RX62T Group

IEC60730 Self Test Code for RX62T Group MCU

R01AN0654EG0100
Rev.1.0
May 25, 2011

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.

In addition to the software modules available, the RX62T also provides additional hardware options which are ideal for safety designs. One key feature is the Port Output Enable (POE) Module. This can be used to force PWM output pins of the MTU (Multi-Function Timer Pulse unit) and large current output pins of the GPT (General PWM Timer) into a high impedance state, regardless of the state of the rest of the CPU. This is a great feature that can be utilised to ensure safety of external loads.

Although VDE cannot certify their operation stand-alone, as they are not part of the IEC60730 software specification, they have assessed the hardware functionality of the RX62T Internal Watch Dogs (Watch Dog Timer WDT and Independent Watch Dog Timer IWDT), the Clock Generation Circuit and the I/O Ports (See VDE Certificate for details).

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

RX62T Group

Contents

1. Tests

1.1 CPU

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

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

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 "Optimisation" 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 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.	
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 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 can not 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 can not 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 – 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, 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 analyse 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 recognises 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 RX62T includes a CRC module that includes support for the CRC16-CCITT. Using this software to drive the CRCmodule 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 can not 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 RX62T 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

(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:

File name
ramtest_march_c.h
ramtest_march_c.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_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:

File name
ramtest_march_x_wom.h
ramtest_march_x_wom.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) 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.

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 RX62T GPT module has a LOCO function that uses the Low-speed OCO clock to detect deviation in the main clock (ICLK) frequency during run time.

Note: The IWDT module must be enabled before using this as the IWDT enables the low-speed OCO.

IMPORTANT NOTE: The E1 debugger can not be used with this LOCO function as the E1 disables the IWDT when code is not running. This in turn affects the LOCO clock which results in the LOCO function wrongly reporting an ICLK deviation.

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 ICLK allowed before an
error is reported. (integer value)*/
#define CLOCK_TOLERANCE_PERCENT  5
```

In addition to the LOCO function the RX62T has an Oscillation Stop Detection Circuit. This is enabled by default from a power on reset. 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.

NOTE: The Oscillation Stop Detection Circuit does not detect if the Low Speed On-Chip oscillator stops, only if the main clock stops.

Table 6: Source files:

File name
clock_monitor.h
clock_monitor.c

Syntax	
`void ClockMonitor_Init (CLOCK_MONITOR_ERROR_CALL_BACK CallBack)`	
Description	
Monitor main clock (ICLK) using LOCO(Low Speed OCO). Also enables NMI interrupt for Oscillation Stop Detection.	
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

1.4.1 Design Note

The absolute value of the main clock can not be measured by the LOCO as the accuracy of the LOCO can not be guaranteed. What can be detected is a deviation in the main clock from its value when the software is first run.

The LOCO function of the General Purpose Timer calculates a running average count that is proportional to the ICLK. Before deviation in the ICLK can be detected against this average vale the clocks must run until an average value can be calculated. This module does this using the following flow:

1. User calls the ClockMonitor_Init function:

   The LOCO function is enabled to calculate an average but not to detect a deviation. An interrupt is configured to happen after 255 frequency LOCO divided clocks.

2. LOCO Interrupt occurs:

   Configure allowable ICLK count range based on the calculated average count value. Enable detection of deviation.

3. If another LOCO Interrupt occurs:

   This means the most recent ICLK count value has deviated beyond the configured allowable range.

   Call the callback function registered by the user.

1.5 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 RX62T is used for this. It will generate an internal reset if the watchdog times out. 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)`	
Description	
Initialise and start the watchdog timer. The parameters specify the time period before the watchdog will time out. After calling this the IWDT_kick function must then be called regularly enough to prevent the watchdog timing out.	
Input Parameters	
`IWDT_TOP TimeOutperiod`	See declaration of enumerated type IWDT_TOP in IWDT.h for details.
`IWDT_CKS_DIV ClockSelection`	See declaration of enumerated type IWDT_CKS_DIV in IWDT.h for details.
Output Parameters	
`NONE`	N/A
Return Values	
`None`	N/A

Syntax	
`void IWDT_Kick(void)`	
Description	
Refresh the watchdog count. This must be called before the watchdog count times out.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`None`	N/A

Syntax
`bool_t IWDT_DidReset(void)`
Description
Returns if the IWDT has timed out. 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.

1.6 Voltage

The RX62T has a Voltage Detection Circuit. This can be used to detect the power supply voltage (Vcc) falling below reference voltages Vdet1 and Vdet2. For each reference voltage it can be configured to produce either an in interrupt or an internal reset. A function is provided to be used after a reset to decide if the Voltage Monitor has caused the reset.

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.LVDST flag.

Table 8: Source files:

File name
Voltage.h
Voltage.c

Syntax

```
void VoltageMonitor_Init(bool_t bEnable_LVD1, bool_t bResetLVD1,
                         bool_t bEnable_LVD2, bool_t bResetLVD2,
                         VOLTAGE_MONITOR_ERROR_CALL_BACK Callback)
```

Description

Initialise and start voltage monitoring.

NOTE: If configured to produce an interrupt then this will be the Non Maskable Interrupt (NMI). This must be handled by user code.

Input Parameters	
`bool_t bEnable_LVD1`	Enable LVD1 monitoring
`bool_t bResetLVD1`	Set TRUE if want a reset following LVD1 low voltage detection. Set FALSE to generate an interrupt.
`bool_t bEnable_LVD2`	Enable LVD2 monitoring
`bool_t bResetLVD2`	Set TRUE if want a reset following LVD2 low voltage detection. Set FALSE to generate an interrupt.
Output Parameters	
`NONE`	N/A
Return Values	

`None`	N/A

Syntax	
`bool_t VoltageMonitor_DidReset(void)`	
Description	
Returns if the Voltage Monitor has detected a low voltage. This can be called after a reset to decide if the reset was caused by the voltage monitor.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	TRUE if low voltage level detected, otherwise FALSE.

1.7 ADC10

The ADC10 module has a diagnostic mode that can be used to test the ADC10 module.

While testing the ADC10 the ADC10 can not be used for normal operation.

The software can be configured to either supply the ADC10 completion interrupt handler (and vector entry) or to supply a function that must be called from a users ADC10 completion interrupt handler.

See `#define ADC10_PROVIDE_INTERRUPT_HANDLER` for details.

Table 9: Source files:

File name
ADC10.h
ADC 10.c

Syntax	
`void Test_ADC10_Init(void)`	
Description	
Initialise the ADC10 module.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A

Return Values	
`NONE`	N/A

Syntax	
`bool_t Test_ADC10_Wait(void);`	
Description	
Test ADC10 module and returns result. Uses reference voltages 0 and VREF. Function waits for two conversions. This is suitable as a power up test.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	TRUE = test passed. FALSE = test failed.

Syntax	
`void Test_ADC10_Start(ADC10_ERROR_CALL_BACK Callback)`	
Description	
Starts an ADC conversion for a diagnostic test. On completion of the conversion the result must be checked. If ADC10_PROVIDE_INTERRUPT_HANDLER is defined then this module will automatically do this. If not defined, the user must call function ADC10_Interrupt when the conversion completes. Each time this function is called it toggles between the reference voltages of 0V and VREF.	
Input Parameters	
`ADC10_ERROR_CALL_BACK Callback`	Function to be called if an error is detected.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

1.8 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 RX62T has two AD12 modules. These functions can be used to independently test each module.

Table 10: Source files:

File name
ADC12.h
ADC 12.c

Syntax	
`void Test_ADC12_Init(ADC12_MODULE module)`	
Description	
Initialise the specified ADC12 module.	
Input Parameters	
`ADC12_MODULE module`	The ADC module (ADC12_MODULE_0 or ADC12_MODULE_1)
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`bool_t Test_ADC12_Wait(ADC12_MODULE module)`	
Description	
Test the specified ADC12 module. This function waits while two ADC conversions are made. 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	
`ADC12_MODULE module`	The ADC module (ADC12_MODULE_0 or ADC12_MODULE_1)
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	TRUE = test passed. FALSE = test failed.

Syntax	
`void Test_ADC12_Start(ADC12_MODULE module, ADC12_ERROR_CALL_BACK Callback)`	
Description	
Setup ADC module so diagnostic test will be performed each time ADC is used. 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_MODULE module`	The ADC module (ADC12_MODULE_0 or ADC12_MODULE_1)
`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_t Test_ADC12_CheckResult(ADC12_MODULE module, bool_t bCallErrorHandler)`	
Description	
Check the ADC diagnostic result is as expected. This must be called after Test_ADC12_Start and then be called periodically or whenever an ADC conversion completes.	
Input Parameters	
`ADC12_MODULE module`	The ADC module (ADC12_MODULE_0 or ADC12_MODULE_1)
`bool_t bCallErrorHandler`	Set TRUE if want the error call-back function supplied to function Test_ADC12_Start to be called, otherwise FALSE.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	TRUE = test passed. FALSE = test failed.

2. Example Usage

In addition to the actual test software source files, a HEW 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.

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.
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 RX62T 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 RX62T is used in a diagnostic mode to continuously monitor something. Hence the test can not 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

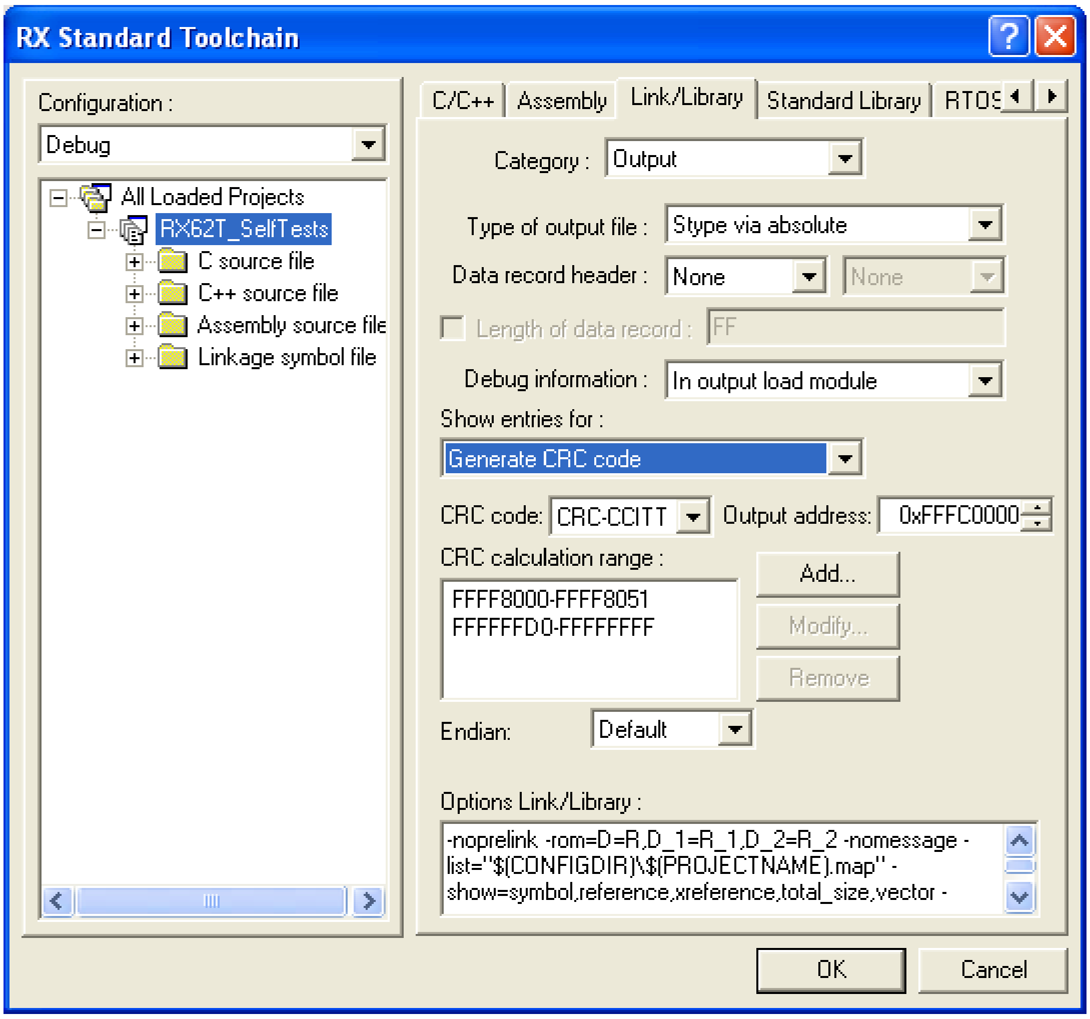
If testing the CPU periodically the function `CPU_Test_All` can be used, as it is for the power-up tests, to automatically run all CPU tests. Alternatively, to reduce the amount of testing done in a single function call, the user can choose to call each of the individual CPU test functions in turn each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

2.2 ROM

The ROM is tested by calculating a CRC value (CRC-CCITT) 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 Renesas RX Standard Toolchain can be used to calculate and add a CRC value to the built mot file at a location specified by the user. This can be done via a dialog in HEW – see Figure 1: Adding Reference CRC.

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



Category : Output

Type of output file : Stype via absolute

Data record header : None None

Length of data record : FF

Debug information : In output load module

Show entries for : Generate CRC code

CRC code: CRC-CCITT Output address: 0xFFFC0000

CRC calculation range :

```
FFFF8000-FFFF8051
FFFFFFD0-FFFFFFFF
```

Endian: Default

Options Link/Library :

```
-noprelink -rom=D=R,D_1=R_1,D_2=R_2 -nomessage -
list="$(CONFIGDIR)\$(PROJECTNAME).map" -
show=symbol,reference,xreference,total_size,vector -
```

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.

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.

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

1. RAM being tested can not 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. For example:

```
ClockMonitor_Init(Clock_Test_Failure);
```

This can be called as soon as the main clock has been configured and the IWDT has been enabled. See section ‘1.5 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 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_16384, IWDT_CKS_DIV_256);
```

After this the watchdog must be refreshed regularly enough so as to stop the watchdog timing out and performing a reset. This is performed by calling;

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

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 perform a reset if the voltage level drops below either VDET1 or VDET2.

```
VoltageMonitor_Init(TRUE, TRUE, TRUE, TRUE);
```

Following a reset the code should check if the Voltage Monitor caused the reset by calling VoltageMonitor_DidReset:

```
if(TRUE == VoltageMonitor_DidReset())
{
  Volatge_Test_Failure();
}
```

The VoltageMonitor_Init function can also be used to configure the Voltage Detection peripheral to generate an NMI interrupt rather than a reset. In this case the NMI handler must check if low voltage was detected as in this example:

```
if(1 == ICU.NMISR.BIT.LVDST)
{
   Volatge_Test_Failure();
}
```

2.7 ADC10

The ADC10 module has a built in diagnostic mode which allows various reference voltages to be tested against.

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

```
#define ADC10_TOLERANCE
```

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

2.7.1 Power-Up

At power up the ADC10 module can be tested using the Test_ADC10_Wait function. This blocks while two AD conversions are performed, one using reference voltage of VREF and the other 0V.

2.7.2 Periodic

To avoid waiting for an AD conversion the periodic test should use the Test_ADC10_Start function. This is a non-blocking function. Each time the function is called the reference voltage for the test is toggled between VREF and 0V.

The result of a conversion must be checked for correctness when the conversion has finished. For this the ADC10 completed interrupt must be handled.

The test module will provide this interrupt handler and automatically test the result if 'ADC10_PROVIDE_INTERRUPT_HANDLER' is #defined. NOTE: If an interrupt occurs when not in diagnostic mode a user supplied function called 'ADC10_Interrupt' will be called.

If 'ADC10_PROVIDE_INTERRUPT_HANDLER' is not #defined then the user must call function 'ADC10_Interrupt' from their own interrupt handler.

2.8 ADC12

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

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

#define ADC12_TOLERANCE

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

2.8.1 Power-Up

At power up the ADC12 module can be tested using the Test_ADC12_Wait function. This blocks while two AD conversions are performed, one using reference voltage of VREF and the other 0V.

2.8.2 Periodic

The periodic testing should start with a single call to Test_ADC12_Start. Following that the ADC12 module will perform a reference conversion each time it is used. The reference voltage is rotated between 0V,VREF/2 and VREF. The result of these reference conversions must be checked periodically using a call to Test_ADC12_CheckResult.

3. Benchmarking

3.1 Environment

Development board: RSKRX62T,

Clock: EXTAL = 12.5MHz,. ICLK = 100MHz, PCLK = 50MHZ

MCU: R5F562TA

Tool chain: RX Standard Toolchain 1.0.0.0

In-circuit debugger: Renesas E1

Complier Settings

Max Level Optimize for Size.	-cpu=rx600 -lang=c -include="$(PROJDIR)\Tests\Common","$(PROJDIR)\Tests\CPU","$(PROJDIR)\Tests\RAM","$(PROJDIR)\Tests\ROM","$(PROJDIR)","$(PROJDIR)\Tests\Clock","$(PROJDIR)\Tests\voltage","$(PROJDIR)\Tests\adc10","$(PROJDIR)\Tests\adc12","$(PROJDIR)\Tests\iwdt" -output=obj="$(CONFIGDIR)\$(FILELEAF).obj" -optimize=max -map="$(CONFIGDIR)\$(PROJECTNAME).bls" -nologo
Max Level Optimize for Speed.	-cpu=rx600 -lang=c -include="$(PROJDIR)\Tests\Common","$(PROJDIR)\Tests\CPU","$(PROJDIR)\Tests\RAM","$(PROJDIR)\Tests\ROM","$(PROJDIR)","$(PROJDIR)\Tests\Clock","$(PROJDIR)\Tests\voltage","$(PROJDIR)\Tests\adc10","$(PROJDIR)\Tests\adc12","$(PROJDIR)\Tests\iwdt" -output=obj="$(CONFIGDIR)\$(FILELEAF).obj" -optimize=max -speed -map="$(CONFIGDIR)\$(PROJECTNAME).bls" -nologo

NOTE: CPU Test files are built with no optimization.

Linker Settings

Optimize = Speed	-map="$(CONFIGDIR)\$(PROJECTNAME).bls" -noprelink -nodebug -rom=D=R,D_1=R_1,D_2=R_2 -nomessage -list="$(CONFIGDIR)\$(PROJECTNAME).map" -optimize=speed -start=B_1,R_1,B_2,R_2,B,R,SU,SI,BADC10_TEST_1,BRAM_TEST_STACK/01000,PResetPRG/0FFFF8000,C_1,C_2,C,C$*,D*,P,PIntPRG,W*,PADC10_TEST,PADC12_TEST,PCPU_TEST,PCRC,PPCTEST_TESTFUNCTION,PCLOCK_MONITOR_TEST,PVOLTAGE_TEST,PIWDT_TEST,PRAM_TEST_MarchC,PRAM_TEST_MarchXWOM,PRAM_TEST_STACK/0FFFF8100,FIXEDVECT/0FFFFFFD0 -nologo -stack -output="$(CONFIGDIR)\$(PROJECTNAME).abs" -end -input="$(CONFIGDIR)\$(PROJECTNAME).abs" -form=stype -output="$(CONFIGDIR)\$(PROJECTNAME).mot" -exit

3.2 Results

3.2.1 CPU

Note: Optimization cannot be used for these tests.

Table 11: CPU test results

Measurement	Result Non-CouplingTest	Result Coupling Test
Code size.	768 bytes	3764 bytes
Stack usage for CPU_TestAll	24 bytes	24 bytes
Execution time to of function CPU_TestAll	290 Clocks	1281 clocks
	3.02 uS	13.35 uS

3.2.2 ROM

Table 12: Test results for CRC16-CCITT

		Optimisation	
Measurement		Size	Speed
Code size / bytes		90	518
Stack usage / bytes		16	4
Clock cycle count (/ 1000)	1k bytes	7968	7872
	4k bytes	31584	31488
	16k bytes	125760	125760
Time Measured (ms)	1k bytes	0.08	0.08
	16k bytes	0.33	0.33
	64k bytes	1.31	1.31

3.2.3 RAM

The tests were executed in 8, 16 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)

Measurement			Optimization	
			Size	Speed
Code size (bytes)			365	1272
Stack usage (bytes)			48	44
Stack usage Extra (bytes)			64	120
Clock cycle count	Destructive	1024 bytes	677760	289920
		500 bytes	331200	141120
		100 bytes	66432	28032
	Non-destructive	1024 bytes	693120	295680
		500 bytes	337920	144960
		100 bytes	67968	29184
	Extra	1024 bytes	1370880	584640
		500 bytes	670080	286080
		100 bytes	134400	57216
Time Measured (ms)	Destructive	1024 bytes	7.06	3.02
		500 bytes	3.45	1.47
		100 bytes	692	292
	Non-destructive	1024 bytes	7.22	3.08
		500 bytes	3.52	1.51
		100 bytes	708	304
	Extra	1024 bytes	14.28	6.09
		500 bytes	6.98	2.98
		100 bytes	1.4	596

Table 14: March C test results (16-bit access, 32-bit word limit)

Measurement	Optimization	
	Size	Speed
Code size (bytes)	409	3921
Stack usage (bytes)	48	72

		Stack usage Extra (bytes)	64	144
Clock cycle count	Destructive	1024 bytes	641280	278400
		500 bytes	312960	136320
		100 bytes	62976	27648
	Non-destructive	1024 bytes	650880	281280
		500 bytes	317760	138240
		100 bytes	63360	28032
	Extra	1024 bytes	1291200	560640
		500 bytes	630720	274560
		100 bytes	126720	55296
Time Measured (ms)	Destructive	1024 bytes	6.68	2.90
		500 bytes	3.26	1.42
		100 bytes	0.66	0.29
	Non-destructive	1024 bytes	6.78	2.93
		500 bytes	3.32	1.44
		100 bytes	0.66	0.29
	Extra	1024 bytes	13.45	5.84
		500 bytes	6.57	2.86
		100 bytes	1.32	0.58

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

			Optimization	
Measurement			**Size**	**Speed**
		Code size (bytes)	408	4843
		Stack usage (bytes)	44	36
		Stack usage Extra (bytes)	60	76
Clock cycle count	Destructive	1024 bytes	611520	233280
		500 bytes	299520	114240
		100 bytes	59904	23040
	Non-destructive	1024 bytes	616320	235200
		500 bytes	301440	115200
		100 bytes	60672	23424
	Extra	1024 bytes	1227840	468480
		500 bytes	600000	229440
		100 bytes	120960	46464
Time Measured (ms)	Destructive	1024 bytes	6.37	2.43
		500 bytes	3.12	1.19

		100 bytes	0.624	0.240
	Non-destructive	1024 bytes	6.42	2.45
		500 bytes	3.14	1.2
		100 bytes	0.632	0.244
	Extra	1024 bytes	12.79	4.88
		500 bytes	6.25	2.39
		100 bytes	1.26	0.484

3.2.5 March X WOM

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

			Optimization	
Measurement			**Size**	**Speed**
		Code size (bytes)	295	2085
		Stack usage (bytes)	32	20
		Stack usage Extra (bytes)	48	44
Clock cycle count	Destructive	1024 bytes	74880	59520
		500 bytes	36864	28800
		100 bytes	7680	5760
	Non-destructive	1024 bytes	92544	65664
		500 bytes	45312	32640
		100 bytes	9216	6528
	Extra	1024 bytes	167040	126720
		500 bytes	81792	62208
		100 bytes	16512	12672
Time Measured (ms)	Destructive	1024 bytes	0.78	0.62
		500 bytes	0.38	0.30
		100 bytes	0.08	0.06
	Non-destructive	1024 bytes	0.96	0.68
		500 bytes	0.47	0.34
		100 bytes	0.10	0.07
	Extra	1024 bytes	1.74	1.32
		500 bytes	0.85	0.65
		100 bytes	0.17	0.13

Table 17: March X WOM test results (16-bit access, 32-bit word limit)

	Optimization

Measurement			Size	Speed
Code size (bytes)			347	2411
Stack usage (bytes)			32	20
Stack usage Extra (bytes)			48	52
Clock cycle count	Destructive	1024 bytes	38784	33024
		500 bytes	18816	16512
		100 bytes	3840	3456
	Non-destructive	1024 bytes	48768	36480
		500 bytes	23808	18432
		100 bytes	4608	3840
	Extra	1024 bytes	87936	69120
		500 bytes	43008	34560
		100 bytes	8832	7680
Time Measured (ms)	Destructive	1024 bytes	0.40	0.34
		500 bytes	0.20	0.17
		100 bytes	0.04	0.04
	Non-destructive	1024 bytes	0.51	0.38
		500 bytes	0.25	0.19
		100 bytes	0.05	0.04
	Extra	1024 bytes	0.92	0.72
		500 bytes	0.45	0.36
		100 bytes	0.09	0.08

3.2.6

Table 18: March X WOM test results (32-bit access, 32-bit word limit)

			Optimization	
Measurement			Size	Speed
Code size (bytes)			353	3627
Stack usage (bytes)			32	20
Stack usage Extra (bytes)			48	36
Clock cycle count	Destructive	1024 bytes	20064	16416
		500 bytes	9888	8160
		100 bytes	2112	1824
	Non-destructive	1024 bytes	25056	18048
		500 bytes	12288	9312
		100 bytes	2592	2304
	Extra	1024 bytes	45120	34560

		500 bytes	22176	17760
		100 bytes	4704	4320
Time Measured (ms)	Destructive	1024 bytes	0.22	0.17
		500 bytes	0.10	0.09
		100 bytes	0.02	0.02
	Non-destructive	1024 bytes	0.26	0.19
		500 bytes	0.13	0.10
		100 bytes	0.03	0.02
	Extra	1024 bytes	0.47	0.36
		500 bytes	0.23	0.19
		100 bytes	0.05	0.05

3.2.7 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 optimisation because inline assembly is used.

	Optimisation	
Measurement	**Size**	**Speed**
Code size (bytes) Program	281	281
Code size (bytes) RAM	36	36
Stack usage (bytes)	12	12

3.2.8 ADC10

Stack usage and execution time is for function Test_ADC10_Wait.

	Optimisation	
Measurement	**Size**	**Speed**
ROM Size Program + Data	363 Bytes	393 Bytes
RAM Size (excluding stack)	4 Bytes	4 Bytes
Stack usage	16 Bytes	4 Bytes
Execution Time: uS	16 uS	16 uS
Execution Time: Clocks Cycles	1536	1536

3.2.9 ADC12

Stack usage and execution time is for function Test_ADC12_Wait.

	Optimisation	
Measurement	**Size**	**Speed**
ROM Size Program + Data	318 Bytes	422 Bytes
RAM Size (excluding stack)	0 Bytes	0 Bytes
Stack usage	12 Bytes	4 Bytes
Execution Time: uS	45 uS	45 uS
Execution Time: Clocks Cycles	4320	4320

3.2.10 Voltage Monitor

This software configures a peripheral that continually monitors and is therefore not applicable for benchmarking for execution time.

	Optimisation	
Measurement (Bytes**)**	**Size**	**Speed**
ROM Size: Program + Data	229	96
RAM Size (excluding stack)	0	0
Stack usage	4	4

3.2.11 Clock Monitor

This software configures a peripheral that continually monitors and is therefore not applicable for benchmarking for execution time.

	Optimisation	
Measurement (Bytes**)**	**Size**	**Speed**
ROM Size: Program + Data	275	133
RAM Size (excluding stack)	0	0
Stack usage	4	4

3.2.12 Watchdog

This software configures a peripheral that continually monitors and is therefore not applicable for benchmarking for execution time.

	Optimisation	
Measurement (Bytes**)**	**Size**	**Speed**
ROM Size: Program + Data	147	78
RAM Size (excluding stack)	0	0
Stack usage	12	4

4. Utilities

4.1 Low Power Control

The RX62T supports the following Low Power Modes as presented in this extract from the Hardware manual:

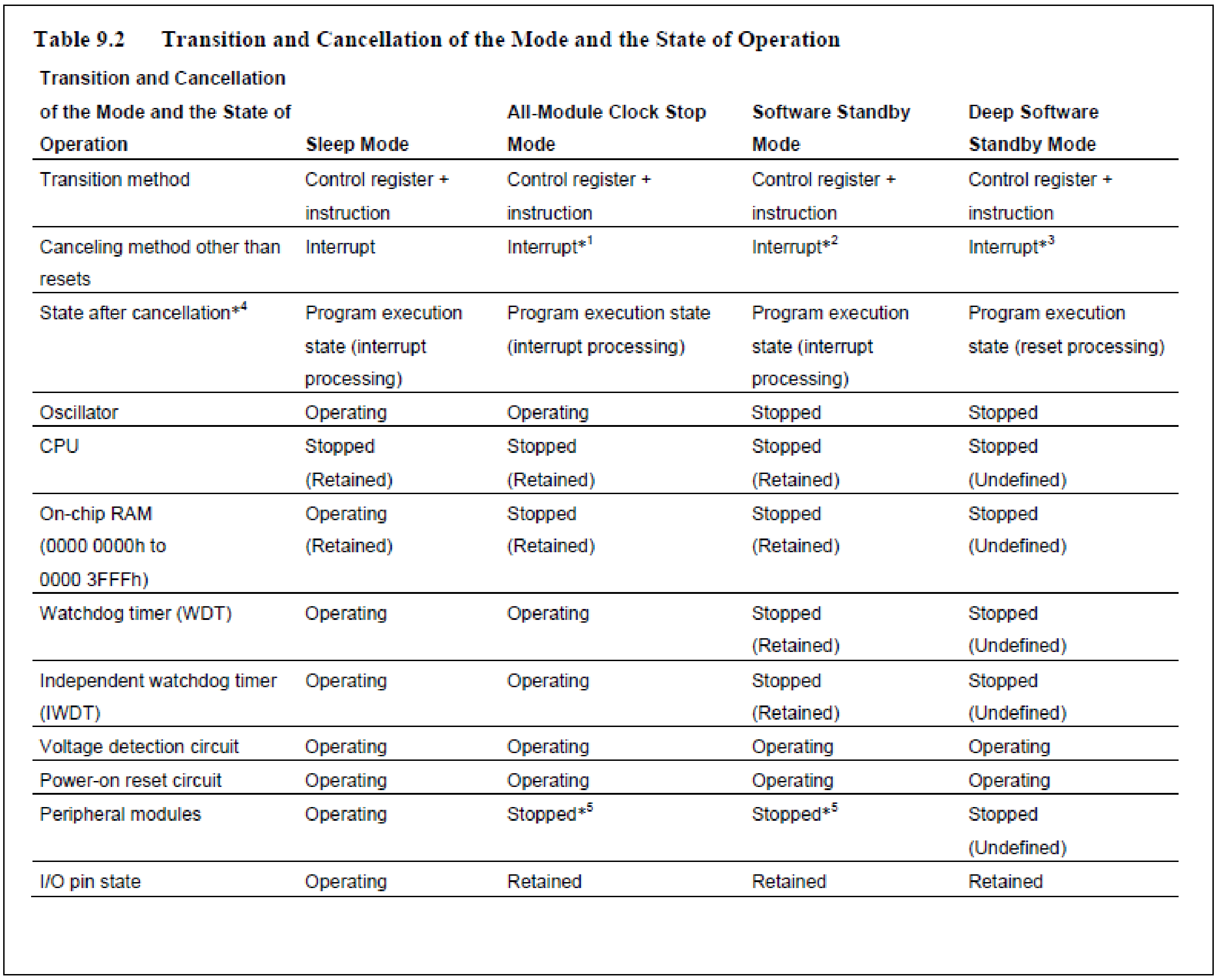


Table 9.2 Transition and Cancellation of the Mode and the State of Operation

Transition and Cancellation of the Mode and the State of Operation	Sleep Mode	All-Module Clock Stop Mode	Software Standby Mode	Deep Software Standby Mode
Transition method	Control register + instruction	Control register + instruction	Control register + instruction	Control register + instruction
Canceling method other than resets	Interrupt	Interrupt*1	Interrupt*2	Interrupt*3
State after cancellation*4	Program execution state (interrupt processing)	Program execution state (interrupt processing)	Program execution state (interrupt processing)	Program execution state (reset processing)
Oscillator	Operating	Operating	Stopped	Stopped
CPU	Stopped (Retained)	Stopped (Retained)	Stopped (Retained)	Stopped (Undefined)
On-chip RAM (0000 0000h to 0000 3FFFh)	Operating (Retained)	Stopped (Retained)	Stopped (Retained)	Stopped (Undefined)
Watchdog timer (WDT)	Operating	Operating	Stopped (Retained)	Stopped (Undefined)
Independent watchdog timer (IWDT)	Operating	Operating	Stopped (Retained)	Stopped (Undefined)
Voltage detection circuit	Operating	Operating	Operating	Operating
Power-on reset circuit	Operating	Operating	Operating	Operating
Peripheral modules	Operating	Stopped*5	Stopped*5	Stopped (Undefined)
I/O pin state	Operating	Retained	Retained	Retained

Figure 2: Low Power Modes

The Power software module supports a switching between all of these modes however if using the IWDT the Software Standby and the Deep Software Standby Modes are not accessible. The

Table 19: Source files:

File name
Power.h
Power.c

Syntax	
`void Power_Init(void)`	
Description	
Initialise the power module. Call this function before using any of the other functions in this module.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

`void Power_Set(etPOWER_MODE eMode)`	
Description	
Set device to specified low power mode. NOTE: Modes Software Standby and Deep Software Standby can not be entered if using the WDT or IWDT. NOTE: If using the IWDT it must still be kicked regularly after going to Sleep or All Module Clock Stop mode. If "`IWDT_KICK_IN_SLEEP_MODE`" is defined then this module will do this automatically by using the WDT module to regularly wake us up so we can kick IWDT and then go back to sleep. This function generates a BRK interrupt. The handler of which must then call the Power_BRK_InterruptHandler function which will then actually perform the switch to the low power mode.	
Input Parameters	
`eMode`	Required low power mode.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

`bool_t Power_Did_DeepStandbyReset (void)`	
Description	
Following a reset this function can be used to see if the reset was caused by an exit from Deep Software Standby Mode. NOTE: This function is only applicable if using Deep Software Standby Mode.	
Input Parameters	
`NONE`	N/A
Output Parameters	

`NONE`	N/A
Return Values	
`bool_t`	Returns TRUE if Deep Software Standby Mode has been exited, otherwise FALSE.

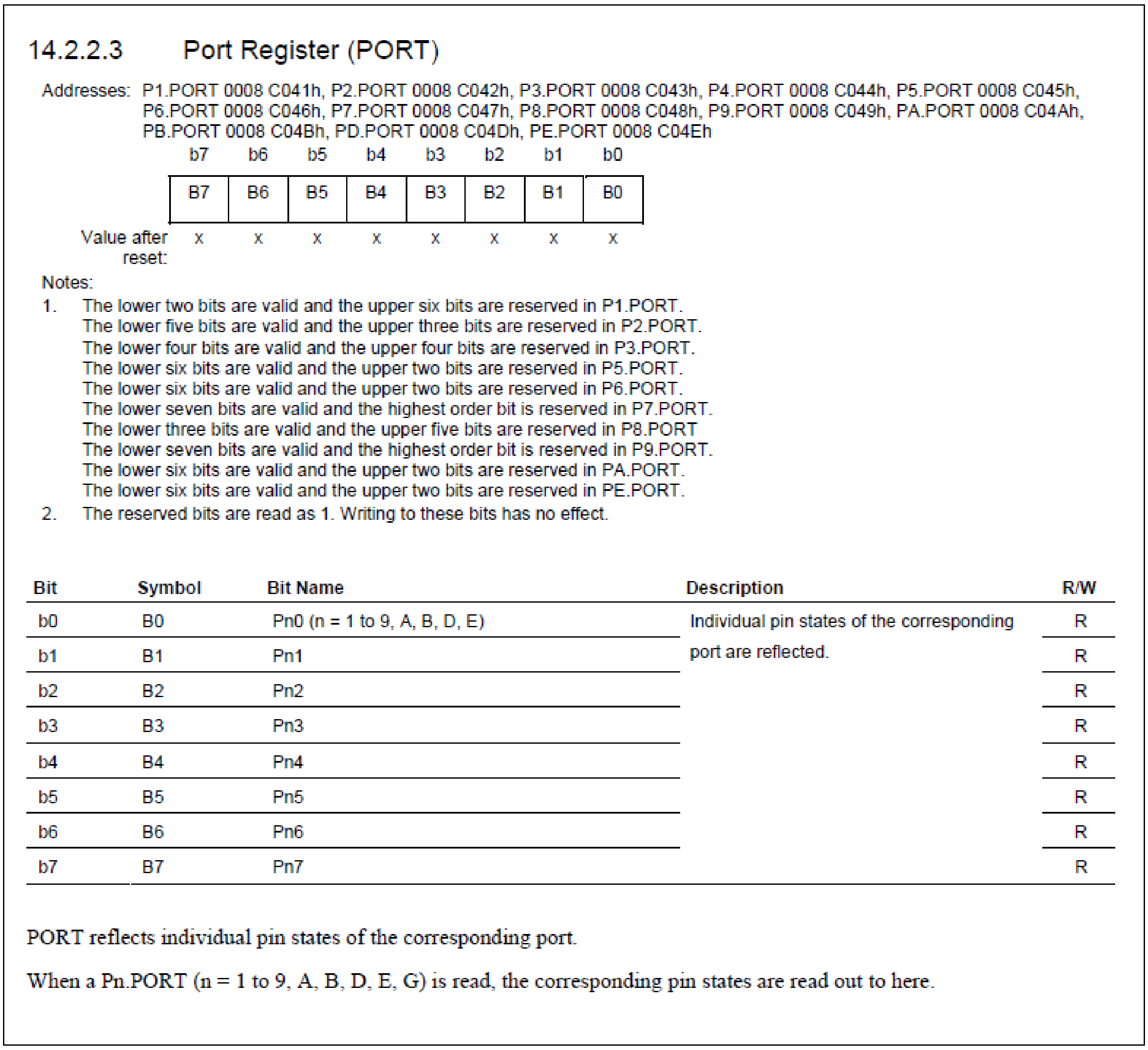
`void Power_DeepStandby_IOResume (void)`	
Description	
Following a reset to exit Deep Software Standby Mode the IO pin states shall be retained regardless of the LSI Internal state until this function is called. NOTE: This function is only applicable if using Deep Software Standby Mode.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

`void Power_BRK_InterruptHandler (void)`	
Description	
The BRK interrupt handler must call this function. This function will therefore be called in Supervisor Mode and can actually execute the privileged WAIT instruction to enter the low power mode requested in function Power_Set.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

5. Additional Information

5.1 Reading an IO Pin State

The actual value of an IO pin can always be read by reading the corresponding pin's PORT register as this extract from the Hardware manual specifies:



14.2.2.3 Port Register (PORT)

Addresses: P1.PORT 0008 C041h, P2.PORT 0008 C042h, P3.PORT 0008 C043h, P4.PORT 0008 C044h, P5.PORT 0008 C045h, P6.PORT 0008 C046h, P7.PORT 0008 C047h, P8.PORT 0008 C048h, P9.PORT 0008 C049h, PA.PORT 0008 C04Ah, PB.PORT 0008 C04Bh, PD.PORT 0008 C04Dh, PE.PORT 0008 C04Eh

	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

Notes:

1. The lower two bits are valid and the upper six bits are reserved in P1.PORT.
   The lower five bits are valid and the upper three bits are reserved in P2.PORT.
   The lower four bits are valid and the upper four bits are reserved in P3.PORT.
   The lower six bits are valid and the upper two bits are reserved in P5.PORT.
   The lower six bits are valid and the upper two bits are reserved in P6.PORT.
   The lower seven bits are valid and the highest order bit is reserved in P7.PORT.
   The lower three bits are valid and the upper five bits are reserved in P8.PORT
   The lower seven bits are valid and the highest order bit is reserved in P9.PORT.
   The lower six bits are valid and the upper two bits are reserved in PA.PORT.
   The lower six bits are valid and the upper two bits are reserved in PE.PORT.
2. The reserved bits are read as 1. Writing to these bits has no effect.

Bit	Symbol	Bit Name	Description	R/W
b0	B0	Pn0 (n = 1 to 9, A, B, D, E)	Individual pin states of the corresponding port are reflected.	R
b1	B1	Pn1		R
b2	B2	Pn2		R
b3	B3	Pn3		R
b4	B4	Pn4		R
b5	B5	Pn5		R
b6	B6	Pn6		R
b7	B7	Pn7		R

PORT reflects individual pin states of the corresponding port.

When a Pn.PORT (n = 1 to 9, A, B, D, E, G) is read, the corresponding pin states are read out to here.

Figure 3: PORT Register

5.2 Port Output Enable Module

The RX62T has a POE module. This can be used to force PWM output pins of the MTU (Multi-Function Timer Pulse unit) and large current output pins of the GPT (General PWM Timer) into the high impedance state regardless of the state of the rest of the CPU. The condition when this occurs can be configured for when:

- POE Input pins see a falling edge or low-level.
- Oscillation-stop detection circuit in the clock pulse generator detects stopped oscillation.

See the RX62T Hardware Manual for details.

6. Website and Support

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

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

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	May 25, 2011	—	First edition issued

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 accord 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.

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.
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: +44-1628-585-100, Fax: +44-1628-585-900

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 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

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.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

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.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141