass/fail result of TS execution 0 If TS failed 1 If TS passed.

3.3 Software Integration Rules

This section provides guidelines for how to integrate the CPU test software within the user's own project.

3.3.1 Code Integration

3.3.1.1 Environment for *coreTest* call

Follow the instructions below to call the *coreTest* function.

1. Include *coreTest.h*
2. Create a variable to hold the result of the test as *uint32_t result*. Then the address of the variable is passed to *coreTest* function (see the example below).
3. Define input variables to pass to *coreTest*
   a. *uint8_t steps*
   b. *uint8_t forceFail*
   c. *uint32_t *result*

Example

```
#include "coreTest.h"

uint8_t steps=1;
uint32_t result=0;
uint8_t forceFail = 11;

void main
{
    coreTestInit(); //init index

    /* Launch the core test function in order to perform Diagnosis SW*/
    coreTest(steps, forceFail, &result);
    if(result != 1) {
            errorHandler(); /*Fault handling*/
}
```

After *coreTest* function returns, fault detection can be done by checking the *result* output value as shown in the example above.

A complete example of the *coreTest* function, which calls all *testSegment* is provided in

Appendix B – CPU Test Example.

3.3.2 Compiler Warnings

Please note that in Test Segment 17 two warnings are raised by the compiler respectively at rows 278 and 286. They are related to the utilization of the stack pointer as source register. The warnings come from the fact that the SP cannot assume an a-priori well known value, since it strongly depends on the application. Therefore its utilization could lead to unpredictable behaviors.

Anyway this is not the case of this SW, because only the offset of the SP between two pre-defined assembly instruction blocks is used (accumulated in the signature). Since the offset value is fixed (this part of code is critical, then exceptions are disabled in it), the SW behavior is completely predictable.

3.3.3 Usage Conditions

Table 3.6 summarises usage conditions.

Table 3.6 Conditions of use

ID	Topic	Constraint	Description
1	Interrupt	Avoid corruption of function context.	When interrupting the Diagnostic SW the context of all General Purpose Registers, system register, including APSR and FPSCR, have to be saved and restored once returning from interrupt handling. See Document Reference [1] for details of the CPU register definitions.
2	CPU mode	Correct execution of the SW.	Launch Diagnostic SW in privileged mode
3	Stack	Correct execution of the SW.	Use Main Stack Pointer as stack pointer for the function call.
4	Diagnostic coverage	Execute all the *coreTest* steps during application SW execution.	If a subset of *coreTest* steps are executed from the CPU Test the overall diagnostic coverage of the CPU Test will be lower than the value achieved with the full CPU Test.
5	Interrupt	Avoid corruption of function context.	The following condition applies if there is an Interrupt Service Routine making use of floating point instructions. Inside the application code isolate in a critical section with interrupt disabled the part of the code making use of floating point instructions.

3.4 Define Directives for Software Configuration

No specific define directives are needed.

3.5 Software Package Description

This section details how to identify the supplied software package and also provides a description in tabular format for each design file type.

3.5.1 Identification and Contents of Package

The Software package version is identified as follows:

- Revision 1.0.2
- File list

Table 3.7 CPU Software Test Package and related MD5 signatures.

Nome File	MD5 Signature
closeTest.asm	50c2d658a53cbd2cc01dd65c96060b81
coreTest.c	d73b1c130c736f21b365fe82bf4a49cc
coreTest.h	354338ad61d6d344ad42a6582b46454c
globVar.h	0eec1261c9ba66b25214ebd3b5729b13
initTest.asm	86d528b427395364b9d071b3991aba47

testSegment00.asm	ffa26ea3e0695a1737c81167ab95c140
testSegment00.h	e7a098b362787264230173ab43907ab6
testSegment01.asm	828822cd7a811b3701ff7722053aa1e6
testSegment01.h	4e0d0de4d40176a2747de0217ad0d2d8
testSegment02.asm	718e51540781736c882ca29ee847a785
testSegment02.h	65884aca0f87bfb1a23f0f2a29ba88f6
testSegment03.asm	64fadf7d06a0acf9c030e19d734c0bd9
testSegment03.h	7e406791d29fe3289887ed60af5fc1f8
testSegment04.asm	4e1e681a7d77b6c080b125de333e3f7c
testSegment04.h	8825b9d1ca2bd456c34d8523899f73db
testSegment05.asm	354a456ce9e98ca5cd5b52c280b17745
testSegment05.h	8a97222eec7fae0a2594514df8d4ad2b
testSegment06.asm	c172c5123942a5c3c7a4db0741676afa
testSegment06.h	36efb828af9b33ef3ef360efd717b510
testSegment07.asm	df3a697e716e3dbdb82a5f4a40d924a4
testSegment07.h	688ecb7e16f64129b35f7696a1ad7c5d
testSegment08.asm	1a2fb67f94d0bc36ce923c064e7cc86a
testSegment08.h	fa8e2c6904513c6067d58011292bb297
testSegment09.asm	2187925108eeea813b93076dc4d64d4a
testSegment09.h	2b0056b7fd5917187ec99846d3503f9c
testSegment10.asm	fb8018e88175106feab08d2d198d4ca8
testSegment10.h	c61233702c83f65b3346a766513053b1
testSegment11.asm	6c314801590bad445abbe0503b3485a1
testSegment11.h	22c072c0375dd92a7dd002c4af1378f0
testSegment12.asm	313194fda5acf288922b9d54c6f21702
testSegment12.h	9d01d6d1a7c2eba4bc6ab14c169f4315
testSegment13.asm	40ee680270a3a486bbab1810b8723aea
testSegment13.h	76c207764c711ee676ec70efa063bffd
testSegment14.asm	36a61640cf1cad6a46a842b9c63a42eb
testSegment14.h	663e49cef53ab71bcc5036f50e3b3587
testSegment15.asm	1dfe95ffbcf91bfafc0941a668160836
testSegment15.h	8a96899cecdbd157494d23105988da29
testSegment16.asm	0a9753495c4ae7f9724f024012ba8604
testSegment16.h	27a814c49a3d3de951eebf3b44d0646e
testSegment17.asm	5644dc9a9f6a1d1640ae4eb3a063c4dd
testSegment17.h	7fd2d0b9359f06ee69f910b82943846f
testSegment18.asm	f1dac4d00c4172439b02ad3d4aebb88b
testSegment18.h	30bfeb664848f071c8afe6be4c7495a6
testSegment19.asm	dd61f120b9e697b22f1ca723e710615d
testSegment19.h	8d88b5d41646c51b072e29cfa4723923
testSegmentMgm.c	9024072e409bb0f57ce385324550c77f
testSegmentMgm.h	0f5334d44e33ca6d734a609410c3da25

3.5.2 Description of Design Files

Table 3.8 Design files

Table ID	File Name	Description
1	globVar.h	This file contains the compile option definitions, through which it is possible to select which TSs have to be included in the SW. This file also contains the definition of the LUT, signature vector sizes and other constants.
3	coreTest.h	This file contains the API of the diagnostic SW. In particular contains the coreTest function declaration to be called by the application SW.
4	coreTest.c	This file contains the definition of coreTest function.
	testSegmentMgm.h	This file contains the API of the TS execution progress management. In particular contains the testSegmentMgm function declaration to be called by the coreTest function.
	testSegmentMgm.c	This file contains the definition of testSegmentMgm function.
5	testSegment**xx**.h with **xx**=0,..,19.	This file contains the declaration of the *testSegment* functions.
7	testSegment**xx**.asm with **xx**=0,..,19.	This file contains the assembler definition of the *testSegment* function.
8	initTest.asm	This file contains the TS signature accumulation register initialization.
9	closTest.asm	This file finalize the TS and state whether it is passed or not.

Table 3.9 - Design files

3.6 Resources Usage

Table 3.10 provides an overview of the memory resources used by the code.

Take care that resources related to the *main* file are not part of the *coreTest* function and then not included.

Maximum stack usage is 0 bytes.

Note that no dynamic memory allocation is implemented.

Table 3.10 Memory resources

Module	ROM		RAM
	Code (bytes)	Data (bytes)	rw data (bytes)
coreTest.o	960	6704	0
testSegmentMgm.o	36	0	1
initTest.o	278	0	0
closeTest.o	28	0	0
testSegment00.o	1044	9	0
testSegment01.o	1962	0	0
testSegment02.o	844	0	0
testSegment03.o	2120	0	0
testSegment04.o	1838	0	0
testSegment05.o	1656	0	0
testSegment06.o	1908	0	0
testSegment07.o	604	0	0
testSegment08.o	2398	0	0
testSegment09.o	188	0	0
testSegment10.o	1340	0	0
testSegment11.o	2136	0	0

Module	ROM		RAM
	Code (bytes)	Data (bytes)	rw data (bytes)
testSegment12.o	6320	0	0
testSegment13.o	976	0	0
testSegment14.o	2056	0	0
testSegment15.o	1642	0	0
testSegment16.o	3908	0	0
testSegment17.o	9254	0	0
testSegment18.o	1266	0	0
testSegment19.o	1578	0	0
TOTAL (bytes)	46340	6713	1

Table 3.11 details the execution time for each *testSegment* for all valid values of *dataSet*. Interrupt disable time is also reported when applicable.

Table 3.11 Execution time

testSegment	dataSet	Execution time [clock cycles]	Execution time@48Mhz clock [us]	Maximum interrupt Disable Time [clock cycles]	Maximum interrupt Disable Time @48Mhz clock [us]
testSegment00		679	14,15	0	0
testSegment01		801	16,69	0	0
testSegment02		499	10,40	0	0
testSegment03	Float32_MUL_set0	3129	65,19	47	0,98
testSegment03	Int32_MUL_set0	3087	64,31	47	0,98
testSegment03	Int32_MUL_set1	3143	65,48	47	0,98
testSegment03	Int32_MUL_set2	2987	62,23	47	0,98
testSegment04	Float32_ADD_set0	4833	100,69	48	1
testSegment04	Int32_ADD_set0	2225	46,35	48	1
testSegment04	Int32_ADD_set1	2223	46,31	48	1
testSegment04	Int32_ADD_set2	2231	46,48	48	1
testSegment05	Float32_DIV_set0	2717	56,60	62	1,29
testSegment06		757	15,77	35	0,73
testSegment07		479	9,98	23	0,48
testSegment08	Int32_MUL_set0	1757	36,60	0	0
testSegment08	Int32_MUL_set1	1799	37,48	0	0
testSegment08	Int32_MUL_set2	1739	36,23	0	0
testSegment09	Int32_DIV_set0	1443	30,06	0	0

testSegment	dataSet	Execution time [clock cycles]	Execution time@48Mhz clock [us]	Maximum interrupt Disable Time [clock cycles]	Maximum interrupt Disable Time @48Mhz clock [us]
testSegment09	Int32_DIV_set1	1147	23,90	0	0
testSegment09	Int32_DIV_set2	1289	26,85	0	0
testSegment10		813	16,94	0	0
testSegment11		1097	22,85	50	1,04
testSegment12		4433	92,35	56	1,165
testSegment13	Int32_UMUL_set0	1517	31,60	0	0
testSegment14		996	20,75	43	0
testSegment15		727	15,15	0	0
testSegment16	Int32_ADD_set0	2353	49,02	0	0
testSegment16	Int32_ADD_set1	2375	49,48	0	0
testSegment16	Int32_ADD_set2	2123	44,23	0	0
testSegment17		3059	63,73	27	0,56
testSegment18		621	12,94	0	0
testSegment19	Float32_SQRT_set0	3609	75,19	46	0,955
Total		62687	1305,98	722	15,05

3.7 Requirements for Safety Relevant Applications

Table 3.12 lists requirements for usage in safety relevant applications.

Table 3.12 Safety relevant requirements

ID	Topic	Sub-topic	Description
SW_1	SW integration	Function return	On the return of *coreTest* evaluate the correctness of the execution by checking the value of "result".
SW_2	SW integration	Function call	When calling the *coreTest* function more than once take care to use different variables to store the outcome of the function, specifically the test result. In case the same variable is used consider to initialize it to zero before executing subsequent runs of the function.
SW_3	SW integration	Function environment	Before calling *coreTest* initialize to 0 the variable used by the function to return the *result* value.
PR_1	Project management	User expertise	User has to have good expertise on embedded programming on the target MCU HW Synergy S3 series. Expertise on assembly programming and C level/assembly interface is requested.

3.8 Diagnostic Fault Coverage and Watch Dog Usage

The Diagnostic coverage provided by the CPU Software Test considers that all *testSegments* of type Fixed are launched together with all *testSegments* of type LUT, each one called with all the supported values of the parameter dataSet, as detailed in Table 3.2.

In addition the coverage considers the contribution of a Watchdog. Indeed the use of the CPU Software Test has to be integrated with the use of a Watchdog and Table 3.13 outlines recommendations for its usage.

The necessity of integrating a Watchdog is related to the fact that some hardware faults will make the control flow of the software not to be followed and, in such conditions, the presence of a Watchdog will effectively detect such deviations.

Note also that the CPU Software Test embeds some control flow mechanisms which are required to trigger the activation of such faults. However, as stated above, the fault detection has to be completed by the presence of a Watchdog.

Table 3.13 Recommendations on Watchdog usage

ID	Topic	Description	Comment
1	WD refresh	Consider a control flow monitoring for the WD refresh function: the refresh is done only if the control flow mechanism (e.g. proper value of global variable) is not respected.	
2	WD refresh	Consider a strategy as the following: activate the WD refresh only if all the main tasks having a predictable and periodic timing schedule of the application SW are called in the proper order.	

4. RAM Software Test

4.1 Test Objectives

The objective of the RAM Software Test is to verify the embedded RAM memory of the MCU.

The main features of the software tests are as follows.

a. Whole memory check including stack(s).
   - Memory size programmable at compile time

b. Block-wise implementation of the test.
   - Size of the block programmable at compile time

c. Supports of two test algorithms
   - Extended March C-
   - WALPAT.

d. Word-wise implementation of the test algorithms where the elementary cell under test is considered to be made up by 32 bits width.

e. Support for destructive and non-destructive memory testing.

Please note that information regarding the test algorithms is provided in Appendix B – CPU Test Example.

4.2 API and RAM Test Environment

A RAM block test is called through a main interface function named *testRAM*. The *testRAM* function signature is defined as follows:

void testRAM(unsigned int index, unsigned int selectAlgorithm, unsigned int destructive)

Table 4.1 below describes in more detail the function interface.

Table 4.1 testRAM Interface

Table ID	Parameter type	C type	Name	Description
1	Input	unsigned int	index	Specify RAM block under test: from 0 to *numberOfBUT-1*.
2	Input	unsigned int	selectAlgorithm	Specify algorithm to be run on the RAM block under test: - "0" runs Extended March C- algorithm - "1" runs WALPAT - Other values will produce an error return value (i.e. resultTestRam1 = resultTestRam2 = 0)
3	Input	unsigned int	destructive	Specify the kind of test: - "0" means non-destructive test is run, RAM block content is saved in the buffer; - "1" means destructive test is run. Once a memory block is tested with a destructive procedure its content is initialized with all zeros.

As specified in Table 4.1 *index* indicates the specific RAM block to be tested using the algorithm specified by *selectAlgorithm*. Each RAM block has a size in terms of double words defined by *BUTSize*.

Valid values of *index* range between 0 and *numberOfBUT-1*.

numberOfBUT indicates the number of block in which the RAM is divided and it is derived by dividing the memory size by the size of the block specified by the *BUTSize* parameter.

Calling the function with an invalid value of the block index, that is greater than (*numberOfBUT-1*), will result in the return variables being set to 0 indicating a failed test.

4.3 Test Strategy

The scope of the RAM Software Test is to provide coverage across the whole embedded RAM, adopting a block-wise strategy.

The memory size and the block size are parameters the user can select based on the device and its application needs.

- MUTSize
  - This is the size of the memory under test expressed in number of double word.
- BUTSize
  - This is the size of the block under test in terms of number of double word
- numberOfBUT
  - This is the number of blocks to which the memory is divided.

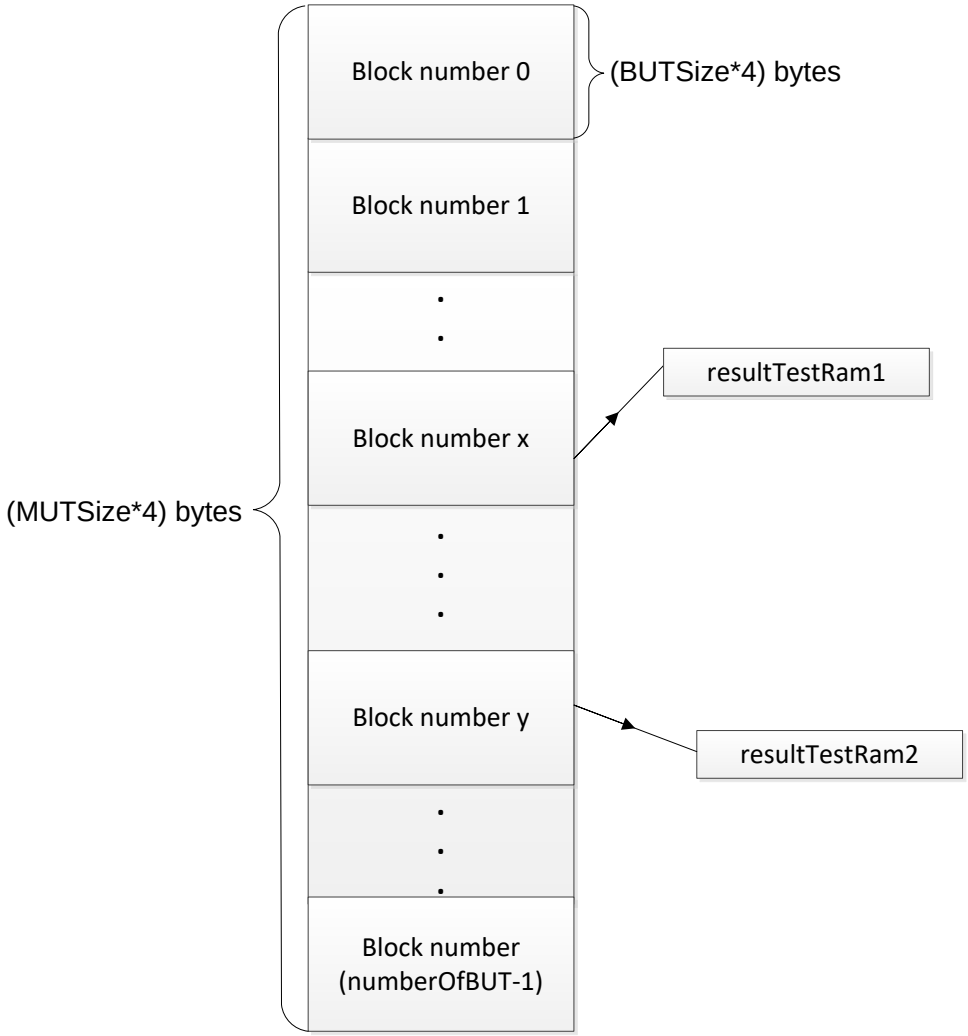

Figure 4.1 RAM block division

Figure 4.1 shows how the memory is divided into a number of blocks equal to *numberOfBUT*.

Each block is then identified with an index ranging from 0 to (*numberOfBUT*-1).

Each block can be tested in a destructive or non-destructive manner.

In order to support non-destructive testing, one block of the RAM is used as a buffer to store the content of the block under test.

The buffer can be tested as well and this can be done with a destructive strategy before testing the other blocks.

A memory reserved area has to be defined for the buffer in order to preserve the integrity of the application software after running the test.

This can be obtained as follows

a. Define the start address of the buffer
    i. This can be done by assigning the label *addressBuffer* inside the file *testRAM.inc*; see Section 4.3 for an example of usage.
b. Define IAR linker commands to reserve the memory buffer locations
    i. Example of linker commands are provided in see section 4.4

The code stores the result of the test in two unused RAM locations accessible from the application software by using two variables: *resultTestRam1* and *resultTestRam2* (see Figure 4.1).

The result variables are located at fixed absolute addresses and they have to be placed into two different blocks.

This strategy has been selected to avoid the issue of not detecting a faulty block due to the fact that the result itself is stored in the same faulty block.

It is worth noting that these two variables are initialized each time the RAM test function is called and the user shall check their values only after having called RAM test function.

By allowing two copies of the test result to be stored into two different blocks, fault detection is still possible because at least one variable won't be stored inside a faulty block.

The location of the result variables can be fixed inside *testRAM.h*.

The application level user then has to check the values of the result variable after the test is completed.

Coding of the test result as follows

i. *resultTestRam1= resultTestRam2=1 implies the test is passed.*
ii. *any other combinations means the test failed.*

An example of a test result check, in addition to definition of addresses for the result variables is provided in Section 4.3

4.4 Software Integration Rules

This section provides guidelines for how to integrate the RAM Test software within the user's own project.

4.4.1 Code Integration

4.4.1.1 Define Memory Size and Block Size

The user has to set the size of the RAM under test and the size of each of the blocks.

This information has to be provided by the directives present in *testRAM.h*.

BUTSize can have one of the values illustrated in Table 4.2 below.

Table 4.2 Relation between BUTSize and MUTSize

BUTSize	Number of Blocks	Index
MUTSize/4	4	0, 1, 2, 3
MUTSize/8	8	0, 1, 2, 3, 4, 5, 6, 7
MUTSize/16	16	0, 1, 2, 3, 4, ..., 15
MUTSize/32	32	0, 1, 2, 3, 4, ..., 31
MUTSize/64	64	0, 1, 2, 3, 4, ..., 63
...	...	...
MUTSize/MUTSize	MUTSize	0, 1, 2, 3, 4, ..., MUTSize-1

Below is a worked example for a 192Kbyte RAM divided in blocks of 1Kbyte size each.

```
//size of the RAM Memory Under Test: 192KB = 192 * 1024 bytes = 196608bytes  = 49152double words
#define MUTSize         49152
//size of the Block of RAM Under Test of 1KB
#define BUTSize          (MUTSize/192)
```

4.4.1.2 Reserve and Place Buffer

In case the user wants to perform non-destructive tests, it is needed a buffer memory area.

A buffer area can be reserved using the IAR linker configuration file (.icf file) and defining a variable *buffer* in the application code.

Assuming the buffer size has to be 1Kbyte (then specify 1024 bytes in hexadecimal format 0x400) and the starting address of the buffer block is 0x2002FC00, then add the following two instructions:

1. *//RAM_TEST:BufferStorage definition*
2. *define block BufferStorage with alignment = 1, size = 0x400 { };*

3. *place at address mem: 0x2002FC00{ block BufferStorage };*

In the file *testRAM.inc* make sure to align the labels *addressBuffer_t* and *addressBuffer_w* to the buffer address, in particular to the most four significant address bytes and the least four significant address bytes.

```
addressBuffer_w  EQU    0xFC00
addressBuffer_t  EQU    0x2002
```

Please note that the RAM buffer shall be stored within the SRAM memory dedicated address range which is specified in the HW manual[2].

In addition, the user shall define a variable *buffer* in the application SW as a global variable and use it to force the linker to allocate it. In particular, considering the above example the user shall insert the following declaration:

```
volatile unsigned int buffer[BUTSize]@ 0x2002FC00 = {0};
```

The user, in order to let the compiler allocate the buffer, shall use this variable, using for example the following instruction:

```
buffer[0] = 0;
```

4.4.1.3 Place Result Variables

The SW stores the result of the test in two unused RAM locations accessible from the application code by using two variables (*resultTestRam1* and *resultTestRam2*).

These two variables have to be placed at two absolute addresses of the RAM.

Declaration of these two variables is defined in *testRAM.h* file.

Considering the case of 192KB RAM divided in blocks of 1KB each, we have for example:

- *resultTestRam1* is placed in the last double word location of the block 0;
- *resultTestRam2* is placed in the last double word location of the block 2.

Code in *testRAM.c* file then has to be as follows:

- unsigned int resultTestRam1 @ 0x20000000 = (unsigned int) 0;
- unsigned int resultTestRam2 @ 0x20000800 = (unsigned int) 0;

4.4.1.4 Word Length

- The chosen RAM algorithm runs using a 32 bit word length.

4.4.1.5 Test Flow and Check Test Results

It is recommended to initially run a destructive test on the buffer. Note that the buffer test has the same result if it is run as destructive or non- destructive; its content are lost.

A recommended flow for the RAM Test is as follows

1. run *testRAM* function on the buffer block;
2. run *testRAM* function on the other blocks of the RAM.

Consider the following instructions to effectively use the *testRAM* function.

1. Include *testRAM.h*
2. Define input variables for parameters to call *testRAM*
    d. *index*
    e. *select Algorithm*
    f. *destructive*
3. Call *testRAM*
4. Check result variables

Worked Example:

```
#include " testRAM.h"

unsigned int index = 7[1];
unsigned int selectAlgorithm = 0;
unsigned int destructive = 0;

testRAM(index, selectAlgorithm, destructive);

 if(!(resultTestRam1&&resultTestRam2)){ /*Fault detection*/
            errorHandler();
 }
```

After the *testRAM* function returns, a fault can be detected by checking the output value as shown in the example above.

Note that the output of *testRAM* is stored in two locations, so if *resultTestRam1* and *resultTestRam2* are both equal to 1 no faults are detected, otherwise fault handling management should start (calling of *errorHandler()* function in the above example).

4.4.2 Usage Conditions

Table 4.3 summarises usage conditions.

[1] Not algorithm specific value, just used as example.

Table 4.3 Conditions of use

ID	Topic	Constraint	Description
1	Interrupt	Avoid corruption of function context.	When interrupting the RAM Software Test the context of all General Purpose Registers, system register, including PSR and FAULTMASK, have to be saved and restored once returning from interrupt handling. See Document Reference [1] for details of the CPU register definitions.
2	CPU mode	Correct execution of the SW.	Launch RAM Software Test in Privileged mode.
3	Stack	Avoid corruption of the stack.	Test RAM blocks corresponding to stack locations in a non-destructive way.
4	Environment	Avoid corruption of variables used to check test results.	In any application code other than the SW test do not overwrite values of *resultTestRam1* and *resultTestRam2* variables.
5	Environment	Avoid data lost	Keep in mind that that data saved by the application inside the buffer will be lost when calling the RAM test.
6	Configuration	Avoid data lost	Do not place the result variables (*resultTestRam1* and *resultTestRam2)* in the same block as the buffer.
7	Configuration	Compliance with SW test strategy	Minimum number of blocks in which RAM is divided has to be 4.
8	Configuration	Compliance with SW test strategy	Range of addresses of the memory under test has to be double word aligned.
9	Configuration	Compliance with SW test strategy	For BUTSize respect the following: BUTSize=MUTSize/2^x with $1<x<=\log_2$(MUTSize)
10	Configuration	Compliance with SW test strategy	Place *resultTestRam1* and *resultTestRam2* variables in two different blocks of the RAM.
11	Diagnostic coverage	Use sufficient block size to guarantee diagnostic coverage value	Both RAM Tests are giving medium coverage (90%) for permanent faults. This coverage value is valid under the condition that for both tests the minimum block size chosen for the test is not lower than 512 bytes.

4.5 Define Directives for Software Configuration

Before compiling the code it is necessary to define the size of the RAM under test, the size of the blocks into which the memory is divided and the word length for the executed RAM test algorithm.

All this information is specified by the directives described in Table 4.4.

Table 4.4 Define directives

Directives	Description
MUTSize	Indicate the size of the RAM under test. Value associated to it expresses size of the RAM in terms of double words. This setting has to be in *testRAM.h*
BUTSize	Indicate the size of the blocks in which the RAM is divided. Value assigned to it has to be of this type: MUTSize/4; MUTSize/8; MUTSize/16; MUTSize/32; ... ; MUTSize/MUTSize This value is always in terms of double words. This setting has to be in *testRAM.h*

4.6 Software Package Description

This section details how to identify the supplied software package, including its MD5 signature and also provides a description in tabular format for each design file type.

4.6.1 Identification and Contents of Package

The Software package version is identified as follows:

- Revision 1.0.1
- File list

Table 4.5 RAM Package and related MD5 signatures

Nome File	MD5 Signature
extendedMarchCminus.asm	aeb6759009a900302e55015424d3a658
extendedMarchCminus.h	eddc772135ebb62c03536ce7649b9b82
testRAM.c	fd62c5f03c3980735ebeb17e618fa0f8
testRAM.h	f161c9d0def145951ff3ea9a8e6230c8
testRAM.inc	393296054a1395d1f664639a901ec2d0
walpat.asm	6219b823cdd280d1aae4f85d6cbe2cb5
walpat.h	7c3e9770144a6d0eeba3568fca019c07

4.6.2 Description of Design Files

Table 4.6 Design files

Table ID	File Name	Description
1	testRAM.h	This file contains the API of the RAM test. In particular contains the *testRAM* function declaration to be called by the application SW. Also it contains declaration of the result variables placed at fixed absolute addresses and define directives
2	testRAM.c	This file contains the definition of *testRAM* function.
3	extendedMarchCminus.h	This file contains the declaration of the Extended March C- algorithm function.
4	extendedMarchCminus.asm	This file contains the definition of the Extended March C- algorithm function.
5	walpat.h	This file contains the declaration of the WALPAT algorithm function.
6	walpat.asm	This file contains the definition of the WALPAT algorithm function.
7	testRAM.inc	This file contains the definition of the patterns for the test execution.

4.7 Resources Usage

Table 4.7 provides an overview of the memory resources used by the code.

The Maximum stack usage is 0bytes.

Table 4.7 Memory resources

Module	ROM		RAM (bytes)
	Code (bytes)	Data (bytes)	
extendedMarchCminus.o	468	0	0
testRAM.o	120	0	8
walpat.o	468	0	0
Total (bytes)	1056	0	8

The timing performance details in Table 4.8 below, are referenced to the test of one RAM block with a size of 1Kb.

Table 4.8 Execution time

Algorithm	NON-Destructive Execution time [clock cycles]	NON-Destructive Execution time@48MHz clock [us]	Destructive Execution time [clock cycles]	Destructive Execution time@48MHz clock [us]
Extended March C-	93476	1947	91176	1899,50
WALPAT	7911754	164828	7909448	164780,17

4.8 Requirements for Safety Relevant Applications

Table 4.9 lists recommendations for usage in safety relevant applications.

Table 4.9 Safety relevant requirements

ID	Topic	Sub-topic	Description
RAM_SW_1	Test flow	Buffer	Before testing blocks other than buffer test the buffer with a destructive testing. Rationale is to avoid corruption of the test result because of a faulty buffer.
RAM_SW_2	Configuration	Number of blocks	Consider to divide the memory under test into a minimum number of blocks, possibly equals to 4. Rationale is to properly detect address faults mainly: the larger the block more efficient the address fault detection.

5. ROM Software Test

5.1 Test Objectives

The objective of the ROM Software Test is to verify the embedded ROM memory of the MCU.

The main features of the software tests are as follows.

- Whole memory check.
- Possibility to test with a block-wise strategy, generating multiple CRC signatures.
- Support of three CRC polynomials.
- Support of incremental mode calculation: calculation of the CRC signature can be time-wise split.

5.2 Test Strategy

The scope of the ROM Software Test is to verify the embedded ROM using a CRC technique. Error detection is achieved as follows:

1. A range of ROM addresses is chosen; this step defines the block under test.
2. A reference checksum value is calculated using the IAR linker and saved inside the memory.
3. During the ROM Software Test execution, the hardware peripheral CRC Calculator (see Document Reference [2] for the peripheral details) is used to produce an actual checksum value of the ROM under test in order to check its integrity.
4. The calculated Checksum value is compared with that stored in memory and an error is detected if the two values do not match.
5. The previous steps are repeated for a different block of memory until the whole ROM area is covered.

5.2.1 Checksum Generation using the IAR linker

Before compiling the ROM Software Test, checksum generation by the IAR linker has to be enabled.

Furthermore, the following information has to be considered.

1. Place a checksum variable for each ROM addresses range under test
2. Start and End addresses of the ROM without considering the location in which checksum value is placed
3. Size and alignment of the checksum variable
4. Initial value of the checksum variable
5. The checksum algorithm used (chosen polynomial)
6. Checksum variable bit order

Further details are provided in Section 5.5

5.2.2 MCU CRC Peripheral

The CRC calculator (refer to Document Reference [2] for peripheral details) generates CRC codes for data blocks. It provides the use of any of the three polynomials listed below.

- 8-bit CRC

1. x^8+x^2+x+1

- 16-bit CRC
  2. $x^{16}+x^{15}+x^2+1$
  3. $x^{16}+x^{12}+x^5+1$

5.3 Top Level Software Structure

Two functions are used to run the CRC calculator module and generate the checksum value

- *crcHwSetup* enables the CRC HW module and configures the control registers to select the selected CRC polynomial to be used
- *crcComputation* calculates checksum on all the bytes of the selected ROM block.

5.3.1 ROM Test APIs

The function signatures are found below

void crcHwSetup(unsigned int crc)

uint16_t crcComputation(unsigned int checksumBegin, unsigned int checksumEnd, unsigned int incrMode)

Table 5.1 describes more details of the interface to the functions.

Table 5.1 ROM test APIs

Table ID	Function	Parameter type	C type	Name	Description
1	crcHwSetup	input	unsigned int	crc	Specify the kind of CRC generating polynomial: -"0": x^8+x^2+x+1 (8-bit CRC) -"1": $x^{16}+x^{15}+x^2+1$ (16-bit CRC) -"2": $x^{16}+x^{12}+x^5+1$ (16-bit CRC) -other values: default is 16-bit CRC $x^{16}+x^{15}+x^2+1$
2	crcComputation	input	unsigned int	checksumBegin	Specify ROM block start address.
3	crcComputation	input	unsigned int	checksumEnd	Specify ROM block end address.
4	crcComputation	input	unsigned int	incrMode	Specify the CRC calculation mode: -"0": incremental mode not active - other values: incremental mode active
5	crcComputation	output	uint16_t	-	The return value of the function is the computed checksum value.

Note that within the *crcComputation* function

- The CRC signature is initialized to 0xff in case of CRC_8 utilization or 0xffff in case of CRC_16 or CRC_16_CCITT.
- The return value is the 1's complement of the calculated checksum.

Note also that the block size of the memory for the CRC calculation is defined by the difference between the end and the start addresses and it has to be a multiple of the CRC length.

5.3.2 Incremental Mode Calculation

The input parameter incrMode allows the user to split the calculation of the CRC signature for the same ROM block in the best way depending on the requirements of its application.

The behaviour is summarized in Figure 5.1:

- the ROM block for which the CRC is to be calculated is divided in sub-blocks identified by a given set of addresses (3 group of addresses in the example);
- then *crcComputation* is run on each set of addresses;
- the first call of *crcComputation* is made with no incremental mode while the following calls need to have the incremental mode active in order to "accumulate" previous partial results;
- after the last function call the total block CRC is returned.

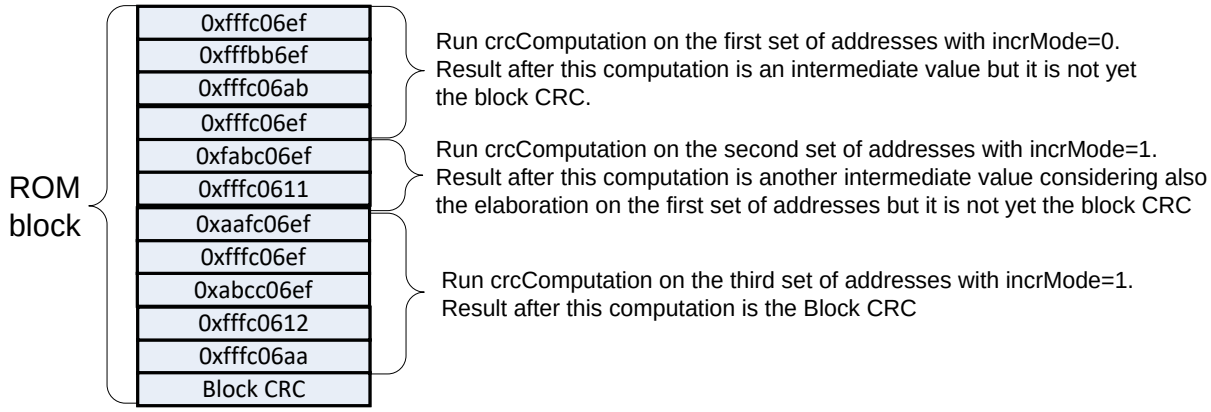

Figure 5.1 Incremental mode calculation

5.4 Software Integration Rules

5.4.1 Code Integration

Follow the instructions below to call the ROM test functions:

1. Include *crc.h*
2. Define extern variables for each CRC signatures generated by the IAR linker and placed in ROM.
3. Define variable for input parameter of *crcHwSetup*:
    a. *crcType*
4. Define variables for input parameter of *crcComputation:*
    a. *checksumBegin*
    b. *checksumEnd*
    c. *incrMode*
5. Define output variable in order to store the result of the *crcComputation*.

Refer to the example in Section 5.4.2.1 which explains a case in which two ROM addresses ranges are tested.

5.4.2 Test Flow and Test Results Check

The recommended test flow is as follows:

1. Initialize the peripheral using *crcHwSetup*.
2. Evaluate the checksum using *crcComputation*.
3. Compare with expected checksum for error detection

5.4.2.1 Worked Example

```
#include “crc.h"
extern const uint16_t __checksum;


  unsigned int type = 1;
  crcHwSetup(type);

  unsigned int checksumStart = 0x00000000;
  unsigned int checksumStop = 0x000FFFFB;
  unsigned int crcIncr = 0;
  uint16_t crcResult;
  crcResult = crcComputation(checksumStart, checksumStop, crcIncr);
  if(crcResult != __checksum){
    errorHandler();
  }
```

After *crcComputation* function returns, a fault can be detected by checking the output value as shown in the example above: *crcResult* achieved by the ROM Software Test is compared with *__checksum*,that is the reference value computed by the IAR linker.

5.4.2.2 Worked Example with Incremental Mode

```
#include “crc.h"

extern const uint16_t _checksum;

unsigned int type;
unsigned int checksumStart;
unsigned int checksumStop;
uint16_t crcResult;
unsigned int crcIncr;

 type = 1;
 crcHwSetup(type);

 crcIncr = 0;
 checksumStart = 0x00000000;
 checksumStop = 0x0007FFFB;            //512KB
 crcResult = crcComputation(checksumStart, checksumStop, crcIncr);

 crcIncr = 1;
 checksumStart = 0x00080000;
 checksumStop = 0x000FFFFB;            //512KB
 crcResult = crcComputation(checksumStart, checksumStop, crcIncr);

 if(crcResult != __checksum){
   errorHandler();
 }
```

The above example shows how the CRC for a 1MB block can be calculated with 2 cumulated runs of the *crcComputation* function.

Note that the *crcResult* is compared with the value computed by the IAR linker only after the last call of the *crcComputation* function.

The above example also shows the 2 calls of the *crcComputation* function are sequential, however this is not a definitive requirement. The calls can be executed in a different order as long as the usage conditions described in section 5.4.3 are maintained.

5.4.3 Usage Conditions

Table 5.2 summarises usage conditions.

Table 5.2 Conditions of use

ID	Topic	Constraint	Description
1	Interrupt	Avoid corruption of function context.	When interrupting the ROM Software Test the context of all General Purpose Registers, system register, including PSR and FAULTMASK, have to be saved and restored once returning from interrupt handling. See Document Reference 1 for details of the CPU register definitions.
2	Incremental mode	Avoid corruption of the calculated CRC value.	When the incremental mode is used do not change the setting or neither use the HW peripheral CRC Calculator until the CRC calculation is completed. This is valid for any kind of SW (e.g. application SW or any interrupt handlers).

5.5 Checksum Generation Using IAR Tools

The ROM Test requires a reference checksum for each addresses range under test. The reference checksum is necessary for comparison with that computed by the CRC calculator.

To ensure accurate control of the error detection performance of the code, it may be necessary to generate multiple checksums.

This section shows how to use the IAR Embedded Workbench for ARM version 8.23.1.17132 to generate the checksum.

The process is outlined in the three steps below.

1. Provide information to the IAR linker as to where to place checksum values. Provide also information about symbols for the start and end addresses of the ROM blocks under test.
2. Use the IAR graphic interface to perform the checksum calculation.
3. In the .icf file, define memory ranges where the checksum values should be placed.

A worked example is provided below, which gives additional clarification for how to use the IAR Tools to generate the required CRCs.

5.5.1 Example Checksum Generation with IAR Tools

Assume the ROM Test addresses range is

- 0x00000000- 0x000FFFFB

and a checksum is required to be generated using the polynomial $x^{16}+x^{12}+ x^5+1$ (16-bit CRC-16CCITT)

1. Go to “Project > Options... > Linker > Checksum” and set the following parameters:
    a. Select “Fill unused code memory” option
    b. File pattern = 0x00
    c. Start Address = 0x00000000
    d. End address = 0x000FFFFB
    e. Select "Generate checksum" option
    f. Checksum size = 2 bytes
    g. Alignment = 1
    h. Algorithm = CRC polynomial, 0x1021
    i. Bit order = MSB
    j. Initial value = 0xFFFF
    k. Checksum unit size = 8 bit.

2. In the .icf file, define memory ranges and locations of the checksums:

    define symbol __ICFEDIT_region_ROMuT_start__ = 0x00000000;
    define symbol __ICFEDIT_region_ROMuT_end__ = 0x000FFFFF;

    define region CHECKSUM_region = mem:[from __ICFEDIT_region_ROMuT_start__ to __ICFEDIT_region_ROMuT_end__];
    place at end of CHECKSUM_region { ro section .checksum };

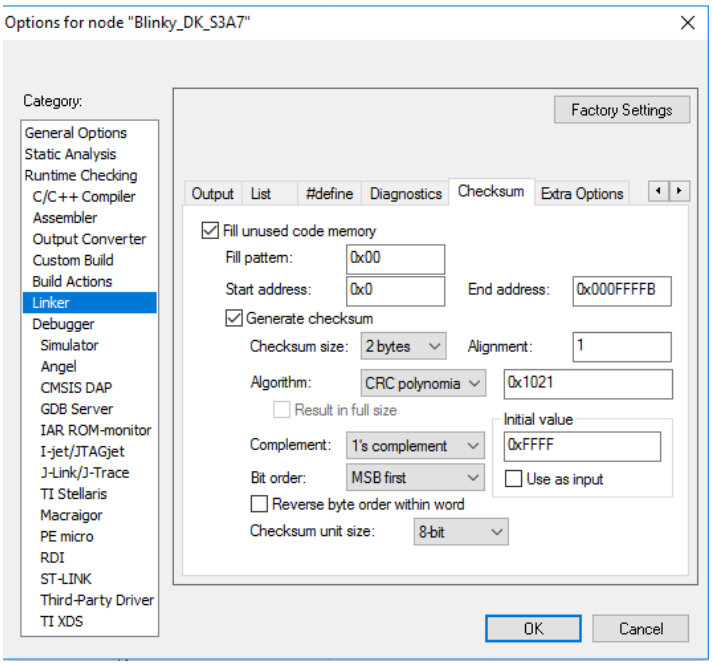

Figure 5.2 IAR Environment Options

For more information about these commands refer to Document Reference 3.

5.6 Software Package Description

This section details how to identify the supplied software package and also provides a description in tabular format for each design file type.

5.6.1 Identification and Contents of Package

The Software package version is identified as follows:

- Revision 1.0.1
- File list

Table 5.3 ROM Package and related MD5 signatures.

Nome File	MD5 Signature
crc.c	d9e82f13ce28208b4d1d2ae25314037d
crc.h	a1277077fe417c19f3a844eb747d67d5
S3A7_registers.h	81c2f1f7d053743283a1debb70f5ecc4

5.6.2 Description of Design Files

Table 5.4 Design files

Table ID	File Name	Description
1	crc.h	• This file contains the declaration of the two functions for the crc calculator: • *crcHwSetup*: It initializes CRC module; • *crcComputation*: It runs CRC on the specified ROM block.
2	crc.c	This file contains the definition of the two functions declared in the file *crc.h*.
3	S3A7_registers.h	This file contains the definitions of the needed peripherals registers.

5.7 Resources Usage

Table 5.5 provides an overview of the memory resources used by the code.

Maximum stack usage is 0 bytes.

Table 5.5 Memory resources

Module	ROM		RAM (bytes)
	Code (bytes)	Data (bytes)	
crc.o	232	0	4
Total (bytes)	232	0	4

Table 5.6 illustrates the execution time for calculating a CRC using the polynomial $x^{16}+x^{15}+x^2+1$ with a block size of 4Kb.

Table 5.6 Execution time

Function	Execution time for a ROM block size of 4Kb [clock cycles]	Execution time for a ROM block of 4Kb @48MHz clock [us]
crcComputation	49188	1024,7

5.8 Requirements for Safety Relevant Applications

Table 5.7 lists recommendations for usage in safety relevant applications.

Table 5.7 Safety relevant requirements

ID	Topic	Sub-topic	Description
ROM_SW_1	CRC type	-	Adopt the following CRC16 polynomial $x^{16}+x^{15}+x^2+1$
ROM_SW_2	Block length	-	Use a block size of 4Kbytes

By following the above mentioned recommendations, it is possible to detect all single and double bit corruptions within one block.

In addition, and regardless of the block size, the use of such a polynomial allows for the detection of an odd number of single bit error, with the following performance in relation to burst error detection, where a burst of length k corresponds to the presence of k consecutive corrupted bits:

- all bursts with length equal and less than 16 bits
- 99.997 percent of bursts of 17 bits
- 99.998 percent of bursts with length greater than 18 bits.

6. CAC Configuration Software

6.1 Test Objectives

The objective of the CAC Configuration Software is to configure the CAC. For safety applications this SW shall be used to:

- Select PCLKB as measurement target clock for the CAC;
- Select Sub-clock oscillator as measurement reference clock for the CAC.

This configuration allows to detect deviations of the Main clock oscillator and PLL due to systematic or random hardware failures.

The CAC Configuration Software also enable the Synergy S3 Oscillation Stop Detection Circuit functionality. This circuit, in case the main clock stops, is in charge to switch to the Middle-Speed On-Chip oscillator and generate an NMI interrupt.

6.2 Test Strategy

The test strategy is to configure the CAC peripheral to monitor PCLKB clock using Sub-clock oscillator.

If the frequency of the monitored clock deviates during runtime from a configured range two types of interrupt can be generated: frequency error interrupt or an overflow interrupt. The user of this module must enable these two kinds of interrupt and handle them.

Note also that it is demanded to the user to enable the Sub-clock oscillator through the SOSCCR register (i.e. SOSCCR.SOSTP = 0b, see User Manual [1]), otherwise the monitoring will not work.

The allowable frequency range is evaluated according to the following equations:

CAULVR (Upper Limit Value) can be computed by rounding down the result from the following equation and converting it into a hexadecimal value:

$$CAULVR = floor\left(\frac{\frac{PCLKB}{CLKT_{DIV}} * \left(1 + 1 - \frac{DC}{100}\right)}{\frac{CLK_{ref}}{CLKR_{DIV}}}\right)$$

Equation 1

CALLVR (Lower Limit Value) can be computed by rounding up the result from the following equation and converting it into a hexadecimal value:

$$CALLVR = ceil\left(\frac{\frac{PCLKB}{CLKT_{DIV}} * \left(\frac{DC}{100}\right)}{\frac{CLK_{ref}}{CLKR_{DIV}}}\right)$$

Equation 2

With parameters having the meaning reported in Table 6.1.

Table 6.1 parameters description for CAULVR, CALLVR

Parameter	Description	Unit
PCLKB	Frequency of the peripheral module clock B	MHz
DC	Target diagnostic coverage	%
CLK_{ref}	Frequency of the reference clock. This is based on the Sub-clock oscillator frequency (32.768 kHz) considering the accuracy of the selected external crystal	MHz
$CLKT_{DIV}$	Division as per Measurement Target Clock Frequency Division Ration Select (TCSS) register	-
$CLKR_{DIV}$	Division as per Measurement Reference Clock Frequency Division Ration Select (RCDS) register	-

In addition to the CAC function the Synergy S3 has an Oscillation Stop Detection Circuit. If the main clock stops, the Middle-Speed On-Chip oscillator will automatically be used instead and an NMI interrupt will be generated. The User of this module must handle the NMI interrupt and check the NMISR.OSTST bit.

6.3 CAC Configuration Software API

The function signatures are found below

void ClockMonitor_Init(double target_clock_frequency, target_clk_div_t target_clock_division,

reference_clk_div_t reference_clock_division, double dc, CLOCK_MONITOR_ERROR_CALL_BACK CallBack);

Table 6.2 describes more details of the interface to the functions.

Table 6.2 CAC Configuration Software APIs

Table ID	Function	Parameter type	C type	Name	Description
1	ClockMonitor_Init	Input	double	target_clock_frequency	The target clock frequency in Hz
2	ClockMonitor_Init	Input	target_clk_div_t	target_clock_division	The target clock division to be set.
3	ClockMonitor_Init	Input	reference_clk_div_t	reference_clock_division	The reference clock division to be set.
4	ClockMonitor_Init	Input	double	dc	The diagnostic coverage in percentage.
5	ClockMonitor_Init	Input	CLOCK_MONITOR_ERROR_CALL_BACK	CallBack	Function to be called if the main clock deviates from the allowable range.

In particular, referring to formulas parameters described in Table 6.1, the function parameters are mapped as the following:

- target_clock_frequency = PCLKB;
- target_clock_division = $CLKT_{DIV}$;
- reference_clock_division = $CLKR_{DIV}$;
- dc = DC.

6.4 Software Integration Rules

This section provides guidelines for how to integrate the CAC Configuration Software within the user's own project.

6.4.1 Code Integration

Follow the instructions below to call the CAC Configuration Software functions:

1. Include *clock_monitor.h*
2. Define variables for input parameters of *ClockMonitor_Init*:
    a. *target_clock_frequency*
    b. *target_clock_division*
    c. *reference_clock_division*
    d. *dc*
    e. *CallBack*

Refer to the example in Section 6.4.2 which explains how to use the Diagnostic SW.

6.4.2 Usage Conditions

The monitoring of the PCLKB clock is set-up with a single function call to ClockMonitor_Init.

For example:

```
#define TARGET_CLOCK_FREQUENCY_HZ          (3000000) // PCLKB: 3MHz (PLL
clock/16)

#define DC (90) // Diagnostic Coverage: 90%



target_clk_div_t target_div = TAR_NO_DIVISION;
reference_clk_div_t ref_div = REF_DIV_32;

/*Enable Sub-Clock*/
PRCR_reg->PRCR = 0xA501;
SOSCCR_reg->SOSCCR_b.SOSTP = 0;
PRCR_reg->PRCR = 0xA500;

ClockMonitor_Init(TARGET_CLOCK_FREQUENCY_HZ, target_div, ref_div, DC,
CAC_Error_Detected_Loop);
```

The clock monitoring is then performed by hardware and so there is nothing that needs to be done by software during the periodic tests.

In order to enable interrupt generation by the CAC, then both Interrupt Controller Unit (ICU) and Cortex-M4 Nested Vectored Interrupt Controller (NVIC) shall be configured in order to handle it.

For configuring the ICU it is necessary to set the ICU Event Link Setting Register (IELSRn) to the event signal number correspondent to the CAC frequency error interrupt (CAC_FERRI = 0x87) and CAC overflow (CAC_OVFI = 0x89). In particular, it is necessary to configure one IELSR register so that it is linked to the aforementioned CAC events:

```
IELSRn.IELS = 0x87;      // (CAC_FERRI)
IELSRn.IELS = 0x89;      // (CAC_OVFI)
```

In addition, in order to enable the Cortex-M4 NVIC to handle the CAC interrupts, the following instructions shall be set:

```
NVIC_EnableIRQ(CAC_FREQUENCY_ERROR_IRQn);
NVIC_EnableIRQ(CAC_OVERFLOW_IRQn);
```

Where CAC_FREQUENCY_ERROR_IRQn and CAC_OVERFLOW_IRQn are the IRQ number that shall be defined by the user[2].

If oscillation stop is detected an NMI interrupt is generated. User code must handle this NMI interrupt and check the NMISR.OSTST flag as shown in this example:

[2] See Table 2-16 of “Cortex-M4 Devices: Generic User Guide”, first release, 16 December 2010 for more details about IRQ numbers.

```
if(1 == R_ICU->NMISR_b.OSTST)
{
   Clock_Stop_Detection();

   /*Clear OSTST bit by writing 1 to NMICLR.OSTCLR bit*/
   R_ICU->NMICLR_b.OSTCLR = 1;
}
```

The OSTDCR.OSTDF status bit can then be read to determine the status of the main clock.

6.5 Define Directives for Software Configuration

No specific directive are present for CAC Configuration Software.

6.6 Software Package Description

This section details how to identify the supplied software package, including its MD5 signature and also provides a description in tabular format for each design file type.

6.6.1 Identification and Contents of Package

The Software package version is identified as follows:

- Revision 1.0.2
- File list

Table 6.3 CAC Configuration Software Package and related MD5 signatures.

Nome File	MD5 Signature
clock_monitor.c	f153ca66e616dfa2db8a8e120634f905
clock_monitor.h	430c7cae882468d4910b4921b18d568e
S3A7_registers.h	4954a0eacdc1b4401f7899735db230cc

6.6.2 Description of Design Files

Table 6.4 Design files

Table ID	File Name	Description
1	clock_monitor.h	This file contains the declaration of the *ClockMonitor_Init* function for the monitoring initialization.
2	clock_monitor.c	This file contains the definition of *clock_monitor* function.
3	S3A7_registers.h	This file contains the definitions of the needed peripherals registers.

6.7 Resources Usage

Table 6.5 provides an overview of the memory resources used by the code.

Maximum stack usage is 120 bytes for both the versions.

Table 6.5 Memory resources

Module	ROM		RAM (bytes)
	Code (bytes)	Data (bytes)	
clock_monitor.o	716	16	4
Total (bytes)	716	16	4

Table 6.6 illustrates the execution time.

Table 6.6 Execution time

Function	Clock Cycle Count	Time measured (us) @ 48MHz
Clock_monitor	2948	61,42

6.8 Requirements for Safety Relevant Applications

Please refer to the Safety Manual [4].

7. IWDT Management Software

7.1 Test Objectives

A watchdog is used to detect abnormal program execution. If a program is not running as expected the watchdog will not be refreshed by software as it is required to be and will therefore detect an error.

7.2 Test Strategy

The Independent Watchdog Timer (iWDT) module of the Synergy S3 is used for this. It includes a windowing feature so that the refresh must happen within a specified 'window' rather than just before a specified time. It can be configured to generate an internal reset or a NMI interrupt if an error is detected. All the configurations for iWDT can be done through OFS0 register whose settings are demanded to the user (see Section 7.4.2 for an example of configuration). A function is provided to be used after a reset to decide if the IWDT has caused the reset.

7.3 IWDT Management Software APIs

The function signatures are found below

void IWDT_Init (void)

void IWDT_Kick (void)

bool IWDT_DidReset (void)

Table 7.1 describes more details of the interface to the functions.

Table 7.1 IWDT Management Software APIs

Table ID	Function	Parameter type	C type	Name	Description
1	IWDT_DidReset	output	Bool	N/A	Returns true if the iWDT has timed out or not been refreshed correctly. This can be called after a reset to decide if the watchdog caused the reset.

7.4 Software Integration Rules

7.4.1 Code Integration

Follow the instructions below to call the IWDT Management Software functions:

1. Include *iwdt.h*
2. Define a boolean variable for output of *IWDT_DidReset.*

Refer to the example in Section 7.4.2 which explains how to use the Diagnostic SW.

7.4.2 Usage Conditions

In order to configure the Independent Watchdog it is necessary to set coherently the OFS0 register. The following code can be used to set the value that has to be stored at the OFS0 memory allocation (OFS0 address = 0x00000400)

```
/* IWDT Start Mode Select */
#define IWDTSTRT_ENABLED  (0x00000000)
#define IWDTSTRT_DISABLED (0x00000001)

/*Time-Out Period selection*/
#define  IWDT_TOP_128    (0x00000000)
#define  IWDT_TOP_512    (0x00000001)
#define  IWDT_TOP_1024   (0x00000002)
#define  IWDT_TOP_2048   (0x00000003)

/*Clock selection. (IWDTCLK/x) */
#define  IWDT_CKS_DIV_1  (0x00000000) // 0b0000
#define  IWDT_CKS_DIV_16 (0x00000002) // 0b0010
#define  IWDT_CKS_DIV_32 (0x00000003) // 0b0011
#define  IWDT_CKS_DIV_64 (0x00000004) // 0b0100
#define  IWDT_CKS_DIV_128 (0x0000000F) // 0b1111
#define  IWDT_CKS_DIV_256 (0x00000005) // 0b0101

/*Window start Position*/
#define  IWDT_WINDOW_START_25    (0x00000000)
#define  IWDT_WINDOW_START_50    (0x00000001)
#define  IWDT_WINDOW_START_75    (0x00000002)
#define  IWDT_WINDOW_START_NO_START (0x00000003) /*100%*/

/*Window end Position*/
#define  IWDT_WINDOW_END_75      (0x00000000)
#define  IWDT_WINDOW_END_50      (0x00000001)
#define  IWDT_WINDOW_END_25      (0x00000002)
#define  IWDT_WINDOW_END_NO_END  (0x00000003) /*0%*/

/*Action when underflow or refresh error */
#define  IWDT_ACTION_NMI       (0x00000000)
#define  IWDT_ACTION_RESET     (0x00000001)

/*IWDT Stop Control*/
#define IWDTSTPCTL_COUNTING_CONTINUE (0x00000000)
#define IWDTSTPCTL_COUNTING_STOP (0x00000001)

#define BIT0_RESERVED (0x00000001)
#define BIT13_RESERVED (BIT0_RESERVED << 13)
#define BIT15_RESERVED (BIT0_RESERVED << 15)

#define OFS0_IWDT_RESET_MASK (0xFFFF0000)

/*This define is used to configure the iWDT peripheral*/
#define OFS0_IWDT_CFG (BIT15_RESERVED | BIT13_RESERVED | BIT0_RESERVED |
(IWDTSTRT_ENABLED << 1) | (IWDT_TOP_1024 << 2) | (IWDT_CKS_DIV_1 << 4) |
(IWDT_WINDOW_END_NO_END << 8) | (IWDT_WINDOW_START_NO_START << 10) |
(IWDT_ACTION_RESET << 12) | (IWDTSTPCTL_COUNTING_CONTINUE << 14))
```

The value OFS0_IWDT__CFG shall be stored at the OFS0 address at compile time in order to configure the Independent Watchdog. In particular, the example enables the iWDT setting a time-out period of 1024 clock cycles at IWDTCLK/1 clock frequency and counting also during sleep mode of the microcontroller. The example does not set any start/end of watchdog window and configure a reset in case of watchdog expiration.

The Independent Watchdog should be initialized as soon as possible following a reset with a call to IWDT_Init:

```
/*Setup the Independent WDT.*/
IWDT_Init();
```

After this the watchdog must be refreshed regularly enough so as to stop the watchdog timing out and performing a reset. Note, if using windowing the refresh must not just be regular enough but also timed to match the specified window. A watchdog refresh is called by calling this:

```
/*Regularly kick the watchdog to prevent it performing a reset. */
IWDT_Kick();
```

If the watchdog has been configured to generate an NMI on error detection then the user must handle the resulting interrupt.

If the watchdog has been configured to perform a reset on error detection then following a reset the code should check if the IWDT caused the watchdog by calling IWDT_DidReset:

```
if(TRUE == IWDT_DidReset())
{
   /*todo: Handle a watchdog reset.*/
   while(1){
      /*DO NOTHING*/
   }
}
```

7.5 Define Directives for Software Configuration

No specific directive are present for IWDT Management Software.

7.6 Software Package Description

This section details how to identify the supplied software package and also provides a description in tabular format for each design file type.

7.6.1 Identification and Contents of Package

The Software package version is identified as follows:

- Revision 1.0.1
- File list

Table 7.2 - iWDT Package and related MD5 signatures

Nome File	MD5 Signature
iwdt.c	88f269058ed774c81f570571a1de1470
iwdt.h	16413964d46e91cc1ddb39c0724d9baa
S3A7_registers.h	c14e2cbb03e58da6bc2a0af960925265

7.6.2 Description of Design Files

Table 7.3 Design files

Table ID	File Name	Description
1	iwdt.h	This file contains the declaration of the functions: • *IWDT_Init*: Initialise the independent watchdog timer. After calling this the IWDT_kick function must then be called at the correct time to prevent a watchdog error. If configured to produce an interrupt then this will be the Non Maskable Interrupt (NMI). This must be handled by user code which must check the NMISR.IWDTST flag; • *IWDT_Kick*: Refresh the watchdog count. • *IWDT_DidReset*: Returns true if the iWDT has timed out or not been refreshed correctly. This can be called after a reset to decide if the watchdog caused the reset.
2	iwdt.c	This file contains the definition of the two functions declared in the file *iwdt.h*.
3	S3A7_registers.h	This file contains the definitions of the needed peripherals registers.

7.7 Resources Usage

Table 5.5 provides an overview of the memory resources used by the code.

Maximum stack usage is 0 bytes.

Table 7.4 Memory resources

Module	ROM		RAM (bytes)
	Code (bytes)	Data (bytes)	
iwdt.o	124	0	0
Total (bytes)	124	0	0

Table 7.5 illustrates the execution time for the specific functions.

Table 7.5 Execution time

Function	Clock Cycles Count	Time measured (us) @ 48MHz
IWDT_Init	26	0,54
IWDT_Kick	19	0,4
IWDT_DidReset	37	0,77

7.8 Requirements for Safety Relevant Applications

Please refer to the Safety Manual [4].

8. LVD Configuration Software

8.1 Test Objectives

The Synergy S3 has a Voltage Detection Circuit. This can be used to detect the power supply voltage (Vcc) falling below a specified voltage.

8.2 Test Strategy

The supplied sample code demonstrates using Voltage Detection Circuit 1 to generate a NMI interrupt when Vcc drops below a specified level. The hardware is also capable of generating a reset but this behavior is not supported in the sample code.

8.3 LVD Configuration Software APIs

The function signatures are found below

void VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL eVoltage)

Table 8.1 describes more details of the interface to the functions.

Table 8.1 LVD Configuration Software APIs

Table ID	Function	Parameter type	C type	Name	Description
1	VoltageMonitor_Init	input	VOLTAGE_MONITOR_LEVEL	eVoltage	The specified low voltage level. See declaration of enumerated type VOLTAGE_MONITOR_LEVEL in voltage.h for details.

8.4 Software Integration Rules

8.4.1 Code Integration

Follow the instructions below to call the LVD Configuration Software functions:

1. Include *voltage.h*
2. Define variable for input parameter of *VoltageMonitor_Init*:
    a. *eVoltage*

Refer to the example in Section 8.4.2 which explains how to use the Diagnostic SW.

8.4.2 Usage Conditions

The Voltage Detection Circuit is configured to monitor the main supply voltage with a call to the VoltageMonitor_Init function. This should be setup as soon as possible following a power on reset.

Please note to set the LVD1SR.DET bit to 0 both before calling VoltageMonitor_init function and in NMI routine, see Section 8.2.2 of 2 for further details.

Please note to set a voltage threshold *eVoltage* lower than the Vcc nominal value.

The following example sets up the voltage monitor to generate an NMI if the voltage drops below 2.99V.

```
VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL_4_29);
```

If a low voltage condition is detected an NMI interrupt will be generated that the user must handle:

```
/*Low Voltage LVD1*/
if(1 == R_ICU->NMISR_b.LVD1ST)
{
 Voltage_Test_Failure();

 /*Clear LVD1ST bit by writing 1 to NMICLR.LVD1CLR bit*/
 R_ICU->NMICLR_b.LVD1CLR = 1;
}
```

8.5 Define Directives for Software Configuration

No specific directive are present for LVD Configuration Software.

8.6 Software Package Description

This section details how to identify the supplied software package and also provides a description in tabular format for each design file type.

8.6.1 Identification and Contents of Package

The Software package version is identified as follows:

- Revision 1.0.1
- File list

Table 8.2 LVD Configuration SW Package and related MD5 signatures.

Nome File	MD5 Signature
S3A7_registers.h	782352821a80f036e8935af878ca2c53
voltage.c	9e603c82f245436b6c5460776c5d422e
voltage.h	216b85df72c30b33091ace537404555a

8.6.2 Description of Design Files

Table 8.3 Design files

Table ID	File Name	Description
1	voltage.h	This file contains the declaration of the functions for voltage monitor: • *VoltageMonitor_Init*: Initialise and start voltage monitoring. An NMI will be generated if Vcc falls below the specified voltage.
2	voltage.c	This file contains the definition of the two functions declared in the file *voltage.h*.
3	S3A7_registers.h	This file contains the definitions of the needed peripherals registers.

8.7 Resources Usage

Table 8.4 provides an overview of the memory resources used by the code.

Maximum stack usage is 0 bytes.

Table 8.4 Memory resources

Module	ROM		RAM (bytes)
	Code (bytes)	Data (bytes)	
voltage.o	188	0	0
Total (bytes)	188	0	0

Table 8.5 illustrates the execution time for the specific functions.

Table 8.5 Execution time

Function	Clock Cycles Count	Time measured (us) @ 48MHz
VoltageMonitor_Init	25243	635

8.8 Requirements for Safety Relevant Applications

Please refer to the Safety Manual [4].

9. Appendix A – RAM Test Algorithms

The following algorithm descriptions are related to 1bit-word memory, but they can be applied to m-bit memories (Word-oriented memory test). The extension to m-bit word is discussed in this Appendix.

9.1 Extended March C-

A March Test consists of a finite sequence of elements called March Elements, delimited by a pair of curling brackets '{ }'.

A March Element is a finite sequence of operations applied to a cell before moving to the next one.

March Elements are delimited by a pair of rounded brackets '( )'. The next cell is defined with respect to the addressing order, which can be, ascending (↑) , descending (↓) or independent (↕) . An operation on a memory cell can be, write 0 (w0), write 1 (w1), read and verify to have read 0 (r0), read and verify to have read 1 (r1).

Extended March C- is represented in Figure 9.1 adopting the notation described above.

$$\{c\ (w0); \Uparrow (r0, w1, r1); \Uparrow (r1, w0);$$
$$\Downarrow (r0, w1); \Downarrow (r1, w0); c\ (r0)\}$$

Figure 9.1 Extended March C- Algorithm

The March C- algorithm detects address faults (AFs), stuck at faults (SAFs), transactional faults (TFs) and coupling faults (CFs) and in addition the Extended March C- algorithm also detects stuck open faults (SOFs) and data retention faults (DRF). Its complexity is equal to 11n where n is the number of addressing cells of the memory.

9.2 WALPAT

The WALPAT algorithm follows the process listed below

1. Write 0 in all cells;
2. For i=0 to n-1
3. { complement cell[i];
    a. For j=0 to n-1, j != i
    b. { read cell[j]; }
4. read cell[i];
5. complement cell[i]; }
6. Write 1 in all cells;
7. For i=0 to n-1
8. { complement cell[i];
    a. For j=0 to n-1, j != i
    b. { read cell[j]; }
9. read cell[i];
10. complement cell[i]; }

The algorithm allows for the detection and location of address faults (AFs), stuck-at faults (SAFs), transactional faults (TFs), coupling faults (CFs) and sense amplifier recovery faults (SARF). Its complexity is equal to 2n2 where n is the number of addressing cells of the memory.

9.3 Word-oriented Memory Test

m-bit memories can be dealt with by repeating each algorithm for a number of times given by:

$$\lceil \log_2 m \rceil + 1$$

For every iteration w1 operation writes a pattern (for instance 00000000) and w0 operation writes the complemented value with respect to that used for w1 (11111111).

Taking into account that the code uses 32bits word access, the algorithm will be repeated 6 times and the following 6 different patterns have to be applied:

00000000000000000000000000000000
00000000000000001111111111111111
00000000111111110000000011111111
00001111000011110000111100001111
00110011001100110011001100110011
01010101010101010101010101010101

10. Appendix B – CPU Test Example

```
#include "coreTest.h"

uint8_t steps=1;
uint32_t result=0;
uint8_t forceFail = 11;


void errorHandler(void);

void main(void)
{
  coreTestInit(); //init index
  steps=36;
  /* Launch the core test function in order to perform Diagnosis SW*/
  coreTest(steps, forceFail, &result);
  if(result != 1) {
      errorHandler();
  }
}
```

11. Appendix C – Pragmas report

The following table reports the pragmas added in the source code to disable specific checks when using the LDRA tool. Related violations have been reviewed in details and judged as not requiring a change to the code.

Table 6 Pragmas report.

Package	File	Code Version	Row	Code (Pragma)	LDRA Rule	MISRA Rule
RAM	testRAM.c	1.0.1	18	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	19	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	20	/*LDRA_INSPECTED 27 D Variable should be declared static. */	27D	R.8.7, R.8.8
RAM	testRAM.c	1.0.1	23	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	24	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	25	/*LDRA_INSPECTED 27 D Variable should be declared static. */	27D	R.8.7, R.8.8
RAM	testRAM.c	1.0.1	28	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	29	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	30	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	33	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6

Package	File	Code Version	Row	Code (Pragma)	LDRA Rule	MISRA Rule
RAM	testRAM.c	1.0.1	36	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	38	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	40	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	43	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	45	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	47	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	61	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	63	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	70	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.c	1.0.1	72	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.h	1.0.1	26	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.h	1.0.1	28	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6

Package	File	Code Version	Row	Code (Pragma)	LDRA Rule	MISRA Rule
RAM	testRAM.h	1.0.1	31	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.h	1.0.1	32	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.h	1.0.1	33	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
ROM	crc.c	1.0.1	21	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
ROM	crc.c	1.0.1	24	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
ROM	crc.c	1.0.1	79	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
ROM	crc.c	1.0.1	80	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
ROM	crc.c	1.0.1	81	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
ROM	crc.c	1.0.1	85	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
ROM	crc.c	1.0.1	90	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
ROM	crc.c	1.0.1	111	/*LDRA_INSPECTED 93 S Value is not of appropriate type. V9.5.0 */	93S	R.10.1, R.10.3, R.10.4, R.10.5, R.11.1
ROM	crc.h	1.0.1	21	/*LDRA_INSPECTED 90 S Basic type declaration	90S	D.4.6

Package	File	Code Version	Row	Code (Pragma)	LDRA Rule	MISRA Rule
				used. */		
ROM	crc.h	1.0.1	24	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
ROM	crc.h	1.0.1	25	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
ROM	crc.h	1.0.1	26	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	66	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	67	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	81	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	83	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	85	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	88	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	90	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	92	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	106	/*LDRA_INSPECTED 90 S Basic	90S	D.4.6

Package	File	Code Version	Row	Code (Pragma)	LDRA Rule	MISRA Rule
				type declaration used. */		
CAC	clock_monitor.c	1.0.2	107	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	108	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	109	/*LDRA_INSPECTED 93 S Value is not of appropriate type. V9.5.0 */	93S	R.10.1, R.10.3, R.10.4, R.10.5, R.11.1
CAC	clock_monitor.c	1.0.2	110	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	111	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	116	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	117	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	118	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	119	/*LDRA_INSPECTED 93 S Value is not of appropriate type. */	93S	R.10.1, R.10.3, R.10.4, R.10.5, R.11.1
CAC	clock_monitor.c	1.0.2	122	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	123	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6

Package	File	Code Version	Row	Code (Pragma)	LDRA Rule	MISRA Rule
CAC	clock_monitor.c	1.0.2	124	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	125	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	128	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	129	/*LDRA_INSPECTED 93 S Value is not of appropriate type. V9.5.0 */	93S	R.10.1, R.10.3, R.10.4, R.10.5, R.11.1
CAC	clock_monitor.c	1.0.2	130	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	131	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	137	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	138	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	139	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	140	/*LDRA_INSPECTED 93 S Value is not of appropriate type. */	93S	R.10.1, R.10.3, R.10.4, R.10.5, R.11.1
CAC	clock_monitor.c	1.0.2	143	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	144	/*LDRA_INSPECTED 90 S Basic type declaration	90S	D.4.6

Package	File	Code Version	Row	Code (Pragma)	LDRA Rule	MISRA Rule
				used. */		
CAC	clock_monitor.c	1.0.2	147	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	148	/*LDRA_INSPECTED 93 S Value is not of appropriate type. V9.5.0 */	93S	R.10.1, R.10.3, R.10.4, R.10.5, R.11.1
CAC	clock_monitor.c	1.0.2	149	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	150	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	176	/*LDRA_INSPECTED 93 S Value is not of appropriate type. V9.5.0 */	93S	R.10.1, R.10.3, R.10.4, R.10.5, R.11.1
CAC	clock_monitor.c	1.0.2	177	/*LDRA_INSPECTED 93 S Value is not of appropriate type. */	93S	R.10.1, R.10.3, R.10.4, R.10.5, R.11.1
CAC	clock_monitor.c	1.0.2	179	/*LDRA_INSPECTED 93 S Value is not of appropriate type. V9.5.0 */	93S	R.10.1, R.10.3, R.10.4, R.10.5, R.11.1
CAC	clock_monitor.c	1.0.2	180	/*LDRA_INSPECTED 93 S Value is not of appropriate type. */	93S	R.10.1, R.10.3, R.10.4, R.10.5, R.11.1
CAC	clock_monitor.h	1.0.2	59	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.h	1.0.2	60	/*LDRA_INSPECTED 90 S Basic type declaration used. */	90S	D.4.6

Document References

1. Cortex-M4 Devices – Generic User Guide, first release, 16/12/2010.
2. Synergy S3 User's Manual: Hardware, Rev. 1.30, February 2018 (Document Reference R01UM0002EU130).
3. IAR C/C++ Development Guide Compiling and linking for Advanced RISC Machines Ltd's ARM Cores, Fifteenth edition, March 2015.
4. Safety Manual, ID=SAF_005_PIA003_S3.

Website and Support

Support: https://synergygallery.renesas.com/support

Technical Contact Details:

- America: https://renesas.zendesk.com/anonymous_requests/new
- Europe: https://www.renesas.com/en-eu/support/contact.html
- Japan: https://www.renesas.com/ja-jp/support/contact.html

All trademarks and registered trademarks are the property of their respective owners.

Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
0.1	Dec 14, 2016	All	First version. Porting of SW User Guide of S7G2 (PA015) to S3A7 (PA024)
0.2	Jan 4, 2017	References	Moved References section at the bottom. Updated References format.
0.3	Jan 26, 2017	§4.4.1.2	Inserted usage condition to reserve buffer area for RAM non destructive tests.
		8.4.2	Added usage condition for LVD SW.
0.4	Feb 9, 2017	All	Updated template.
0.5	Feb 23, 2017	8.4.2	Added usage condition for LVD SW regarding LVD1SR register.
		10.4.2	Added consideration about temperature sensor slope.
0.6	Mar 02, 2017	2.1	Updated C type implementation assumption
0.7	Jun 09, 2017	All	Updated MD5 signature for the final code version of each SW baseline. Updated resource usage for each SW code.
1.0	Jun 14, 2017	All	Internal approval
1.1	Mar 14, 2018	4.7,5.7	Updated Walpat execution time and memory resources used by the code crc.o
		4.4.1	updated integration strategy of RAM test
1.2	Jul 17, 2018	References	Removed revision information from documentation
		3.6,4.7,5.7, 6.7,7.7,8.7	Corrected Resources usage
1.3	Sep 27, 2018	All	Updated the functional safety version of the IAR Embedded Workbench.
		-	Removed “ADC14 Comparator Software” and TSN “Management Software” chapters.
		-	Updated latest release and MD5s of CPU,RAM,ROM,CAC,IWDT and LVD tests.
		All	Replaced “S3A7” Synergy name with “S3”.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
3. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics product.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; and safety equipment etc.
   Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (nuclear reactor control systems, military equipment etc.). You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application for which it is not intended. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You should not use Renesas Electronics products or technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. When exporting the Renesas Electronics products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, who distributes, disposes of, or otherwise places the product with a third party, to notify such third party in advance of the contents and conditions set forth in this document, Renesas Electronics assumes no responsibility for any losses incurred by you or third parties as a result of unauthorized use of Renesas Electronics products.
11. This document may not be reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

RENESAS

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HALII Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141