RX100 Series

VDE Certified IEC60730 Self Test Code for RX100 Series

R01AN2061ED0100
Rev.1.00
Mar 20, 2014

Introduction

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

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

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

1. Class A: Control functions, which are not intended to be relied upon for the safety of the equipment.

   Examples: Room thermostats, humidity controls, lighting controls, timers, and switches.

2. Class B: Control functions, which are intended to prevent unsafe operation of the controlled equipment.

   Examples: Thermal cut-offs and door locks for laundry equipment.

3. Class C: Control functions, which are intended to prevent special hazards

   Examples: Automatic burner controls and thermal cut-outs for closed.

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

This Application Note provides guidelines of how to use flexible sample software routines to assist with compliance with IEC60730 class B safety standards. These routines have been certified by VDE Test and Certification Institute GmbH and a copy of the Test Certificate is available in the download package for this Application Note (See Note 1 below).

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

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

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

RX111

When using this application note with another RX100 Series MCUs, careful evaluation is recommended after making modifications to comply with the alternate MCU.

Contents

1. Tests

1.1 CPU

This section describes CPU tests routines. Reference IEC 60730: 1999+A1:2003 Annex H - Table H.11.12.7 CPU.

The following CPU registers are tested: R0->R15, ISP, USP, INTB, PC, PSW, BPC, BPSW, FINTV and ACC.

The source file 'CPU_Test.c' provides implementation of the CPU test using "C" language with inline assembly to actually access the registers. File CPU_Test_Coupling.c is also required if using the coupling test version of the General Purpose Registers. The source file 'CPU_Test.h' provides the interface to the CPU tests. The file 'MisraTypes.h' includes definitions of MISRA compliant standard data types.

Note: The following statement in file CPU_Test.c must be present to prevent testing of a register (FPSW) that the RX111 does not support but other RX CPU cores do.

```
#define RX111
```

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);
```

This will be jumped to by the CPU test if an error is detected. This function must not return.

The CPU test is split into a number of functions or, if time is permitting, a single function call can be used to run all the tests one after another. See Section 1.1.1 Software API for details.

The test functions all follow the rules of register preservation following a C function call as specified in the Renesas tool chain manual. Therefore the user can call these functions like any normal C function without any additional responsibilities for saving register values beforehand.

IMPORTANT NOTE: Please keep the "Optimization" option "OFF" for the 'CPU_Test.c' file, to prevent modification of the test code.

1.1.1 Software API

Table 1: Source files:

File name
CPU_Test.h
CPU_Test.c, CPU_Test_Coupling.c

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_GeneralA CPU_Test_GeneralB 2. CPU_Test_Control(*2, see below) 3. CPU_Test_Accumulator 4. CPU_Test_PC It is the calling functions responsibility to ensure that the processor is in Supervisor Mode. If this function is called in User Mode the test will fail as some of the register bits are not accessible in User Mode. 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_TestGPRsCoupling' in the code is used to select which functions will be used to test the General Purpose Registers. *2 The RX111 has a slightly different PSW register from other Rx devices. For this reason, if using an RX111, then "RX111" must be defined in the project.	
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 R15.Coupling faults between the registers are detected. This is PartA 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 R15.Coupling faults between the registers are detected. This is PartB 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_GeneralA(void)`
Description
Test registers R1,R2,R3,R4,R5,R14 and R15. These are the general purpose registers that don't need to be preserved by a function. Registers are tested in pairs. For each pair of registers: 1. Write h'55555555 to both. 2. Read both and check they are equal. 3. Write h'AAAAAAAA to both. 4. Read both and check they are equal. It is the calling function's responsibility to 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_GeneralB(void)`
Description
Test registers R0,R6,R7,R8,R9,R10,R11,R12 and R13. These are the general purpose registers that need to be preserved by a function. Registers are tested in pairs. For each pair of registers: 1. Write h'55555555 to both. 2. Read both and check they are equal. 3. Write h'AAAAAAAA to both. 4. Read both and check they are equal. It is the calling function's responsibility to 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_Control(void)`
Description
Tests control registers ISP,USP,INTB,PSW,BPC,BPSW,FINTV and FPSW. NOTE: FPSW is not tested if 'RX210' is #defined. This test assumes registers R1 to R5 are working. Generally the test procedure for each register is as follows: For each register:- 1. Write h'55555555 to. 2. Read back and check value equals h'55555555. 3. Write h'AAAAAAAA to. 4. Read back and check value equals h'AAAAAAAA. Note however that there are some cases where restrictions on certain bits within a register mean this cannot be can followed exactly so other test values have been chosen. It is the calling functions responsibility to ensure that the processor is in Supervisor Mode. If this function is called in User Mode the test will fail as some of the register bits are not accessible in User Mode. It is also the calling function's responsibility to ensure no interrupts occur during this test. The RX610 has a slightly different PSW register from other Rx devices. For this reason, if using an RX610, then "RX610" must be defined in the project. 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_Accumulator(void)`
Description
Tests the ACC register. NOTE: Bits 0-15 cannot be read and are therefore not tested. The register value is preserved by this test. The test procedure is as follows: 1. Write h'55555555 to high order 32 bits. 2. Write h'55555555 to low order 32 bits. 3. Read back high order and check value equals h'55555555. 4. Read back middle order(bits 47 to 16) and check value equals h'55555555. 5. Write h'AAAAAAAA to high order 32 bits. 6. Write h'AAAAAAAA to low order 32 bits. 7. Read back high order and check value equals h'AAAAAAAA. 8. Read back middle order (bits 47 to 16) and check value equals h'AAAAAAAA. This test assumes registers R1 to R5 are working. If an error is detected then external function '`CPU_Test_ErrorHandler`' will be called

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

Syntax
`void CPU_Test_PC(void)`
Description
This function provides the Program Counter (PC) register test.

This provides a confidence check that the PC is working. It tests that the PC is working by calling a function that is located in its own section so that it can be located away from this function, so that when it is called more of the PC Register bits are required for it to work. So that this function can be sure that the function has actually been executed it returns the inverse of the supplied parameter. This return value is checked for correctness. If an error is detected then external function '`CPU_Test_ErrorHandler`' will be called.

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

1.2 ROM

This section describes the ROM / Flash memory test using CRC routines. Reference IEC 60730: 1999+A1:2003 Annex H - Table H.11.12.7 Invariable Memory.

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, which 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 Renesas RX Standard toolchain does using the –CRC option. Therefore if you are using the Renesas Toolchain to automatically insert a reference CRC into the ROM the value can be compared directly with the one calculated.

1.2.1 CRC16-CCITT Algorithm

The RX100 family includes a CRC module that includes support for the CRC16-CCITT. Using this software to drive the CRC module produces this 16-bit CRC16-CCITT:

- Polynomial = 0x1021 ($x^{16} + x^{12} + x^5 + 1$)
- Width = 16 bits
- Initial value = 0xFFFF
- XOR with h'FFFF is performed on the output CRC

1.2.2 CRC Software API

All software is written in ANSI C.

'MisraTypes.h' includes definitions of MISRA-compliant standard data types.

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: Source files:

File name
CRC_Verify.h, CRC_Verify.c
CRC.h, CRC.c

Syntax	
`bool_t CRC_Verify(const uint16_t ui16_NewCRCValue, const uint32_t ui32_AddrRefCRC)`	
Description	
This function compares a new CRC value with a reference CRC by supplying address where reference CRC is stored.	
Input Parameters	
`uint16_t ui16_NewCRCValue`	Value of calculated new CRC value.
`uint32_t ui32_AddrRefCRC`	Address where 16 bit reference CRC value is stored.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	Test result: TRUE = Passed, FALSE = Failed

This following functions are implemented in files CRC.h and CRC.c:

Syntax	
`uint16_t CRC_Init(void)`	
Description	
Initialises the CRC module. This function must be called before any of the other CRC functions can be.	
Input Parameters	
`uint8_t* pui8_DataBuf`	Pointer to start of memory to be tested.
`uint32_t ui32_DataBufSize`	Length of the data in bytes.
Output Parameters	

NONE	N/A
Return Values	
uint16_t	The 16-bit calculated CRC-CCITT value.

Syntax	
uint16_t CRC_Calculate(uint8_t* pui8_Data, uint32_t ui32_Length)	
Description	
This function calculates the CRC of a single specified memory area.	
Input Parameters	
uint8_t* pui8_DataBuf	Pointer to start of memory to be tested.
uint32_t ui32_DataBufSize	Length of the data in bytes.
Output Parameters	
NONE	N/A
Return Values	
uint16_t	The 16-bit calculated CRC-CCITT value.

The following functions are used when the memory area can not 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.

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(uint8_t* pui8_Data, uint32_t ui32_Length)
Description
Use this function rather than CRC_Calculate if wanting 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	
`uint8_t* pui8_DataBuf`	Pointer to start of memory range to be tested.
`uint32_t ui32_DataBufSize`	Length of the data in bytes.
Output Parameters	
`NONE`	N/A
Return Values	
`None`	N/A

`int16_t CRC_Result(void)`	
Description	
Calcualtes 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	
`uint16_t`	The calculated CRC-CCITT 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, while 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 will be its fault coverage 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. Following that is the specification of the software APIs.

1.3.1 Algorithms

(1) **March C**

The March C algorithm (van de Goor 1991) consists of 6 March elements with a total of 10 operations. It detects the following faults:

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

These are the 6 March elements:-

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

(2) **March X**

Note: This algorithm has not been implemented for the RX100 family and is only presented here for information as it relates to the March X WOM version below.

The March X algorithm consists of 4 March elements with a total of 6 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 4 March elements:-

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

(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 using a data pattern of alternating high/low bits then the second using the inverse. The addition of these elements is to detect intra-word coupling faults.

These are the 6 March elements:-

I. Write all zeros to array
II. Starting at lowest address, read zeros, write ones, increment up array word by word.
III. Starting at highest address, read ones, write zeros, decrement down word by word.
IV. Starting at lowest address, read zeros, write h'AAs, increment up array word by word.
V. Starting at highest address, read h'AAs, write h'55s, decrement down word by word.
VI. Read all h'55s 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) to help perform the tests.

Both implementations share the same core API but the 'HW' implementation has some additional functions. Please see details in Section (3) March C and March X WOM HW Implementation specific API.

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. It is recommended that more than h'0F bytes are tested each time.

(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:

- RAMTEST_MARCH_C_ACCESS_SIZE_8BIT
- RAMTEST_MARCH_C_ACCESS_SIZE_16BIT
- 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:

- RAMTEST_MARCH_C_MAX_WORDS_8BIT (Max words in test area is 0xFF)
- RAMTEST_MARCH_C_MAX_WORDS_16BIT (Max words in test area is 0xFFFF)
- 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_c.c	ramtest_march_c_HW.c
	ramtest_march_HW.h
	ramtest_march_HW.c

The source is written in ANSI C and uses MISRA-compliant data types as declared in file MisraTypes.h.

Declaration
`bool_t 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

ui32_StartAddr	The address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
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.
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_t	TRUE = Test passed. FALSE = Test or parameter check failed.

Declaration	
`bool_t RamTest_March_C_Extra(uint32_t ui32_StartAddr, uint32_t ui32_EndAddr, void* p_RAMSafe);`	
Description	
Non Destructive RAM memory test using March C (Goor 1991) algorithm. This function differs from the RamTest_March_C function 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	
ui32_StartAddr	The address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
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.
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_t	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_x_wom.h
ramtest_march_x_wom.c	ramtest_march_x_wom_HW.c
	ramtest_march_HW.h
	ramtest_march_HW.c

The source is written in ANSI C and uses MISRA-compliant data types as declared in file MisraTypes.h.

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_t 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	
`ui32_StartAddr`	Address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
`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.
`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_t	TRUE = Test passed. FALSE = Test or parameter check failed.

Declaration	
bool_t 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_XXBit 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	
ui32_StartAddr	The address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
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.
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_t	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) to help perform the tests. The DOC is used to compare values read back from RAM with expected values.

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

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

Declaration	
`bool_t RamTest_March_HW_PreTest(void);`	
Description	
This may be used to check if the hardware (DOC) are functioning correctly before using. A quick functional test of the DOC is performed.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	TRUE = Test passed. FALSE = Test failed.

Declaration	
`bool_t 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_t`	TRUE = Test passed. FALSE = Test or parameter check failed.

(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 (User or Interrupt) is in the test area or if both are.

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

Table 5: Source files:

File name
ramtest_stack.h
ramtest_stack.c

Declaration	

`bool_t RamTest_Stack_User(uint32_t ui32_StartAddr,`
`uint32_t ui32_EndAddr,`
`void* p_RAMSafe,`
`uint32_t ui32_NewUSP,`
`TEST_FUNC fpTest_Func);`	

Description	
RAM test of an area that includes the User Stack. (but not the Interrupt stack)	
Input Parameters	
`ui32_StartAddr`	The address of the first word of RAM to be tested. This must be compatible with the requirements of the fpTest_Func.
`Ui32_EndAddr`	The address of the last word of RAM to be tested. This must be compatible with the requirements of the fpTest_Func.
`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.
`Ui32_NewUSP`	New Stack pointer value for the User stack to be re-located 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_t`	TRUE = Test passed. FALSE = Test or parameter check failed.

Declaration	

`bool_t RamTest_Stack_Int(uint32_t ui32_StartAddr,`
`uint32_t ui32_EndAddr,`
`void* p_RAMSafe,`
`uint32_t ui32_NewISP,`
`TEST_FUNC fpTest_Func);`	

Description	
RAM test of an area that includes the Interrupt Stack. (but not the User stack)	
Input Parameters	
`ui32_StartAddr`	The address of the first word of RAM to be tested. This must be compatible with the requirements of the fpTest_Func.
`Ui32_EndAddr`	The address of the last word of RAM to be tested. This must be compatible with the requirements of the fpTest_Func.
`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.
`Ui32_NewISP`	New Stack pointer value for the Interrupt stack to be re-located 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_t`	TRUE = Test passed. FALSE = Test or parameter check failed.

Declaration	

`bool_t RamTest_Stacks(uint32_t ui32_StartAddr,`
`uint32_t ui32_EndAddr,`
`void* p_RAMSafe,`
`uint32_t ui32_NewISP,`
`uint32_t ui32_NewUSP,`
`TEST_FUNC fpTest_Func);`	

Description	
RAM test of an area that includes the Interrupt Stack. (but not the User stack)	
Input Parameters	
`ui32_StartAddr`	The address of the first word of RAM to be tested. This must be compatible with the requirements of the fpTest_Func.
`Ui32_EndAddr`	The address of the last word of RAM to be tested. This must be compatible with the requirements of the fpTest_Func.
`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.
`Ui32_NewISP`	New Stack pointer value for the Interrupt stack to be re-located to.
`Ui32_NewUSP`	New Stack pointer value for the User stack to be re-located to.
`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_t`	TRUE = Test passed. FALSE = Test or parameter check failed.

1.4 Clock

The RX100 family has a Clock Frequency Accuracy Measurement Circuit (CAC) which can be used to detect monitor the Main clock frequency during run time.

Either the IWDTCLK or an External clock on the CACREF pin can be used as a reference voltage.

If using an external reference clock:

1. #define CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK in file clock_monitor.c.

If using the IWDCLK:

1. Ensure CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK is not defined.
2. Ensure the definition of CLOCK_COUNT_EXPECTED is correct for the expected Main clock value.

If the frequency of the main clock deviates during runtime from a configured range an error call-back function shall be called. 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 RX100 family has an Oscillation Stop Detection Circuit. If the main clock stops, the Low 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: Source files:

File name
clock_monitor.h
clock_monitor.c

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_MONITOR_ERROR_CALL_BACK CallBack)`	
Description	
1. Start monitoring the Main clock using the CAC module and the IWDCLK as a reference clock. 2. Enables Oscillation Stop Detection and configures an NMI to be generated if detected..	
Input Parameters	
`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 not defined.

Syntax	

`void ClockMonitor_Init (uint32_t MainClockFrequency,`
`uint32_t ExternalRefClockFrequency,`
`CLOCK_MONITOR_CACREF_PIN ePin,`
`CLOCK_MONITOR_ERROR_CALL_BACK CallBack)`	

Description	
1. Start monitoring the Main clock using the CAC module and the CACREF pin as a reference clock. 2. Enables Oscillation Stop Detection and configures an NMI to be generated if detected.	
Input Parameters	
`MainClockFrequency`	Main clock expected frequency in Hz.
`ExternalRefClockFrequency`	External reference clock frequency in Hz.
`ePin`	The pin to use for CACREF. See CLOCK_MONITOR_CACREF_PIN for details.
`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

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

The Independent Watchdog Timer (IWDT) module of the RX100 family 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. A function is provided to be used after a reset to decide if the IWDT has caused the reset.

Table 7: Source files:

File name
IWDT.h
IWDT.c

Syntax	

`void IWDT_Init (IWDT_TOP TimeOutperiod,`
`IWDT_CKS_DIV ClockSelection,`
`IWDT_WINDOW_START WindowStart,`
`IWDT_WINDOW_END WindowEnd,`
`IWDT_ACTION Action)`	

Description	
Initialize and start the independent watchdog timer. After calling this the IWDT_kick function must then be called at the correct time to prevent a watchdog 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	
TimeOutperiod	Time out count. See declaration of enumerated type IWDT_TOP in IWDT.h for details.
ClockSelection	IWDT clock selection. See declaration of enumerated type IWDT_CKS_DIV in IWDT.h for details.
WindowStart	Window start position. See declaration of enumerated type IWDT_WINDOW_START in IWDT.h for details.
WindowEnd	Window start position. See declaration of enumerated type IWDT_WINDOW_END in IWDT.h for details.
Action	Select between generating a reset or NMI when detecting an error.
Output Parameters	
NONE	N/A
Return Values	
None	N/A

Syntax	
`void IWDT_Kick(void)`	
Description	
Refresh the watchdog count.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`None`	N/A

Syntax	
`bool_t 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 caused the reset.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	TRUE if watchdog has timed out, otherwise FALSE.

Syntax	
`void IWDT_SleepMode_CountStop_Disable (void)`	
Description	
By default the IWDT counter is stopped in sleep mode. Call this to change the default so the counter continues counting in sleep mode.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`None`	N/A

1.6 Voltage

The RX100 family has a 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 and monitoring an external pin voltage but this behavior is not supported in the sample code.

Table 8: 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.	
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 Temperature

The RX111 has a Temperature Sensor module that can monitor the MCU temperature. The ADC12 module is also required in conjunction with the Temperature Sensor.

Table 9: 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	
`Temperature_ADC_Value_Min`	Specify the minimum value that the ADC12 should output when reading the temperature sensor.
`Temperature_ADC_Value_Max`	Specify the maximum value that the ADC12 should output when reading the temperature sensor. NOTE: The ADC output is 12 bit so do not specify a value greater than h'FFF.
`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 behaviour is OK. 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_t Temperature_CheckResult(bool_t 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_t	TRUE: Result falls within specified limits. FALSE: Result falls outside specified limits.

1.8 Port Output Enable (POE)

The port output enable 2 (POE2) module can be used to place the states of the pins for complementary PWM output by the MTU2A (MTIOC3B, MTIOC3D, MTIOC4A, MTIOC4B, MTIOC4C, and MTIOC4D), and the states of pins for MTU0 (MTIOC0A, MTIOC0B, MTIOC0C, and MTIOC0D) in the high-impedance in response to changes in the input levels on the POE0# to POE3# and POE8# pins, in the output levels on pins for complementary PWM output by the MTU2A, oscillation stop detection by the clock generation circuit, and changes to register settings (SPOER), and changes to event signal from the event link controller (ELC).

This software demonstrates the setting of certain pins into the high impedance state when falling edge on POE0 input pin is detected or when oscillation stop is detected.

Table 10: Source files:

File name
POE.h
POE.c

Syntax	
`void POE_Init(POE_CALL_BACK Callback);`	
Description	
This software configures the POE: 1. To put the following pins in the high impedance state if a falling edge on the POE0 (PA_3 pin) input pin is detected. An interrupt is also generated. Pins: MTIOC3B, MTIOC3D, MTIOC4A, MTIOC4B, MTIOC4C, and MTIOC4D. 2. To put the following pins in the high impedance state if Oscillation Stop is detected. Pins: MTIOC0A, MTIOC0B, MTIOC0C, MTIOC0D, MTIOC3B, MTIOC3D, MTIOC4A, MTIOC4B, MTIOC4C, and MTIOC4D.	
Input Parameters	
`POE_CALL_BACK Callback`	Function to call if a falling edge on the POE0 (PA_3 pin) input pin is detected.
Output Parameters	
`NONE`	N/A

Syntax	
`void POE_ClearFlags(void);`	
Description	
Clears the Oscillation Stop High-Impedance status flag. Clears the POE0 to POE3 detection status flags. This will release the pins from the high impedance state.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A

2. Example Usage

In addition to the actual test software source files, a e2studio workspace is provided which includes an example application demonstrating how the tests can be run. This code should be examined in conjunction with this document to see how the various test functions are used.

This example configures the tests so that they can be run on an RSK RX111 that is set for 3.3V operation (Board_VCC = 3.3V); If powering the RSK via the E1 then select the 3.3V option when connecting. When building the workspace the following compiler warning may be displayed:

"C5170 (W) Pointer points outside of underlying object"

This warning is issued by the compiler sometimes when using the section address operators (_sectop and _secend). Despite the warnings the code operation has been verified and so it is safe to ignore the warnings.

The testing can be split into three 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 initialisation code before running all the tests, so that for example a faster main clock can be selected. Note: If building an application where it is not expected that a power down will be performed very often, it may be necessary to schedule these tests more than just at power up.
2. Periodic Tests. These are tests that are run regularly through out 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 sample application sets up a Timer module of the RX111 to periodically call a function (`PeriodicTestCallBack`). Each time this function is called a particular test, or part of a test, is performed. The requirements of the user's application will determine how much time can be spent each time the function is called.
3. Monitoring tests. This is where the RX111 is used in a diagnostic mode to continuously monitor something. Hence the test cannot be classed as either Power-Up or Periodic.

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 position, 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 User mode by function Change_PSW_PM_to_UserMode in resetprg.c.

The function `CPU_Test_All` can be used to automatically run all the CPU tests.

2.1.2 Periodic

If testing 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.

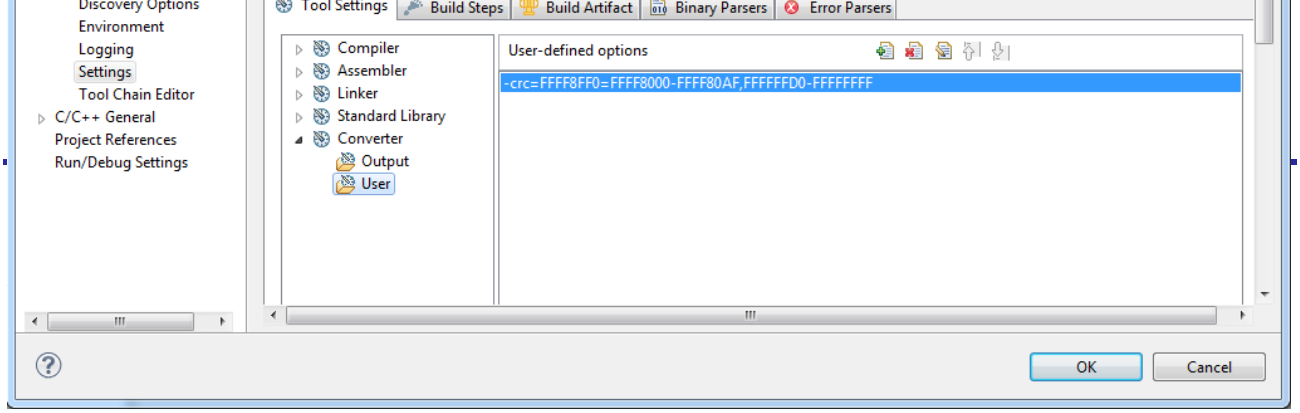

Figure 1: Adding Reference **CRC**.

Note: The e2studio .x file will not contain the CRC reference but is not able to download *.mot file. So the reference CRC value calculated by linker isn't stored on specified address. To fix this problem in Debug configuration on Startup card should be added initialization command. – see Figure 2: Adding initialization command

```
restore HardwareDebug\"projectname".mot
```

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 e2studio and the CRC Test code use so that the results will match.

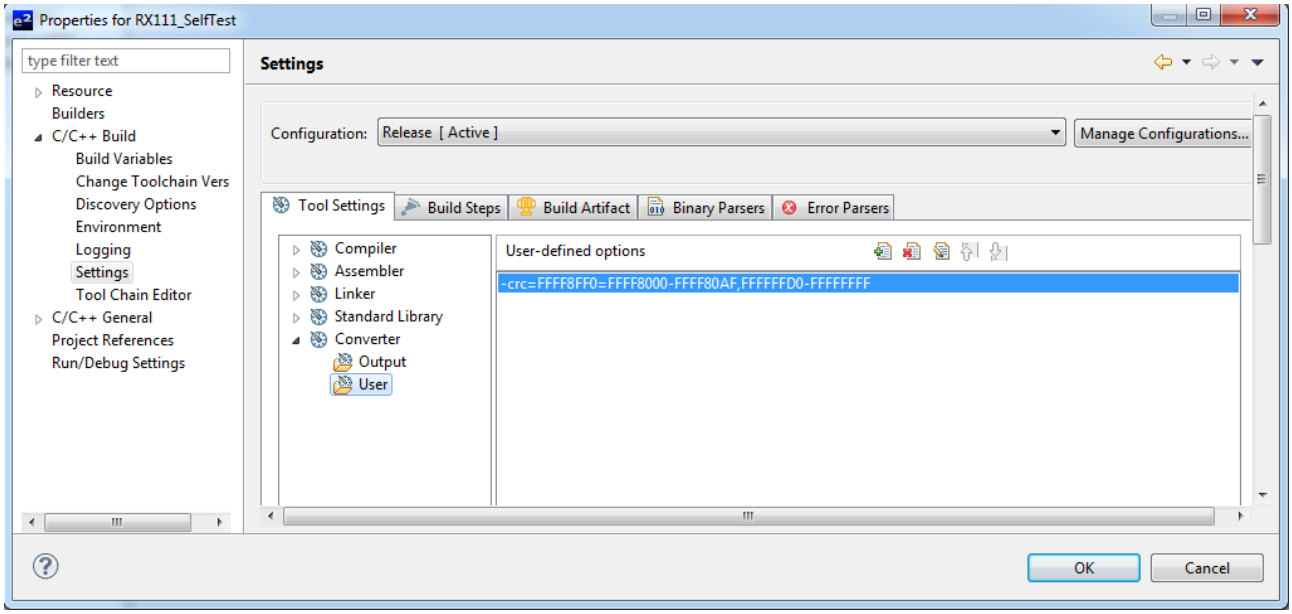

Figure 1: Adding Reference CRC

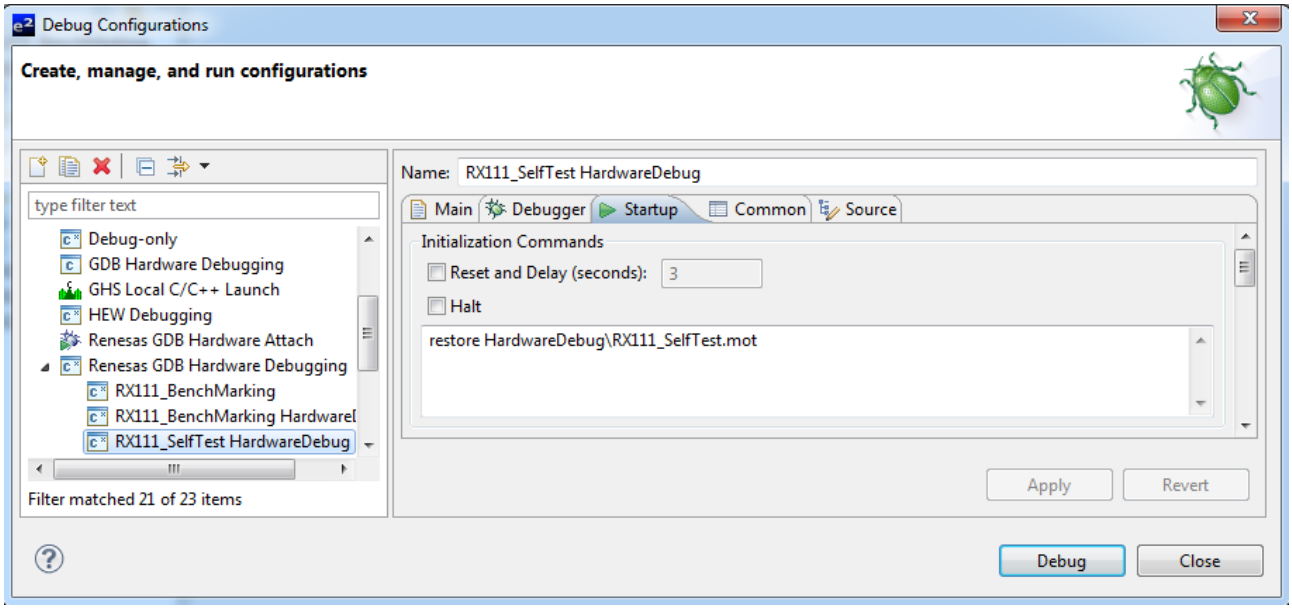

Figure 2: Adding initialization command

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.

The Renesas Rx Compiler provides section address operators, __sectop, __secend and __secsize, that can be used to obtain the addresses of ROM used. The sample application uses these to initialize a structure used during CRC testing:

```
const CRC_RANGE CRC_Ranges[CRC_RANGE_NUM] =
{
  __sectop("PResetPRG"), __secend("PResetPRG"),
  __sectop("C1"), __secend("PPCTEST_TESTFUNCTION"),
  __sectop("FIXEDVECT"), __secend("FIXEDVECT")
};
```

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

The sample application includes the files Test_Usage_RAM.h and .c as an example of testing the RAM.

It is very important to realize, if using this example for your own project, that the area of RAM that needs to be tested may change dramatically depending upon your projects memory map.

The example code makes several assumptions when setting up the #defines which define the RAM areas. See Test_Usage_RAM.c and read the comments carefully when setting them up for your build.

If using the 'HW' versions of the RAM Tests (where the DOC is used) then function RamTest_March_HW_Init must be called prior to running the test. The following #define in file Test_Usage_RAM.c 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 re-located to.
4. There are two stacks, Interrupt and User. It is the current stack that must be re-located before being used.
5. If re-locating the stack the device must be in supervisor mode. The device automatically enters default mode when handling an interrupt.

2.3.1 Power-Up

Providing the RAM power on test is done before global variable initialisation is performed (as done by _INITSCT) a full destructive test can be performed on all 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 is tested using the slower non-destructive test and the rest of the Stack tested with a destructive test.

The sample application provides function Tests_PowerOn_RAM as an example of testing the RAM at power up. The function should be called before the device is put in user mode by function Change_PSW_PM_to_UserMode in resetprg.c.

It uses the March C test algorithm to perform the following steps.

1. A destructive test is performed on the RAM area defined between RAM_START_ADDRESS and RAM_END_ADDRESS. (This area defines all used RAM except for stacks and the RAM_Test_Buffer.)
2. A destructive test is performed on the RAM_Test_Buffer used in the periodic RAM tests.
3. A non-destructive test is performed on the stack area defined between STACK_START_ADDRESS and STACK_END_ADDRESS. The stacks are re-located during this process.

2.3.2 Periodic

The sample code uses the March X WOM test algorithm for all periodic tests. 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 supervisor mode.

The periodic tests are split into three; testing of the stack, testing of the RAM Buffer and testing of the remaining RAM area. The functions PeriodicTest_RAM_Buffer, PeriodicTest_Stack and PeriodicTest_RAM are used for this. The PeriodicTest_Stack and PeriodicTest_RAM functions are both designed to be called repeatedly, by the Periodic test

scheduler, until they return that they have finished. This enables these functions to split the testing up into small enough chunks that a single function call never takes too long.

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 ref3erence clock as decided by the following #define:

```
#define CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK
```

For example:

```
#ifdef CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK

ClockMonitor_Init(CLOCK_FREQ_MAIN,
   1000000,
   eCLOCK_MONITOR_CACREF_PIN_PA0, //JA3_1 on RSK
   //eCLOCK_MONITOR_CACREF_PIN_PC7, //J2_20
   //eCLOCK_MONITOR_CACREF_PIN_PH0, //J2_13
   ClockErrorFunction);
#else
  ClockMonitor_Init(ClockErrorFunction); #endif
```

This can be called as soon as the main clock has been configured and the IWDT has been enabled. See section ‘1.5 Independent Watchdog’ 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.

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 == ICU.NMISR.BIT.OSTST)
{
   Clock_Stop_Detection();

   /*Clear OSTST bit by writing 1 to NMICLR.OSTCLR bit*/
   ICU.NMICLR.BIT.OSTCLR = 1;
}
```

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

2.5 Independent Watchdog

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

```
/*Setup the Independent WDT.
IWDT_Init(IWDT_TOP_2048, IWDT_CKS_DIV_64,
          IWDT_WINDOW_START_NO_START, IWDT_WINDOW_END_NO_END,
          IWDT_ACTION_NMI);
```

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

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

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

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

```
if(TRUE == IWDT_DidReset())
{
   //todo: Handle a watchdog reset.
   while(1){;}
}
```

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. The following example sets up the voltage monitor to generate an NMI if the voltage drops below 2.79V.

```
VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL_2_79);
```

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

```
/*Low Voltage LVD1*/
if(1 == ICU.NMISR.BIT.LVD1ST)
{
 Voltage_Test_Failure();

 /*Clear LVD1ST bit by writing 1 to NMICLR.LVD1CLR bit*/
 ICU.NMICLR.BIT.LVD1CLR = 1;
}
```

2.7 Temperature

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

The temperature sensor must be initialized before use with a call to Temperature_Init. This function must be passed the allowable range of temperatures expressed in terms of the ADC12 output. See the RX111 Hardware Manual for details on how to calculate / find by experiment these values.

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

In the example code the allowable temperature range is set wide to accommodate variations in sensor output so that the code should run without detecting an error at room temperature. However, if an error is detected adjust the allowable range by adjusting the #defines of `TEMPERATURE_ADC_MIN` and `TEMPERATURE_ADC_MAX` to suit your environment / system.

2.7.1 Power-Up

In the example software the temperature is not tested at power up. However, if it is required then the procedure will be the same as explained for the periodic tests.

2.7.2 Periodic

Periodically the use of the ADC12 module must be taken over by the temperature sensor. To make a temperature reading calls 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 for too long it can be arranged 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. See the example usage in function PeriodicTest_Temperature for details.

User's code can use functions Temperature_Is_Finished or Temperature_Wait_Finish to determine when that can resume using the ADC12 to read analog pins.

2.8 POE

The POE initialisation and start up is made using the following call:

```
POE_Init(POE_Event_Detected);
```

The user must carefully study the description of POE_Init and consult the RX111 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 POE.C file may well need to be adapted for the desired behavior.

The sample configuration of the POE means the POE will be triggered by a falling edge on the POE0 (PA_3) pin. To allow the example test usage to run on a default RSK RX111, without triggering due to a floating pin, this pin is internally pulled high by setting `PORTA.PMR.BIT.B3 = 1` before the call to POE_Init.

3. Benchmarking

3.1 Environment

Development board: RSKRX111

Clock: EXTAL = 16MHz, ICLK = 32MHz, PCLKB = 32MHZ, PCLKD = 32MHZ

MCU: R5F51115

Tool chain: RX Standard Toolchain 1.02.01.00

In-circuit debugger: Renesas E1

Complier Settings

Optimize for Size. Level = Max	-include="$(PROJDIR)\src\Tests\Common" -include="$(PROJDIR)\src\Tests\IWDT" -include="$(PROJDIR)\src\Tests\RAM" -include="$(PROJDIR)\src\Tests\ROM" -include="$(PROJDIR)\src\Tests\CPU" -include="$(PROJDIR)\src\Tests\Clock" -include="$(PROJDIR)\src\Tests\Voltage" -include="$(PROJDIR)\src\Tests\Temperature" -include="$(PROJDIR)\src\Tests\POE" -include="$(PROJDIR)\src" -include="C:\PROGRA~2\Renesas\Hew\Tools\Renesas\RX\1_2_1\include"-define=__RX -debug -nologo -change_message=warning -cpu=rx200 -optimize=max
Optimize for Speed. Level = Max	-include="$(PROJDIR)\src\Tests\Common" -include="$(PROJDIR)\src\Tests\IWDT" -include="$(PROJDIR)\src\Tests\RAM" -include="$(PROJDIR)\src\Tests\ROM" -include="$(PROJDIR)\src\Tests\CPU" -include="$(PROJDIR)\src\Tests\Clock" -include="$(PROJDIR)\src\Tests\Voltage" -include="$(PROJDIR)\src\Tests\Temperature" -include="$(PROJDIR)\src\Tests\POE" -include="$(PROJDIR)\src" -include="C:\PROGRA~2\Renesas\Hew\Tools\Renesas\RX\1_2_1\include " -define=__RX -debug -nologo -change_message=warning -cpu=rx200 -optimize=max -speed

NOTE: CPU Test files are built with no optimization.

Linker Settings

Optimize = Speed	-library="$(PROJDIR)\HARDWA~2\RX111_BenchMarking.lib" -noprelink -list="RX111_BenchMarking.map" -show -optimize=speed -rom=D=R,D_1=R_1,D_2=R_2 -nomessage -nologo -output="$(PROJDIR)\HardwareDebug\RX111_BenchMarking.abs"

NOTE: Create following sections in e2studio linker settings for benchmarking tests. Section names are PCPU_TEST, PCRC, PRAM_TEST_MarchC_HW, PRAM_TEST_MarchXWOM_HW, PRAM_TEST_HW, PRAM_TEST_MarchC, PRAM_TEST_MarchXWOM and PRAM_TEST_STACK.

3.2 Results

3.2.1 CPU

Note: Optimization cannot be used for these tests.

Table 11: CPU test results

Measurement	Result Non-Coupling Test	Result Coupling Test
Code size [bytes].	701	3697
Stack usage for CPU_TestAll [bytes]	16	40
Execution time to of function CPU_TestAll	272 [clock cycle count]	1312 [clock cycle count]
	8.5 [µs]	41.0 [µs]

3.2.2 ROM

Table 12: Test results for CRC16-CCITT

		Optimization	
Measurement		**Size**	**Speed**
Code size [bytes]		102	262
Stack usage [bytes]		36	32
Clock cycle count	1k bytes	8320	3840
	4k bytes	32960	14400
	16k bytes	131200	59520
Time Measured [ms]	1k bytes	0.26	0.12
	4k bytes	1.03	0.45
	16k bytes	4.10	1.86

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.

The name ‘Extra’ refers to the function that includes the automatic safe buffer test.

3.2.4 March C

Table 13: March C test results (8-bit access, 32-bit word limit)

			Normal		HW (DOC)	
			Optimization			
Measurement			Size	Speed	Size	Speed
Code size [bytes]			359	2588	365	3676
Stack usage [bytes]			84	148	80	68
Stack usage Extra [bytes]			100	300	80	120
Clock cycle count	Destructive	1024 bytes	695360	297280	652480	299840
		500 bytes	339840	145280	318400	146560
		100 bytes	68160	29120	64000	29440
	Non-destructive	1024 bytes	713600	305280	671040	307520
		500 bytes	348800	149120	327680	150400
		100 bytes	69760	30080	65600	30080
	Extra	1024 bytes	1409280	608640	1323200	600960
		500 bytes	688000	295360	646400	293760
		100 bytes	137920	59840	129920	59200
Time Measured [ms]	Destructive	1024 bytes	21.73	9.29	20.39	9.37
		500 bytes	10.62	4.54	9.95	4.58
		100 bytes	2.13	0.91	2.00	0.92
	Non-destructive	1024 bytes	22.30	9.54	20.97	9.61
		500 bytes	10.90	4.66	10.24	4.70
		100 bytes	2.18	0.94	2.05	0.94
	Extra	1024 bytes	44.04	19.02	41.35	18.78
		500 bytes	21.50	9.23	20.20	9.18
		100 bytes	4.31	1.87	4.06	1.85

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

			Normal		HW (DOC)	
			Optimization			
Measurement			Size	Speed	Size	Speed
Code size [bytes]			408	2740	429	5311
Stack usage [bytes]			80	88	84	100
Stack usage Extra [bytes]			96	140	100	172
Clock cycle count	Destructive	1024 bytes	637120	281920	619200	375360
		500 bytes	311040	137600	302720	183360
		100 bytes	62400	27520	60800	36480
	Non-destructive	1024 bytes	642560	284160	625280	377280
		500 bytes	313920	138880	305280	184320
		100 bytes	63040	27840	61440	36800
	Extra	1024 bytes	1279680	567040	1244480	763200
		500 bytes	625280	277120	607680	373120
		100 bytes	125440	55680	121920	74880
Time Measured [ms]	Destructive	1024 bytes	19.91	8.81	19.35	11.73
		500 bytes	9.72	4.30	9.46	5.73
		100 bytes	1.95	0.86	1.90	1.14
	Non-destructive	1024 bytes	20.08	8.88	19.54	11.79
		500 bytes	9.81	4.34	9.54	5.76
		100 bytes	1.97	0.87	1.92	1.15
	Extra	1024 bytes	39.99	17.72	38.89	23.85
		500 bytes	19.54	8.66	18.99	11.66
		100 bytes	3.92	1.74	3.81	2.34

3.2.5 March X WOM

Table 15: March X WOM test results (8-bit access, 32-bit word limit)

			Normal		HW (DOC)	
			Optimization			
Measurement			Size	Speed	Size	Speed
Code size [bytes]			288	2754	289	2021
Stack usage [bytes]			64	52	68	64
Stack usage Extra [bytes]			80	80	84	92
Clock cycle count	Destructive	1024 bytes	80000	47360	54400	28800
		500 bytes	39040	23360	26560	14080
		100 bytes	8000	4800	5440	2880
	Non-destructive	1024 bytes	98560	55040	72960	36480
		500 bytes	48000	27200	37440	17920
		100 bytes	9920	5760	7360	3840
	Extra	1024 bytes	178560	102080	127360	65280
		500 bytes	87360	50240	62400	32320
		100 bytes	17600	10560	12800	6720
Time Measured [ms]	Destructive	1024 bytes	2.50	1.48	1.70	0.9
		500 bytes	1.22	0.73	0.83	0.44
		100 bytes	0.25	0.15	0.17	0.09
	Non-destructive	1024 bytes	3.08	1.72	2.28	1.14
		500 bytes	1.50	0.85	1.17	0.56
		100 bytes	0.31	0.18	0.23	0.12
	Extra	1024 bytes	5.58	3.19	3.98	2.04
		500 bytes	2.73	1.57	1.95	1.01
		100 bytes	0.55	0.33	0.40	0.21

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

			Normal		HW (DOC)	
			Optimization			
Measurement			Size	Speed	Size	Speed
Code size [bytes]			352	3531	356	3102
Stack usage [bytes]			64	64	68	64
Stack usage Extra [bytes]			80	100	84	96
Clock cycle count	Destructive	1024 bytes	20480	12800	17280	10880
		500 bytes	9920	6400	8640	5440
		100 bytes	2240	1280	1920	1280
	Non-destructive	1024 bytes	25920	15040	23040	13120
		500 bytes	12800	7680	11200	6720
		100 bytes	2560	1600	2560	1280
	Extra	1024 bytes	46400	27520	40320	24320
		500 bytes	22720	14080	19840	12480
		100 bytes	4800	3200	4160	2560
Time Measured [ms]	Destructive	1024 bytes	0.64	0.40	0.54	0.34
		500 bytes	0.31	0.20	0.27	0.17
		100 bytes	0.07	0.04	0.06	0.04
	Non-destructive	1024 bytes	0.81	0.47	0.72	0.41
		500 bytes	0.4	0.24	0.35	0.21
		100 bytes	0.08	0.05	0.08	0.04
	Extra	1024 bytes	1.45	0.86	1.26	0.76
		500 bytes	0.71	0.44	0.62	0.39
		100 bytes	0.15	0.10	0.13	0.08

3.2.6 Stack Test

Note: This does not contain timing information as that depends upon the specific algorithm used. The time to move the stack is negligible compared with the actual memory test, so see the normal RAM test results.

Note: The results are the same regardless of the optimization because inline assembly is used.

	Optimization	
Measurement	**Size**	**Speed**
Code size [bytes] Program	389	389
Code size [bytes] RAM	36	36
Stack usage [bytes]	88	76

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 Input Register as this extract from the Hardware manual specifies:

18.3.3 Port Input Data Register (PIDR)

Address(es): PORT0.PIDR: 0008 C040h, PORT1.PIDR: 0008 C041h, PORT2.PIDR: 0008 C042h, PORT3.PIDR: 0008 C043h, PORT4.PIDR: 0008 C044h, PORT5.PIDR: 0008 C045h, PORTA.PIDR: 0008 C04Ah, PORTB.PIDR: 0008 C04Bh, PORTC.PIDR: 0008 C04Ch, PORTE.PIDR: 0008 C04Eh, PORTH.PIDR: 0008 C051h, PORTJ.PIDR: 0008 C052h

	b7	b6	b5	b4	b3	b2	b1	b0
	B7	B6	B5	B4	B3	B2	B1	B0
Value after reset:	x	x	x	x	x	x	x	x

x: Undefined

Bit	Symbol	Bit Name	Description	R/W
b0	B0	Pm0	Indicates individual pin states of the corresponding port.	R/W
b1	B1	Pm1		R/W
b2	B2	Pm2		R/W
b3	B3	Pm3		R/W
b4	B4	Pm4		R/W
b5	B5	Pm5		R/W
b6	B6	Pm6		R/W
b7	B7	Pm7		R/W

m = 0 to 5, A to C, E, H, J

PIDR indicates individual pin states of port m.

The pin states of port m can be read with the PORTm.PIDR, regardless of the values of PORTm.PDR and PORTm.PMR.

The NMI pin state is reflected in the P35 bit.

The bit corresponding to a pin that does not exist is reserved. A reserved bit is read as undefined and cannot be modified.

Notes when using PH7 as a general input port and PJ6 and PJ7 as general I/O ports

When using PH7 as a general input port, follow the procedure below to make settings.

1. Set the sub-clock oscillator control bit in RTC control register 3 (RCR3.RTCEN) to 0. For details on this register, refer to section 23.2.19, RTC Control Register 3 (RCR3).
2. Set the sub-clock oscillator stop bit in the sub-clock oscillator control register (SOSCCR.SOSTP) to 1. For details on the functions and rewriting of this register, refer to section 9.2.6, Sub-Clock Oscillator Control Register (SOSCCR).

When using PJ6 and PJ7 as a general input port, follow the procedure below to make settings.

1. Set the PJ6PFS.ASEL bit to 0. (When using the PJ6 port)
   Set the PJ7PFS.ASEL bit to 0. (When using the PJ7 port)
2. Set the PORTJ.PMR.B6 to 0. (When using the PJ6 port)
   Set the PORTJ.PMR.B7 to 0. (When using the PJ7 port)

Figure 3: PIDR Register

5. Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry

Revision History

		Description	
Rev.	**Date**	**Section**	**Summary**
1.00	Mar 20, 2014	All	Initial document release

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Handling of Unused Pins

   Handle unused pins in accordance with the directions given under Handling of Unused Pins in the manual.

   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

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

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of an MPU or MCU in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

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

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

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

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

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

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700, Fax: +44-1628-651-804

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-65030, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, 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 Rd., Putuo District, Shanghai, China
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei, 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 906, 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 Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141