Renesas Synergy™ Platform

S1 Series MCU Diagnostic Software User Guide

R11AN0260EU0100
Rev. 1.0
Dec 11, 2017

Introduction

Today, as automatic electronic controls systems continue to expand into many diverse applications, the requirement of reliability and safety are becoming an ever increasing factor in system design.

For example, the introduction of the IEC60730 safety standard for household appliances requires manufactures to design automatic electronic controls that ensure safe and reliable operation of their products.

The IEC60730 standard covers all aspects of product design but Annex H is of key importance for design of Microcontroller based control systems. This provides three software classifications for automatic electronic controls:

1. Class A: Control functions, which are not intended to be relied upon for the safety of the equipment.
   Examples: Room thermostats, humidity controls, lighting controls, timers, and switches.
2. Class B: Control functions, which are intended to prevent unsafe operation of the controlled equipment.
   Examples: Thermal cut-offs and door locks for laundry equipment.
3. Class C: Control functions, which are intended to prevent special hazards.
   Examples: Automatic burner controls and thermal cut-outs for closed.

Appliances such as washing machines, dishwashers, dryers, refrigerators, freezers, and cookers / stoves tend to fall under the classification of Class B.

This Application Note provides guidelines on flexible sample software routines to aid in complying with IEC60730 class B safety standards. These routines have been certified by VDE Test and Certification Institute GmbH and a copy of the Test Certificate is available in the download package for this Application Note (See Note below).

Although these routines were developed using IEC60730 compliance as a basis, they can be implemented in any system for self-testing of Renesas MCUs.

The software routines provided are to be used after reset and also during the program execution. The end user has the flexibility of how to integrate these routines into their overall system design, but this document and the accompanying sample code provide an example of how to do this.

It is worth noting that error-handling routines are as demanding to the user as interrupt handler routines. Since errors covered by the software routines are very critical (e.g., PC failure), and the correct functionality cannot be assured, it is strongly recommended that the user not only rely solely on software error handling, but also use hardware safety mechanisms, such as using the Independent Watchdog (iWDT).

Note. This document is based on the European Norm EN60335-1:2002/A1:2004 Annex R, in which the Norm IEC 60730-1 (EN60730-1:2000) is used in some points. The Annex R of the mentioned Norm has a single sheet that jumps to IEC 60730-1 for definitions, information and applicable paragraphs.

Target Device

Renesas Synergy S1 Series MCU

Contents

1. Tests 4
1.1 CPU 4
1.1.1 Software API 5
1.2 ROM 8
1.2.1 CRC32C Algorithm 8
1.2.2 CRC Software API 9
1.3 RAM 10
1.3.1 Algorithms 11
1.3.2 Software API 12
1.4 Clock 17
1.5 Independent Watchdog Timer 19
1.6 Voltage 20
1.7 ADC14 21
1.8 Temperature 22
1.9 Port Output Enable (POE) 24

2. Example Usage 25
2.1 CPU 25
2.1.1 Power-Up 25
2.1.2 Periodic 26
2.2 ROM 26
2.2.1 Power-Up 26
2.2.2 Periodic 26
2.3 RAM 27
2.3.1 Power-Up 27
2.3.2 Periodic 27
2.4 Clock 27
2.5 Independent Watchdog Timer 28
2.6 Voltage 30
2.7 ADC14 30
2.7.1 Power-Up 30
2.7.2 Periodic 30
2.8 Temperature 30
2.8.1 Power-Up 30
2.8.2 Periodic 31
2.9 POE 31

3. Benchmarking 31
3.1 Environment 31
3.2 Results 32

3.2.1 CPU 32
3.2.2 ROM 32
3.2.3 RAM 32
3.2.4 Clock 35
3.2.5 Independent Watchdog 36
3.2.6 Voltage 36
3.2.7 ADC14 36
3.2.8 Temperature 36
3.2.9 Port Output Enable 37

4. Additional Information 37
4.1 Reading an IO Pin State 37

1. Tests

1.1 CPU

This section describes CPU tests routines. Reference IEC 60730: 1999+A1:2003 Annex H – Table H.11.12.1 CPU.

The following CPU registers are tested: R0->R12, MSP, PSP, LR, APSR, BASEPRI and CONTROL.

The source file `cpu_test.c` provides implementation of the CPU test using C language and relies on assembly language function to access the registers (that is, CPU_Test_Control).

File `cpu_test_coupling.c` is also required to use the coupling test version of the General Purpose Registers. Coupling test relies on assembly language functions (TestGPRsCouplingStart_A,

- `TestGPRsCouplingR1_R3_A`
- `TestGPRsCouplingR4_R6_A`
- `TestGPRsCouplingR7_R9_A`
- `TestGPRsCouplingR10_R12_A`
- `TestGPRsCouplingR0_A`
- `TestGPRsCouplingStart_B`
- `TestGPRsCouplingR1_R3_B`
- `TestGPRsCouplingR4_R6_B`
- `TestGPRsCouplingR7_R9_B`
- `TestGPRsCouplingR10_R12_B`
- `TestGPRsCouplingR0_B`
- `TestGPRsCouplingEnd`

Alternatively, the assembly language functions, CPU_Test_General_Low, CPU_Test_General_High, are used to test GPRS registers.

The source file `cpu_test.h` provides the interface to the CPU tests. The file `S124_registers.h` includes definitions of S124 registers.

These tests are testing such fundamental aspects of the CPU operation; the API functions do not have return values to indicate the result of a test. Instead the user of these tests must provide an error handling function with the following declaration:-

```
extern void CPU_Test_ErrorHandler(void);
```

The CPU test jumps to this function if an error is detected. This function must not return.

All the test functions follow the rules of register preservation following a C function call. The user can call these functions like any normal C function without any additional responsibilities to save register values beforehand.

1.1.1 Software API

Table 1 Software API Source files

File name
cpu_test.h,
cpu_test_coupling.c, cpu_test.c
TestGPRsCouplingStart_A.asm TestGPRsCouplingR1_R3_A.asm TestGPRsCouplingR4_R6_A.asm TestGPRsCouplingR7_R9_A.asm TestGPRsCouplingR10_R12_A.asm TestGPRsCouplingR0_A.asm TestGPRsCouplingStart_B.asm TestGPRsCouplingR1_R3_B.asm TestGPRsCouplingR4_R6_B.asm TestGPRsCouplingR7_R9_B.asm TestGPRsCouplingR10_R12_B.asm TestGPRsCouplingR0_B.asm TestGPRsCouplingEnd.asm CPU_Test_Control.asm CPU_Test_General_Low.asm CPU_Test_General_High.asm

Syntax	
`void CPU_TestAll(void)`	
Description	

Runs through all the tests detailed below in the following order:
1. If using Coupling GPR Tests (*1. See below):
`CPU_Test_GPRsCouplingPartA`
`CPU_Test_GPRsCouplingPartB`
 	
If not using Coupling GPR test:
`CPU_Test_General_Low`
`CPU_Test_General_High`
 	
2. CPU_Test_Control
3. CPU_Test_PC
 	
It is the calling function's responsibility to ensure the processor is in Privileged Mode. If this function is called in unprivileged mode, the test fails; some of the register bits are not accessible in unprivileged mode. In addition, since in `CPU_Test_Control` function tests stack pointer registers (that is, MSP and PSP), disable stack pointer monitoring (MSPMPUCTL.ENABLE = 0, PSPMPUCTL.ENABLE = 0) before running `CPU_TestAll` function and restore its setting after a function return. The calling function also ensures no interrupts occur during this test. If an error is detected, then the external function `CPU_Test_ErrorHandler` will be called.
For a full description, see the individual tests.
 	
*1. A "#define USE_TEST_GPRS_COUPLING" in the code is used to select which functions are used to test the General Purpose Registers.	

Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void CPU_Test_GPRsCouplingPartA(void)`	
Description	
Tests general purpose registers R0 to R12. Coupling faults between the registers are detected. This is Part A of a complete GPR test. Use function `CPU_Test_GPRsCouplingPartB` to complete the test. It is the calling function's responsibility to ensure no interrupts occur during this test. If an error is detected, then the external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void CPU_Test_GPRsCouplingPartB(void)`	
Description	
Tests general purpose registers R0 to R12.Coupling faults between the registers are detected. This is Part B of a complete GPR test. Use function `CPU_Test_GPRsCouplingPartA` to complete the test. It is the calling function's responsibility to ensure no interrupts occur during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void CPU_Test_General_Low(void)`	
Description	
Test registers R1, R2, R3, R4, R5, R6 and R7. These are the general purpose registers. Registers are tested in pairs. For each pair of registers: 1. Write h'55555555 to both. 2. Read both and check they are equal. 3. Write h'AAAAAAAA to both. 4. Read both and check they are equal. It is the calling function's responsibility to disable exception during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void CPU_Test_General_High(void)`	
Description	
Test registers R8, R9, R10, R11 and R12. These are the general purpose registers. Registers are tested in pairs. For each pair of registers: 1. Write h'55555555 to both. 2. Read both and check they are equal. 3. Write h'AAAAAAAA to both. 4. Read both and check they are equal. It is the calling function's responsibility to disable exceptions during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void CPU_Test_Control(void)`	
Description	
Tests control registers PSP, MSP, LR, APSR, BASEPRI, CONTROL. This test assumes registers R1 to R4 are working. Generally the test procedure for each register is as follows: For each register:- 1. Write h'55555555 to. 2. Read back and check value equals h'55555555. 3. Write h'AAAAAAAA to. 4. Read back and check value equals h'AAAAAAAA. Note that there are some cases where restrictions on specific bits within a register do not allow for following this procedure. For these cases other test values have been chosen. It is the calling functions responsibility to ensure that the processor is in Privileged Mode. If this function is called in Unprivileged Mode the test will fail, as some of the register bits are not accessible in Unprivileged Mode. It is also the calling function's responsibility to disable exceptions during this test. Note: FAULTMASK and PRIMASK are not tested since this test requires exception disabled. Thus they are not activated during the test modifying FAULTMASK and PRIMASK. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void CPU_Test_PC (void)`	
Description	
This function provides the Program Counter (PC) register test. This provides a confidence check that the PC is working. It tests that the PC is working by calling a function that is located in its own section so that it can be located away from this function, so that when it is called more of the PC Register bits are required for it to work. So that this function can be sure that the function has actually been executed it returns the inverse of the supplied parameter. This return value is checked for correctness. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

1.2 ROM

This section describes the ROM / Flash memory test using CRC routines. Reference IEC 60730: 1999+A1:2003 Annex H – H2.19.4.1 CRC – Single Word.

A cyclic redundancy check (CRC) is a fault or error control technique that generates a single word or checksum to represent the contents of memory. A CRC checksum is the remainder after a binary division, with a no bit carry (XOR used instead of subtraction) of the message bit stream done by a predefined (short) bit stream of length n + 1, representing the coefficients of a polynomial with degree n. Before the division, *n* zeros are appended to the message stream. CRCs are popular because they are simple to implement in binary hardware and easy to analyze mathematically.

The ROM test can be achieved by generating a CRC value for the contents of the ROM and saving it.

During the memory self-test, the same CRC algorithm is used to generate another CRC value, which is compared with the saved CRC value. The technique recognizes all one-bit errors and a high percentage of multi-bit errors.

CRCs get complicated when you need to generate a CRC value to compare to CRC values produced by other CRC generators. Factors may change the resulting CRC value, even when the basic CRC algorithm is the same. The combination of the order data is supplied to the algorithm, the assumed bit order in any look-up table used, and the required order of the bits of the actual CRC value — are all factors. These complications have arisen because big and little endian systems were developed to work together and employ serial data transfers where bit order became important. It produces the same result as the IAR for Arm® toolchain does using the checksum option. If you are using the IAR for Arm toolchain to automatically insert a reference CRC into the ROM, the value can be compared directly with the one calculated.

1.2.1 CRC32C Algorithm

The Synergy S124 includes a CRC module that supports CRC-32C. This software configures the CRC module to produce a 32-bit CRC-32C:

- Polynomial = 0x1EDC6F41 (x32 + x28 + x27 + x26 + x25 + x23 + x22 + x20 + x19 + x18 + x14 + x13 + x11 + x10 + x9 + x8 + x6 + 1)
- Width = 32 bits
- Initial value = 0xFFFFFFFF
- XOR with h'FFFFFFFF is performed on the output CRC

1.2.2 CRC Software API

All software is written in ANSI C. The file `S124_registers.h` includes definitions of S124 registers. The functions listed calculate a CRC value and verify its correctness against a value stored in ROM.

Table 2 CRC Software API Source files

File name
crc.h, crc_verify.h
crc.c, CRC_Verify.c

These following functions are implemented in files `CRC_Verify.h` and `CRC_Verify.c`:

Syntax	
`bool CRC_Verify(const uint32_t ui32_NewCRCValue, const uint32_t ui32_AddrRefCRC)`	
Description	
This function compares a new CRC value to a reference CRC and supplies an address where reference CRC is stored.	
Input Parameters	
`uint32_t ui32_NewCRCValue`	Value of calculated new CRC value.
`uint32_t ui32_AddrRefCRC`	Address where 32 bit reference CRC value is stored.
Output Parameters	
`NONE`	N/A
Return Values	
`bool`	True = Passed, false = Failed

These following functions are implemented in files crc.h and crc.c:

Syntax	
void CRC_Init(void)	
Description	
Initializes the CRC module. This function must be called before any of the other CRC functions can be.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A
Syntax	
`uint32_t CRC_Calculate(uint32_t* pui32_Data, uint32_t ui32_Length)`	
Description	
This function calculates the CRC of a single specified memory area.	
Input Parameters	
uint32_t* pui32_Data	Pointer to start of memory to be tested.
uint32_t ui32_Length	Length of the data in long words.
Output Parameters	
`NONE`	N/A
Return Values	
`Uint32_t`	The 32-bit calculated CRC32C value.

The following functions are used when the memory area cannot simply be specified by a start address and length. They provide a way of adding memory areas in ranges/sections. This method can also be used if the function, CRC_Calculate, takes too long in a single function call.

void CRC_Start(void)	
Description	
Prepares the module for starting to receive data. Call this once prior to using function CRC_AddRange.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
void CRC_AddRange(uint32_t* pui32_Data, uint32_t ui32_Length)	
Description	
Use this function (rather than CRC_Calculate) to calculate the CRC on data with more than one address range. Call CRC_Start first, then CRC_AddRange, for each address range required, and then call CRC_Result to get the CRC value.	
Input Parameters	
uint32_t* pui32_Data	Pointer to start of memory range to be tested.
uint32_t ui32_Length	Length of the data in long words.
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
uint32_t CRC_Result(void)	
Description	
Calculates the CRC value for all memory ranges added using function CRC_AddRange, since CRC_Start was called.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
uint32_t	The calculated CRC32C value.

1.3 RAM

March tests are a family of tests that are well recognized as an effective way to test RAM.

A March test consists of a finite sequence of March elements A March element is a finite sequence of operations applied to every cell in the memory array before proceeding to the next cell.

In general, the more March elements the algorithm has the better its fault coverage, but at the expense of a slower execution time.

The algorithms are destructive (they do not preserve the current RAM values), but the supplied test functions provide a non-destructive option, so the memory's contents can be preserved. Preservation is achieved by copying the memory to a supplied buffer — before running the actual algorithm — then restoring the memory from the buffer at the end of the test. The API includes an option to automatically test the buffer, as well as the RAM test area.

The area of RAM undergoing testing cannot be used for anything else during the test. This issue makes testing RAM used in the stack difficult. To alleviate this issue, the API has functions that can be used to test the stack.

The following section introduces specific March tests. Following that is the specification of the software APIs.

1.3.1 Algorithms

March C

The March C algorithm (van de Goor 1991) consists of six March elements with a total of 10 operations. It detects the following faults:

1. Stuck At Faults (SAF)
   The logic value of a cell or a line is always 0 or 1.
2. Transition Faults (TF)
   A cell or a line that fails to undergo a 0→1 or a 1→0 transition.
3. Coupling Faults (CF)
   A write operation to one cell changes the content of a second cell.
4. Address Decoder Faults (AF)
   - Any fault that affects the address decoder:
   - With a certain address, no cell will be accessed.
   - A certain cell is never accessed.
   - With a certain address, multiple cells are accessed simultaneously.
   - A certain cell can be accessed by multiple addresses.

These are the 6 March elements:-

I. Write all zeros to array
II. Starting at lowest address, read zeros, write ones, increment up array bit by bit.
III. Starting at lowest address, read ones, write zeros, increment up array bit by bit.
IV. Starting at highest address, read zeros, write ones, decrement down array bit by bit.
V. Starting at highest address, read ones, write zeros, decrement down array bit by bit.
VI. Read all zeros from array.

March X

Note: This algorithm has not been implemented for the Synergy and is only presented here for information as it relates to the March X WOM version below.

The March X algorithm consists of four March elements with a total of six operations. It detects the following faults:

1. Stuck At Faults (SAF)
2. Transition Faults (TF)
3. Inversion Coupling Faults (Cfin)
4. Address Decoder Faults (AF)

These are the four March elements:-

I. Write all zeros to array
II. Starting at lowest address, read zeros, write ones, increment up array bit by bit.
III. Starting at highest address, read ones, write zeros, decrement down array bit by bit.
IV. Read all zeros from array.

March X (Word-Oriented Memory version)

The March X Word-Oriented Memory (WOM) algorithm has been created from a standard March X algorithm in two stages. First the standard March X is converted from using a single bit data pattern to using a data pattern equal to the memory access width. At this stage, the test is primarily detecting inter-word faults, including Address Decoder faults. The second stage adds two additional March elements. The first element uses a data pattern of alternating high/low bits; the second uses the inverse. These elements are added to detect intra-word coupling faults.

These are the six March elements:-

1. Write all zeros to an array
2. Starting at lowest address, read zeros, write ones, increment up array word by word.
3. Starting at highest address, read ones, write zeros, decrement down word by word.
4. Starting at lowest address, read zeros, write h'AAs, increment up array word by word.
5. Starting at highest address, read h'AAs, write h'55s, decrement down word by word.
6. Read all h'55s from the array.

1.3.2 Software API

Two implementations of the RAM tests are available;

1. Standard implementation.
2. Hardware (HW) implementation. This version uses the Data Operation Circuit (DOC) to help perform the tests.

Both implementations share the same core API, but the 'HW' implementation has some additional functions detailed in applicable sections.

March C API

This test can be configured to use 8, 16 or 32-bit RAM accesses.

This is achieved by #defining RAMTEST_MARCH_C_ACCESS_SIZE in the header file to be one of the following:

1. RAMTEST_MARCH_C_ACCESS_SIZE_8BIT
2. RAMTEST_MARCH_C_ACCESS_SIZE_16BIT
3. RAMTEST_MARCH_C_ACCESS_SIZE_32BIT

Sometimes limiting the maximum size of RAM that can be tested with a single function call can speed up the test, as well as reducing the stack and code size. It is done by limiting the size of the variable used to hold the number of 'words' that the test area contains. The 'word' size is the selected access width.

This is achieved by #defining RAMTEST_MARCH_C_MAX_WORDS in the header file to be one of the following:

1. RAMTEST_MARCH_C_MAX_WORDS_8BIT (Max words in test area is 0xFF)
2. RAMTEST_MARCH_C_MAX_WORDS_16BIT (Max words in test area is 0xFFFF)
3. RAMTEST_MARCH_C_MAX_WORDS_32BIT (Max words in test area is 0xFFFFFFFF)

Table 3 Source files

Standard	HW
ramtest_march_c.h,	ramtest_march_c.h, ramtest_march_HW.h
ramtest_march_c.c,	ramtest_march_c_HW.c, ramtest_march_HW.c.

The source is written in ANSI C and uses `S124_registers.h` to access peripheral registers.

Note: The API allows just a single word to be tested with a function call. However, to test coupling faults between words, it is important to use the functions to test a data range bigger than one word.

Declaration	
`bool RamTest_March_C(uint32_t ui32_StartAddr, uint32_t ui32_EndAddr, void* p_RAMSafe);`	
Description	
RAM memory test using March C (Goor 1991) algosithm.	
Input Parameters	
`uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This address must align with the selected memory access width.
`uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This address must align with the selected memory access width and be a value greater or equal to ui32_StartAddr.
`void* p_RAMSafe`	For a destructive memory test set to NULL. For a non-destructive memory test, it is set to the start of a buffer that is large enough to copy the contents of the test area into it and that is aligned with the selected memory access width.
Output Parameters	
`NONE`	N/A
Return Values	
`bool`	True = Test passed. False = Test or parameter check failed.

Declaration	
`bool RamTest_March_C_Extra(uint32_t ui32_StartAddr, uint32_t ui32_EndAddr, void* p_RAMSafe);`	
Description	
Non Destructive RAM memory test using March C (Goor 1991) algorithm. This function differs from the RamTest_March_C function, it tests the 'RAMSafe' buffer before using it. If the test of the 'RAMSafe' buffer fails, then the test is aborted and the function returns false.	
Input Parameters	
`uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This address must align with the selected memory access width.
`uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This address must align with the selected memory access width and be a value greater or equal to ui32_StartAddr.
`void* p_RAMSafe`	Set to the start of a buffer that is large enough to copy the contents of the test area into it and that is aligned with the selected memory access width.
Output Parameters	
`NONE`	N/A
Return Values	
`bool`	True = Test passed. False = Test or parameter check failed.

March X WOM API

This test can be configured to use 8, 16 or 32-bit RAM accesses.

This test is achieved by #defining RAMTEST_MARCH_X_WOM_ACCESS_SIZE in the header file to be one of the following:

1. RAMTEST_MARCH_ X_WOM_ACCESS_SIZE_8BIT
2. RAMTEST_MARCH_ X_WOM_ACCESS_SIZE_16BIT
3. RAMTEST_MARCH_ X_WOM_ACCESS_SIZE_32BIT

To speed up the test run time, you can choose to limit the maximum size of RAM that can be tested with a single function call. This is done by limiting the size of the variable used to hold the number of 'words' that the test area contains. The 'word' size is the same as the selected access width.

This is achieved by #defining RAMTEST_MARCH_ X_WOM_MAX_WORDS in the header file to be one of the following:

1. RAMTEST_MARCH_ X_WOM_MAX_WORDS_8BIT (Max words in test area is 0xFF)
2. RAMTEST_MARCH_ X_WOM_MAX_WORDS_16BIT (Max words in test area is 0xFFFF)
3. RAMTEST_MARCH_ X_WOM_MAX_WORDS_32BIT (Max words in test area is 0xFFFFFFFF)

Table 4 Source files

Standard	HW
ramtest_march_x_wom.h	ramtest_march_HW.h, ramtest_march_x_wom.h
ramtest_march_x_wom.c	ramtest_march_HW.c, ramtest_march_x_wom_HW.c

The source is written in ANSI C and uses `S124_registers.h` to access peripheral registers.

Note: The API allows just a single word to be tested with a function call. However, to test coupling faults between words, it is important to use the functions to test a data range bigger than one word.

Declaration	
`bool RamTest_March_X_WOM(uint32_t ui32_StartAddr, uint32_t ui32_EndAddr, void* p_RAMSafe);`	
Description	
RAM memory test based on March X algorithm converted for WOM.	
Input Parameters	
uint32_t ui32_StartAddr	Address of the first word of RAM to be tested. This address must align with the selected memory access width.
uint32_t ui32_EndAddr	Address of the last word of RAM to be tested. This address must align with the selected memory access width and be a value greater or equal to ui32_StartAddr.
void* p_RAMSafe	For a destructive memory test set to NULL. For a non-destructive memory test, set to the start of a buffer that is large enough to copy the contents of the test area into it and that is aligned with the selected memory access width.
Output Parameters	
NONE	N/A
Return Values	
bool	True = Test passed. False = Test or parameter check failed.

Declaration	
`bool RamTest_March_X_WOM_Extra(uint32_t ui32_StartAddr, uint32_t ui32_EndAddr, void* p_RAMSafe);`	
Description	
Non Destructive RAM memory test based on March X algorithm converted for WOM. This function differs from the RamTest_March_X_WOM function, it tests the 'RAMSafe' buffer before using it. If the test of the 'RAMSafe' buffer fails, then the test is aborted and the function returns false.	
Input Parameters	
uint32_t ui32_StartAddr	The address of the first word of RAM to be tested. This address must align with the selected memory access width.
uint32_t ui32_EndAddr	The address of the last word of RAM to be tested. This address must align with the selected memory access width and be a value greater or equal to ui32_StartAddr.
void* p_RAMSafe	Set to the start of a buffer that is large enough to copy the contents of the test area into it and that is aligned with the selected memory access width.
Output Parameters	
NONE	N/A
Return Values	
bool	True = Test passed. False = Test or parameter check failed.

March C and March X WOM HW Implementation specific API.

The 'HW' implementations of the March C and the March X WOM tests use the Data Operation Circuit (DOC) to help perform the tests. The DOC is used to compare values read back from RAM with expected values.

It is the user's responsibility to ensure that nothing else accesses the DOC during the RAM tests.

Declaration	
`void RamTest_March_HW_Init(void);`	
Description	
Initializes the hardware (DOC) used by the 'HW' implementations of the RAM tests. Call this function before using any other RAM Test function that uses a HW implementation.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`void`	N/A

Declaration	
`bool RamTest_March_HW_PreTest(void);`	
Description	
Checks whether the hardware (DOC) are functioning correctly before using. A quick functional test of the DOC is performed.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`bool`	True = Test passed. False = Test failed.

Declaration	
`bool RamTest_March_HW_Is_Init(void);`	
Description	
Checks if RamTest_March_HW_Init has been called. Used by specific RAM tests to check that the hardware has been initialized before trying to use it. A user does not have to use this function.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
bool	True = Test passed. False = Test or parameter check failed.

RAM Test Stack API

This API enables a RAM test on an area of RAM that includes the stack. The function that performs the RAM test requires a stack, so these functions re-locate the stack to a supplied new RAM area, allowing the original stack area to be tested. Three functions can be called, depending on which stack (Main or Process) is in the test area, or if both are.

It is the calling function's responsibility to ensure that the processor is in Privileged Mode. If this function is called in unprivileged mode the test will fail as some of the register bits are not accessible in unprivileged mode.

Note: The stack testing functions use one of the March RAM tests presented previously, passing it in as a function pointer. If using a test that requires initialization before use, it is the user's responsibility to ensure this initialization has been done before trying to use the test—by calling one of these functions.

Table 5 Source files

File name
ramtest_stack.h
ramtest_stack.c
StartBothTestAssembly.asm, StartMainTestAssembly.asm, StartProcTestAssembly.asm

Declaration	
`bool RamTest_Stack_Main(uint32_t ui32_StartAddr, uint32_t ui32_EndAddr, void* p_RAMSafe, uint32_t ui32_NewMSP, TEST_FUNC fpTest_Func);`	
Description	
RAM test of an area that includes the Main Stack. (but not the Process stack)	
Input Parameters	
`uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This address must be compatible with the requirements of the fpTest_Func.
`uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This address must be compatible with the requirements of the fpTest_Func.
`void* p_RAMSafe`	Set to the start of a buffer that is the same size as the test RAM area. This address must be compatible with the requirements of the fpTest_Func.
`uint32_t ui32_NewUSP`	New Stack pointer value for the Main stack to be relocated.
`TEST_FUNC fpTest_Func`	Function pointer of type TEST_FUNC to the actual memory test to be used. Typedef bool_t(*TEST_FUNC)(uint32_t, uint32_t, void*); For example 'RamTest_March_X_WOM'.
Output Parameters	
`NONE`	N/A
Return Values	
`bool`	True = Test passed. False = Test or parameter check failed.

```
bool RamTest_Stack_Main(uint32_t ui32_StartAddr,
                        uint32_t ui32_EndAddr,
                        void* p_RAMSafe,
                        uint32_t ui32_NewMSP,
                        TEST_FUNC fpTest_Func);
```

Declaration	
`bool RamTest_Stack_Proc(uint32_t ui32_StartAddr, uint32_t ui32_EndAddr, void* p_RAMSafe, uint32_t ui32_NewPSP, TEST_FUNC fpTest_Func);`	
Description	
RAM test of an area that includes the Process Stack. (but not the Main stack)	
Input Parameters	
`uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This address must be compatible with the requirements of the fpTest_Func.
`uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This address must be compatible with the requirements of the fpTest_Func.
`void* p_RAMSafe`	Set to the start of a buffer that is the same size as the test RAM area. This setting must be compatible with the requirements of the fpTest_Func.
`uint32_t ui32_NewPSP`	New Stack pointer value for the Process stack to be relocated to.
`fpTest_Func`	Function pointer of type TEST_FUNC to the actual memory test to be used. Typedef bool_t(*TEST_FUNC)(uint32_t, uint32_t, void*); For example 'RamTest_March_X_WOM'.
Output Parameters	
`NONE`	N/A
Return Values	
`bool`	True = Test passed. False = Test or parameter check failed.

```
bool RamTest_Stack_Proc(uint32_t ui32_StartAddr,
                        uint32_t ui32_EndAddr,
                        void* p_RAMSafe,
                        uint32_t ui32_NewPSP,
                        TEST_FUNC fpTest_Func);
```

Declaration	
`bool RamTest_Stacks(uint32_t ui32_StartAddr, uint32_t ui32_EndAddr, void* p_RAMSafe, uint32_t ui32_NewPSP, uint32_t ui32_NewMSP, TEST_FUNC fpTest_Func);`	
Description	
RAM test of an area that includes both the Stacks (that is, Main and Process stacks).	
Input Parameters	
uint32_t ui32_StartAddr	The address of the first word of RAM to be tested. This address must be compatible with the requirements of the fpTest_Func.
uint32_t ui32_EndAddr	The address of the last word of RAM to be tested. This address must be compatible with the requirements of the fpTest_Func.
void* p_RAMSafe	Set to the start of a buffer that is the same size as the test RAM area. This setting must be compatible with the requirements of the fpTest_Func.
uint32_t ui32_NewPSP	New Stack pointer value for the Process stack to be relocated to.
uint32_t ui32_NewMSP	New Stack pointer value for the Main stack to be relocated to.
TEST_FUNC fpTest_Func	Function pointer of type TEST_FUNC to the actual memory test to be used. Typedef bool_t(*TEST_FUNC)(const uint32_t, const uint32_t, void* const); For example 'RamTest_March_X_WOM'.
Output Parameters	
NONE	N/A
Return Values	
bool	True = Test passed. False = Test or parameter check failed.

1.4 Clock

The Renesas Synergy S124 MCU has a Clock Frequency Accuracy Measurement Circuit (CAC) used to monitor the Main clock frequency during run time.

Either one of the MAIN, SUB_CLOCK, HOCO, MOCO, LOCO, IWDTCLK, and PCLKB clocks, or an External clock on the CACREF pin, can be used as a reference clock source.

If using an external reference clock:

1. #define CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK in file `clock_monitor.h`.
2. Be sure to provide target and reference clocks frequency in Hz.

If using one of the internal source clocks:

1. Be sure CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK is not defined.
2. Be sure to select the reference clock (through ref_clock input parameter).
3. Be sure to provide target and reference clocks frequency in Hz.

If the frequency of the main clock deviates during runtime from a configured range two types of interrupt can be generated: frequency error interrupt or an overflow interrupt. The user of this module must enable these two kinds of interrupt and handle them. See Section 2.4 for an example of interrupt activation. The allowable frequency range can be adjusted using:

```
/*Percentage tolerance of main clock allowed before an error is reported.*/

#define CLOCK_TOLERANCE_PERCENT 10
```

In addition to the CAC function the Synergy S124 has an Oscillation Stop Detection Circuit. If the main clock stops, the Middle-Speed On-Chip oscillator will automatically be used instead and an NMI interrupt will be generated. The User of this module must handle the NMI interrupt and check the NMISR.OSTST bit.

Table 6 Clock Source files

File name
clock_monitor.h
clock_monitor.c

Software relies on `S124_registers.h` to access peripheral registers.

There are two versions of the ClockMonitor_Init function:

1. ClockMonitor_Init function if CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK is not defined.

Syntax	
`void ClockMonitor_Init(clock_source_t target_clock, clock_source_t ref_clock, uint32_t target_clock_frequency, uint32_t ref_clock_frequency, CLOCK_MONITOR_ERROR_CALL_BACK CallBack);`	
Description	
1. Start monitoring the target clock selected through target_clock input parameter using the CAC module and the reference clock selected through ref_clock input parameter. 2. Enables Oscillation Stop Detection and configures an NMI to be generated if detected.	
Input Parameters	
`clock_source_t target_clock`	The target clock to be monitored (one of the following: Main, Sub, HOCO, MOCO, LOCO, IWDTCLK, and PCLKB).
`clock_source_t ref_clock`	The reference clock used by CAC to monitor the target clock (one of the following: Main, Sub, HOCO, MOCO, LOCO, IWDTCLK, and PCLKB).
`uint32_t target_clock_frequency`	The target clock frequency in Hz
`uint32_t ref_clock_frequency`	The reference clock frequency in Hz.
`CLOCK_MONITOR_ERROR_CALL_BACK CallBack`	Function called if the main clock deviates from the allowable range.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

2. ClockMonitor_Init function, if CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK is defined.

Syntax
`void ClockMonitor_Init(clock_source_t target_clock, uint32_t MainClockFrequency, uint32_t ExternalRefClockFrequency, CLOCK_MONITOR_CACREF_PIN ePin, CLOCK_MONITOR_ERROR_CALL_BACK CallBack);`
Description
1. Start monitoring the target clock selected through target_clock input parameter using the CAC module and the CACREF pin as a reference clock. SW can select two possible pins: eCLOCK_MONITOR_CACREF_A (pin P204) and eCLOCK_MONITOR_CACREF_B (pin P400), and it is the user's responsibility to select the pin based on the board set-up. 2. Enables Oscillation Stop Detection and configures an NMI to be generated if detected.

Input Parameters	
clock_source_t target_clock	The target clock to be monitored. The clock shall be one among Main clock, Sub clock, HOCO clock, MOCO clock, LOCO clock, IWDTCLK clock and PCLKB clock.
uint32_t MainClockFrequency	Main clock expected frequency in Hz.
uint32_t ExternalRefClockFrequency	External reference clock frequency in Hz.
CLOCK_MONITOR_CACREF_PIN ePin	The pin to use for CACREF.
CLOCK_MONITOR_ERROR_CALL_BACK CallBack	Function to be called if the main clock deviates from the allowable range or if this function fails.
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

1.5 Independent Watchdog Timer

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

The Independent Watchdog Timer (iWDT) module of the Synergy S124 is used to detect abnormal execution. It includes a windowing feature so that the refresh must happen within a specified 'window' rather than just before a specified time. It can be configured to generate an internal reset or a NMI interrupt if an error is detected. Configuring the iWDT can be done through the OFS0 register whose settings are controlled by the user (see Section 2.5). A function to be used after a reset is provided to decide whether the IWDT has caused the reset. The test module relies on the `S124_registers.h` file to access to peripheral registers.

Table 7 Independent Watchdog Timer Source files

File name
iwdt.h
iwdt.c

Syntax	
void IWDT_Init (void)	
Description	
Initialize the iWDT. After calling, the IWDT_kick function is called at the correct time to prevent a watchdog timer error. Note: If configured to produce an interrupt, it will be the Non Maskable Interrupt (NMI). This NMI must be handled by user code to check the NMISR.IWDTST flag.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
None	N/A

Syntax
void IWDT_Kick(void)
Description
Refresh the watchdog timer count.

Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`bool IWDT_DidReset(void)`	
Description	
Returns true if the iWDT has timed out or not been refreshed correctly. Can be called after a reset to decide if the watchdog timer caused the reset.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
`bool`	True if watchdog timer has timed out, otherwise false.

1.6 Voltage

The Synergy S124 MCU has a Voltage Detection Circuit that can be used to detect when the power supply voltage (VCC) falls below a specified voltage. The sample code demonstrates using the Voltage Detection Circuit 1 to generate a NMI interrupt when VCC drops below a specified level. The hardware can also generate a reset, but this behavior is not supported in the sample code. This module relies on `S124_registers.h` file to access peripheral registers.

Table 8 Voltage Source files

File name
voltage.h
voltage.c

Syntax	
`void VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL eVoltage)`	
Description	
Initialize and start voltage monitoring. An NMI is generated if the VCC falls below the specified voltage. Note: The NMI must be handled by user code, which must check the NMISR.LVDST flag. The voltage threshold eVoltage is set to a value lower than the nominal VCC one.	
Input Parameters	
VOLTAGE_MONITOR_LEVEL eVoltage	The specified low voltage level. For details, see declaration of enumerated type VOLTAGE_MONITOR_LEVEL in `voltage.h`.
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

1.7 ADC14

The ADC14 has a diagnostic mode used to test the ADC. The diagnostic mode can be configured so a test is performed every time the ADC is used normally for a conversion. The diagnostic reference voltage and the expected result are automatically rotated between zero, half scale, and full scale. The diagnostic software provides two automatic conversions (zero and full scale). The module relies on `S124_registers.h` file to access peripheral registers.

Table 9 ADC14 Source files

File name
test_ADC14.h
test_ADC14.c

Syntax	
`void Test_ADC14_Init (void)`	
Description	
Initialize the ADC14 module. This must be called before using any other ADC functions.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`bool Test_ADC14_Wait (void)`	
Description	
This function waits while two ADC conversions are made by ADC14 module. This test does not preserve ADC configuration and is therefore suitable as a power on test rather than as a run-time periodic test.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`bool`	True = Test passed. False = test failed.

Syntax	
`void Test_ADC14_Start(ADC14_ERROR_CALL_BACK Callback)`	
Description	
Set up the ADC module so diagnostic tests are performed each time the ADC is used. The diagnostic reference voltage is automatically rotated (Zero, half VREF, and VREH). User code must call the Test_ADC14_CheckResult function, either periodically or by following each ADC completion to check the diagnostic result.	
Input Parameters	
`ADC14_ERROR_CALL_BACK Callback`	Function to call if an error is detected. Note: This function is only called if the Test_ADC14_CheckResult is called, with parameter bCallErrorHandler set to true.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`bool Test_ADC14_CheckResult(bool bCallErrorHandler)`	
Description	
Check that ADC diagnostic result is as expected. This must be called after Test_ADC14_Start and then be called periodically or whenever an ADC conversion completes. Note: The actual result is allowed to be with a certain tolerance of the expected result. See ADC14_TOLERANCE in test_ad12.c for details.	
Input Parameters	
`bool bCallErrorHandler`	Set true to call the error callback function supplied to `Test_ADC14_Start` function; otherwise, it is false.
Output Parameters	
`NONE`	N/A
Return Values	
`bool`	True = Test passed. False = Test failed.

1.8 Temperature

Synergy S124 has a Temperature Sensor module to monitor MCU temperature. The ADC14 module is also required with the Temperature Sensor. The module relies on the `S124_registers.h` file to access peripheral registers.

Table 10 Temperature Source files

File name
temperature.h
temperature.c

Syntax	

`void Temperature_Init(uint16_t Temperature_ADC_Value_Min,`
`uint16_t Temperature_ADC_Value_Max,`
`TEMPERATURE_ERROR_CALL_BACK Error_callback)`	

Description	
Initialize the Temperature Sensor and enable the ADC14 module. Specify an allowed temperature range in terms of ADC14 output values. After calling this function, the Temperature_Start function is called periodically to perform an ADC conversion on the Temperature Sensor output, then the remaining functions are used to check the result.	
Input Parameters	
`uint16_t Temperature_ADC_Value_Min`	Specifies the minimum value the ADC14 should output when reading the temperature sensor.
`uint16_t Temperature_ADC_Value_Max`	Specifies the maximum value the ADC14 should output when reading the temperature sensor.
`TEMPERATURE_ERROR_CALL_BACK Error_callback`	This function is called by function Temperature_CheckResult if the temperature (ADC14 Value) is outside the specified allowable range.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void Temperature_Start(void);`	
Description	
Start an ADC conversion to read the temperature. Uses the ADC14 module, destroying its current settings. It is the user's responsibility to ensure this behavior is not a problem. Following this function, use the Temperature_Read_Wait or Temperature_CheckResult function.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void Temperature_Wait_Finish (void);`	
Description	
This function blocks until a temperature conversion, started by Temperature_Start, has completed.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A
Syntax	
`uint16_t Temperature_Read_Wait (void);`	
Description	
This function blocks until a temperature conversion, started by Temperature_Start, has completed and then returns the ADC14 value.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
`uint16_t`	ADC14 output value

`bool Temperature_CheckResult(bool bCallErrorHandler)`	
Description	
This function blocks until a temperature conversion, started by Temperature_Start, has completed and then checks if the ADC14 value is within the range specified in Temperatire_Init.	
Input Parameters	
`bCallErrorHandler`	Set true to get the callback registered in Temperature_Init called if the temperature falls outside the specified limits; otherwise, set false.
Output Parameters	
NONE	N/A
Return Values	
`bool`	True: Result falls within specified limits. False: Result falls outside specified limits.

1.9 Port Output Enable (POE)

The Port Output Enable for the GPT (POEG) module can be used to place General PWM Timer (GPT) output pins in the output disable state in one of the following ways: input level detection of the GTETRG pins; Output-disable request from the GPT; Comparator interrupt request detection; Oscillation stop detection of the clock generation circuit; Register settings.

This software demonstrates the setting of certain pins into the high impedance state when a rising edge on GTETRGn (n = A, B) input pin is detected or when oscillation stop is detected. Note that the user must configure the GTETRGn pin within POE_init function as well as enable handling interrupts generated by the POE. See Section 2.9 for more details about enabling the handling of POE interrupt. The software module relies on `S124_registers.h` header file to access peripheral registers.

Table 11 Port Output Enable Source files

File name
POE.h, GPT.h
POE.c, GPT.c

Syntax	
`void POE_Init(POE_CALL_BACK Callback, POE_group_t group);`	
Description	
This software configures the POE: 1. To put the GTIOCA and GTIOCB pins of all GPT channels in the high impedance state if a rising edge on the GTETRGn (n = A, B) input pin is detected. In particular the SW configures pin P100 to be used as GTETRGA pin. An interrupt is also generated. Note that user shall ensure the configuration of GTETRGA pin which strictly depends on the board where the microcontroller is placed. 2. To put the GTIOCA and GTIOCB pins of all GPT channels in the high impedance state if Oscillation Stop is detected.	
Input Parameters	
`POE_CALL_BACK Callback`	Function to call if a rising edge on the GTETRGn input pin is detected.
`POE_group_t group`	The POEG group to be set by the initialization function.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void POE_ClearFlags_ga(void);`	
Description	
For POEG group A, this function clears the Port Input Detection Flag, the Detection Flag for GPT or ACMPHS Output-Disable Request, the Oscillation Stop Detection Flag and Software stop flag. This will release the pins from the high impedance state.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void POE_ClearFlags_gb(void);`	
Description	
For POEG group B, this function clears the Port Input Detection Flag, the Detection Flag for GPT or ACMPHS Output-Disable Request, the Oscillation Stop Detection Flag and Software stop flag. This will release the pins from the high impedance state.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void GPT_Init(POE_group_t group);`	
Description	
This function configures the GPT in order to associate GTIOCA and GTIOCB pins of each GPT channel to the POE group stated by input parameter 'group'.	
Input Parameters	
POE_group_t group	POE group to associate the GTP channels.
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

2. Example Usage

This section provides some useful suggestions about how to apply the released software.

The testing can be split into two parts:

1. **Power-Up Tests**. These are tests run once following a reset. They should be run as soon as possible, but especially if start-up time is important. It may be permissible to run some initialization code before running all the tests so, for example, a faster main clock can be selected.
2. **Periodic Tests**. These are tests that are run regularly throughout normal program operation. This user guide does not determine how often a particular test should be run. How often periodic tests are scheduled is up to the user and depends on how their application is structured.

The following sections provide an example of how each test type should be used.

2.1 CPU

If a fault is detected by any of the CPU tests, then a user supplied function called `CPU_Test_ErrorHandler` is called. Since any error in the CPU is very serious, the aim of this function should be to get to a safe state as soon as possible, where software execution is not relied on.

2.1.1 Power-Up

All the CPU tests should be run as soon as possible following a reset.

Note: The function must be called before the device is put in Unprivileged mode. The function `CPU_Test_All` can be used to automatically run all the CPU tests.

2.1.2 Periodic

To test the CPU periodically, the function CPU_Test_All can be used—as it is for the power-up tests—to automatically run all CPU tests. Alternatively, to reduce the amount of testing done in a single function call, the user can choose to call each individual CPU test function in turn each time the CPU periodic test is scheduled.

2.2 ROM

The ROM is tested by calculating a CRC value (CRC32C) of its contents and comparing with a reference CRC value that must be added to a specific location in the ROM that is not included in the CRC calculation.

The IAR for Arm Toolchain can be used to calculate and add a CRC value to the built file at a location specified by the user. This process can be done via a dialog in IAR. See Figure 1 Adding Reference CRC.

The CRC module must be initialized before its use with a call to the CRC_Init function.

Ensure that all ROM sections used are included in the CRC calculation the IAR and the CRC Test code both use, so that the results match.

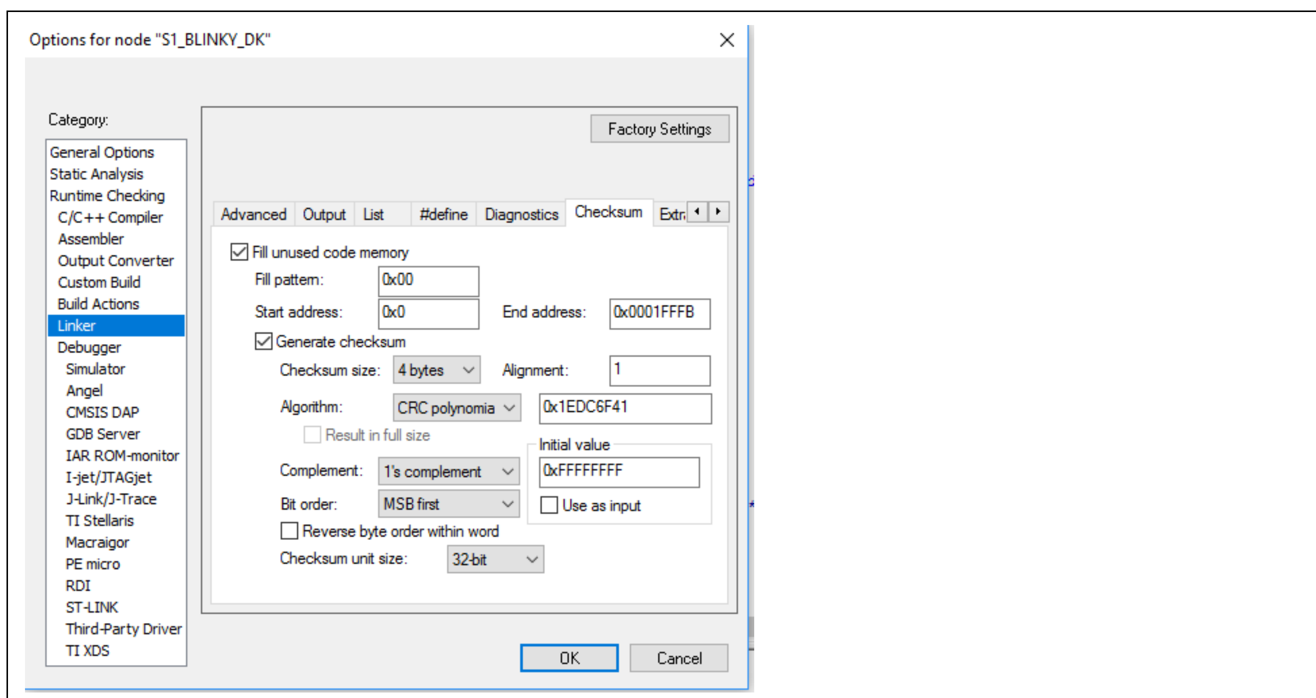


Figure 1 Adding Reference CRC

2.2.1 Power-Up

All the ROM memory used must be tested at power-up.

If the ROM area is one contiguous block, then the `CRC_Calculate` function can be used to calculate and return a calculated CRC value.

If the ROM used is not in one contiguous block then the following procedure must be used.

1. Call CRC_Start.
2. Call CRC_AddRange for each area of memory to be included in the CRC calculation.
3. Call CRC_Result to get the calculated CRC value.

The calculated CRC value can be compared with the reference CRC value stored in the ROM using the `CRC_Verify` function.

It is a user's responsibility to ensure all ROM areas used by their project are included in the CRC calculations.

2.2.2 Periodic

It is suggested the periodic testing of ROM be done using the `CRC_AddRange` method, even if the ROM is contiguous, as this method allows the CRC value to be calculated in sections, so that no single function call takes too long. Follow the procedure specified for power-up tests. Ensure that each address range is the smallest possible, so a call to `CRC_AddRange` does not take too long.

2.3 RAM

Note: RAM are to be tested may change dramatically, depending on your project's memory map.

If using 'HW' versions of RAM Tests (where DOC is used), then call the `RamTest_March_HW_Init` function prior to running the test. The following #define in file `ramtest_march_HW.h` makes this selection:

```
#define USE_HW_VERSION_OF_RAM_TESTS
```

When testing RAM, it is important to remember the following:

1. RAM being tested cannot be used for anything else including the current stack.
2. Any non-destructive test requires a RAM buffer where memory contents can be safely copied to and restored from.
3. Any test of the stack requires a RAM buffer where the stack can be relocated.
4. There are two stacks, Main and Process. It is the current stack that must be relocated before being used.
5. To relocate the stack the device must be in supervisor mode. The device automatically enters default mode when handling an interrupt.

2.3.1 Power-Up

At power-up a full destructive test can be performed on the RAM other than the Stack. The Stack must be tested with a non-destructive test. If startup time is very important, it might be possible to fine-tune this time, so only the area of Stack used before the power-up RAM test is performed by using the slower non-destructive test, with the rest of the Stack tested with a destructive test.

2.3.2 Periodic

All periodic tests must be non-destructive.

It is assumed that the periodic tests are called from an interrupt handler and therefore the device is in privileged mode.

2.4 Clock

Main clock monitoring is set-up with a single function call to `ClockMonitor_Init`. There are two versions of this file, depending on the choice between using an external or internal reference clock, as decided by the following #define:

```
#define CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK
```

For example:

```
#ifdef CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK

#define MAIN_CLOCK_FREQUENCY_HZ         (16000000) // 16 MHz
#define EXTERNAL_REF_CLOCK_FREQUENCY_HZ (15000) // 15kHz

    ClockMonitor_Init(MAIN,
MAIN_CLOCK_FREQUENCY_HZ,EXTERNAL_REF_CLOCK_FREQUENCY_HZ,eCLOCK_MONITOR_CACREF_A,CAC_Er
ror_Detected_Loop);

#else

#define TARGET_CLOCK_FREQUENCY_HZ         (16000000) // 16 MHz
#define REFERENCE_CLOCK_FREQUENCY_HZ      (15000) // 15kHz

    ClockMonitor_Init(MAIN,            IWDTCLK,            TARGET_CLOCK_FREQUENCY_HZ,
REFERENCE_CLOCK_FREQUENCY_HZ, CAC_Error_Detected_Loop);
    /*NOTE: The IWDTCLK clock must be enabled before starting the clock monitoring.*/

#endif
```

This define can be called as soon as the main clock has been configured and the iWDT has been enabled (see section 1.5).

Clock monitoring is performed by hardware, so nothing needs to be done by software during the periodic tests.

To enable interrupt generation by the CAC, both Interrupt Controller Unit (ICU) and Arm Cortex-M0+ Nested Vectored Interrupt Controller (NVIC) should be configured to handle it.

To configure the ICU, it is necessary to set the ICU Event Link Setting Register (IELSRn) to the event signal number correspondent to the CAC frequency error interrupt (CAC_FERRI = 0x34) and CAC overflow (CAC_OVFI = 0x36). In particular, it is necessary to configure one IELSR register so that it is linked to CAC events mentioned previously:

```
IELSRn.IELS = 0x34; // (CAC_FERRI)

IELSRn.IELS = 0x36; // (CAC_OVFI)
```

In addition, to enable the Cortex-M0+ NVIC to handle the CAC interrupts, the following instructions are set:

```
NVIC_EnableIRQ(CAC_FREQUENCY_ERROR_IRQn);

NVIC_EnableIRQ(CAC_OVERFLOW_IRQn);
```

Where CAC_FREQUENCY_ERROR_IRQn and CAC_OVERFLOW_IRQn are the IRQ number that are defined by the user[1].

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

```
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.

2.5 Independent Watchdog Timer

To configure the iWDT, it is necessary to coherently set the OFS0 register. The following code sets the value 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)
```

[1] For details on IRQ numbers, see Table 2-11 of "Cortex-M0+ Devices: Generic User Guide", second release, 18 December 2012.

```
#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 OFS0_IWDT__CFG value is stored at the OFS0 address at compile time to configure the iWDT. In particular, the example enables the iWDT to set a time-out period of 1024 clock cycles at IWDTCLK/1 clock frequency, counting also during sleep mode of the microcontroller. The example does not set any start/end of watchdog window and configures a reset in case there is a 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();
```

Afterwards, the watchdog timer must be refreshed regularly to prevent WDT timeouts and initiating a reset. Note, if using windowing, the refresh must not just be sufficiently regular, but also timed to match the specified window. A watchdog refresh is called by calling the following:

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

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

If the WDT has been configured to initiate a reset on error detection, then after a reset, the code should check whether the IWDT caused the reset by calling `IWDT_DidReset`:

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

2.6 Voltage

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

Note: Set 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 3.10 V.

```
VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL_3_10);
```

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

2.7 ADC14

The ADC14 module has a built in diagnostic mode that allows various reference voltages to be tested.

To account for allowed inaccuracies, the expected result is allowed to fall within a tolerance defined using

```
#define ADC14_TOLERANCE 8
```

This value is set as the maximum absolute accuracy that the ADC is rated to. In a calibrated system this tolerance could be tightened.

The ADC14 Test module must be initialized with a call to Test_ADC14_Init.

2.7.1 Power-Up

At power up the ADC14 module can be tested using the Test_ADC14_Wait function. This function waits until two AD conversions are performed, one using reference voltage of VREF, and the other 0 V. The return value of this function must be checked for the result.

2.7.2 Periodic

The periodic testing should start with a single call to Test_ADC14_Start. Following that the ADC14 module will perform a reference conversion each time it is used. The reference voltage is rotated between 0 V, VREF/2 and VREF. The result of these reference conversions must be checked periodically using a call to Test_ADC14_CheckResult.

2.8 Temperature

When testing the MCU temperature the ADC14 module is used. If the user's code also has the ADC14 monitoring analog pins, it is important to carefully consider resource-sharing the ADC14 module.

The temperature sensor must be initialized before it is used with a call to `Temperature_Init`. This function must be passed using the allowable range of temperatures expressed in terms of the ADC14 output. For details, see the Synergy S124 MCU Hardware Manual.

```
/*Temperature Sensor*/
Temperature_Init(TEMPERATURE_ADC_MIN,
                 TEMPERATURE_ADC_MAX,
                 Temperature_Test_Failure);
```

2.8.1 Power-Up

The temperature test procedure at power-up is the same as the procedure used for periodic tests.

2.8.2 Periodic

Periodically, the use of the ADC14 module must be driven by the temperature sensor. To make a temperature reading, the user calls this function:

```
/*Start ADC reading temperature sensor output.*/
  Temperature_Start();
```

The result can be checked against the allowable range supplied in the Temperature_Init function with a call to:

```
/*The registered Error callback will be called if there is an error. */
  Temperature_CheckResult(TRUE);
```

To avoid the periodic test from blocking the software application too long, it can be arranged that each time the periodic test is scheduled; it checks the temperature test result started on the prior scheduled test, then starts a new conversion.

The user's code can use functions `Temperature_Is_Finished` or `Temperature_Wait_Finish` to determine when the application can resume using the ADC14 to read analog pins.

2.9 POE

The POE initialization and start-up can be made using the following call:

```
  POE_Init(POE_Event_Detected, GROUP_A);
```

The POEG group choice is up to the user. The user must carefully study the description of POE_Init and consult the Synergy S124 Hardware Manual to determine if the POE sample configuration meets user's system requirements. Depending on the pins used in the user's system, the `POE.c` file may need to be adapted for the desired behavior.

To enable interrupt generation by the POE, both the Interrupt Controller Unit (ICU) and Cortex-M0+ Nested Vectored Interrupt Controller (NVIC) must be configured to handle it.

To configure the ICU, it is necessary to set the ICU Event Link Setting Register (IELSRn) to the event signal number corresponding to the POE group events (POEG_GROUP0 = 0x40, POEG_GROUP1 = 0x41). In particular, it is necessary to configure one IELSR register so that it is linked to the aforementioned CAC events:

```
IELSRn.IELS = 0x40; // (POEG_GROUP0)

IELSRn.IELS = 0x41; // (POEG_GROUP1)
```

In addition, to enable the Arm Cortex-M0+ NVIC to handle the CAC interrupts, the following instructions must be set:

```
NVIC_EnableIRQ(POEG0_EVENT_IRQn);

NVIC_EnableIRQ(POEG1_EVENT_IRQn);
```

Where `POEG0_EVENT_IRQn, POEG1_EVENT_IRQn` are the IRQ numbers that must be defined by the user[2].

3. Benchmarking

3.1 Environment

Development board	DK-S124M v3.0
Clock	EXTAL = 16 MHz, HOCO = 64 MHz, ICLK = 32 MHz, PCLKB = 32 MHz, PCLKD = 64 MHz
MCU	R7FS1247H2A01CBD
Tool chain	IAR Embedded Workbench for Arm , Functional Safety, v.7.40.6.9816
In-circuit debugger	Arm Debug + ETM connector and SEGGER J-link® on board

[2] For details on IRQ numbers, see Table 2-11 of "Cortex-M0+ Devices: Generic User Guide," second release, 18 December 2012.

Build option:

General option	Target = Renesas R7FS1247H
Complier Settings	Language = C C-dialect = C-99 Language Conformance = Standard with IAR extension Plain 'char' is: Unsigned Floating-point semantics: Strict conformance
Optimization Level	None

3.2 Results

3.2.1 CPU

Table 12 CPU test results

Measurement	Result Non-CouplingTest	Result Coulping Test
ROM usage (bytes)	2454	2518
RAM usage (bytes)	0	0
Stack usage (bytes)	24	24
Clock Cycle Count – CPU_TestAll	397	1311
Time Measured (µs) @32 MHz – CPU_TestAll	12.40	40,96

3.2.2 ROM

Table 13 Test results for CRC32C

Measurement	Result
ROM usage (bytes)	172
RAM usage (bytes)	0
Stack usage (bytes)	20
Clock Cycle Count – CRC_Init	82
Time Measured (µs) @32 MHz – CRC_Init	2.56
Clock Cycle Count – CRC_Calculate (ROM overall, that is, 128 kB)	426073
Time Measured (ms) @32 MHz – CRC_Calculate (128 kB)	13.314
Clock Cycle Count – CRC_Calculate (1 kB)	3417
Time Measured (µs) @32 MHz – CRC_Calculate (1 kB)	106.78
Clock Cycle Count – CRC_Calculate (4 kB)	13401
Time Measured (µs) @32 MHz – CRC_Calculate (4 kB)	418.78
Clock Cycle Count – CRC_Calculate (16 kB)	53337
Time Measured (ms) @32 MHz – CRC_Calculate (16 kB)	1,67
Clock Cycle Count – CRC_Verify	57
Time Measured (us) @32 MHz – CRC_Verify	1.78

3.2.3 RAM

The tests were executed in 8 and 32-bit access width configurations. The 32-bit word limit was always used; it was found that using a smaller limit did not improve performance.

March C

Table 14 March C test results (8-bit access, 32-bit word limit)

Measurement	Normal Result	HW (DOC) Result
ROM usage (bytes)	562	550
RAM usage (bytes)	0	0
Stack usage (bytes)	96	92
Stack usage Extra (bytes)	128	124

Measurement			Normal Result	HW (DOC) Result
Clock cycle count	Destructive	1024 bytes	1939833	1928661
		500 bytes	947374	941966
		100 bytes	189742	188734
	Non-destructive	1024 bytes	1958304	1947132
		500 bytes	956413	951005
		100 bytes	191581	190573
	Extra	1024 bytes	3898249	3875905
		500 bytes	1903896	1893080
		100 bytes	381400	379384
Time Measured (ms) @ 32 MHz	Destructive	1024 bytes	60,61978	60,27066
		500 bytes	29,60544	29,43644
		100 bytes	5,929438	5,897938
	Non-destructive	1024 bytes	61,197	60,84788
		500 bytes	29,88791	29,71891
		100 bytes	5,986906	5,955406
	Extra	1024 bytes	121,8203	121,122
		500 bytes	59,49675	59,15875
		100 bytes	11,91875	11,85575

Table 15 March C test results (32-bit access, 32-bit word limit)

Measurement			Normal Result	HW (DOC) Result
ROM usage (bytes)			600	622
RAM usage (bytes)			0	0
Stack usage (bytes)			88	88
Stack usage Extra (bytes)			120	120
Clock cycle count	Destructive	1024 bytes	1040683	1171585
		500 bytes	508299	572260
		100 bytes	101899	114760
	Non-destructive	1024 bytes	1047386	1178288
		500 bytes	511596	575557
		100 bytes	102596	115457
	Extra	1024 bytes	2088090	2349894
		500 bytes	1019916	1147838
		100 bytes	204516	230238
Time Measured (ms) @ 32 MHz	Destructive	1024 bytes	32,52134	36,61203
		500 bytes	15,88434	17,88313
		100 bytes	3,184344	3,58625
	Non-destructive	1024 bytes	32,73081	36,8215
		500 bytes	15,98738	17,98616
		100 bytes	3,206125	3,608031
	Extra	1024 bytes	65,25281	73,43419
		500 bytes	31,87238	35,86994
		100 bytes	6,391125	7,194938

March X WOM

Table 16 March X WOM test results (8-bit access, 32-bit word limit).

Measurement			Normal Result	HW (DOC) Result
ROM usage (bytes)			428	434
RAM usage (bytes)			0	0
Stack usage (bytes)			0	0
Stack usage Extra (bytes)			0	0
Clock cycle count	Destructive	1024 bytes	96613	89505
		500 bytes	47354	43914
		100 bytes	9722	9082
	Non-destructive	1024 bytes	115084	107976
		500 bytes	56393	52953
		100 bytes	11561	10921
	Extra	1024 bytes	211809	197593
		500 bytes	103856	96976
		100 bytes	21360	20080
Time Measured (ms) @ 32 MHz	Destructive	1024 bytes	3,019156	2,797031
		500 bytes	1,479813	1,372313
		100 bytes	0,303813	0,283813
	Non-destructive	1024 bytes	3,596375	3,37425
		500 bytes	1,762281	1,654781
		100 bytes	0,361281	0,341281
	Extra	1024 bytes	6,619031	6,174781
		500 bytes	3,2455	3,0305
		100 bytes	0,6675	0,6275

Table 17 March X WOM test results (32-bit access, 32-bit word limit)

Measurement			Normal Result	HW (DOC) Result
ROM usage (bytes)			428	434
RAM usage (bytes)			0	0
Stack usage (bytes)			0	0
Stack usage Extra (bytes)			0	0
Clock cycle count	Destructive	1024 bytes	96613	89505
		500 bytes	47354	43914
		100 bytes	9722	9082
	Non-destructive	1024 bytes	115084	107976
		500 bytes	56393	52953
		100 bytes	11561	10921
	Extra	1024 bytes	211809	197593
		500 bytes	103856	96976
		100 bytes	21360	20080
Time Measured (ms) @ 240 MHz	Destructive	1024 bytes	3,019156	2,797031
		500 bytes	1,479813	1,372313
		100 bytes	0,303813	0,283813
	Non-destructive	1024 bytes	3,596375	3,37425
		500 bytes	1,762281	1,654781
		100 bytes	0,361281	0,341281
	Extra	1024 bytes	6,619031	6,174781
		500 bytes	3,2455	3,0305
		100 bytes	0,6675	0,6275

Stack Test

Note: Whether normal or hardware implementation, the results are the same, because the stack test does not rely on hardware.

Table 18 Stack test results

Measurement	Result
ROM usage (bytes)	364
RAM usage (bytes)	33
Stack usage (bytes) – RamTest_Stack_Main	12
Stack usage (bytes) – RamTest_Stack_Proc	12
Stack usage (bytes) – RamTest_Stacks	12
Clock Cycle Count – RamTest_Stack_Main (only stack relocation)	135
Time Measured (us) @32 MHz – RamTest_Stack_Main	4.21
Clock Cycle Count – RamTest_Stack_Proc (only stack relocation)	136
Time Measured (us) @32 MHz – RamTest_Stack_Proc	4.25
Clock Cycle Count – RamTest_Stacks (only stack relocation)	170
Time Measured (us) @32 MHz – RamTest_Stacks	5.31

HW supporting functions

The following table lists the ROM and RAM resources needed to support using the peripherals DOC.

Table 19 HW supporting function results

Measurement	Result
ROM usage (bytes)	248
RAM usage (bytes)	0
Stack usage (bytes)	0
Clock Cycle Count – RamTest_March_HW_Init	71
Time Measured (us) @32 MHz – RamTest_March_HW_Init	2.22
Clock Cycle Count – RamTest_March_HW_PreTest	452
Time Measured (us) @32 MHz – RamTest_March_HW_PreTest	14.13
Clock Cycle Count – RamTest_March_HW_Is_Init	57
Time Measured (us) @32 MHz – RamTest_March_HW_Is_Init	1.78

3.2.4 Clock

Table 20 Clock test results

Measurement	Internal Reference Clock Result	External Reference Clock Result
ROM usage (bytes)	608	1028
RAM usage (bytes)	4	4
Stack usage (bytes)	56	56
Clock Cycle Count	10375	2238
Time measured (us) @ 32 MHz	324.22	69.94

3.2.5 Independent Watchdog

Table 21 Independent Watchdog test results

Measurement	Result
ROM usage (bytes)	132
RAM usage (bytes)	0
Stack usage (bytes)	0
Clock Cycles Count – IWDT_Init	58
Time measured (us) @ 32 MHz – IWDT_Init	1.81
Clock Cycles Count – IWDT_Kick	50
Time measured (us) @ 32 MHz – IWDT_Kick	1.56
Clock Cycles Count – IWDT_DidReset	68
Time measured (us) @ 32 MHz – IWDT_DidReset	2.13

3.2.6 Voltage

Table 22 Voltage Monitoring test results

Measurement	Result
ROM usage (bytes)	200
RAM usage (bytes)	0
Stack usage (bytes)	0
Clock Cycles Count	27728
Time measured (us) @ 32 MHz	866.5

3.2.7 ADC14

Table 23 12-bit ADC Converter test results

Measurement	Result
ROM usage (bytes)	524
RAM usage (bytes)	4
Stack usage (bytes)	24
Clock Cycles Count – Test_ADC14_Init	65
Time measured (us) @ 32 MHz – Test_ADC14_Init	2.03
Clock Cycles Count – Test_ADC14_Wait	354
Time measured (us) @ 32 MHz – Test_ADC14_Wait	11.06
Clock Cycles Count – Test_ADC14_Start	82
Time measured (us) @ 32 MHz- Test_ADC14_Start	2.56
Clock Cycles Count (clock cycles) – Test_ADC14_CheckResult	120
Time measured (us) @ 32 MHz – Test_ADC14_CheckResult	3.75

3.2.8 Temperature

Table 24 Temperature sensor test results

Measurement	Result
ROM usage (bytes)	316
RAM usage (bytes)	8
Stack usage (bytes)	36
Clock Cycles Count – Temperature_Init	87
Time measured (us) @ 32 MHz – Temperature_Init	2.72
Clock Cycles Count – Temperature_Start	121
Time measured (us) @ 32 MHz – Temperature_Start	3.78
Clock Cycles Count – Temperature_CheckResult	134
Time measured (us) @ 32 MHz – Temperature_CheckResult	4.19

3.2.9 Port Output Enable

Table 25 Port Output Enable test results

Measurement	Result
ROM usage (bytes)	1068
RAM usage (bytes)	4
Stack usage (bytes)	0
Clock Cycles Count – GPT_init	628
Time measured (us) @ 240MHz – GPT_init	19.63
Clock Cycles Count – POE_Init	267
Time measured (us) @ 240MHz – POE_Init	8.34

4. Additional Information

4.1 Reading an IO Pin State

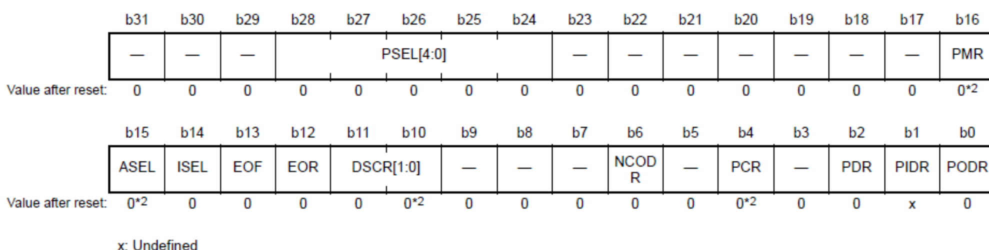
The actual value of an IO pin can always be read by reading the corresponding pin's Port mn Pin Function Select Register (PmnPFS). For details, see section 16.2.5 of Synergy S1 Hardware Manual:

20.2.5 Port mn Pin Function Select Register (PmnPFS/PmnPFS_HA/PmnPFS_BY) (m = 0 to 9, A, B; n = 00 to 15)

Address(es): PFS.P000PFS 4004 0800h to PFS.P015PFS 4004 083Ch, PFS.P100PFS 4004 0840h to PFS.P115PFS 4004 087Ch,
PFS.P200PFS 4004 0880h to PFS.P215PFS 4004 08BCh, PFS.P300PFS 4004 08C0h to PFS.P315PFS 4004 08FCh,
PFS.P400PFS 4004 0900h to PFS.P415PFS 4004 093Ch, PFS.P500PFS 4004 0940h to PFS.P515PFS 4004 097Ch,
PFS.P600PFS 4004 0980h to PFS.P615PFS 4004 09BCh, PFS.P700PFS 4004 09C0h to PFS.P715PFS 4004 09FCh,
PFS.P800PFS 4004 0A00h to PFS.P815PFS 4004 0A3Ch, PFS.P900PFS 4004 0A40h to PFS.P915PFS 4004 0A7Ch,
PFS.PA00PFS 4004 0A80h to PFS.PA15PFS 4004 0ABCh, PFS.PB00PFS 4004 0AC0h to PFS.PB07PFS 4004 0ADCh

PFS.P000PFS_HA 4004 0802h to PFS.P015PFS_HA 4004 083Eh, PFS.P100PFS_HA 4004 0842h to PFS.P115PFS_HA 4004 087Eh,
PFS.P200PFS_HA 4004 0882h to PFS.P215PFS_HA 4004 08BEh, PFS.P300PFS_HA 4004 08C2h to PFS.P315PFS_HA 4004 08FEh,
PFS.P400PFS_HA 4004 0902h to PFS.P415PFS_HA 4004 093Eh, PFS.P500PFS_HA 4004 0942h to PFS.P515PFS_HA 4004 097Eh,
PFS.P600PFS_HA 4004 0982h to PFS.P615PFS_HA 4004 09BEh, PFS.P700PFS_HA 4004 09C2h to PFS.P715PFS_HA 4004 09FEh,
PFS.P800PFS_HA 4004 0A02h to PFS.P815PFS_HA 4004 0A3Eh, PFS.P900PFS_HA 4004 0A42h to PFS.P915PFS_HA 4004 0A7Eh,
PFS.PA00PFS_HA 4004 0A82h to PFS.PA15PFS_HA 4004 0ABEh, PFS.PB00PFS_HA 4004 0AC2h to PFS.PB07PFS_HA 4004 0ADEh

PFS.P000PFS_BY 4004 0803h to PFS.P015PFS_BY 4004 083Fh, PFS.P100PFS_BY 4004 0843h to PFS.P115PFS_BY 4004 087Fh,
PFS.P200PFS_BY 4004 0883h to PFS.P215PFS_BY 4004 08BFh, PFS.P300PFS_BY 4004 08C3h to PFS.P315PFS_BY 4004 08FFh,
PFS.P400PFS_BY 4004 0903h to PFS.P415PFS_BY 4004 093Fh, PFS.P500PFS_BY 4004 0943h to PFS.P515PFS_BY 4004 097Fh,
PFS.P600PFS_BY 4004 0983h to PFS.P615PFS_BY 4004 09BFh, PFS.P700PFS_BY 4004 09C3h to PFS.P715PFS_BY 4004 09FFh,
PFS.P800PFS_BY 4004 0A03h to PFS.P815PFS_BY 4004 0A3Fh, PFS.P900PFS_BY 4004 0A43h to PFS.P915PFS_BY 4004 0A7Fh,
PFS.PA00PFS_BY 4004 0A83h to PFS.PA15PFS_BY 4004 0ABFh, PFS.PB00PFS_BY 4004 0AC3h to PFS.PB07PFS_BY 4004 0ADFh



	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	—	—	—	PSEL[4:0]					—	—	—	—	—	—	—	PMR
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0*2

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	ASEL	ISEL	EOF	EOR	DSCR[1:0]		—	—	—	NCODR	—	PCR	—	PDR	PIDR	PODR
Value after reset:	0*2	0	0	0	0	0*2	0	0	0	0	0	0*2	0	0	x	0

x: Undefined

Bit	Symbol	Bit name	Description	R/W
b0	PODR	Port Output Data	0: Output low 1: Output high.	R/W
b1	PIDR	Port Input Data	0: Input low 1: Input high.	R

Figure 2 PmnPFS Register

Website and Support

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

Technical Contact Details:

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

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

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Dec 11, 2017	—	Initial release

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

1. Precaution against Electrostatic Discharge (ESD)

A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.

2. Processing at power-on

The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.

3. Input of signal during power-off state

Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.

4. Handling of unused pins

Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.

5. Clock signals

After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.

6. Voltage application waveform at input pin

Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between VIL (Max.) and VIH (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between VIL (Max.) and VIH (Min.).

7. Prohibition of access to reserved addresses

Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.

8. Differences between products

Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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