Renesas Synergy™ Platform

IEC 60730 Self-Test Code for Synergy S5D9 MCU

R20AN0453EU0100
Rev.1.00
Jun 26, 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 IEC 60730 safety standard for household appliances requires manufactures to design automatic electronic controls that ensure safe and reliable operation of their products.

The IEC 60730 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 will tend to fall under Class B.

This Application Note provides guidelines on how to use flexible sample software routines to assist with compliance with IEC 60730 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 1 below).

Although these routines were developed using IEC 60730 compliance as a basis, they can be implemented in any system for self-testing 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 the definition of error handling routines is the responsibility of the user as well as interrupt handler routines. The errors that are covered by the software routines are very critical (for example, PC failure) and the correct software functionality cannot be assured. Therefore, we strongly recommend that you not only rely on software error handling, but to also use hardware safety mechanisms. An example would be the use of the Independent Watchdog Timer (iWDT).

Note: 1. 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 contains just a single sheet that jumps to the IEC 60730-1 for definitions, information and applicable paragraphs.

Target Device

Renesas Synergy S5D9 Group MCU.

Contents

1. Tests

1.1 CPU

This section describes CPU test 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. In addition, these FPU registers are also tested: S0-S31, CPACR, FPCCR, FPCAR, FPSCR, and FPDSCR.

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

The `cpu_test.c` source file relies also on FPU_Control assembly language function to access the FPU control registers. File `fpu_test_coupling.c` is also required if using the coupling test version of the FPU extension registers:

- TestFPUCouplingStart_A
- TestFPUCouplingS0_S3_A
- TestFPUCouplingS4_S7_A
- TestFPUCouplingS8_S11_A
- TestFPUCouplingS12_S15_A
- TestFPUCouplingS16_S19_A
- TestFPUCouplingS20_S23_A
- TestFPUCouplingS24_S27_A
- TestFPUCouplingS28_S31_A
- TestFPUCouplingStart_B
- TestFPUCouplingS0_S3_B
- TestFPUCouplingS4_S7_B
- TestFPUCouplingS8_S11_B
- TestFPUCouplingS12_S15_B
- TestFPUCouplingS16_S19_B
- TestFPUCouplingS20_S23_B
- TestFPUCouplingS24_S27_B
- TestFPUCouplingS28_S31_B
- TestFPUCouplingEnd

Alternatively, the FPU_Exten assembly language function is used to test FPU extension registers

The source file `cpu_test.h` provides the interface to the CPU tests. The file `S5D9_registers.h` includes definitions of S5D9 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 will jump 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. Therefore the user can call these functions like any normal C function without any additional responsibilities for saving register values beforehand.

1.1.1 Software API

Table 1 Software API Source files

File name
cpu_test.h, fpu_test.h
cpu_test_coupling.c, cpu_test.c, fpu_test_coupling.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, fpu_control.asm, TestFPUCouplingStart_A.asm, TestFPUCouplingS0_S3_A.asm, TestFPUCouplingS4_S7_A.asm, TestFPUCouplingS8_S11_A.asm, TestFPUCouplingS12_S15_A.asm, TestFPUCouplingS16_S19_A.asm, TestFPUCouplingS20_S23_A.asm, TestFPUCouplingS24_S27_A.asm, TestFPUCouplingS28_S31_A.asm, TestFPUCouplingStart_B.asm, TestFPUCouplingS0_S3_B.asm, TestFPUCouplingS4_S7_B.asm, TestFPUCouplingS8_S11_B.asm, TestFPUCouplingS12_S15_B.asm, TestFPUCouplingS16_S19_B.asm, TestFPUCouplingS20_S23_B.asm, TestFPUCouplingS24_S27_B.asm, TestFPUCouplingS28_S31_B.asm, TestFPUCouplingEnd.asm, fpu_exten.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 4. If using Coupling FPU extension registers Tests (*2. see below): FPU_Test_FPUCouplingPartA FPU_Test_FPUCouplingPartB If not using Coupling GPR test: FPU_Exten 5. FPU_Control 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. In addition, since the CPU_Test_Control function tests stack pointer registers (that is, MSP and PSP), then it is necessary to disable stack pointer monitoring (MSPMPUCTL.ENABLE = 0. PSPMPUCTL.ENABLE = 0) before running CPU_TestAll function and restore its setting after function return. It is also 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. See the individual tests for a full description. *1. A `#define USE_TEST_GPRS_COUPLING` in the code is used to select which functions will be used to test the General Purpose Registers. *2 A `#define USE_TEST_FPU_COUPLING` in the code is used to select which functions will be used to test the FPU extension 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 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 general-purpose registers R1, R2, R3, R4, R5, R6, and R7. Registers are tested in pairs. For each pair of registers: 1. Write 0x55555555 to both. 2. Read both and check they are equal. 3. Write 0xAAAAAAAA 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 general-purpose registers R8, R9, R10, R11, and R12. These are the registers. Registers are tested in pairs. For each pair of registers: 1. Write 0x55555555 to both. 2. Read both and check they are equal. 3. Write 0xAAAAAAAA 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, and 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 0x55555555 to both. 2. Read back and check value equals 0x55555555. 3. Write 0xAAAAAAAA to both. 4. Read back and check value equals 0xAAAAAAAA. Note however that there are some cases where restrictions on specific bits within a register do not allow this procedure. For these cases other test values have been chosen. 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. 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 exceptions be 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	
Tests the Program Counter (PC) register. 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. Therefore, 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

Syntax	
`void FPU_Test_FPUCouplingPartA (void)`	
Description	
Tests FPU extension registers S0 to S31. Coupling faults between the registers are detected. This is Part A of a complete FPU extension register test, use function FPU_Test_FPUCouplingPartB 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 FPU_Test_FPUCouplingPartB(void)`	
Description	
Tests FPU extension registers S0 to S31. Coupling faults between the registers are detected. This is Part B of a complete FPU extension register test, use function FPU_Test_FPUCouplingPartA 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 FPU_Exten(void)`	
Description	
Test FPU extension registers S0 to S31. Registers are tested in pairs. For each pair of registers: 1. Write 0x55555555 to both. 2. Read both and check they are equal. 3. Write 0xAAAAAAAA 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 FPU _Control(void)`	
Description	
Tests FPU control registers CPACR, FPCCR, FPCAR, FPSCR and FPDSCR. This test assumes registers R1 to R10 are working. Generally the test procedure for each register is as follows: For each register: 1. Write 0x55555555 to. 2. Read back and check value equals 0x55555555. 3. Write 0xAAAAAAAA to. 4. Read back and check value equals 0xAAAAAAAA. Note however that there are some cases where restrictions on specific bits within a register do not allow this procedure. For these cases other test values have been chosen. 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. It is also 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

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.

CRC is a fault/error control technique which generates a single word or checksum to represent the contents of memory. A CRC checksum is the remainder of a binary division with no bit carry (XOR used instead of subtraction) of the message bit stream, by a predefined (short) bit stream of length n + 1. This represents 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 are 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.

The complicated part of using CRCs is if you need to generate a CRC value that will then be compared with other CRC values produced by other CRC generators. This proves difficult because there are a number of factors that can change the resulting CRC value even if the basic CRC algorithm is the same. This includes the combination of the order that the 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. This complication has arisen because big and little endian systems were developed to work together that employed serial data transfers where bit order became important. This implementation will produce the same result as the IAR for ARM toolchain does using the Checksum option. Therefore 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 S5D9 includes a CRC module with support for the CRC32C. This software set the CRC module to produce a 32-bit CRC32C:

- Polynomial = 0x1EDC6F41 ($x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{20} + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10} + x^{9} + x^{8} + x^{6} + 1$)
- Width = 32 bits
- Initial value = 0xFFFFFFFF
- XOR with 0xFFFFFFFF is performed on the output CRC

1.2.2 CRC Software API

All software is written in ANSI C.

The file `S5D9_registers.h` includes definitions of S5D9 registers.

The functions in the remainder of this section are used to 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	
Compares a new CRC value with a reference CRC by supplying 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	
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 can also be used if function `CRC_Calculate` takes too long in a single function call.

Syntax	
`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 made up of 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 the 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 of testing 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 consists of, the better its fault coverage will be but at the expense of a slower execution time.

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

The area of RAM being tested cannot be used for anything else while it is being tested. This makes the testing of RAM used for the stack particularly difficult. To help with this problem the API includes functions which can be used for testing the stack.

The following section introduces the specific March Tests with the specification of the software APIs.

1.3.1 Algorithms

(1) March C

The March C algorithm (van de Goor 1991) consists of six March elements with a total of ten 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 six March elements:

1. Write all zeros to array.
2. Starting at lowest address, read zeros, write ones, increment up array bit by bit.
3. Starting at lowest address, read ones, write zeros, increment up array bit by bit.
4. Starting at highest address, read zeros, write ones, decrement down array bit by bit.
5. Starting at highest address, read ones, write zeros, decrement down array bit by bit.
6. Read all zeros from array.

(2) 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:

1. Write all zeros to array.
2. Starting at lowest address, read zeros, write ones, increment up array bit by bit.
3. Starting at highest address, read ones, write zeros, decrement down array bit by bit.
4. Read all zeros from array.

(3) 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 is to add an additional two March elements. The first uses a data pattern of alternating high/low bits then the second uses the inverse. The addition of these elements is to detect intra-word coupling faults.

These are the six March elements:

1. Write all zeros to 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 0xAAs, increment up array word by word.
5. Starting at highest address, read 0xAAs, write 0x55s, decrement down word by word.
6. Read all 0x55s from 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) and optionally a DMAC channel to help perform the tests.

Both implementations share the same core API but the HW implementation has some additional functions. Please see details below.

(1) 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 the test up as well as reducing stack and code size. 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 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 `S5D9_registers.h` to access peripheral registers.

Note: The API allows just a single word to be tested with a function call. However, for coupling faults to be tested 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) algorithm.	
Input Parameters	
`uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
`uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This must be aligned 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_C_Extra(uint32_t ui32_StartAddr, uint32_t ui32_EndAddr, void* p_RAMSafe);`	
Description	
Nondestructive RAM memory test using March C (Goor 1991) algorithm. This function differs from the RamTest_March_C function by testing the 'RAMSafe' buffer before using it. If the test of the 'RAMSafe' buffer fails then the test will be aborted and the function will return false.	
Input Parameters	
`uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
`uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This must be aligned 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.

(2) March X WOM API

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

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

- RAMTEST_MARCH_ X_WOM_ACCESS_SIZE_8BIT
- RAMTEST_MARCH_ X_WOM_ACCESS_SIZE_16BIT
- RAMTEST_MARCH_ X_WOM_ACCESS_SIZE_32BIT

In order to speed up the run time of the test 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:

- RAMTEST_MARCH_ X_WOM_MAX_WORDS_8BIT (Max words in test area is 0xFF)
- RAMTEST_MARCH_ X_WOM_MAX_WORDS_16BIT (Max words in test area is 0xFFFF)
- 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 `S5D9_registers.h` to access peripheral registers.

Note: The API allows just a single word to be tested with a function call. However, for coupling faults to be tested 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 must be aligned with the selected memory access width.
`uint32_t ui32_EndAddr`	Address of the last word of RAM to be tested. This must be aligned 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 by testing the RAMSafe buffer before using it. If the test of the RAMSafe buffer fails then the test will be aborted and the function will return false.	
Input Parameters	
`uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
`uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This must be aligned 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.

(3) 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) and optionally a DMAC channel to help perform the tests. The DMAC is used to initialize the RAM under test and 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 or chosen DMAC channel during the RAM tests.

The optional use of the DMAC is controlled using the following #defines in file `ramtest_march_HW.h`:

#define	Meaning if #defined
'RAMTEST_USE_DMAC'	The DMAC will be initialized.
'DMAC_CHANNEL'	Select the DMAC channel to use. See file for details.

If 'RAMTEST_USE_DMAC' has been defined than a specific HW test may enable use of the DMAC. This is done using the following:

#define	File where defined
'RAMTEST_MARCH_C_USE_DMAC'	`ramtest_march_c_HW.c`
'RAMTEST_MARCH_X_WOM_USE_DMAC'	`ramtest_march_x_wom_HW.c`

Declaration	
`void RamTest_March_HW_Init(void);`	
Description	
Initialize the hardware (DOC and optionally DMAC) used by the HW implementations of the RAM tests. The DMAC is only used if 'RAMTEST_USE_DMAC' is defined. 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	
Use to check if the hardware (DOC and DMAC) are functioning correctly before using. Performs a quick functional test of the DOC and (if RAMTEST_USE_DMAC is #defined) the DMAC.	
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. This is used by specific RAM tests to check that the HW 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.

Declaration	
`void RamTest_March_HW_Wait_DMAC(void);`	
Description	
Wait for the DMAC channel to complete a transfer. This is used by specific RAM tests and does not need to be called by a user. Note: In theory a user could add some code into this blocking loop. However, as this is called during RAM testing, they would need to be very careful not to use any RAM that is involved in the current RAM test. Note: Only available if RAMTEST_USE_DMAC is #defined.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

(4) RAM Test Stack API

This API enables a RAM test to be performed on an area of RAM that includes the stack. As the function that performs the RAM test requires a stack these functions will, re-locate the stack to a supplied new RAM area allowing the original stack area to be tested. Three functions are provided that can be called depending upon 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 by passing it in as a function pointer. If you are using a test that requires initialization before use, it is your responsibility to ensure this has been done before trying to use the test by calling one of these functions.

Table 5: RAM Test Stack API 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 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 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 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 to.
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.

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 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 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 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.

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 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 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 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 Synergy S5D9 has a Clock Frequency Accuracy Measurement Circuit (CAC) which can be used to detect monitor the Main clock frequency during run time.

Either one of MAIN, SUB_CLOCK, HOCO, MOCO, LOCO, IWDTCLK, and PCLKB 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. Ensure 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 S5D9 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

The SW relies on `S5D9_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. The clock shall be one of Main clock, Sub clock, HOCO clock, MOCO clock, LOCO clock, IWDTCLK clock, and PCLKB clock.
`clock_source_t ref_clock`	The reference clock to be used by CAC to monitor the target clock. The clock shall be one of Main clock, Sub clock, HOCO clock, MOCO clock, LOCO clock, IWDTCLK clock, and PCLKB clock.
`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 to be 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. The SW can select five possible pins: eCLOCK_MONITOR_CACREF_A (pin P204), eCLOCK_MONITOR_CACREF_B (pin P402), eCLOCK_MONITOR_CACREF_C (pin P600), eCLOCK_MONITOR_CACREF_D (pin P611) and eCLOCK_MONITOR_CACREF_E (pin P708) and it is the user 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 timer will not be refreshed by software as it is required to be and will therefore detect an error.

The Independent Watchdog Timer (iWDT) module of the Synergy S5D9 is used for this. It includes a windowing feature so that the refresh must happen within a specified ‘window’ rather than just before a specified time. It can be configured to generate an internal reset or a NMI interrupt if an error is detected. All the configurations for iWDT can be done through the OFS0 register whose settings are controlled by the user (see Section 2.5 for a configuration example). A function is provided to be used after a reset to decide if the iWDT has caused the reset. The test module relies on the `S5D9_registers.h` header 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 independent watchdog timer. After calling this function, the `IWDT_kick` function must then be called at the correct time to prevent a watchdog timer error. Note: 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.	
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. This 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 S5D9 has a Low Voltage Detection circuit. This can be used to detect the power supply voltage (VCC) falling below a specified voltage. The supplied sample code demonstrates using Voltage Detection Circuit 1 to generate a NMI interrupt when VCC drops below a specified level. The hardware is also capable of generating a reset but this behavior is not supported in the sample code. The SW module relies on `S5D9_registers.h` header 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 will be generated if VCC falls below the specified voltage. Note: The Non-Maskable Interrupt (NMI) must be handled by user code which must check the NMISR.LVDST flag. Note: The voltage threshold eVoltage shall be set to a value lower than the nominal VCC one.	
Input Parameters	
`VOLTAGE_MONITOR_LEVEL eVoltage`	The specified low voltage level. See declaration of enumerated type VOLTAGE_MONITOR_LEVEL in `voltage.h` for details.
Output Parameters	
`NONE`	N/A
Return Values	
`None`	N/A

1.7 ADC12

The ADC12 has a diagnostic mode that can be used to test the ADC. The diagnostic mode can be configured so that a test is performed every time the ADC is used normally for a conversion. The diagnostic reference voltage and hence the expected result is automatically rotated between zero, half scale and full scale. The diagnostic SW provides two automatic conversions (zero and full scale). The SW module relies on `S5D9_registers.h` header file to access peripheral registers.

Table 9: ADC12 Source files

File name
`test_adc12.h`
`test_adc12.c`

Syntax	
`void Test_ADC12_Init_u0(void)`	
Description	
Initializes unit 0 of ADC12 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	
`void Test_ADC12_Init_u1(void)`	
Description	
Initializes unit 1 of ADC12 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_ADC12_Wait_u0(void)`	
Description	
Waits while two ADC conversions are made by unit 0 of ADC12 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	
`bool Test_ADC12_Wait_u1(void)`	
Description	
Waits while two ADC conversions are made by unit 1 of ADC12 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_ADC12_Start_u0(ADC12_ERROR_CALL_BACK Callback)`	
Description	
Sets up unit 0 of ADC module so diagnostic tests will be performed each time ADC is used. The diagnostic reference voltage is automatically rotated (Zero, half VREF and VREH). User code must now call the `Test_ADC12_CheckResult` function either periodically or following every ADC completion to check the diagnostic result.	
Input Parameters	
`ADC12_ERROR_CALL_BACK Callback`	Function to call if an error is detected. Note: This function will only get called if `Test_ADC12_CheckResult` is called with parameter bCallErrorHandler set true.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void Test_ADC12_Start_u1(ADC12_ERROR_CALL_BACK Callback)`	
Description	
Sets up unit 1 of ADC module so diagnostic tests will be performed each time ADC is used. The diagnostic reference voltage is automatically rotated (Zero, half VREF and VREH). User code must now call the Test_ADC12_CheckResult function either periodically or following every ADC completion to check the diagnostic result.	
Input Parameters	
`ADC12_ERROR_CALL_BACK Callback`	Function to call if an error is detected. Note: This function will only get called if `Test_ADC12_CheckResult` is called with parameter bCallErrorHandler set true.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`bool Test_ADC12_CheckResult_u0(bool bCallErrorHandler)`	
Description	
Checks that ADC unit 0 diagnostic result is as expected. This must be called after Test_ADC12_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 ADC12_TOLERANCE in `test_ad12.c` for details.	
Input Parameters	
`bool bCallErrorHandler`	Set true to call the error call-back function supplied to function `Test_ADC12_Start`, otherwise false.
Output Parameters	
`NONE`	N/A
Return Values	
`bool`	True = Test passed. False = test failed.

Syntax	
`bool Test_ADC12_CheckResult_u1(bool bCallErrorHandler)`	
Description	
Checks the ADC unit 1 diagnostic result is as expected. This must be called after Test_ADC12_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 ADC12_TOLERANCE in `test_ad12.c` for details.	
Input Parameters	
`bool bCallErrorHandler`	Set true to call the error call-back function supplied to function `Test_ADC12_Start`, otherwise false.
Output Parameters	
`NONE`	N/A
Return Values	
`bool`	True = Test passed. False = test failed.

1.8 Temperature

The Synergy S5D9 has a Temperature Sensor module that can monitor the MCU temperature. The ADC12 module unit 1 is also required in conjunction with the Temperature Sensor. The SW module relies on `S5D9_registers.h` header 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 ADC12 module. Specify an allowed temperature range in terms of ADC12 output values. After calling this function the Temperature_Start function must be called periodically to perform an ADC conversion on the Temperature Sensor output and then the remaining functions must be used to check the result.	
Input Parameters	
`uint16_t Temperature_ADC_Value_Min`	Specify the minimum value that the ADC12 should output when reading the temperature sensor.
`uint16_t Temperature_ADC_Value_Max`	Specify the maximum value that the ADC12 should output when reading the temperature sensor.
`TEMPERATURE_ERROR_CALL_BACK Error_callback`	This function will be called by function Temperature_CheckResult if the temperature (ADC12 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. This will use the ADC12 module, destroying its current settings. It is the user's responsibility to ensure this behavior is not a problem. Following this function use function Temperature_Read_Wait or Temperature_CheckResult.	
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 ADC12 value.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`uint16_t`	ADC12 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 ADC12 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, C, D) input pin is detected or when oscillation stop is detected. Note that the user must configuration of 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 SW module relies on `S5D9_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, C, D) 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 POE_ClearFlags_gc(void);`	
Description	
For POEG group C, 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_gd(void);`
Description
For POEG group D, 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 gives to the user 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 that for example a faster main clock can be selected.
2. Periodic Tests. These are tests that are run regularly throughout normal program operation. This document does not provide a judgment of how often a particular test should be ran. How the scheduling of the periodic tests is performed is up to the user depending upon 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 will be called. As any error in the CPU is very serious the aim of this function should be to get to a safe state, where software execution is not relied upon, as soon as possible.

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.

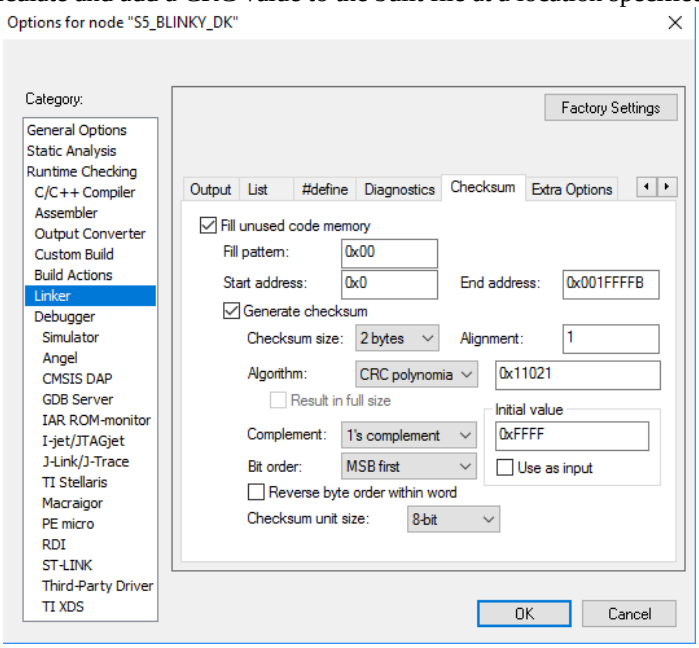
2.1.2 Periodic

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 of the individual CPU test functions 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 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 can be done via a dialog in IAR. See Figure 1: Adding Reference CRC.



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

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

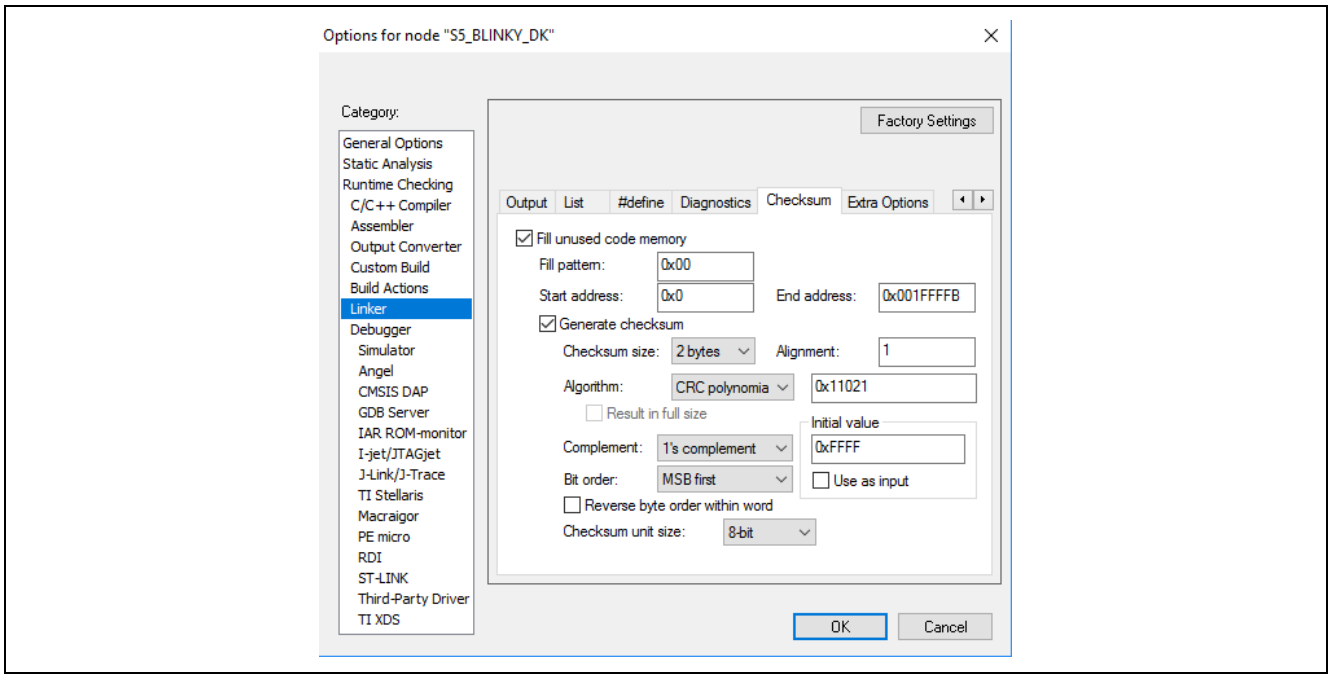


Figure 1: Adding Reference CRC

2.2.1 Power-Up

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

If this area is one contiguous block then function CRC_Calculate 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 then be compared with the reference CRC value stored in the ROM using function CRC_Verify.

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

2.2.2 Periodic

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

2.3 RAM

It is very important to realize that the area of RAM that needs to be tested may change dramatically depending upon your project's memory map.

If you are using the 'HW' versions of the RAM Tests (where the DOC and possibly DMAC are used), then you must call function RamTest_March_HW_Init 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 points:

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 to.
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. However, if startup time is very important, it might be possible to fine tune this so that only the area of Stack used before the power-up RAM test is performed using the slower non-destructive test and 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

The monitoring of the main clock 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         (24000000) // 24 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         (24000000) // 24 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 can be called as soon as the main clock has been configured and the IWDT has been enabled. See Section 1.5 for enabling the iWDT.

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

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

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

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

In addition, in order to enable the Cortex-M4 NVIC to handle the CAC interrupts, the following instructions 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 (See Table 2-16 of “Cortex-M4 Devices: Generic User Guide”, first release, 16 December 2010 for more details about IRQ numbers).

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

```
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

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

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

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

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

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

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

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

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

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

#define OFS0_IWDT_RESET_MASK (0xFFFF0000)

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

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

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

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

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

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

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

If the watchdog timer has been configured to perform a reset on error detection then following a reset the code should check if 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 should be setup as soon as possible following a power on reset.

Please be sure to set the LVD1SR.DET bit to 0 both before calling the `VoltageMonitor_Init` function and in the NMI routine. See Section 8.2.2 of Synergy S5D9 Hardware Manual for further details.

Please be sure to set a voltage threshold eVoltage lower than the VCC nominal value.

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

```
VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL_2_99);
```

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

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

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

2.7 ADC12

The ADC12 module has a built in diagnostic mode which allows various reference voltages to be tested against.

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

```
#define ADC12_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 ADC12 Test module must be initialized with a call to `Test_ADC12_Init_uX` (X = 0. 1).

2.7.1 Power-Up

At power-up, the ADC12 module can be tested using the `Test_ADC12_Wait_uX` function. This function waits until two AD conversions are performed, one using reference voltage 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_ADC12_Start_uX`. Following that, the ADC12 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_ADC12_CheckResult_uX`.

2.8 Temperature

When testing the MCU temperature, it is important to remember that the ADC12 module unit 1 will be used. Therefore if the user's code also uses the ADC12 to monitor analog pins, resource sharing of the ADC12 module must be carefully considered.

The temperature sensor must be initialized before use with a call to `Temperature_Init`. This function must be passed for the allowable range of temperatures expressed in terms of the ADC12 output. See the Synergy S5D9 Hardware Manual for details on how to calculate/find by experimenting with these values.

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

2.8.1 Power-Up

Temperature test procedure at power-up will be the same as explained for the periodic tests.

2.8.2 Periodic

Periodically, the use of the ADC12 module must be taken over by the temperature sensor. To make a temperature reading, call this function:

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

The result can then 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 blocking the SW application for too long, it can be arranged so that each time the periodic test is scheduled it actually checks the result of the temperature test started on the previous scheduled test and then start 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 ADC12 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);
```

Note that the POEG group choice is up to the user. The user must carefully study the description of `POE_Init` and consult the Synergy S5D9 Hardware manual to determine if the sample configuration of the POE meets the requirements of the user's system. Depending upon the pins used in the user's system, the `S5D9_registers.h` header file may need to be adapted for the desired behavior. In particular, the value of the P100PFS_reg must be updated to the address of the pin to be used as GTETRGn input of POE peripheral.

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

For configuring 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 = 0x9A, POEG_GROUP1 = 0x9B, POEG_GROUP2 =

0x9C, POEG_GROUP3 = 0x9D). In particular, it is necessary to configure one IELSR register so that it is linked to the aforementioned CAC events:

```
IELSRn.IELS = 0x9A; // (POEG_GROUP0)
IELSRn.IELS = 0x9B; // (POEG_GROUP1)
IELSRn.IELS = 0x9C; // (POEG_GROUP2)
IELSRn.IELS = 0x9D; // (POEG_GROUP3)
```

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

```
NVIC_EnableIRQ(POEG0_EVENT_IRQn);
NVIC_EnableIRQ(POEG1_EVENT_IRQn);
NVIC_EnableIRQ(POEG2_EVENT_IRQn);
NVIC_EnableIRQ(POEG3_EVENT_IRQn);
```

Where `POEG0_EVENT_IRQn`, `POEG1_EVENT_IRQn`, `POEG2_EVENT_IRQn` and `POEG3_EVENT_IRQn` are the IRQ numbers that must be defined by the user. See Table 2-16 of “Cortex-M4 Devices: Generic User Guide”, first release, 16 December 2010 for more details about IRQ numbers.

3. Benchmarking

3.1 Environment

1. Development board: DK-S5D9M v3.0
2. Clocks: EXTAL = 24 MHz, ICLK = 120 MHz, PCLKB = 60 MHz, PCLKD = 120 MHz
3. MCU: R7FS5D97H2A01CBD
4. Tool chain: IAR Embedded Workbench for ARM , Functional Safety, v.7.40.6.9816
5. In-circuit debugger: ARM Debug + ETM connector and SEGGER J-link on board

Build option:

1. General option:
    1. Target = Renesas R7FS5D97H
2. Complier Settings:
    1. Language = C
    2. C-dialect = C-99
    3. Language Conformance = Standard with IAR extension
    4. Plain ‘char’ is: Unsigned
    5. Floating-point semantics: Strict conformance
    6. Optimization Level: None

3.2 Results

3.2.1 CPU

Table 12: CPU test results

Measurement	Result Non-CouplingTest (both CPU and FPU)	Result Coulping Test (both CPU and FPU)
ROM usage (bytes)	27346	27502
RAM usage (bytes)	0	0
Stack usage (bytes)	0	0
Clock Cycle Count – CPU_TestAll	891	9732
Time Measured (µs) @120 MHz – CPU_TestAll	7.43	81.1

3.2.2 ROM

Table 13: Test results for CRC32C

Measurement	Result
ROM usage (bytes)	170
RAM usage (bytes)	0
Stack usage (bytes)	0
Clock Cycle Count – CRC_Init	52
Time Measured (µs) @120 MHz – CRC_Init	0.43
Clock Cycle Count – CRC_Calculate (ROM overall, that is, 2 MB)	6291498
Time Measured (ms) @120 MHz – CRC_Calculate (2 MB)	52.43
Clock Cycle Count – CRC_Calculate (1kB)	3122
Time Measured (µs) @120 MHz – CRC_Calculate (1 kB)	26.01
Clock Cycle Count – CRC_Calculate (4kB)	12338
Time Measured (µs) @120 MHz – CRC_Calculate (4 kB)	4102.82
Clock Cycle Count – CRC_Calculate (16kB)	49202
Time Measured (µs) @120 MHz – CRC_Calculate (16 kB)	410.02
Clock Cycle Count – CRC_Verify	25
Time Measured (us) @120 MHz – CRC_Verify	0.21

3.2.3 RAM

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

(1) March C

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

Measurement			Normal Result	HW (DOC+DMAC) Result
ROM usage (bytes)			586	652
RAM usage (bytes)			0	0
Stack usage (bytes)			80	88
Stack usage Extra (bytes)			112	120
Clock cycle count	Destructive	1024 bytes	1067270	1046968
		500 bytes	521262	511442
		100 bytes	104460	102640
	Non-destructive	1024 bytes	1086763	1064623
		500 bytes	530796	520183
		100 bytes	106397	104579
	Extra	1024 bytes	2187843	2143345
		500 bytes	1068582	1047137
		100 bytes	214178	210339
Time Measured (ms) @ 120 MHz	Destructive	1024 bytes	8,89	8,72
		500 bytes	4,34	4,26
		100 bytes	0,87	0,86
	Non-destructive	1024 bytes	9,06	8,87
		500 bytes	4,42	4,33
		100 bytes	0,89	0,87
	Extra	1024 bytes	18,23	17,86
		500 bytes	8,90	8,73
		100 bytes	1,78	1,75

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

Measurement			Normal Result	HW (DOC+DMAC) Result
ROM usage (bytes)			544	688
RAM usage (bytes)			0	0
Stack usage (bytes)			80	88
Stack usage Extra (bytes)			112	120
Clock cycle count	Destructive	1024 bytes	858381	849834
		500 bytes	419269	415176
		100 bytes	84068	83376
	Non-destructive	1024 bytes	863284	854433
		500 bytes	421680	417543
		100 bytes	84582	84043
	Extra	1024 bytes	1754700	1737075
		500 bytes	857091	848761
		100 bytes	171890	170661
Time Measured (ms) @ 120 MHz	Destructive	1024 bytes	7,15	7,08
		500 bytes	3,49	3,46
		100 bytes	0,70	0,69
	Non-destructive	1024 bytes	7,19	7,12
		500 bytes	3,51	3,48
		100 bytes	0,70	0,70
	Extra	1024 bytes	14,62	14,48
		500 bytes	7,14	7,07
		100 bytes	1,43	1,42

(2) March X WOM

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

Measurement			Normal Result	HW (DOC+DMAC) Result
ROM usage (bytes)			444	442
RAM usage (bytes)			0	0
Stack usage (bytes)			0	0
Stack usage Extra (bytes)			0	0
Clock cycle count	Destructive	1024 bytes	98548	99634
		500 bytes	48243	48806
		100 bytes	9842	10004
	Non-destructive	1024 bytes	118037	119123
		500 bytes	57779	58335
		100 bytes	11777	11937
	Extra	1024 bytes	221730	219795
		500 bytes	108545	107661
		100 bytes	22139	22059
Time Measured (ms) @ 120 MHz	Destructive	1024 bytes	0.82	0.83
		500 bytes	0.42	0.41
		100 bytes	0.08	0.08
	Non-destructive	1024 bytes	1.06	0.99
		500 bytes	0.52	0.49
		100 bytes	0.1	0.10
	Extra	1024 bytes	2	1.83
		500 bytes	0.98	0.90
		100 bytes	0.2	0.18

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

Measurement			Normal Result	HW (DOC+DMAC) Result
ROM usage (bytes)			424	442
RAM usage (bytes)			0	0
Stack usage (bytes)			0	0
Stack usage Extra (bytes)			0	0
Clock cycle count	Destructive	1024 bytes	98549	99610
		500 bytes	48244	48781
		100 bytes	9843	9981
	Non-destructive	1024 bytes	118039	119102
		500 bytes	57781	58317
		100 bytes	11779	11917
	Extra	1024 bytes	221732	219787
		500 bytes	108547	107652
		100 bytes	22141	22049
Time Measured (ms) @ 120 MHz	Destructive	1024 bytes	0.82	0.83
		500 bytes	0.40	0.41
		100 bytes	0.08	0.08
	Non-destructive	1024 bytes	0.98	0.99
		500 bytes	0.48	0.49
		100 bytes	0.10	0.10
	Extra	1024 bytes	1.85	1.83
		500 bytes	0.90	0.90
		100 bytes	0.18	0.18

(3) Stack Test

Note: The results are the same regardless of the normal or HW implementation, because the stack test does not rely on HW.

Table 18: Stack test results

Measurement	Result
ROM usage (bytes)	404
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)	131
Time Measured (us) @120 MHz – RamTest_Stack_Main	1.09
Clock Cycle Count – RamTest_Stack_Proc (only stack relocation)	134
Time Measured (us) @120 MHz – RamTest_Stack_Proc	1.12
Clock Cycle Count – RamTest_Stacks (only stack relocation)	156
Time Measured (us) @120 MHz – RamTest_Stacks	1.3

(4) HW supporting functions

This section reports the ROM and RAM resources needed to support the use of HW peripherals DOC and DMAC.

Table 19: HW supporting function results

Measurement	Result
ROM usage (bytes)	556
RAM usage (bytes)	0
Stack usage (bytes)	0
Clock Cycle Count – RamTest_March_HW_Init	120
Time Measured (us) @120 MHz – RamTest_March_HW_Init	1
Clock Cycle Count – RamTest_March_HW_PreTest	863
Time Measured (us) @120 MHz – RamTest_March_HW_PreTest	7.19
Clock Cycle Count – RamTest_March_HW_Is_Init	22
Time Measured (us) @120 MHz – RamTest_March_HW_Is_Init	0.18

3.2.4 Clock

Table 20: Clock test results

Measurement	Internal reference Clock Result	External Reference Clock Result
ROM usage (bytes)	592	1480
RAM usage (bytes)	4	4
Stack usage (bytes)	56	56
Clock Cycle Count	2370	876
Time measured (us) @ 120 MHz	19.75	7.3

3.2.5 Independent Watchdog

Table 21: Independent Watchdog test results

Measurement	Result
ROM usage (bytes)	124
RAM usage (bytes)	0
Stack usage (bytes)	0
Clock Cycles Count – IWDT_Init	30
Time measured (us) @ 120 MHz – IWDT_Init	0.25
Clock Cycles Count – IWDT_Kick	17
Time measured (us) @ 120 MHz – IWDT_Kick	0.14
Clock Cycles Count – IWDT_DidReset	35
Time measured (us) @ 120 MHz – IWDT_DidReset	0.29

3.2.6 Voltage

Table 22: Voltage Monitoring test results

Measurement	Result
ROM usage (bytes)	188
RAM usage (bytes)	0
Stack usage (bytes)	0
Clock Cycles Count	25279
Time measured (us) @ 120 MHz	210.66

3.2.7 ADC12

Table 23: 12-bit ADC Converter test results

Measurement	Result
ROM usage (bytes)	908
RAM usage (bytes)	8
Stack usage (bytes)	0
Clock Cycles Count – Test_ADC12_Init_uX	33
Time measured (us) @ 120 MHz – Test_ADC12_Init_uX	0.275
Clock Cycles Count – Test_ADC12_Wait_uX	347
Time measured (us) @ 120 MHz – Test_ADC12_Wait_uX	2.89
Clock Cycles Count – Test_ADC12_Start_uX	42
Time measured (us) @ 120 MHz- Test_ADC12_Start_uX	0.35
Clock Cycles Count (clock cycles) – Test_ADC12_CheckResult_uX	73
Time measured (us) @ 120 MHz – Test_ADC12_CheckResult_uX	0.61

3.2.8 Temperature

Table 24: Temperature sensor test results

Measurement	Result
ROM usage (bytes)	344
RAM usage (bytes)	8
Stack usage (bytes)	0
Clock Cycles Count – Temperature_Init	71
Time measured (us) @ 120 MHz – Temperature_Init	0.59
Clock Cycles Count – Temperature_Start	43328
Time measured (us) @ 120 MHz – Temperature_Start	361
Clock Cycles Count – Temperature_CheckResult	111
Time measured (us) @ 120 MHz – Temperature_CheckResult	0.925

3.2.9 Port Output Enable

Table 25: Port Output Enable test results

Measurement	Result
ROM usage (bytes)	1504
RAM usage (bytes)	4
Stack usage (bytes)	0
Clock Cycles Count – GPT_init	793
Time measured (us) @ 120 MHz – GPT_init	6.61
Clock Cycles Count – POE_Init	261
Time measured (us) @ 120 MHz – POE_Init	2.17

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) (see section 20.2.5 of Synergy S5D9 Hardware Manual for details):

20.2.5 Port mn Pin Function Select Register (PmnPFS) (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

	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	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: Low output 1: High output.	R/W
b1	PIDR	Port Input Data	0: Low output 1: High output.	R

Figure 2 PmnPFS Register

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	26 Jun, 2017	-	Initial Release

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 or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other disputes involving 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, including but not limited to, the product data, drawing, chart, program, algorithm, application examples.
3. 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 shall not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics products.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended 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 (traffic lights); large-scale communication equipment; key financial terminal systems; safety control 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 (space and undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability 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. When using the Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat radiation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions or failure or accident 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 Renesas Electronics 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 ensure to implement safety measures to guard them against the possibility of bodily injury, injury or damage caused by fire, and social damage in the event of failure or malfunction of Renesas Electronics products, 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 by your own responsibility as warranty for your products/system. Because the evaluation of microcomputer software alone is very difficult and not practical, 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 investigate applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive carefully and sufficiently and use Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall 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 shall not use Renesas Electronics products or technologies for (1) any purpose relating to the development, design, manufacture, use, stockpiling, etc., of weapons of mass destruction, such as nuclear weapons, chemical weapons, or biological weapons, or missiles (including unmanned aerial vehicles (UAVs)) for delivering such weapons, (2) any purpose relating to the development, design, manufacture, or use of conventional weapons, or (3) any other purpose of disturbing international peace and security, and you shall not sell, export, lease, transfer, or release Renesas Electronics products or technologies to any third party whether directly or indirectly with knowledge or reason to know that the third party or any other party will engage in the activities described above. When exporting, selling, transferring, etc., Renesas Electronics products or technologies, you shall comply with any applicable export control laws and regulations promulgated and administered by the governments of the countries asserting jurisdiction over the parties or transactions.
10. Please acknowledge and agree that you shall bear all the losses and damages which are incurred from the misuse or violation of the terms and conditions described in this document, including this notice, and hold Renesas Electronics harmless, if such misuse or violation results from your resale or making Renesas Electronics products available any third party.
11. This document shall not be reprinted, 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.

(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.

(Rev.3.0-1 November 2016)

RENESAS

SALES OFFICES

Renesas Electronics Corporation

http://www.renesas.com

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

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

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

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

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

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

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

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

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

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

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

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL II 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

Colophon 6.0