Renesas Synergy™ Platform

S7 Series MCU Diagnostic Software User Guide

R11AN0081EU0107
Rev. 1.07
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 the critical areas of the MCU.

The Renesas Diagnostics Software is designed for use with the Synergy S7 Microcontroller Family. Tests are provided for coverage of the following critical areas of MCU operation: the Central Processing Unit (CPU), embedded flash ROM memory, 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 8.23.17132 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.

Note: In the code, some pragmas have been added in the form of comments, for example, "`/*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 Appendix C – Pragmas report for details of the pragmas inserted.

Target Device

Synergy S7 Series MCU

Contents

1. Common Terminology 5
1.1 Acronyms 5
2. Compiler Environment 5
2.1 C Type Implementation 5
2.2 IAR Environment Settings 5
3. CPU Software Test 6
3.1 Test Objectives 6
3.2 Software Structure 6
3.2.1 API and CPU Test Environment 8
3.3 Software Integration Rules 10
3.3.1 Code Integration 10
3.3.2 Compiler Warnings 11
3.3.3 Usage Conditions 12
3.4 Define Directives for Software Configuration 12
3.5 Software Package Description 12
3.5.1 Identification and Contents of Package 12

3.5.2 Description of design files 14
3.6 Resources Usage 15
3.7 Requirements for Safety Relevant Applications 17
3.8 Diagnostic Fault Coverage and Watch Dog Usage 17

4. RAM Software Test 17
4.1 Test Objectives 17
4.2 Test Strategy 18
4.3 API and RAM Test Environment 19
4.4 Software Integration Rules 20
4.4.1 Code integration 20
4.4.2 Usage Conditions 23
4.5 Define Directives for Software Configuration 23
4.6 Software Package Description 24
4.6.1 Identification and Contents of Package 24
4.6.2 Description of design files 24
4.7 Resources Usage 25
4.8 Requirements for Safety Relevant Applications 25

5. ROM Software Test 25
5.1 Test Objectives 25
5.2 Test Strategy 25
5.2.1 Checksum Generation using the IAR linker 26
5.2.2 MCU CRC Peripheral 26
5.3 Top Level Software Structure 26
5.3.1 ROM Test APIs 26
5.3.2 Incremental mode calculation 27
5.4 Software Integration Rules 28
5.4.1 Code integration 28
5.4.2 Test flow and test result check 28
5.4.3 Usage Conditions 30
5.5 Checksum Generation Using IAR Tools 30
5.5.1 Example Checksum Generation with IAR Tools 30
5.6 Software Package Description 31
5.6.1 Identification and contents of package 31
6.1.1 Description of Design Files 32
6.2 Resources Usage 32
6.3 Requirements for Safety Relevant Applications 32

7. CAC Configuration Software 33
7.1 Test Objectives 33
7.2 Test Strategy 33

7.3 CAC Configuration Software API 34
7.4 Software Integration Rules 34
7.4.1 Code integration 35
7.4.2 Usage Conditions 35
7.5 Define Directives for Software Configuration 36
7.6 Software Package Description 36
7.6.1 Identification and Contents of Package 36
7.6.2 Description of Design Files 36
7.7 Resource Usage 36
7.8 Requirements for Safety Relevant Applications 37

8. IWDT Management Software 37
8.1 Test Objectives 37
8.2 Test Strategy 37
8.3 IWDT Management Software APIs 37
8.4 Software Integration Rules 37
8.4.1 Code integration 37
8.4.2 Usage conditions 37
8.5 Define Directives for Software Configuration 39
8.6 Software Package Description 39
8.6.1 Identification and Contents of Package 39
8.6.2 Description of design files 40
8.7 Resources Usage 40
8.8 Requirements for Safety Relevant Applications 40

9. LVD Configuration Software 40
9.1 Test Objectives 40
9.2 Test Strategy 40
9.3 LVD Configuration Software APIs 41
9.4 Software Integration Rules 41
9.4.1 Code integration 41
9.4.2 Usage conditions 41
9.5 Define Directives for Software Configuration 41
9.6 Software Package Description 41
9.6.1 Identification and Contents of Package 42
9.6.2 Description of design files 42
9.7 Resource Usage 42
9.8 Requirements for Safety Relevant Applications 42

10. Requirements for Safety Relevant Applications 42

11. Appendix A - RAM Test Algorithms 42
11.1 Extended March C- 43

11.2 WALPAT 43
11.3 Word-oriented Memory Test 43

12. Appendix B - CPU Test Example 44

13. Appendix C – Pragmas report 44

14. Document References 50

Website and Support 51

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, 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 is represented in 32-bit format in 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

Table ID	Category	Sub-category	Setting description	Comment
1	General options	Target	• Device := Renesas R7FS7G27H • 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 to be 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 actions	For RAM test specific testing see section 4 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 predominantly instruction based diagnosis, with the aim of detecting permanent hardware failures of the CPU core.

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

See reference document [REF.1] for the complete list of instructions. 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 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, while the `coreTest` allows the user to select and run 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 provides an overview of the `testSegment` functions.

Two types of `testSegment` functions are defined as follows:

- `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
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 the number of steps to be performed by the `coreTest` function, so that the user can control the execution progress of the CPU core test. If the user has specific execution time constraints, the user can decide how many steps execute for the execution time constraints to be fulfilled.

Table 3-2 Association of execution steps with respect to `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 3-3 describes the input and output of each function in more detail.

Using the forceFail input makes it 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	Specifies how many execution progresses have to be executed. Each execution of a LUT TS with a specific dataset counts for 1 step (see Table 3-2 for details on the 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 behaviour.
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 the valid range information).

In order to correctly use coreTest function, two other functions, coreTestInit and getcoreTestStatus are provided.

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 the output of the function in more detail.

Table 3-4 getcoreTestStatus interface

Table ID	Parameter type	C type	Name	Description
1	Output	*unsigned int 8 bit	N/A	Indicates the next execution step to be executed

testSegments functions are implemented using ARM Cortex-M4 assembly code, with a C code interface.

Note that the need for a hardware low level control makes the use of an 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), we have the following two types of function signatures:

1. Fixed
   - `void testSegmenty (const uint8_t forceFail, uint32_t *result) with y=00, 01, 02, 06, 07, 10, 11, 12, 14, 15, 17, 18.`
2. LUT
   - `void testSegmentx (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 input and output of the functions in more detail.

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, forces the TS to fail, generating a failure signature that is a non-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 project.

3.3.1 Code Integration

Environment for `coreTest` call

To call the `coreTest` function:

1. Include `coreTest.h`.
2. Create a variable to hold the result of the test as `uint32_t` result. The address of the variable is then passed to `coreTest` function (see the following example).
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 the `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

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

However, this is not the case in this software, 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 the code is critical, and exceptions are disabled in it), the software behavior is completely predictable.

3.3.3 Usage Conditions

The usage conditions are summarized in Table 3-6.

Table 3-6 Usage Conditions

ID	Topic	Constraint	Description
1	Interrupt	Avoid corruption of function context	When interrupting the diagnostic software, the context of all general purpose registers, system register, including APSR and FPSCR, have to be saved and restored after returning from interrupt handling. See reference document [REF.1] for the CPU register definitions.
2	CPU mode	Correct execution of the software	Launch diagnostic software in privileged mode
3	Stack	Correct execution of the software	Use the main stack pointer as stack pointer for the function call
4	Environment	Avoid corruption of software flow by corruption of a control flow variable	Do not write to the `testExcp` variable
5	Diagnostic coverage	Execute all the `coreTest` steps during application software execution	If a subset of `coreTest` steps are executed from the CPU test, the overall diagnostic coverage of the CPU test is lower than the value achieved with the full CPU test
6	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 a table format for each design file type.

3.5.1 Identification and Contents of Package

The software package version is:

- Revision 1.0.3
- File list.

Table 3-7 CPU Software Test Package and related MD5 signatures

File Name	MD5 Signature
closeTest.asm	1ceb60324c1b5653eddf9a0d25ef7732
coreTest.c	7b9bfb92fcf1b4d47c7ef21881e4bcc2
coreTest.h	647410b11f8c049c0a3c70341c75041f
globVar.h	61ebb216e6f98dc08c0fcbf906415b1e
initTest.asm	d72749c9df087a65ecab81f1339cc0af
testSegment00.asm	301aa75e0285956f86aa410336dbeb19
testSegment00.h	557784953af6d2b43298ba6e5e45fcc0
testSegment01.asm	4d67ea08005e8286be2030411cfb3e04
testSegment01.h	bd7f3370e24ff175e433e10d25d1df1d
testSegment02.asm	ff4dda0f01b651ccab65e47bd4a559a6
testSegment02.h	27d1a1efd77de1f3a5eca69ba2fcd943
testSegment03.asm	ee6fc066ca1c3bd8e46ba2c01e549474
testSegment03.h	b296d52facc6efc77fa21cf3ff8119f8
testSegment04.asm	5699c6d06fee35740b667e06de127ba0
testSegment04.h	7b82f6dd015b353bcb8b4e588aa2d32f
testSegment05.asm	57e2b5f11f330cc665ce9e726400aad5
testSegment05.h	e42445714f84d31fb8d4b514e4aa6261
testSegment06.asm	5f9466182f4a9584b287a7034b043b03
testSegment06.h	8ae790aa4e0683cc4cd669f619c4bb16
testSegment07.asm	8452bbb8cb22922495ca13dc4a24b06a
testSegment07.h	c5f204de84871bf84f5e9b89ddf49756
testSegment08.asm	a7e31d2abe88c211b48f012561d1585d
testSegment08.h	92604b9629d916e5f799629064e7a403
testSegment09.asm	ea87df5e3c11901354ccc608d215e1c8
testSegment09.h	853ee64ad838c6b0113ef765bc0a6834
testSegment10.asm	12580630c8f898575f7f4e7e7aeb0b9c
testSegment10.h	26a3caeda6473948dd4dccf700f7dc48
testSegment11.asm	c76a0b533e3ec1235106984ada374594
testSegment11.h	76525ecf711e921ab1aca21b0ba4a342
testSegment12.asm	8be38b28576b57b6db2f82ef7ff815a4
testSegment12.h	114f3bc90f5aefb4f7ae0d49201ce1e5
testSegment13.asm	e2fef8bf9cedbdf5ec40672aaea7df40
testSegment13.h	ebb9f7b5cfb596d6273235f4da271750
testSegment14.asm	1c3b02acef9a169bb9e1d79cd934e3d4
testSegment14.h	b0fe14f8fb2b794133c5fc4ce1d81a39
testSegment15.asm	383d4795095c07580ea12c10866961ea
testSegment15.h	9c0c62fec8cdc65372f0a45df62e009b
testSegment16.asm	0db83e3d411022658d012372c05b5369
testSegment16.h	1cdade74f94167841af65c2a96e1fbff
testSegment17.asm	58b2c69f33c4ef4aa78c139c03db1861
testSegment17.h	767a83085c8bfafa2ccfb65f7f448e48
testSegment18.asm	bd9f2f611659ab925a1e6f06a89bb0f5
testSegment18.h	7cdd0ed12725d2cf54623468629a54f0
testSegment19.asm	df47f17dfe4bda9dfbf20c61830f1e89

testSegment19.h	e0370a39dc267c37ccf1a0e58546d63c
testSegmentMgm.c	5e2072e0901c82b6fe5c0e856f1267ec
testSegmentMgm.h	b26103e3c01e14593793d28e93de1e2d

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 software. 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 software. In particular, it contains the `coreTest` function declaration to be called by the application software.
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, it contains the `testSegmentMgm` function declaration to be called by the `coreTest` function.
	`testSegmentMgm.c`	This file contains the definition of `testSegmentMgm` function
5	`testSegmentxx.h with xx=0,..,19`	This file contains the declaration of the `testSegment` functions
7	`testSegmentxx.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

3.6 Resources Usage

Resources related to the main file should not be part of the `coreTest` function, and should not be included.

The maximum stack usage is 212 bytes.

Note: No dynamic memory allocation is implemented.
Table 3-9 provides an overview of the memory resources used by the code.

Table 3-9 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
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 3-10 details the execution time for each `testSegment` for all valid values of `dataSet`. Interrupt disable time is also reported when applicable.

Table 3-10 Execution time

testSegment	dataSet	Execution time [clock cycles]	Execution time with 240 MHz clock [µs]	Maximum interrupt Disable Time [clock cycles]	Maximum interrupt Disable Time with 240 MHz clock [µs]
testSegment00		686	2,86	0	0
testSegment01		816	3,4	0	0
testSegment02		528	2,2	0	0
testSegment03	Float32_MUL_set0	2630	11,0	49	0,204
testSegment03	Int32_MUL_set0	2604	10,9	49	0,204
testSegment03	Int32_MUL_set1	2610	10,9	49	0,204
testSegment03	Int32_MUL_set2	2522	10,5	49	0,204
testSegment04	Float32_ADD_set0	4508	18,8	48	0,2
testSegment04	Int32_ADD_set0	2106	8,78	48	0,2
testSegment04	Int32_ADD_set1	2106	8,78	48	0,2
testSegment04	Int32_ADD_set2	2112	8,80	48	0,2
testSegment05	Float32_DIV_set0	2640	11	62	0,258
testSegment06		794	3,31	35	0,146
testSegment07		514	2,14	23	0,096
testSegment08	Int32_MUL_set0	1706	7,11	0	0
testSegment08	Int32_MUL_set1	1736	7,23	0	0
testSegment08	Int32_MUL_set2	1682	7,01	0	0
testSegment09	Int32_DIV_set0	1394	5,81	0	0
testSegment09	Int32_DIV_set1	1096	4,57	0	0
testSegment09	Int32_DIV_set2	1230	5,13	0	0
testSegment10		822	3,43	0	0
testSegment11		1086	4,52	52	0,216
testSegment12		4250	17,7	56	0,233
testSegment13	Int32_UMUL_set0	1494	6,23	0	0
testSegment14		1016	4,23	43	
testSegment15		752	3,13	0	0
testSegment16	Int32_ADD_set0	2296	9,57	0	0
testSegment16	Int32_ADD_set1	2316	9,65	0	0
testSegment16	Int32_ADD_set2	2074	8,64	0	0
testSegment17		3020	12,6	27	0,112
testSegment18		652	2,72	0	0
testSegment19	Float32_SQRT_set0	3556	14,8	46	0,191
Total		59354	247,45	732	3,04

3.7 Requirements for Safety Relevant Applications

Table 3-11 lists requirements for usage in safety relevant applications.

Table 3-11 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 the “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 initializing 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	The user has to have good expertise in embedded programming on the target MCU HW Synergy S7 series. Expertise on assembly programming and C level/assembly interface is needed.

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. The use of the CPU software test should be integrated with the use of a Watchdog. Table 3-12 outlines the recommendations for its usage.

A Watchdog needs to be integrated due to the fact that some hardware faults prevent following the control flow of the software and, in such conditions, the presence of a Watchdog effectively detects such deviations.

Also, 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 needs the presence of a Watchdog.

Table 3-12 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, for example, proper value of the global variable, is not respected.	
2	WD refresh	Consider the following strategy: Activate the WD refresh only if all the main tasks of the application software that have a predictable and periodic timing schedule 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:

1. Whole memory check including stack(s).
   - Memory size programmable at compile time.
2. Block-wise implementation of the test.
   - Size of the block programmable at compile time.
3. Two test algorithm support
   - Extended March C-

— WALPAT.

4. Word-wise implementation of the test algorithms, where the elementary cell under test is considered to be of 32-bit width.
5. Support for destructive and non-destructive memory testing.

Information regarding test algorithms is provided in Appendix B - CPU Test Example.

4.2 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 that the user can select based on the device and its application needs, described as follows:

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

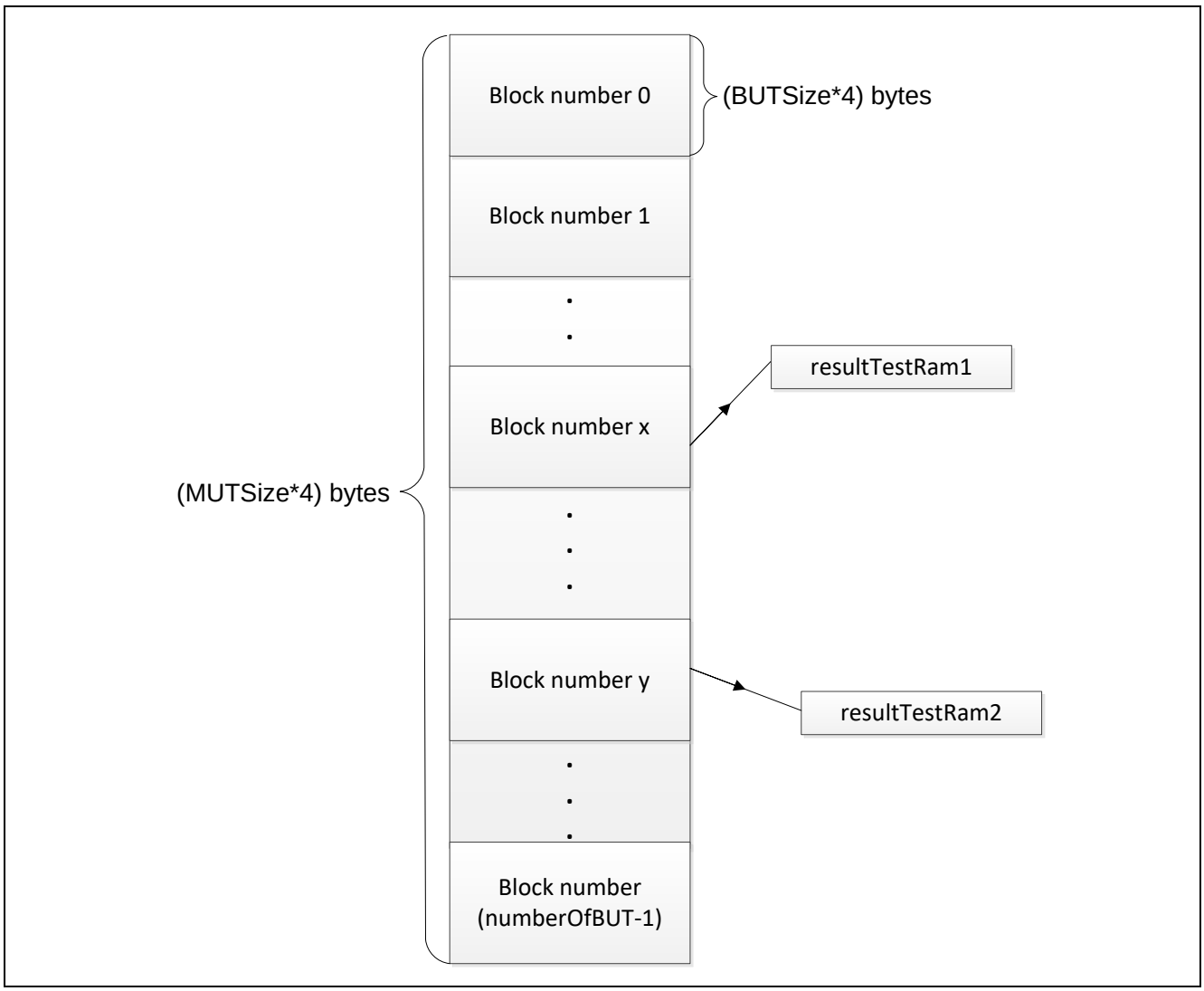

Figure 4-1: RAM block division.

Figure 4-1shows 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.

To do this:

1. Define the start address of the buffer.
   This can be done by assigning the label `addressBuffer` inside the file `testRAM.inc`. See section 4.4 for a usage example.
2. Define IAR linker commands to reserve the memory buffer locations.
   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 not detecting a faulty block because the result itself is stored in the same faulty block.

Note: These two variables are initialized each time the RAM test function is called, and the user needs to check their values only after having called the RAM test function.

Allowing two copies of the test result to be stored into two different blocks makes fault detection possible because at least one variable is not 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 is as follows:

1. `resultTestRam1= resultTestRam2=1 implies the test is passed.`
2. `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 4.4.

4.3 API and RAM Test Environment

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

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

testRAM interface in Table 4-1 describes the function interface in more detail.

Table 4-1 testRAM interface

Table ID	Parameter type	C type	Name	Description
1	Input	unsigned int	`index`	Specify the RAM block under test, from 0 to `numberOfBUT`-1
2	Input	unsigned int	`selectAlgorithm`	Specify the algorithm to be run on the RAM block under test: • 0 runs Extended March C- algorithm • 1 runs WALPAT. Other values produce an error return value (that is, `resultTestRam1` = `resultTestRam2` = 0).
3	Input	unsigned int	`destructive`	Specify the kind of test: • 0 means non-destructive test is run, and 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 blocks in which the RAM is divided, and it is obtained 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), results in the return variables being set to 0, indicating a failed test.

4.4 Software Integration Rules

This section provides guidelines for how to integrate the RAM test software within the user project.

4.4.1 Code integration

Defining 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 shown in Table 4-2.

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

Following is a working example for a 640 KB RAM, divided in blocks of 64 KB size each.

```
//size of the RAM Memory Under Test: 640KB = 640 * 1024 bytes = 655360 bytes  =
163840double words

#define MUTSize   163840

//size of the Block of RAM Under Test of 64KB

#define BUTSize             (MUTSize/10)
```

Reserving and placing the 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 that the buffer size has to be 64 KB (specify 1024 bytes in hexadecimal format 0x10000) and the starting address of the buffer block is 0x20030000, add the following two instructions:

```
//RAM_TEST:BufferStorage definition
```

1. `define block BufferStorage with alignment = 1, size = 0x10000 { };`
2. `place at address mem:0x20030000 { 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    0x0000
addressBuffer_t  EQU    0x2003
```

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

In addition, in order to minimize possible interference with the application SW, it is recommended to define a variable buffer in the application SW as a global variable and use it to force the allocation through the linker. Below is an example for the case reported above:

```
volatile unsigned int buffer[BUTSize]@ 0x20030000 = {0};
```

The user could then instruct the compiler to allocate the buffer to this variable, using for example the following instruction:

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

Placing result variables

The software stores the result of the test in two unused RAM locations that are 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.

The following is an example with 640 KB RAM divided in blocks of 64 KB each:

- `resultTestRam1` is placed in the last double word location of the block 3
- `resultTestRam2` is placed in the last double word location of the block 4.

The code in `testRAM.c` file is:

- unsigned int resultTestRam1 @ 0x20000000 = (unsigned int) 0;
- unsigned int resultTestRam2 @ 0x20010000 = (unsigned int) 0;.

Word length

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

Test flow and checking test results

It is recommended to initially run a destructive test on the buffer. The buffer test has the same result if it is run as destructive or non- destructive, and 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`:
   A. `index`.
   B. `select Algorithm`.
   C. `destructive`.
3. Call `testRAM`.
4. Check result variables.

Working 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.

The output of `testRAM` is stored in two locations. So, if `resultTestRam1` and `resultTestRam2` are both equal to 1, no faults are detected. Otherwise, the fault handling management should start (calling of `errorHandler()` function in the above example).

[1] Not algorithm specific value. Only used as an example.

4.4.2 Usage Conditions

Table 4-3 summarizes usage conditions.

Table 4-3 Usage conditions

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 when returning from interrupt handling. See reference document [REF.1] for the detailed 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 manner
4	Environment	Avoid corruption of variables used to check test results	In any application code other than the software test, do not overwrite values of `resultTestRam1` and `resultTestRam2` variables
5	Environment	Avoid data lost	The data saved by the application inside the buffer is 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	The minimum number of blocks into which the 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`, use the following formula: $BUTSize=MUTSize/2^x$ 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 give 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. The value associated with it expresses the 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 the following 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 table 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

File Name	MD5 Signature
extendedMarchCminus.asm	8d29d2c4ef1b516ace04e7403b986d5d
extendedMarchCminus.h	cf8ad143080603ae2aed9beeec3dfb64
testRAM.c	242961ce5f3ca457811f9797d15dab02
testRAM.h	03afee8c63ff96e4d3a3c8acecb3f42d
testRAM.inc	dc4cb561dc5fc9a154917b5d271ff418
walpat.asm	656312c044114043de5d6bf8904f8e0c
walpat.h	caf2c03440ea9f2ce8d2be2b7cc7894c

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, it contains the testRAM function declaration to be called by the application software. It also contains the declaration of the result variables placed at fixed absolute addresses, and define directives.
2	testRAM.c	This file contains the definition of the 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 60 bytes.

Table 4-7 Memory resources

Module	ROM		RAM (bytes)
	Code (bytes)	Data (bytes)	
`extendedMarchCminus.o`	468	0	0
`testRAM.o`	124	0	8
`walpat.o`	468	0	0
Total (bytes)	1060	0	8

The timing performance details in Table 4-8, are referenced to the test of one 1 Kb RAM block.

Table 4-8 Execution time

Algorithm	Non-destructive execution time [clock cycles]	Non-destructive execution time with 240 MHz clock [µs]	Destructive execution time [clock cycles]	Destructive execution time with 240 MHz clock [µs]
Extended March C-	104106	433	100524	418,85
WALPAT	8699562	36248	8695982	36233,26

4.8 Requirements for Safety Relevant Applications

Table 4-9 lists the 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 the buffer, perform destructive testing on the buffer. This should be done to avoid corruption of the test result because of a faulty buffer.
RAM_SW_2	Configuration	Number of blocks	Consider dividing the memory under test into a minimum number of blocks, possibly equal to 4. This is to properly detect address faults. 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 for three CRC polynomials
- Support for incremental mode calculation, that is, 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 reference document [REF.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.

In addition, use the following steps:

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. Consider the size and alignment of the checksum variable.
4. Consider the initial value of the checksum variable.
5. Consider the checksum algorithm used (chosen polynomial).
6. Consider the checksum variable bit order.

Further details are provided in section 5.5.

5.2.2 MCU CRC Peripheral

The CRC calculator generates CRC codes for data blocks. For details on the peripherals, see document reference [REF.2]. It provides the use of any of the three polynomials listed as follows:

- 8-bit CRC
  - x^8+x^2+x+1
- 16-bit CRC
  - $x^{16}+x^{15}+x^2+1$
  - $x^{16}+x^{12}+x^5+1$.

5.3 Top Level Software Structure

The following 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 as follows:

```
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: 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 th' 1's complement of the calculated checksum.

Note: 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 as best suited to the requirements of its application.

The behavior, as summarized in Figure 5-1 is as follows:

- The ROM block for which the CRC is to be calculated is divided in sub-blocks identified by a given set of addresses (3 groups of addresses in the example)
- The `crcComputation` is then 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.

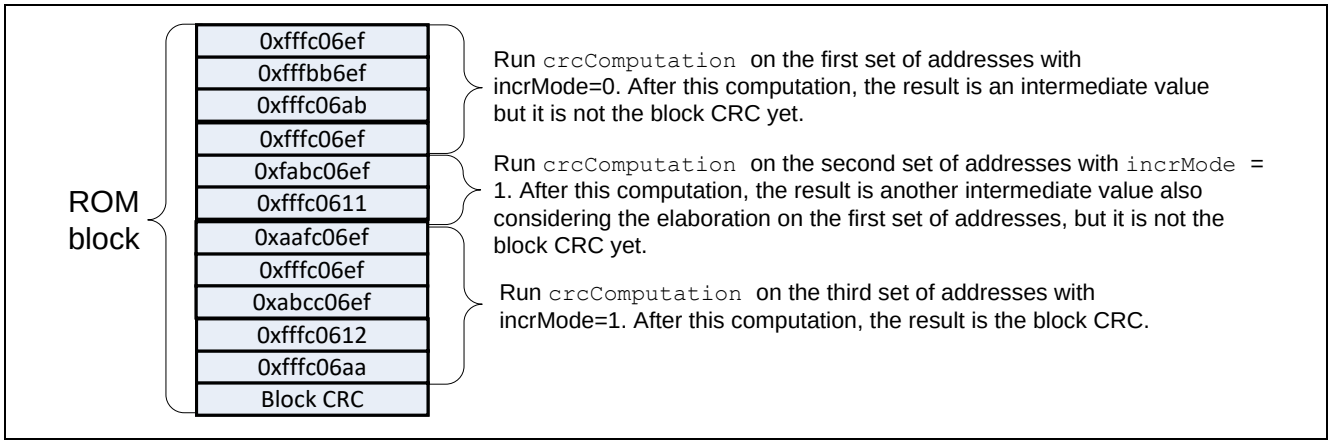

Figure 5-1: Incremental mode calculation.

5.4 Software Integration Rules

5.4.1 Code integration

To call the ROM test functions, use the following steps:

1. Include crc.h.
2. Define external variables for each CRC signatures generated by the IAR linker and placed in ROM.
   1. Define variable for input parameter of `crcHwSetuA.crcType`.
3. Define variables for input parameter of `crcComputation`:
   A. `checksumBegin`.
   B. `checksumEnd`.
   C. `incrMode`.
4. Define output variable in order to store the result of the `crcComputation`.

Refer to the example in section 5.4.2, which explains a case in which two ROM address ranges are tested.

5.4.2 Test flow and test result check

The recommended test flow is as follows:

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

Working example

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


  unsigned int type = 1;
  crcHwSetup(type);

  unsigned int checksumStart = 0x00000000;
  unsigned int checksumStop = 0x003FFFFB;
  unsigned int crcIncr = 0;
```

```
  uint16_t crcResult;
  crcResult = crcComputation(checksumStart, checksumStop, crcIncr);
  if(crcResult != __checksum){
    errorHandler();
  }
```

After the `crcComputation` function returns, a fault can be detected by checking the output value as shown in the example above. The `crcResult` achieved by the ROM software test is compared with `__checksum`, which is the reference value computed by the IAR linker.

Working 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 = 0x000FFFFB;                //1MB
  crcResult = crcComputation(checksumStart, checksumStop, crcIncr);

  crcIncr = 1;
  checksumStart = 0x00100000;
  checksumStop = 0x001FFFFB;                //1MB
  crcResult = crcComputation(checksumStart, checksumStop, crcIncr);

  crcIncr = 1;
  checksumStart = 0x00200000;
  checksumStop = 0x002FFFFB;                //1MB
  crcResult = crcComputation(checksumStart, checksumStop, crcIncr);

  crcIncr = 1;
  checksumStart = 0x00300000;
  checksumStop = 0x003FFFFB;                //1MB

  crcResult = crcComputation(checksumStart, checksumStop, crcIncr);
```

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

The above example shows how the CRC for a 4 MB block can be calculated with 4 cumulated runs of the `crcComputation` function.

Note: 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 that the 4 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 summarizes usage conditions.

Table 5-2 Usage conditions

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 after returning from interrupt handling. See reference document [REF.1] for 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 use the HW peripheral CRC calculator until the CRC calculation is completed. This is valid for any kind of software such as application software or any interrupt handlers.

5.5 Checksum Generation Using IAR Tools

The ROM test requires a reference checksum for each address 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 steps are as follows:

1. Provide information to the IAR linker as to where to place checksum values. Also, provide information about the 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.

The working example provided in the following section gives additional information on how to use the IAR tools to generate the required CRCs.

5.5.1 Example Checksum Generation with IAR Tools

Assume that the ROM test address range is 0x00000000 — 0x003FFFFF, and a checksum is required to be generated using the polynomial $x^{16}+x^{12}+ x^5+1$ (16-bit CRC-16CCITT).

Use the following steps:

1. Go to **Project > OptiI... > 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 = 0x003FFFFB.
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__      = 0x003FFFFF;
define region CHECKSUM_region = mem:[from __ICFEDIT_region_ROMuT_start__ to
__ICFEDIT_region_ROMuT_end__];
place at end of CHECKSUM_region { ro section .checksum };
```

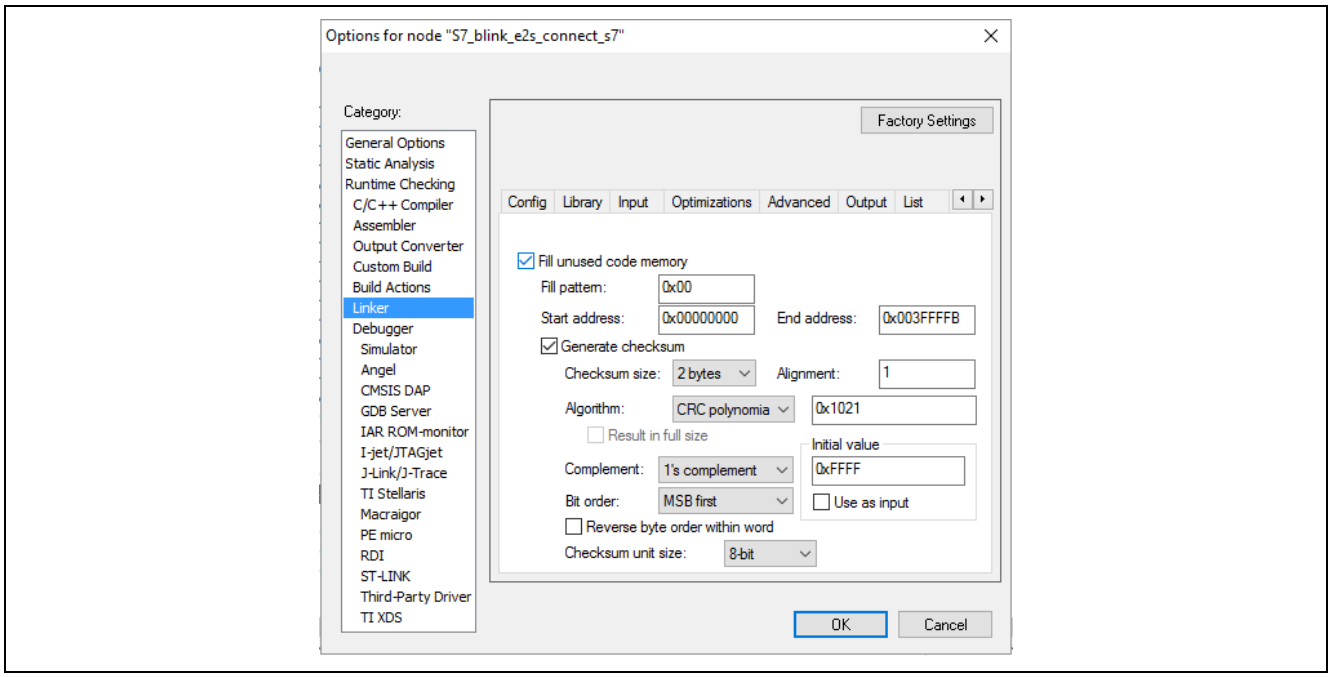

Figure 5-2: IAR environment options.

For more information about these commands, refer to reference document [REF.3].

5.6 Software Package Description

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

5.6.1 Identification and contents of package

The software package version is as follows:

- Revision 1.0.1
- File list.

Table 5-3 ROM Package and related MD5 signatures

File Name	MD5 Signature
6. `crc.c`	66d4c9c03eb5906ce5364f5d8b804858
`crc.h`	2d5cdb92e1acaf76bb3d5dd5f4c90c48
`S7G2_registers.h`	4a2dfba75ed595991e87d34b4fb4db74

6.1.1 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`: Initializes CRC module • `crcComputation`: 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	`S7G2_registers.h`	This file contains the definitions of the needed peripheral registers.

6.2 Resources Usage

Table 5-5 provides an overview of the memory resources used by the code.

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

Table 5-6 Execution time

Function	Execution time for a ROM block size of 4 Kb (clock cycles)	Execution time for a ROM block of 4 Kb at 240 MHz clock (µs)
`crcComputation`	57434	239

6.3 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 4 KB

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

In addition, regardless of the block size, the use of such a polynomial allows for the detection of an odd number of single bit errors, 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.

7. CAC Configuration Software

7.1 Test Objectives

The objective of the CAC configuration software is to configure the CAC. For safety applications, this software is used to:

- Select PCLKB as the measurement target clock for the CAC
- Select the sub-clock oscillator as a measurement reference clock for the CAC.

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

The CAC configuration software also enables the Synergy S7 oscillation stop detection circuit functionality. In case the main clock stops, this circuit is in charge of switching to the middle-speed on-chip oscillator, and generating an NMI interrupt.

7.2 Test Strategy

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

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

Note: The user must enable the sub-clock oscillator through the SOSCCR register (that is, SOSCCR.SOSTP = 0b. See document reference [REF.2]). Otherwise, the monitoring will not work.

The allowed 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:

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

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

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

The parameters are described in Table 7-1.

Table 7-1 Parameter description for CAULVR, CALLVR

Parameter	Description	Unit
PCLKB	Frequency of the peripheral module clock B	MHz
DC	Target diagnostic coverage. The user has to add a safety margin to the claimed DC, for example, 90% + 1% margin Renesas allows a DC range from, for example, 60 … 95%	%
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 according to the Measurement Target Clock Frequency Division Ration Select (TCSS) register	-
$CLKR_{DIV}$	Division according to the Measurement Reference Clock Frequency Division Ration Select (RCDS) register	-

In addition to the CAC function, the Synergy S7 has an oscillation stop detection circuit. If the main clock stops, the middle-speed on-chip oscillator is automatically used instead, and an NMI interrupt is generated. The user of this module must handle the NMI interrupt and check the NMISR.OSTST bit.

7.3 CAC Configuration Software API

The function signatures are as follows:

```
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 7-2 describes more details of the interface to the functions.

Table 7-2 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 reference to the formula parameters described in Table 7-1, the function parameters are mapped as follows:

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

7.4 Software Integration Rules

This section provides guidelines on integrating the CAC configuration software within the user project.

7.4.1 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 7.4.2 on how to use the diagnostic software.

7.4.2 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          (60000000) // PCLKB: 60MHz
#define DC (90) // Diagnostic Coverage: 90%
target_clk_div_t target_div = TAR_DIV_4;
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 hardware performs the clock monitoring, and so the software does not need to do anything during the periodic tests.

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

For configuring the ICU, it is necessary to set the ICU Event Link Setting Register (IELSRn) to the event signal number corresponding 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 previously mentioned 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 should 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 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 the following example:

[2] See Table 2-16 in reference document [1] 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.

7.5 Define Directives for Software Configuration

No specific directives are present for CAC configuration software.

7.6 Software Package Description

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

7.6.1 Identification and Contents of Package

The software package version is listed as follows:

- Revision 1.0.2
- File list.

Table 7-3 CAC configuration software package and related MD5 signatures

File Name	MD5 Signature
clock_monitor.c	1bdc9c2713d2a51bfd38a9724bb0be85
clock_monitor.h	78f648e238cbbdbfeaefc94beaf5de89
S7G2_registers.h	ac539ac998214ac9cba73eeef86985fd

7.6.2 Description of Design Files

Table 7-4 Design files

Table ID	File Name	Description
1	clock_monitor.h	This file contains the declaration of the ClockMonitor_Init function for monitoring initialization
2	clock_monitor.c	This file contains the definition of clock monitor function
3	S7G2_registers.h	This file contains the definitions of the needed peripheral registers

7.7 Resource Usage

Table 7-5 provides an overview of the memory resources used by the code.

Maximum stack usage is 152 bytes for both versions.

Table 7-5 Memory resources

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

Table 7-6 illustrates the execution time.

Table 7-6 Execution time

Function	Clock Cycle Count	Time measured (µs) at 240 MHz
Clock_monitor	3105	12,94

7.8 Requirements for Safety Relevant Applications

Refer to the Safety Manual [REF.4].

8. IWDT Management Software

8.1 Test Objectives

A watchdog is used to detect abnormal program execution. If a program is not running as expected, the Watchdog is not refreshed by software as required, and so, detects an error.

8.2 Test Strategy

The Independent Watchdog Timer (IWDT) module of the Synergy S7 is used for this purpose. The IWDT includes a windowing feature where 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 the OFS0 register whose settings are controlled by the user (see section 8.4.2 for a configuration example). A function is provided to be used after a reset, to decide if the IWDT has caused the reset.

8.3 IWDT Management Software APIs

The function signatures are as follows:

```
void IWDT_Init (void)
void IWDT_Kick (void)
bool IWDT_DidReset (void)
```

Table 8-1 describes more details of the interface to the functions.

Table 8-1 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 times out or is not refreshed correctly. This can be called after a reset, to decide if the Watchdog caused the reset.

8.4 Software Integration Rules

8.4.1 Code integration

The instructions to call the IWDT management software function are as follows:

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

Refer to the example in section 8.4.2, which explains how to use the diagnostic software.

8.4.2 Usage conditions

In order to configure the IWDT, it is necessary to set the OFS0 register correctly. 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` is stored at the OFS0 address at compile time, in order to configure the IWDT. In particular, the example enables the IWDT setting a time-out period of 1024 clock cycles at IWDTCLK/1 clock frequency, also counting during the sleep mode of the microcontroller. The example does not set any start/end of the Watchdog window, or configure a reset in case of Watchdog expiration.

The IWDT should be initialized as soon as possible, following a reset with a call to `IWDT_Init`:

```
/*Setup the Independent WDT.*/
```

```
IWDT_Init();
```

Then, the watchdog must be refreshed regularly to stop the Watchdog from timing out and performing a reset. If using windowing, the refresh must not just be regular, but also timed to match the specified window. A Watchdog refresh is called as follows:

```
/*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 reset by calling `IWDT_DidReset`:

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

8.5 Define Directives for Software Configuration

No specific directive are present for IWDT management software.

8.6 Software Package Description

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

8.6.1 Identification and Contents of Package

The software package version is listed as follows:

- Revision 1.0.1
- File list.

Table 8-2 IWDT package and related MD5 signatures

File Name	MD5 Signature
iwdt.c	c1ff175e73414577ebed6545d137963f
iwdt.h	136b2dd867a8551137d6ab80a85f4230
S7G2_registers.h	b64c20dfea0a3d0667d8fcf86e154b2e

8.6.2 Description of design files

Table 8-3 Design files

Table ID	File Name	Description
1	`iwdt.h`	This file contains the declaration of the functions: • `IWDT_Init`: Initializes the Independent Watchdog Timer. After calling this, the IWDT kick function must 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`: Refreshes 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	`S7G2_registers.h`	This file contains the definitions of the needed peripheral 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)	
`iwdt.o`	124	0	0
Total (bytes)	124	0	0

Table 8-5 illustrates the execution time for the specific functions.

Table 8-5 Execution time

Function	Clock cycles count	Time measured (µs) at 240 MHz
`IWDT_Init`	86	0,3
`IWDT_Kick`	80	0,3
`IWDT_DidReset`	96	0,4

8.8 Requirements for Safety Relevant Applications

Refer to the Safety Manual [REF.4].

9. LVD Configuration Software

9.1 Test Objectives

The Synergy S7 has a voltage detection circuit. This can be used to detect when the power supply voltage (Vcc) falls below a specified voltage.

9.2 Test Strategy

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

9.3 LVD Configuration Software APIs

The function signatures are as follows:

```
void VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL eVoltage)
```

Table 9-1 describes more details of the interface to the functions.

Table 9-1 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.

9.4 Software Integration Rules

9.4.1 Code integration

To call the LVD configuration software functions, use the following steps:

1. Include `voltage.h`.
2. Define variable for input parameter of `VoltageMonitor_t`:
   1. A. `eVoltage`

Refer to the example in section 9.4.2, which explains how to use the diagnostic software.

9.4.2 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 [REF.2] for further details.

Please 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_2_99);
```

If a low voltage condition is detected, an NMI interrupt is 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;
}
```

9.5 Define Directives for Software Configuration

No specific directives are present for LVD configuration software.

9.6 Software Package Description

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

9.6.1 Identification and Contents of Package

The software package version is listed as follows:

- Revision 1.0.1
- File list.

Table 9-2 LVD package and related MD5 signatures

File Name	MD5 Signature
S7G2_registers.h	b64c20dfea0a3d0667d8fcf86e154b2e
voltage.c	15e89e618e92fe6f2fb89bd995a820a8
voltage.h	072694c1d415b5bab51acc4464dff5b8

9.6.2 Description of design files

Table 9-3 Design files

Table ID	File Name	Description
1	voltage.h	This file contains the declaration of the functions for voltage monitor: • Bullet list item <table 1 unordered,t1u>
2	voltage.c	This file contains the definition of the two functions declared in the file voltage.h.
3	S7G2_registers.h	This file contains the definitions of the needed peripheral registers.

9.7 Resource Usage

Table 9-4 provides an overview of the memory resources used by the code.

Maximum stack usage is 0 bytes.

Table 9-4 Memory resources

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

Table 9-5 illustrates the execution time for the specific functions.

Table 9-5 Execution time

Function	Clock Cycle Count	Time measured (µs) at 240 MHz
VoltageMonitor_Init	30504	127

9.8 Requirements for Safety Relevant Applications

Refer to the Safety Manual [REF.4].

10. Requirements for Safety Relevant Applications

Refer to the Safety Manual [REF.4].

11. Appendix A - RAM Test Algorithms

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

11.1 Extended March C-

A March Test consists of a finite sequence of elements called March Elements, delimited by a pair of curly 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 11-1, using the preceding notation described in this section.

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

Figure 11-1: Extended March C- Algorithm.

The March C- algorithm detects address faults (AFs), stuck at faults (SAFs), transactional faults (TFs), and coupling faults (CFs). 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.

11.2 WALPAT

The WALPAT algorithm follows the process shown below:

```
Write 0 in all cells;
For i=0 to n-1
{ complement cell[i];
For j=0 to n-1, j != i
{ read cell[j]; }
read cell[i];
complement cell[i]; }
Write 1 in all cells;
For i=0 to n-1
{ complement cell[i];
For j=0 to n-1, j != i
{ read cell[j]; }
read cell[i];
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.

11.3 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 32-bit word access, the algorithm is repeated 6 times, and the following 6 different patterns have to be applied:

00000000000000000000000000000000
00000000000000001111111111111111
00000000111111110000000011111111
00001111000011110000111100001111
00110011001100110011001100110011
01010101010101010101010101010101

12. Appendix B - CPU Test Example

```
#include "coretest.h"

uint8_t s teps=1;
uint32_t re sult=0;
uint8_t forceFai l = 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();
  }
}
```

13. Appendix C – Pragmas report

Table 13-1 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 13-1 Pragma 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
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

Package	File	Code Version	Row	Code (Pragma)	LDRA Rule	MISRA Rule
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	25	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.h	1.0.1	27	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
RAM	testRAM.h	1.0.1	30	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
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
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 used. */	90S	D.4.6

Package	File	Code Version	Row	Code (Pragma)	LDRA Rule	MISRA Rule
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	67	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	68	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	82	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	84	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	86	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	89	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	91	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	93	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
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 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	110	/*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	111	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	112	/*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	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 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	120	/*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	123	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
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	126	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	129	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	130	/*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	131	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	132	/*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 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	141	/*LDRA INSPECTED 93 S Value is not of appropriate type. */	93S	R.10.1, R.10.3, R.10.4, R.10.5,

Package	File	Code Version	Row	Code (Pragma)	LDRA Rule	MISRA Rule
						R.11.1
CAC	clock_monitor.c	1.0.2	144	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	145	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	148	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	149	/*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	150	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	151	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.c	1.0.2	177	/*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		178	/*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	180	/*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		181	/*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	58	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6
CAC	clock_monitor.h	1.0.2	59	/*LDRA INSPECTED 90 S Basic type declaration used. */	90S	D.4.6

14. Document References

[REF.1] Cortex-M4 Devices – Generic User Guide, first release, 16/12/2010.
[REF.2] Synergy S7 User's Manual: Hardware, Rev. 1.30, January 2018 (Document Reference R01UM0001EU0130).
[REF.3] IAR C/C++ Development Guide Compiling and linking for Advanced RISC Machines Ltd's ARM Cores, Fifteenth edition, March 2015.
[REF.4] Safety Manual, ID=SAF_005_PIA003_S7.

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

Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Feb 7, 2017	-	Initial version
1.01	Feb 17, 2017	§4.4.1	Inserted clarification in “Reserving and placing the buffer” section.
		§8.4.2	Inserted an additional note on setting of “eVoltage” value.
1.02	Feb 23, 2017	§8.4.2	Inserted an additional note on LVD usage for LVD1SR register
1.03	Mar 02, 2017	§2.1	Updated C type implementation assumption
1.04	Mar 09, 2017	§3.5.1	Updated CPU latest release and MD5s.
		§3.6	Updated resource usage to align to “1.0.1”.
1.05	March 2018	§5.7	Updated memory resources used by the code crc.o
		§4.4.1	Inserted usage condition to reserve buffer area for RAM non destructive tests.
		§15	Updated reference document for the User’s Manual
1.06	July 17, 2018	References	Removed revision information from documentation
		3.6,4.7,7.7,8.7	Corrected Resources usage
1.07	Sep 27, 2018	All	Updated the functional safety version of the IAR Embedded Workbench.
		-	Removed “ADC12 Comparator Software” and TSN “Management Software” chapters.
		-	Updated latest release and MD5s of CPU,RAM,ROM,CAC,IWDT and LVD tests.
		All	Replaced “S7G2” Synergy name with “S7”.

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (http://www.renesas.com)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
   (1) artificial life support devices or systems
   (2) surgical implantations
   (3) healthcare intervention (e.g., excision, administration of medication, etc.)
   (4) any other purposes that pose a direct threat to human life

   Renesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas 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 applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.