AN11208

NXP LPC Cortex-M0 IEC60335 Class B library

Rev. 1.1 — 22 January 2014 **Application note**

Document information

Info	Content
Keywords	NXP ARM Cortex-M0, IEC60335 Class B, VDE, LPC1100, LPC1200
Abstract	This application note describes the IEC60335 Class B certified library for the NXP ARM Cortex-M0 family members. All tests implemented and the library usage are described in detail.

Revision history

Rev	Date	Description
1.1	20140122	Minor edits.
1	20120601	Initial version.

Contact information

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

Modern day home appliances require a certain level of protection in order to avoid hazardous situations if the appliance fails. Since 2007, home appliances must comply with the IEC60335 standard. Home appliance manufacturers therefore need to ensure that the requirements are met.

This document describes the IEC60335 standard requirements with respect to software for microcontrollers and the implementation of these requirements. NXP has developed a software library for the NXP ARM Cortex-M0 family, based on these requirements; this document discusses the tests and the usage of these tests in detail.

ATTENTION!

The usage of this library does not make a certified application of your project. It is still necessary to have the complete application software certified.

This library should not be changed, and it should be used as explained. Otherwise, a new certification for the changed parts will be necessary.

The library is usable, as-is, for **all** NXP ARM Cortex-M0 products, including those not specifically mentioned in this application note.

1.1 How to read this application note

This application note is a guide in using and implementing the library functions provided. It will first discuss the requirements the IEC60335 standard sets, and then briefly discuss the products the library is developed for.

The main part of the document describes how the Class B tests are done and how it can and should be implemented. Details on the tested peripherals are given in the last chapter.

2. IEC60335 Class B

The IEC60335 standard specifies design enhancements for home appliance manufacturers that design appliances with electronic controls and controls using software with respect to safe and reliable operation. This standard requires inclusion of features that will avoid or at least minimize the change of hazardous situations when the appliance fails.

Referring to IEC60730, this deals with standard various assets of safety and reliability precautions required to be taken for all home appliances. Annex H of the IEC60730 standard software and hardware requirements are defined to be taken in order to comply with this standard.

2.1 Software classification

Within the IEC60730 Annex H, details for testing and diagnostic implementation in microcontroller software are classified as A, B or C.

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

- Class **B**: Control functions intended to prevent unsafe operation of the controlled equipment.

- Class **C**: Control functions which are intended to prevent special hazards (e.g. explosion of the controlled equipment such as burner controls).

The majority of home appliances, like white goods (refrigerator, dishwasher, cooker etc.) and personal appliances (electrical tooth brush, shaver etc.), require the Class B level of precautions.

IEC60370 Class B specifies that measures must be taken to avoid software related faults and errors in data and segments of the software that are safety related. Periodic monitoring of the system therefore is required.

2.2 Class B components

Table H.11.12.7 of IEC60730 Annex H specifies the components to be tested and monitored during operation of the controller. Table 1 shows a summary of table H.11.12.7.

Table 1.   IEC60335 Class B tests as defined by IEC60730 Annex H

Test number	Component	Fault/error	In library
1.1.	CPU registers	Stuck at	YES
1.3.	Program Counter	Stuck at	YES
2.	Interrupt handling and execution	No interrupt or too frequent interrupt	YES
3.	Clock	Wrong frequency (for quartz synchronized clock: harmonics/subharmonics only)	YES[1]
4.1.	Invariable memory	All single bit faults	YES
4.2.	Variable memory	DC Fault	YES
4.3.	Addressing (relevant to variable and invariable memory)	Stuck at	YES
5.1.[2]	Internal data path	Stuck at	NO
5.2.[2]	Addressing	Wrong address	NO
6.	External communications	Hamming distance 3	NO
6.3.	Timing	Wrong point in time and sequence	NO
7.[3]	Input/output periphery	Fault conditions specified in H.27	NO
7.2.1.[3]	A/D and D/A converters	Fault conditions specified in H.27	NO
7.2.2.[3]	Analog multiplexer	Wrong addressing	NO
			NO

[1]   Only applicable of the NXP ARM Cortex-M0 family members with RTC domain.

[2]   Only when using external memory.

[3]   Production plausibility check.

3. NXP ARM Cortex-M0 microcontrollers

This chapter gives a general description of the NXP ARM Cortex-M0 family members for which the IEC60335 Class B self-test libraries are written.

3.1 The NXP ARM Cortex-M0 microcontrollers

The NXP LPC1100(L) is the world's first Cortex-M0 based microcontroller series. It offers users a cost-effective, very easy-to-use 32-bit MCU which is code- and tool-compatible with other NXP ARM-based MCU products. With 32-bit performance and multiple power modes including very low, deep sleep power, the LPC1100(L) series offers industry-leading energy efficiency, greatly extending battery life. The LPC1100(L) series sets new benchmarks in performance efficiency with dramatically improved code density enabling longer battery life and lower system costs.

3.1.1 The ARM Cortex-M0 core

The ARM Cortex-M0 processor is the smallest, lowest-power and most energy-efficient ARM processor available. The exceptionally small silicon area, low power, and minimal code footprint of the processor achieves 32-bit performance at an 8-bit price point, bypassing the step to 16-bit devices.

The Cortex-M0 processor promises substantial savings in system cost while retaining tool and binary compatibility with feature-rich processors such as the Cortex-M0 processor. It consumes as little as 85 microwatts/MHz (0.085 milliwatts) in an area of typically under 12 K gates, enabling the creation of ultra low-power analog and mixed signal devices.

Fig 1.   The Cortex-M0 core
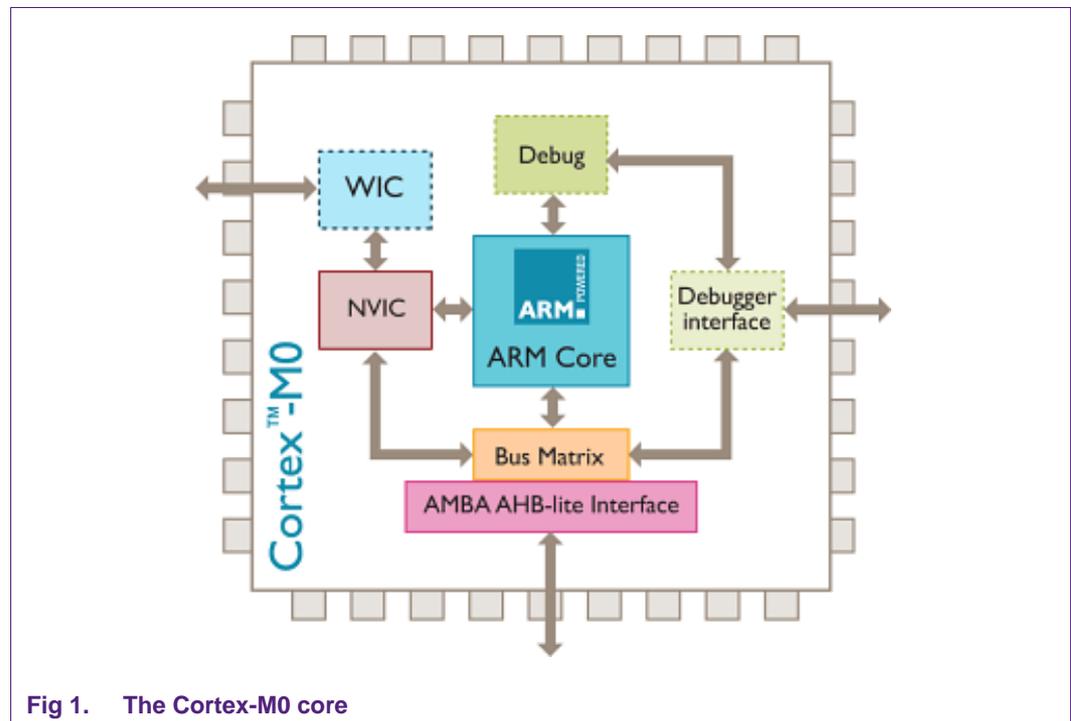
The Cortex-M0 processor is built on a highly area and power optimized 32-bit processor core, with a 3-stage pipeline von Neumann architecture. The processor delivers exceptional energy efficiency through a small but powerful instruction set and extensively optimized design, providing high-end processing hardware including a single-cycle multiplier.

The Cortex-M0 processor implements the ARMv6-M architecture, which is based on the 16-bit Thumb instruction set and includes Thumb-2 technology. This provides the exceptional performance expected of a modern 32-bit architecture, with a higher code density than other 8-bit and 16-bit microcontrollers.

The Cortex-M0 processor closely integrates a configurable Nested Vectored Interrupt Controller (NVIC), to deliver industry-leading interrupt performance. The NVIC:

- Includes a non-maskable interrupt (NMI). The NMI is not implemented on the LPC111x/LPC11Cxx.
- Provides zero jitter interrupt option.
- Provides four interrupt priority levels.

The tight integration of the processor core and NVIC provides fast execution of interrupt service routines (ISRs), dramatically reducing the interrupt latency. This is achieved through the hardware stacking of registers, and the ability to abandon and restart load-multiple and store-multiple operations. Interrupt handlers do not require any assembler wrapper code, removing any code overhead from the ISRs. Tail-chaining optimization also significantly reduces the overhead when switching from one ISR to another.

To optimize low-power designs, the NVIC integrates with the sleep modes that include a Deep-sleep function that enables the entire device to be rapidly powered down.

3.2 Product options

The NXP Cortex-M0 product portfolio is growing rapidly. Discover the latest ARM Cortex-M0 processors on our website:

http://www.nxp.com/products/microcontrollers/cortex_m0_m0/

4. IEC60335 Class B library

The chapter gives an overview about the functionality of the various functions and illustrates how the functions are implemented. It gives you knowledge about the library and helps with understanding the self-test philosophy. Please note by changing any library functionality it need to be re-certified again. If a special part needs to be modified, then there will be an explicit description and explanation.

4.1 POST and BIST

POST (Pre Operation System Test) means the testing as part of the start-up procedure. These tests are destroyable, i.e., the data contents are not restored after executing the test. Also, in this state of application, there are normally no interrupts active.

Note, at start-up all tests must be executed: CPU registers, PC, RAM and ROM. For this reason special POST functions are available. The POST testing is developed such that it reduces test time and therefore is monolithic and destroyable.

For runtime testing or Build-In System Test (Build-In System Test), the test functions are non-destructible. To avoid system failures in time critical applications, these test are not monolithic. Functions are implemented for testing the variable and non-variable in smaller blocks.

The POST routines for testing the CPU registers, the variable memory (RAM) and invariable memory (Flash) are available as separate modules, and should be run in sequence at system startup (after reset).

AN11208

Application note **Rev. 1.1 — 22 January 2014** **8 of 76**

4.2 CPU Register Test (1.1)

4.2.1 Test description

As described in Chapter 5.1 the ARM Cortex-M0 core has a number of registers used during program execution. Nineteen of these registers are read/write.

Since these registers are all used during program execution in the various core operation modes, they are tested for stuck-at faults and direct coupling faults.

These tests are to be executed as POST and BIST. POST testing is a destroyable test, so the CPU registers are not retained. Since the POST CPU register tests don't retain register data, it is mandatory to execute this test prior any other application or system initialisation. Preferably execute this test prior to branch to main.

CPU BIST testing isn't destroyable, so all data is restored after testing. To decrease test time and therefore CPU resources, the CPU register BIST testing is divided into five separate tests. The first three test the general purpose registers, the fourth tests the stack pointer. To prevent the system from crashing, all interrupts and exceptions are disabled while running this part of the CPU register BIST. The fifth and last BIST test will test the other special registers.

Both BIST and POST use the same test methodology when testing the registers. First, a pattern is stored in the register, then read back and compared. Then, the inverse of that pattern is stored in the register, read and compared.

4.2.1.1 Failure

If a failure in the POST testing occurs, a function hook has been implemented. The user has the option to handle a failing CPU POST test by implementing the **_CPUpostTestFailureHook** function.

The basic pattern used for the CPU register tests is the following:

- Normal: `0xAAAA.AAAA`
- Inverted: `0x5555.55555`

AN11208

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2014. All rights reserved.

Application note **Rev. 1.1 — 22 January 2014** **9 of 76**

4.2.2 Test usage

This chapter describes the files used and summarizes all function calls used in CPU register POST and BIST testing.

The tests are developed in assembly code since for POST the CPU registers need to be tested before branching to the `main` routine. The C environment (stack, initialised variables, and zero initialised variables) is not available at the time the POST test needs to be performed. Additionally, the CPU register test POST does not require usage of memory.

The BIST tests are developed in assembly code because most of the registers of the core are not directly accessible from C code.

4.2.2.1 IEC60335_B_CPUregTest.h

File name	Function prototyping
IEC60335_B_CPUregTest.h	**type_testResult** IEC60335_CPUregTest_BIST(void);
	extern void _CPUregTestLOW(**void**);
	extern void _CPUregTestMID(**void**);
	extern void _CPUregTestHIGH(**void**);
	extern void _CPUregTestSP(**void**);
	extern void _CPUregTestSPEC(**void**);
	Type definition
	IEC60335_CPUreg_struct

This header file contains all function prototypes and the structure type definition used during the CPU BIST register tests. It therefore enables the C source files to call the Assembly source routines for the BIST test.

AN11208

Application note **Rev. 1.1 — 22 January 2014** **10 of 76**

4.2.2.2 IEC60335_B_CPUregTest.c

File name	Function prototyping
IEC60335_B_CPUregTest.c	**type_testResult** IEC60335_CPUregTest_BIST (**void**)

This file is responsible for the full CPU BIST test routine definition.

Function:

type_testResult IEC60335_CPUregTest_BIST

Purpose:

The type_testResult IEC60335_CPUregTest_BIST (void) function executes the full BIST test. After this test is executed, the CPUregTestBIST_struct contains the full pass/fail indication, testPassed. The testState will also be updated, which indicates the passing tests according to Table 3.

Return value:

IEC60335_testPassed

IEC60335_testFailed

Important file or function notifications:

- The full IEC60335_CPUregTest_BIST may only be executed in thread mode since the test performs checking of the SP register, which cannot be modified while the core is executing in handler mode (during an exception).
- Test pass/fail available through function return and also available in the testPassed member of the CPUregTestBIST structure.

AN11208

Application note **Rev. 1.1 — 22 January 2014** **11 of 76**

4.2.2.3 IEC60335_B_CPUregTestBIST_nnn.asm

File name	Function prototyping
`IEC60335_B_CPUregTestBIST_nnn`[1]`.asm`	**void** _CPUregTestLOW(**void**);
	void _CPUregTestMID(**void**);
	void _CPUregTestHIGH(**void**);
	void _CPUregTestSP(**void**);
	void _CPUregTestSPEC(**void**);

[1] The `nnn` in the .asm file names must be replaced by a compiler indicator.
   `gnu` = GNU GCC compiler
   `arm` = ARM Realview compiler
   `iar` = IAR EWARM compiler

This file contains all routines for testing the CPU registers during program execution and it gives the user access to the required functions used by the CPU register BIST testing.

The registers tested by the test functions are:

Table 2.   CPU register BIST functions

Test function name	Register tested
`_CPUregTestLOW`	R0 - R7
`_CPUregTestMID`	R4 – R10
`_CPUregTestHIGH`	R8 – R12
`_CPUregTestSP`	R13, stackpointer (MSP, PSP)
`_CPUregTestSPEC`	LR, APSR, PRIMASK,

After each individual test the test structure is updated and therefore contains the latest test values. Each test will reset the testPassed structure member and write the new pass or fail status. The testState member will also be updated after each test with the status of all passing tested registers.

Important file or function notifications:

- The _CPUregTestSP can be performed only in thread mode, since it requires modification of the CONTROL register. As per ARM CPU specification, all direct writes to the CONTROL register are ignored whilst in handler mode, so this function cannot be used within a handler routine. All other functions can be executed also in handler mode (within an ISR exception).

- After test execution, the passing tests will be given a PASS bit in the `CPUregTestBIST_struct testState` member according to Table 3.

- After test execution, and all containing tests pass, `CPUregTestBIST_struct testPassed` will be set to IEC60335_testPassed = 1.

AN11208

Application note

All information provided in this document is subject to legal disclaimers.

Rev. 1.1 — 22 January 2014

© NXP B.V. 2014. All rights reserved.

12 of 76

4.2.2.4 IEC60335_B_CPUregTestPOST_nnn.asm

File name	Function prototyping
IEC60335_B_CPUregTestPOST_nnn[1].asm	**void** _CPUregTestPOST(**void**);

[1] The nnn in the .asm file names must be replaced by a compiler indicator.
gnu = GNU GCC compiler

arm = ARM Realview compiler

iar = IAR EWARM compiler

This file contains the POST testing routing of the complete set of CPU registers. It gives the user access to the CPU register POST.

The routine is made available by means of the _CPUregTestPOST assembly label.

Important file or function notifications:

- The CPUregTestPOST function must be executed prior to the branch to main. It should also execute in Privileged Thread mode.

- After test execution, and all included tests pass, the variable type_testResult CPUregTestPOSTStatus will be set to IEC60335_testPassed = 1

- The variable CPUregTestPOSTStatus needs to be defined but can be located in any software module which is part of the application code, as long as its scope is made visible (so it cannot be declared as a C *static* variable). This status variable should preferably be defined in a dedicated module, which the user could place in a specific section of the device RAM memory, according to its application requirements.

- The variable CPUregTestPOSTStatus needs to be defined as being "not initialised", to prevent its value being changed by the application initialisation code before reaching *main* , so that the POST test result is preserved.

- In case of failure during test execution, the variable type_testResult CPUregTestPOSTStatus will be set to IEC60335_testPassed = 0

- The variable CPUregTestPOSTStatus is defined as a C language enumerative value. This implies the compiler might choose to represent the value with a single byte of data; in this case, the module including the CPUregTestPOSTStatus variable shall be compiled with the appropriate compiler specific option so that a 4 byte value is used to define enumerative values. As an alternative, the return value can be stored in an unsigned integer variable, which is also guaranteed to be 4 byte size.

- In case of failure, the CPUregTestPOST will behave in two possible ways:

  - (default) The CPU will be kept in a safe state, executing an infinite loop.

    This behavior can be overridden by the user application, by re-defining the function _CPUpostTestFailureHook in an assembly module included within the application code

  - (application specific) The function CPUpostTestFailureHook is an optional assembly function, located in a module included in the user application.

    This allows the system to perform different or additional recovery actions than the one described in point 1 above.

4.2.2.5 CPU register test numbers

During the BIST testing the testState member of the test structure is updated with the passing tests. Table 3 depicts the tested register and its corresponding bit value found in the testState.

Table 3. **CPU register test table**

Test number	Hexadecimal bit value	Register	Bits tested
0	0x0000 0001	R0	31:0
1	0x0000 0002	R1	31:0
2	0x0000 0004	R2	31:0
3	0x0000 0008	R3	31:0
4	0x0000 0010	R4	31:0
5	0x0000 0020	R5	31:0
6	0x0000 0040	R6	31:0
7	0x0000 0080	R7	31:0
8	0x0000 0100	R8	31:0
9	0x0000 0200	R9	31:0
10	0x0000 0400	R10	31:0
11	0x0000 0800	R11	31:0
12	0x0000 1000	R12	31:0
13	0x0000 2000	R13 (default SP, MSP)	31:2
14	0x0000 4000	R13 (alternative SP)	31:2
15	0x0000 8000	R14 (LR)	31:0
16	0x0001 0000	APSR	31:27
17	0x0002 0000	PRIMASK	0

AN11208

Application note **Rev. 1.1 — 22 January 2014** **14 of 76**

4.3 Program Counter (PC) Test (1.3)

4.3.1 Test description

The Program Counter test checks whether the PC is able to branch throughout the whole program and data memory space. To test the branching, dummy functions are allocated throughout the whole used program and data memory space as depicted in Fig 2.


Fig 2.　Program counter test function placement example
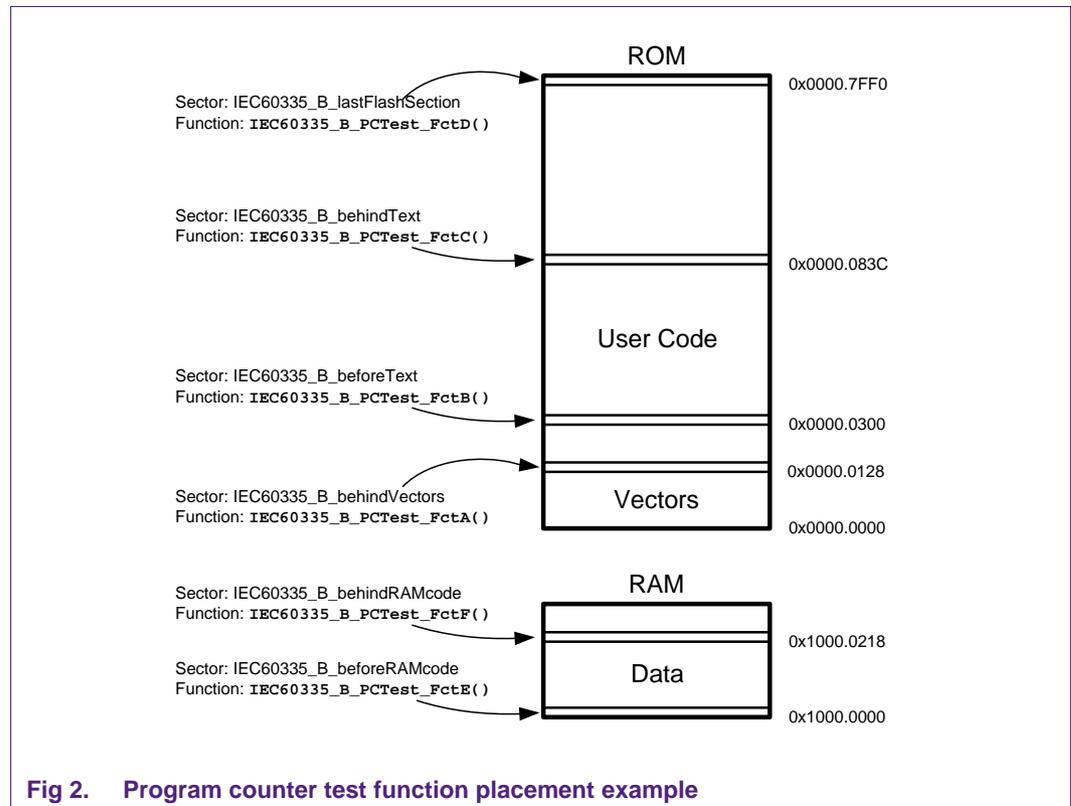
The allocation of the PC dummy test functions are placed accordingly by use of sections defined in the linker file.

The PC test routines call the dummy functions and check the returned value. Each dummy function returns a unique value. Thereby it is possible to check if the PC has jumped to the correct address.

Note that an enabled memory protection unit may trigger an exception when dummy functions are executable code areas that are protected.

In principle, the test results always show as okay, because a defective program counter results in program crashes in any way.

There are two different implementations available for this test: BIST and POST. The POST will check each dummy function at once. This is implemented by a loop. The BIST will only test one dummy function per call. All functions will be called after each other like a ring buffer.
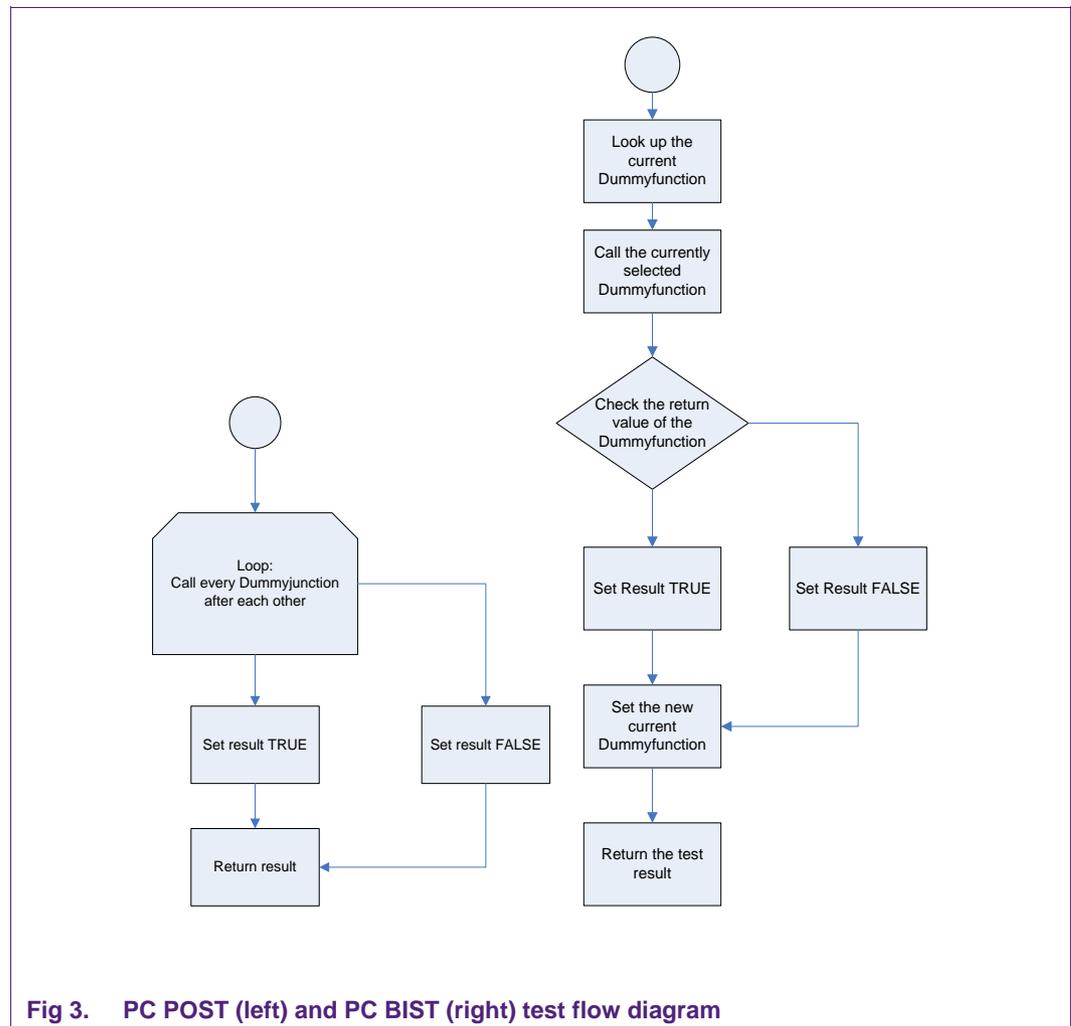
Fig 3. **PC POST (left) and PC BIST (right) test flow diagram**

4.3.2 Test usage

This chapter describes the usage of the PC POST and BIST.

4.3.2.1 IEC60335_B_ProgramCounterTest.h

File name	Function prototyping
IEC60335_B_ProgramCounterTest.h	**type_testResult** IEC60335_B_PCTest_POST(**void**);
	type_testResult IEC60335_B_PCTest_BIST(**void**);

This header file contains all function prototypes used during the PC tests.

4.3.2.2 IEC60335_B_ProgramCounterTest.c

File name	Definitions
IEC60335_B_ProgramCounterTest.c	RET_FCT_A = 1
	RET_FCT_B = 2
	RET_FCT_C = 3
	RET_FCT_D = 5
	RET_FCT_E = 7
	RET_FCT_F = 11
	Global variable
	UINT32 IEC60335_B_PCTest_lastFctTested
	Functions
	type_testResult IEC60335_B_PCTest_POST(**void**)
	type_testResult IEC60335_B_PCTest_BIST(**void**)

The PC test should be done pre-operation (POST) and during program execution (BIST). The PC POST and BIST functions are to be called in the corresponding state of the controller.

Function:

`type_testResult` IEC60335_B_PCTest_POST(**void**)

Purpose:

This function should be executed prior running the main application. It will call the test functions throughout the program and data memory and check the return value against the expected value.

Return value:

IEC60335_testPassed

IEC60335_testFailed

Function:

`type_testResult` IEC60335_B_PCTest_BIST(**void**)

Purpose:

The PC BIST function `IEC60335_B_PCTest_BIST(void)` executes at every call one PC test, saves the current executed test and returns a PASS/FAIL. It will automatically run through all six tests.

Return value:

IEC60335_testPassed

IEC60335_testFailed

4.4 Interrupt Handling and Execution Test (2)

4.4.1 Test description

The test for interrupt handling and execution is application dependent. In this test, the library delivers some templates to enable the users testing the functionality in an abstract way.

The interrupts will be checked with the aid of counter variables. The different interrupts, which are observed by counter mechanisms, should have individual up-counting values instead of simply adding one.

To check the interrupts, the counter value has to be checked cyclically in a known equidistant time and compared to boundaries estimated by the user. A timer interrupt service handler should solve this.
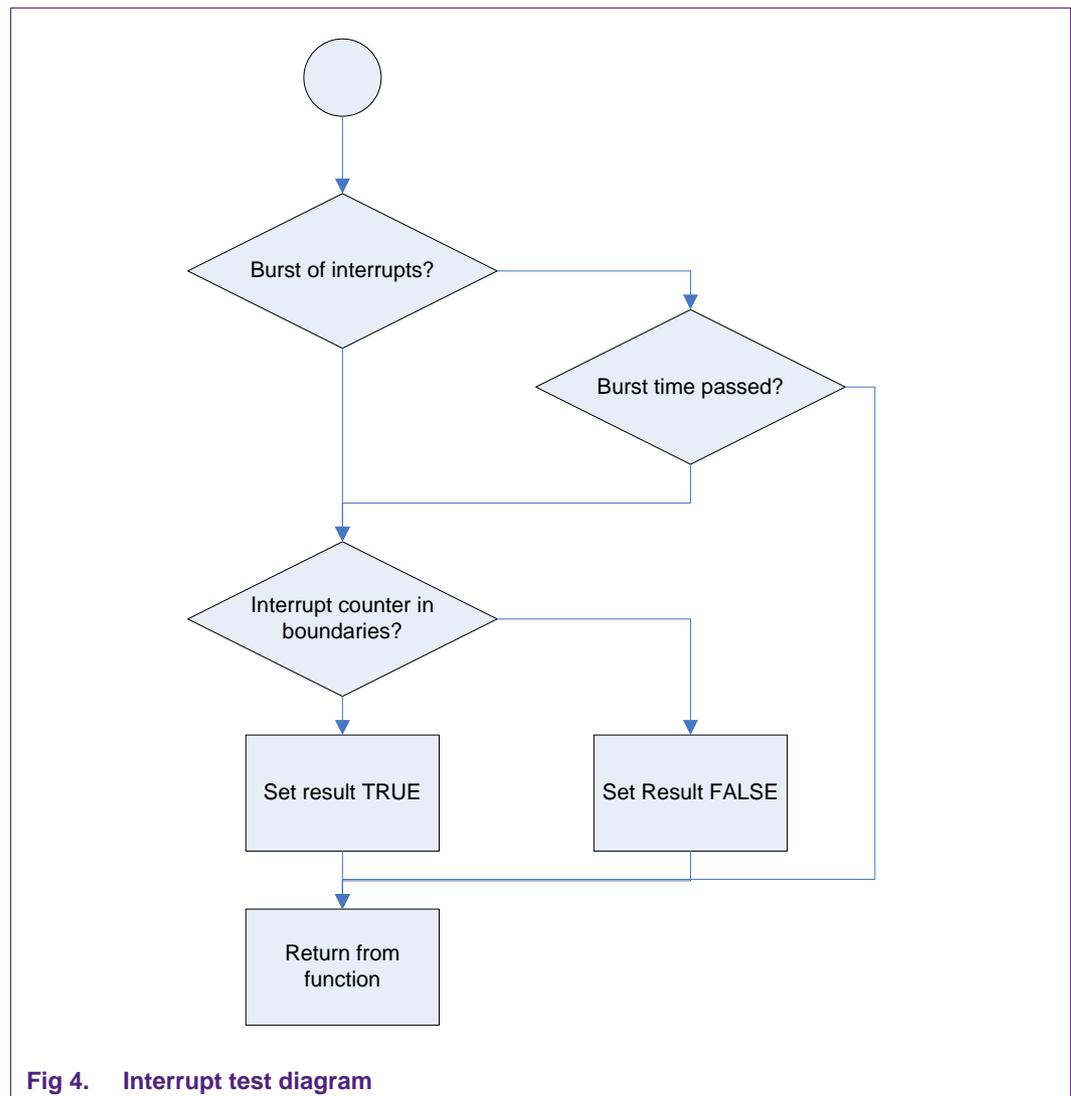
Fig 4.   Interrupt test diagram

The interrupt check routine first checks the interrupt configuration for the type of interrupt. The test usage details are described in chapter 4.4.2

If the interrupt that needs to be checked is a burst of interrupts, the routine will check if the time to wait for all interrupts has elapsed. If the time has passed, it will check the interrupt count to be within the boundaries. If not, the check function will return directly, without setting any Result.

If the interrupt to check is not a burst of interrupts, the routine will check the interrupt counter to be within the boundary directly.

4.4.2 Test usage

4.4.2.1 IEC60335_B_Interrupts.h

File name	Type definition
`IEC60335_B_Interrupts.h`	`type_InterruptTest`[1]
	Function prototyping

	void `IEC60335_InitInterruptTest`
	`(`
	type_InterruptTest `*pIRQ,`
	UINT32 `lowerBound,`
	UINT32 `upperBound,`
	UINT32 `individualValue`
	`);`

	void `IEC60335_InterruptOcurred`
	`(`
	type_InterruptTest `*pIRQ`
	`);`

	type_testResult `IEC60335_InterruptCheck`
	`(`
	type_InterruptTest `*pIRQ`
	`);`

[1]   See the detailed type description in Table 4.

The IEC60335_B_Interrupts header file contains the function prototypes of the interrupt testing. A type defined structure contains all variables needed for interrupt testing.

Table 4. `Type_InterruptTest` **type description**

Member name	Description
`UINT32 count`	The counter variable
`UINT32 lower`	The estimated minimum count value of the interrupt concurrencies
`UINT32 upper`	The estimated maximum count value of the interrupt concurrencies
`UINT32 individualValue`	The individual up-counting value
`BOOL CountOverflow`	Counter overflow bit
`BOOL cyclic`	
`UINT32 minTime`	The time count that has to be waited, before the check is done

4.4.2.2 IEC60335_B_Interrupts.c

File name	Function
`IEC60335_B_Interrupts.c`	`void IEC60335_InitInterruptTest` `(` `type_InterruptTest *pIRQ,` `UINT32 lowerBound,` `UINT32 upperBound,` `UINT32 individualValue` `)`
	`void IEC60335_InterruptOcurred` `(` `type_InterruptTest *pIRQ` `)`
	`type_testResult IEC60335_InterruptCheck` `(` `type_InterruptTest *pIRQ` `)`

This file contains the functions needed for the Interrupt testing.

Function:

void IEC60335_InitInterruptTest
(
type_InterruptTest *pIRQ,
UINT32 lowerBound,
UINT32 upperBound,
UINT32 individualValue

)

Purpose:

The IEC60335_InitInterruptTest function initialises the interrupt test structure. This function must be called prior to any interrupt initialisations.

Input variables:

type_InterruptTest *pIRQ

This structure pointer is used to set the default values to the interrupt test structure members during the interrupt test initialisation.

UINT32 lowerBound

The estimated minimum count value of the interrupt concurrencies.

UINT32 upperBound

The estimated maximum count value of the interrupt concurrencies.

UINT32 individualValue

The internal individual up-counting value.

Return value:

None

Function:

void IEC60335_InterruptOcurred
(
type_InterruptTest *pIRQ

)

Purpose:

This function must be called from any interrupt service handler which has to be tested.

Input variables:

type_InterruptTest *pIRQ

Pointer to the interrupt test structure.

Return value:

None

Function:

type_testResult IEC60335_InterruptCheck
(
type_InterruptTest *pIRQ

)

Purpose:

This function should be called periodically and it will do a 'quantity of interrupts' check for the interrupt under test.

Input variables:

type_InterruptTest *pIRQ

Pointer to the interrupt test structure.

Return value:

IEC60335_testPassed

*IEC60335_testFailed*

4.5 Clock System Test (3)

This chapter is only applicable for the NXP Cortex-M0 family members with an RTC.

4.5.1 Test description

This test is intended to check the CPU clock source and frequency. This requires a second independent clock source. For a part of the NXP ARM Cortex-M0 family, the only possibility to get interrupts triggered, sourced by an independent clock, is to use the RTC peripheral.

Three test functions are implemented; the first one is cyclically called from the main loop of the user application.

As depicted in the Fig 5, the main loop function checks both the timer and RTC interrupt occurrence functions. If one or both of them are missing within a rough time frame, which has to be estimated empirically, the function will return failed as result. This function also checks the result of the timer check, which is performed by the RTC function.


Fig 5.   Clock system test flow
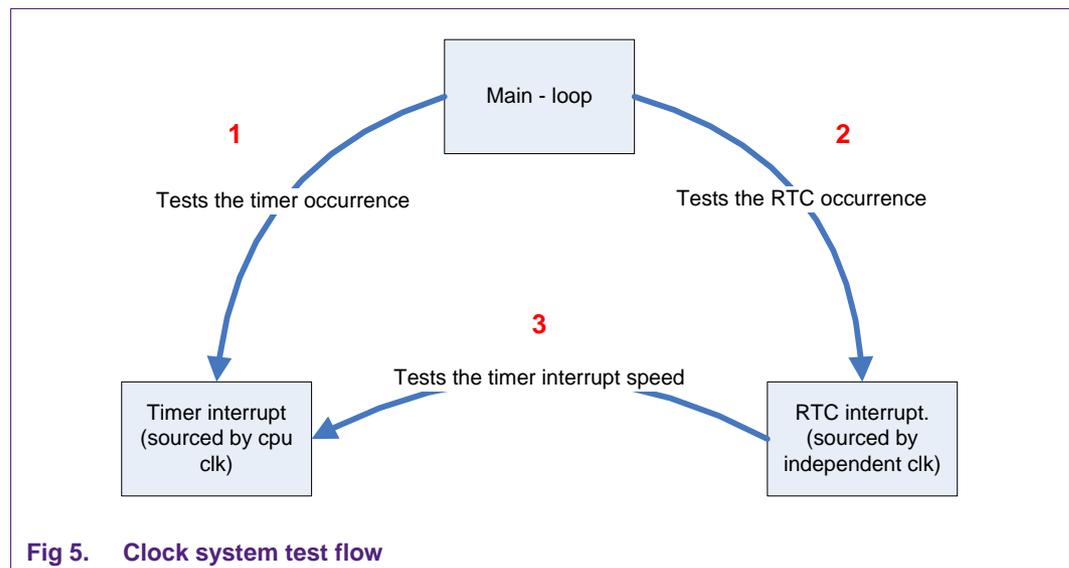
The second function is intended to be called from a timer interrupt service handler. This Timer needs to have the same clock source as the CPU.

The last function is intended to be called from the RTC interrupt service handler. This function is intended to check the frequency of the timer interrupts.

The timer simply counts how often the timer interrupt has occurred. This value is then checked by the RTC function. Additionally, it sets the occurrence semaphore, which is used for occurrence recognition inside of the main function. See Fig 6.

Fig 6. **Setting the occurrence semaphore**
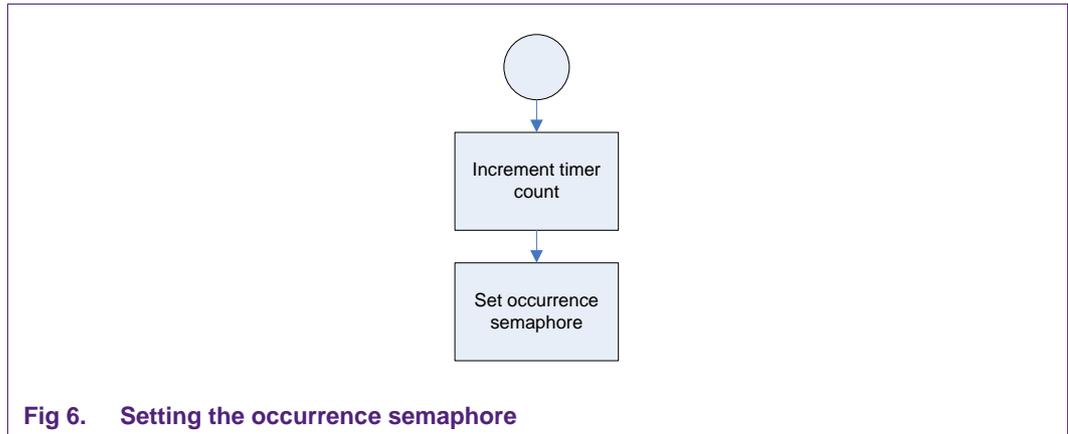
The RTC function also sets an occurrence semaphore, to be tested from the main function. Then it checks the timer counter variable to be within the estimated boundaries. The result of this check is stored into a result semaphore.

Fig 7. **Setting semaphore and boundary check**

Fig 8.   Main-loop test flow
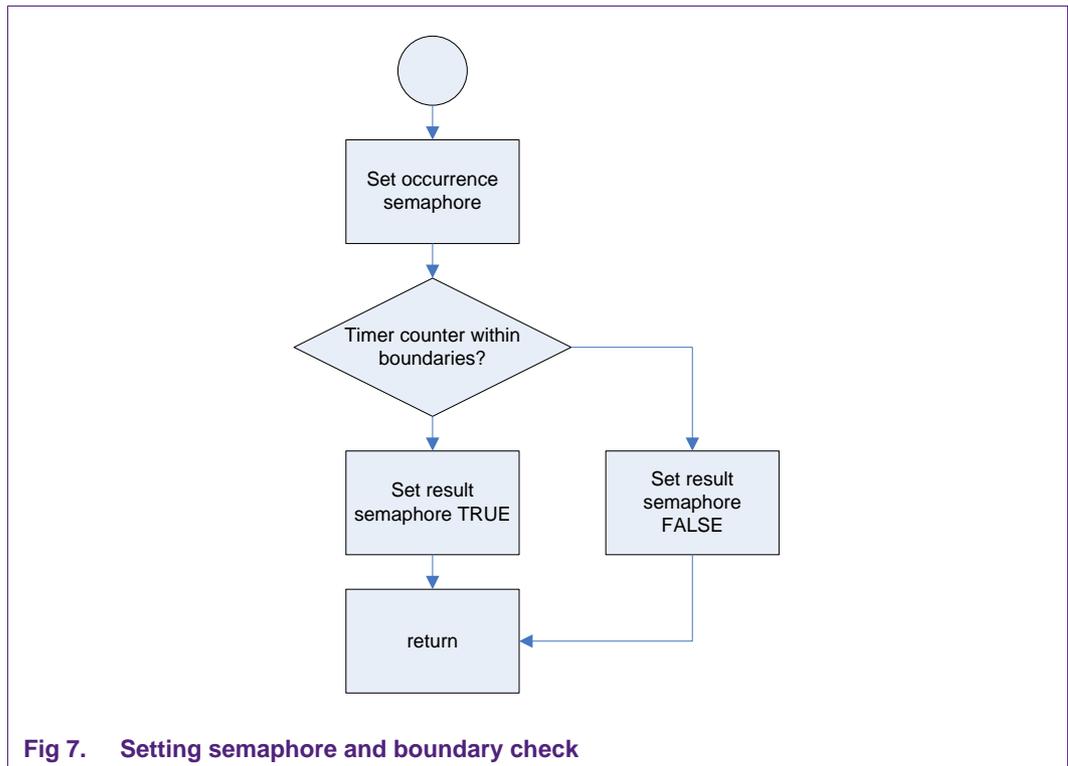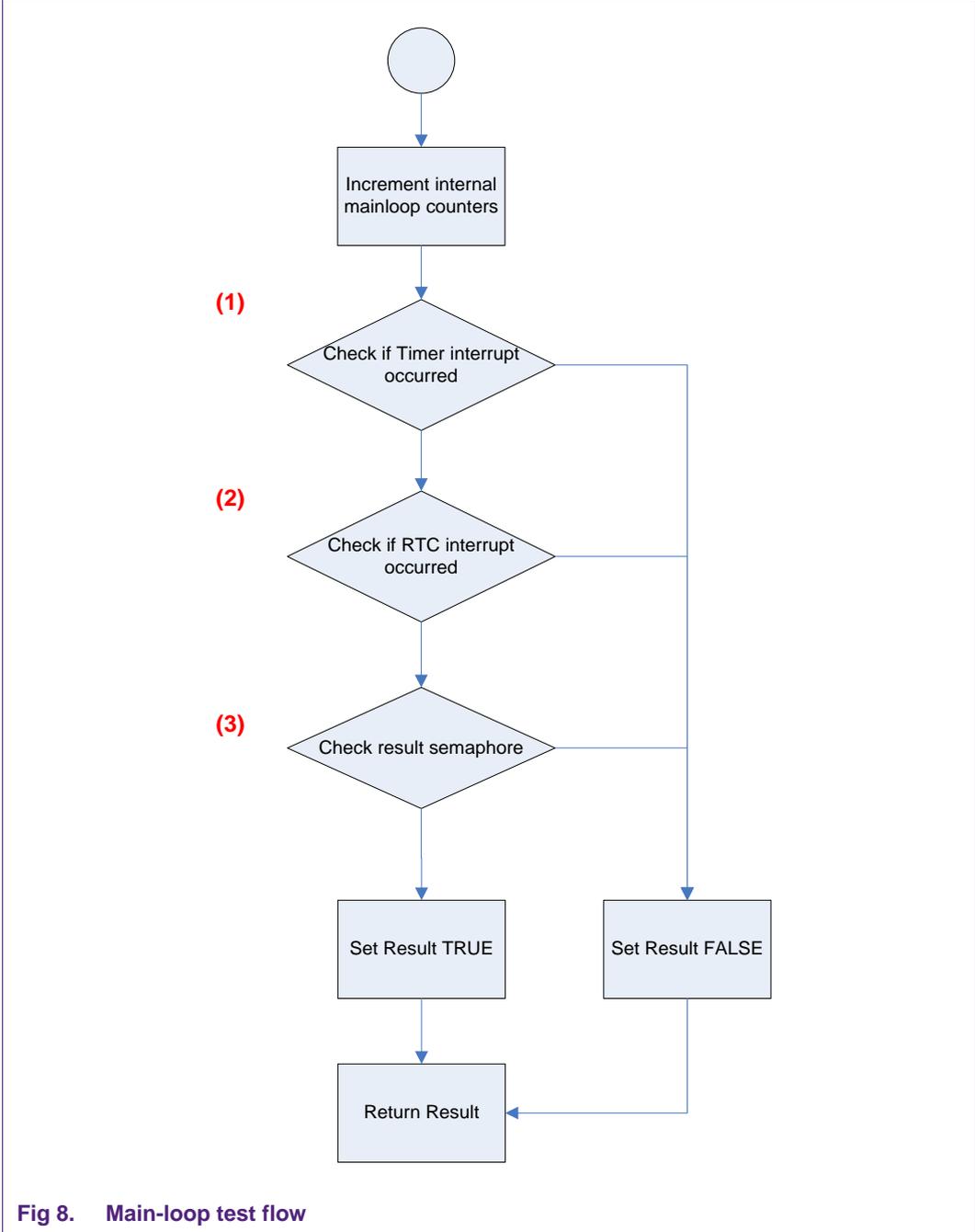
4.5.2 Test usage

4.5.2.1 IEC60335_B_ClockTest.h

File name	Function prototyping

`IEC60335_B_ClockTest.h`	**void** `IEC60335_initClockTest`
	 `(`
	 UINT32 `timerOccThreshold,`
	 UINT32 `rtcOccThreshold,`
	 UINT32 `timerLowerBound,`
	 UINT32 `timerUpperBound`
	 `)`

	type_testResult `IEC60335_Clocktest_MainLoopHandler(`**void**`)`
	void `IEC60335_Clocktest_TimerIntHandler(`**void**`)`
	void `IEC60335_Clocktest_RTCHandler(`**void**`)`

The IEC60335_B_ClockTest.h file contains all prototypes needed for the ClockTest.

4.5.2.2 IEC60335_B_ClockTest.c

File name	Type definition
`IEC60335_B_ClockTest.c`	`type_ClockTest`[1]
	Functions
	void `IEC60335_resetClockTest(`**void**`)`

	void `IEC60335_initClockTest`
	 `(`
	 UINT32 `timerOccThreshold,`
	 UINT32 `rtcOccThreshold,`
	 UINT32 `timerLowerBound,`
	 UINT32 `timerUpperBound`
	 `)`

	type_testResult `IEC60335_Clocktest_MainLoopHandler(`**void**`)`
	void `IEC60335_Clocktest_TimerIntHandler(`**void**`)`
	type_testResult `IEC60335_Clocktest_MainLoopHandler(`**void**`)`
	void `IEC60335_Clocktest_RTCHandler(`**void**`)`

[1] Structure members described in Table 5

Table 5. `type_ClockTest` **structure**

Member name	Description
UINT32 timerTestThreshold	Used in the mainloop function, defines the number of calls to start occurrence test
UINT32 rtcTestThreshold	Used in the mainloop function, defines the number of calls to start occurrence test
UINT32 rtcOccCounter	Counter variable for the mainloop, if value reached the defined threshold, the occurrence test is started
UINT32 timerOccCounter	Counter variable for the mainloop, if value reached the defined threshold, the occurrence test is started
BOOL timerOccured	This bool will be set in the timer function, and is reset during occurrence test
BOOL rtcOccured	This bool will be set in the rtc function, and is reset during occurrence test
UINT32 timerCounter	The counter Variable, to test the timer to be within its boundaries
UINT32 timerBoundLower	The estimated minimum count of cycle occurrences (Threshold for timer test).
UINT32 timerBoundUpper	The estimated maximum count of cycle occurrences (Threshold for timer test).
BOOL timerOutOfBounds	Within this bool, the rtc timer test signals the error state to the main function
BOOL timerCounterOverflow	Reflects, if the TimerCounter overflows due to an error

Function:

void IEC60335_resetClockTest**(void)**

Purpose:

The IEC60335_resetClockTest function clears and resets all used Clock Test variables

Return value:

None

AN11208

Application note

All information provided in this document is subject to legal disclaimers.

Rev. 1.1 — 22 January 2014

© NXP B.V. 2014. All rights reserved.

28 of 76

Function:

void IEC60335_initClockTest

(

UINT32 timerOccThreshold,

UINT32 rtcOccThreshold,

UINT32 timerLowerBound,

UINT32 timerUpperBound

)

Purpose:

This function initiates the various variables used during the Clock Test.

Input variables:

UINT32 timerOccThreshold

The timerOccThreshold variable initiates the threshold value that defines the number of calls that started the timer occurrence test.

UINT32 rtcOccThreshold

The rtcOccThreshold variable initiates the threshold value that defines the number of calls that started the RTC occurrence test.

UINT32 timerLowerBound

This variable sets the lower bound value of the number of timer or RTC test occurrences.

UINT32 timerUpperBound

This variable sets the upper bound value of the number of timer or RTC test occurrences.

Return value:

None

AN11208

Application note **Rev. 1.1 — 22 January 2014** **29 of 76**

Function:

`type_testResult` IEC60335_Clocktest_MainLoopHandler(**void**)

Purpose:

This function represents the part of the IEC60335 Class B clock test that must be executed within the main loop.

This function tests the following criteria:

- The clock test timer interrupts were triggered
- The clock test RTC interrupt was triggered
- In any of the two interrupts an error was detected.

Return value:

IEC60335_testPassed

IEC60335_testFailed

Important function notifications:

- This function must be called once inside the main loop.
- For this function, it is necessary to estimate the count of how often this function could be called. This is important to find valid threshold values, which are used to test timer and RTC interrupt occurrence.

AN11208

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2014. All rights reserved.

Application note **Rev. 1.1 — 22 January 2014** **30 of 76**

Function:

void IEC60335_Clocktest_TimerIntHandler(**void**)

Purpose:

This function is intended to use as timer interrupt service handler or to be called once inside the timer interrupt service handler.

Return value:

None

Function:

void IEC60335_Clocktest_RTCHandler(**void**)

Purpose:

This function should be called inside the custom RTC interrupt service handler. It can't be used as service handler by itself, because of the return value that has to be evaluated after the call.

This function tests the timer-time-frame, in this case the CPU frequency.

Also, this function checks if the main loop function was called.

Return value:

None

4.6 Invariable memory Test (4.1)

4.6.1 Test description:

The invariable memory must be checked for single bit faults. During POST testing the complete Flash memory where the user application is located gets tested.

During BIST testing it is advisable to test the Flash memory in smaller segments to prevent the CPU from being blocked.

4.6.1.1 Multiple Input Signature Register

The NXP Cortex-M0 based devices within the LPC11x family have an integrated flash module that incorporates a 128-bit signature generator, called the Multiple Input Signature Register (MISR).

This MISR can be used for generating a signature of the used safety critical memory region.

Since this module is integrated in the flash module, it generates a signature faster than when implemented in software, decreasing the time required for the test.

A signature can be generated for any part of the Flash contents. The address range to be used for the signature generation is defined by writing the start address to the `FMSSTART` register and the stop address to the `FMSSTOP` register.

The flash address should first be aligned with a flash word (128 bits) in the array; this is done by right - shifting the start and stop address by 4.

```
/* align flash address to refer the flash word in the array */
startAddr = (startAddr >> 4) & 0x0001ffff;
length    = ((startAddr + length) >> 4)  & 0x0001ffff;

/* write start address of the flash contents to the register*/
LPC_FMC->FMSSTART = startAddr;

/* write stop address of the flash contents to the register, start generating
the signature*/
LPC_FMC->FMSSTOP = length | MISR_START;
```

The signature generation is started by writing '1' to the `MISR_START` bit (17) in the `FMSSTOP` register.

Since the MISR is implemented in hardware, it is much faster than doing the same MISR check in software. The time that the signature generation takes is proportional to the address range for which the signature is generated.

4.6.1.2 Signature generation time

A safe estimation for the duration of the signature generation is:

$$T_{MISR} = \text{int}\left(\frac{60ns}{T_{cclk}} + 3\right) \times (FMSSTOP - FMSSTART + 1)$$

(1)

with Tcclk the core clock. See the device user manual for more information on the clock system.

After completion of the hardware MISR the 128 bits signature can be read from the `FMSW0`…`FMSW3` registers.

4.6.1.3 Signature verification

The signatures generated by the hardware MISR must be verified and equal to the reference signatures. The algorithm for deriving the reference signatures is illustrated in the pseudo code below.

```
Sign_word0 = 0
Sign_word1 = 0
Sign_word2 = 0
Sign_word3 = 0

FOR address = FMSTART TO FMSTOP
{
nextSign_word0 = flashWord_word0 XOR (Sign_word0>>1) XOR (Sign_word1<<31)
nextSign_word1 = flashWord_word1 XOR (Sign_word1>>1) XOR (Sign_word2<<31)
nextSign_word2 = flashWord_word2 XOR (Sign_word2>>1) XOR (Sign_word3<<31)

nextSign_word3 = flashWord_word3 XOR (Sign_word3>>1)
XOR (Sign_word0 AND 1<<29) << 2
XOR (Sign_word0 AND 1<<27) << 4
XOR (Sign_word0 AND 1<<2) << 29
XOR (Sign_word0 AND 1<<0) << 31

Sign_word0 = nextSign0
Sign_word1 = nextSign1
Sign_word2 = nextSign2
Sign_word3 = nextSign3
}
```

Important notification:

The hardware MISR signature generator is *blocking* for the Flash, this means no flash read or write access is possible during signature generation. The MISR Code should run from SRAM. It is therefore advisable to make sure while using the hardware MISR the flash will not be accessed.

4.6.1.4 CRC generator

The NXP Cortex-M0 based devices within the LPC12x family have a Cyclic Redundancy Check (CRC) module integrated. The CRC generator with programmable polynomial settings supports several CRC standards commonly used.

The following features are supported:

- Three common polynomials: CRC-CCITT, CRC-16, and CRC-32.
  - CRC-CCITT: $x_{16} + x_{12} + x_5 + 1$
  - CRC-16: $x_{16} + x_{15} + x_2 + 1$

- CRC-32: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
- Bit order reverse and 1's complement programmable setting for input data and CRC sum
- Programmable seed number setting.
- Accept any size of data width per write: 8, 16 or 32-bit.

The CRC module can be used for generating a signature of the used safety critical memory region.

Since this module is integrated in the device, it generates a signature faster than when implemented in software, decreasing the time required for the test. A signature can be generated for any part of the Flash contents.

The device can be programmed for supporting a specific standard by using the following setups:

CRC-CCITT set-up

Polynomial = $x^{16} + x^{12} + x^5 + 1$
Seed Value = 0xFFFF
Bit order reverse for data input: NO
1's complement for data input: NO
Bit order reverse for CRC sum: NO
1's complement for CRC sum: NO
CRC_MODE = 0x0000 0000
CRC_SEED = 0x0000 FFFF

CRC-16 set-up

Polynomial = $x^{16} + x^{15} + x^2 + 1$
Seed Value = 0x0000
Bit order reverse for data input: YES
1's complement for data input: NO
Bit order reverse for CRC sum: YES
1's complement for CRC sum: NO
CRC_MODE = 0x0000 0015
CRC_SEED = 0x0000 0000

CRC-32 set-up

Polynomial = $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
Seed Value = 0xFFFF FFFF
Bit order reverse for data input: YES
1's complement for data input: NO
Bit order reverse for CRC sum: YES
1's complement for CRC sum: YES
CRC_MODE = 0x0000 0036
CRC_SEED = 0xFFFF FFFF

After configuring the CRC engine by programming the parameters above in the `MODE` and `SEED` registers, the data over which the signature needs to be calculated can be fed sequentially to its `WR_DATA` input register. The resulting CRC signature can be read out from the read only `SUM` register

4.6.1.5 Critical content

If there is a stored critical constant periodically used in critical calculations, then it is necessary to check this variable before every usage.

AN11208

Application note

All information provided in this document is subject to legal disclaimers.

Rev. 1.1 — 22 January 2014

© NXP B.V. 2014. All rights reserved.

34 of 76

Refer to chapter 4.8 Secure Data storage.

4.6.1.6 Usage notes

There are the following possible definitions for the flash signature options:

LPC111x: MISR_FLASH_CRC
LPC12x: CCITT_FLASH_CRC, CRC16_FLASH_CRC, CRC32_FLASH_CRC

All of them are referenced within the LPC1xxx_TargetConfig.c file.

The values of the signatures depend on the compiler and linker version number, and may need to be updated from the default values provided within the example application. The computed values will also be different whenever any modifications are made inside the application code.

To determine the signature, either:

- Run the test application once, to find out the actual signature, then substitute the actual value into the appropriate #define directive.

  For convenience, place a breakpoint at the routine check loop labels (_misr_hw_sigcheck, _crc16_verify, _crc32_verify) within the file IEC60335_B_FlashTestxxx.s, then copy the computed values back in the signature definition;

- Calculate the signature automatically, and patch it in the desired location, by using the build tools (if supported) or;

- Use a custom script to patch the image before it gets burned in flash.

AN11208

Application note **Rev. 1.1 — 22 January 2014** 35 of 76

4.6.2 Test usage

This chapter explains how the invariable testing is implemented and can be used.

4.6.2.1 IEC60335_B_FlashTest.h

File name	Function prototyping
IEC60335_B_FlashTest.h	**void** StartHardSignatureGen (**UINT32** startAddr, **UINT32** length, **FlashSign_t** *ResultSign);
	void StartSoftSignatureGen (**UINT32** startAddr, **UINT32** length, **FlashSign_t** *ResultSign);
	type_testResult IEC60335_FLASHtest_BIST (**UINT32** startAddr, **UINT32** length, **FlashSign_t** *TestSign, **UINT8** selectHS);
	type_testResult IEC60335_testSignatures (**FlashSign_t** *sign1, FlashSign_t *sign2);
Type definition	
FlashSign_t Crc32Sig_t; Crc16Sig_t; CcittSig_t;	
Definitions	
FLASH_HARD_SIGN = 1 FLASH_SOFT_SIGN = 2 MISR_START = (1<<17) EOM = (0x01<<2)	

Functions:

All functions are described in detail in

Type definitions:

A type `FlashSign_t` is defined; this type contains four `UINT32` variables named word0…word3. These four words represent the 128 bits used for the hardware and software 128-bit signature generation.

The types `Crc32Sig_t`, `Crc16Sig_t`, `CcittSig_t` define the type of signature used with the LPC12x devices for the various algorithms supported by the CRC engine. `Crc32Sig` is a `UINT32` type, whereas `Crc16Sig_t` and `CcittSig_t` are `UINT16` variables.

Definitions:

The various flash sizes (32, 64, 128, 256 or 512 kB) available on the NXP Cortex-M0 family are defined within the file IEC60335_B_Config.h (described in 4.8.3), as a convenience to the user for determining the ranges of flash which can be tested within the application.

The actual flash size for the chosen device can be specified in the definition of symbol FLASH_SIZE within the target specific configuration header file included by IEC60335_B_Config.h file.

There are two defines which differentiate between the hardware or software generation, used by the `IEC60335_FLASHtest_BIST` function.

`FLASH_HARD_SIGN` indicates the usage of the hardware signature generator, and `FLASH_SOFT_SIGN` the software signature generator, on the LPC11x (using the MISR).

`MISR_START` is the hardware MISR start bit in the FMC `FMSSTOP` register.

`EOM` is the END OF MISR status in the FMC `STATUS` register.

AN11208

Application note **Rev. 1.1 — 22 January 2014** **37 of 76**

4.6.2.2 IEC60335_B_FlashTest.c

File name	Function prototyping
IEC60335_B_FlashTest.c	**void** StartHardSignatureGen (**UINT32** startAddr, **UINT32** length, **FlashSign_t** *ResultSign);
	void StartSoftSignatureGen (**UINT32** startAddr, **UINT32** length, **FlashSign_t** *ResultSign);
	type_testResult IEC60335_FLASHtestMISR_BIST (**UINT32** startAddr, **UINT32** length, **FlashSign_t** *TestSign, **UINT8** selectHS);
	type_testResult IEC60335_testMISRSignatures (**FlashSign_t** *sign1, FlashSign_t *sign2);
	Structure definitions
	FlashSign_t IEC60335_Flash_Sign_BIST

AN11208

Application note **Rev. 1.1 — 22 January 2014** **38 of 76**

Function:

void StartHardSignatureGen

(

UINT32 startAddr,

UINT32 length,

FlashSign_t *ResultSign

);

Purpose:

This function starts the execution of the hardware signature generation. It will do the signature generation from the start address (startAddr) with a length (length). After completion the signature will be copied to the location the pResultSign pointer points to.

Input variables:

UINT32 startAddr

This variable is the starting address of where the signature generation will start.

UINT32 length

The length variable is the region size to be used for the signature generation.

FlashSign_t *pResultSign

The result after generation completion will be put in the pointed location by the pResultSign pointer.

Return value:

None

Important notification:

This function is BLOCKING. It blocks all access to the flash memory. It is advisable to make sure no flash memory needs to be accessed during the execution of this function. The time required for this function is explained in the test description chapter.

Function:

void StartSoftSignatureGen

(

UINT32 startAddr,

UINT32 length,

FlashSign_t *ResultSign

);

Purpose:

This function starts the execution of the software signature generation. It will do the signature generation from the start address (startAddr) with a length (length).

The algorithm explained in the test description chapter is used for generation of the software signature.

This function can be used for the reference signature with which the hardware generated signature must be equal to.

After completion the signature will be copied to the location the pResultSign pointer points to.

Input variables:

UINT32 startAddr

This variable is the starting address of where the signature generation will start.

UINT32 length

The length variable is the region size used for the signature generation.

FlashSign_t *pResultSign

The result after generation completion will be put in the pointed location by the pResultSign pointer.

Return value:

None

Function:

type_testResult IEC60335_FLASHtestMISR_BIST

(

UINT32 startAddr,

UINT32 length,

FlashSign_t *TestSign,

UINT8 selectHS

);

Purpose:

This is the general IEC60335 Flash test function for BIST on the LPC11x. It must periodically be executed for testing the safety critical region. The start address and region length is passed as well as the reference signature to which the newly generated signature must match.

The user can select whether the hardware or software generator will be used during Flash BIST.

The comparison of the reference signature and the generated signature is integrated in this function and therefore it will return a pass or fail for this test.

Input variables:

UINT32 startAddr

This variable is the starting address of where the signature generation will start.

UINT32 length

The length variable is the region size used for the signature generation.

FlashSign_t *TestSign

Pointer to the reference signature.

UINT8 selectHS

Hardware or software signature generation selection byte, FLASH_HARD_SIGN or FLASH_SOFT_SIGN should be used.

Return value:

IEC60335_testPassed

IEC60335_testFailed

Function:

type_testResult IEC60335_testMISRSignatures

(

FlashSign_t *sign1,

FlashSign_t *sign2

);

Purpose:

This function compares two signatures generated using the MISR algorithm on the LPC11x and returns a pass or fail if equal or not.

Input variables:

FlashSign_t *sign1

Pointer to the first signature to be tested.

FlashSign_t *sign2

Pointer to the second signature to be tested.

Return value:

IEC60335_testPassed

IEC60335_testFailed

4.6.2.3 IEC60335_B_FLASHTestPOST_nnn.asm

File name	Function prototyping
IEC60335_B_FLASHTestPOST_nnn.asm[1]	**_FLASHTestPOST**

	Included Definitions
	SELECTED_CRC_TYPE
	IEC60335_BOTTOM_ROM_POST
	IEC60335_TOP_ROM_POST
	CRC_MODE_ADDR
	CRC_SEED_ADDR
	CRC_SUM_ADDR
	CRC_WR_DATA_ADDR
	FMSSTART_ADDR
	FMSSTOP_ADDR
	FMSW0_ADDR
	FMSW1_ADDR
	FMSW2_ADDR
	FMSW3_ADDR
	FMSTAT_ADDR
	FMSTATCLR_ADDR
	CRC_SIGNATURE_ADDR
	MISR_SIGNATURE_ADDR

[1] The nnn in the .asm file names must be replaced by a compiler indicator.
gnu = GNU GCC compiler

arm = ARM Realview compiler

iar = IAR EWARM compiler

This file contains the POST testing routing of the used non volatile (flash) memory image used by the application. It gives the user access to the Flash POST.

The routine is made available by means of the **_FLASHTestPOST** assembly label.

To configure the POST test, the user needs to define which type of signature test needs to be performed at startup. The options are:

- CRC16_ALGO, CRC32_ALGO, CCITT_ALGO (usable on the LPC12x targets)
- MISRHW_ALGO (usable on the LPC11x targets)

The particular algorithm choice is defined by the symbol SELECTED_CRC_TYPE and is specified in a target specific configuration header file (example: LPC1227_TargetConfig.h), which the user needs to include within the global library IEC60335_B_Config.h file (explained in more detail in section 4.8.3).

The actual range of memory being tested is determined by the symbols included from the target specific configuration header file. These symbols are called IEC60335_BOTTOM_ROM_POST and IEC60335_TOP_ROM_POST.

Note: these might be automatically determined by means of the symbols exported by the tool chain used for generating (compiling, assembling, linking) the application code, as shown in the example applications provided with the library.

AN11208

Application note **Rev. 1.1 — 22 January 2014** **43 of 76**

The memory address where the reference signature is stored is specified by the macros called CRC_SIGNATURE_ADDR (used for a CRC type signature) or MISR_SIGNATURE_ADDR (used for a MISR type signature).

These also both need to be specified within the target specific configuration header file.

Note: these symbols might be automatically determined by means of the symbols determined and exported by the tool chain used for generating (compiling, assembling, linking) the application code, as shown in the example applications provided by the library. In the examples, the signature is located on the next 128 bit boundary just after the end of the application image. This will leave the rest of the Flash memory available for application specific use (which might be verified with BIST tests at runtime).

The POST test will compute the signature on the actual Flash contents, and compare it with the reference signature located at address CRC_SIGNATURE_ADDR / MISR_SIGNATURE_ADDR.

CRC_MODE_ADDR, CRC_SEED_ADDR, CRC_SUM_ADDR, CRC_WR_DATA_ADDR are macros which include in the `IEC60335_B_FLASHTestPOST_nnn.asm` assembly module the register addresses specific to the CRC engine available in the LPC12xx family of devices.

Note: the actual values (register addresses) of these macros are predefined within the IEC60335_B_Config.h file, and apply to the current LPC12xx family of devices. The user has the option to override those values if needed. This ensures compatibility of the library for future devices in which the actual address assignment might be different than the ones available at the time the library is released.

FMSSTART_ADDR, FMSSTOP_ADDR, FMSW0_ADDR, FMSW1_ADDR, FMSW2_ADDR, FMSW3_ADDR, FMSTAT_ADDR, FMSTATCLR_ADDR are macros which include in the assembly module `IEC60335_B_FLASHTestPOST_nnn.asm` the register addresses which are specific to the MISR engine available in the LPC11xx family of devices.

Note: the actual values (register addresses) of these macros are predefined within the IEC60335_B_Config.h file, and apply to the current LPC11xx family of devices. The user has the option to override those values if needed. This ensures compatibility of the library for future devices in which the actual address assignment might be different than the ones available at the time the library is released.

Note that while performing the MISR algorithm computation, a part of the test will be BLOCKING. It blocks all access to the flash memory until the MISR engine has completed the generation of the signature.

Important file or function notifications:

- The `FLASHTestPOST` function must be executed prior to the branch to `main`. It should also execute in Privileged Thread mode.

- After test execution, and all included tests pass, the variable `type_testResult FlashPostTestStatus` will be set to IEC60335_testPassed = 1

- The variable `FlashPostTestStatus` must be defined by the user but can be located in any software module which is part of the application code, as long as its scope is made visible (so it cannot be declared as a C *static* variable). This status variable should preferably be defined in a dedicated module, which the user could

place in a specific section of the device RAM memory, according to its application requirements.

- The variable `FlashPostTestStatus` should be defined as being "not initialised", to prevent its value being changed by the application initialisation code before reaching *main*, so that the POST test result is preserved. This is tool chain specific and left to the user.

- In case of failure during test execution, the variable `type_testResult FlashPostTestStatus` will be set to IEC60335_testPassed = 0

- In case of failure, the function will behave in the following way:

  - (default) The CPU will be kept in a safe state, executing an infinite loop.

    This behavior can be overridden by the user application, by re-defining the function `_flashPostTestFailureHook` in an assembly module included within the application code.

  - (application specific) The function `flashPostTestFailureHook` is an optional assembly function, located in a module included in the user application.

This allows the system to perform different or additional recovery actions than the one described in point 1 above.

- An MISR type of signature should be located in memory at an address which is aligned to (a multiple of) 16 bytes (128 bits) because of the MISR hardware requirements.

AN11208

Application note **Rev. 1.1 — 22 January 2014** **45 of 76**

4.7 Variable Memory (4.2)

4.7.1 Test description

The variable memory (SRAM) must be tested for direct coupling and stuck-at faults. A pattern therefore must be written and checked. This pattern is chosen such that it could determine not only stuck-at faults but also direct coupling and even retention faults.

The March test algorithm is developed for efficient testing and detecting direct coupling and stuck-at faults in the variable memory, or in this case RAM, array.

The March algorithm used during the variable memory testing is depicted in Fig 10. The algorithm can be divided in 8 individual tests, called March tests 0 to 8. Each test has an *even* and an *odd* test.

Even represents even addressing and odd, odd addressing during the test; this is indicated as nnnn in Fig 9. Increasing and decreasing addressing is indicated by use of an up pointing or down pointing arrow. Read or write execution is indicated by r or w.

There are two patterns used during the variable memory test, the dbg (0x5555.5555) and the dbgN (0xAAAA.AAAA) pattern. The pattern layout depends on the invariable memory structure.
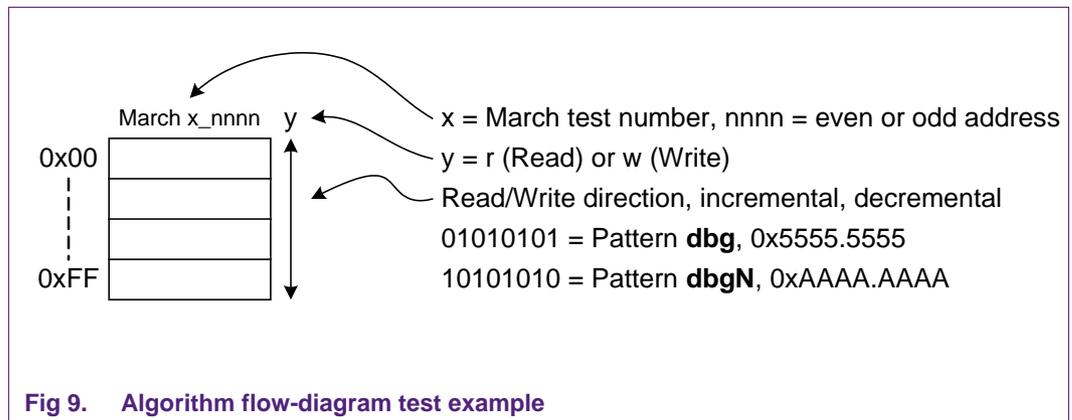
Fig 9. Algorithm flow-diagram test example

This algorithm is designed to cover both stuck-at faults and direct coupling faults in the fastest possible way.

March 0 and 1 test in increasing addressing order whether the full tested variable memory region dbg pattern is written and read correctly. This covers stuck-at 0 faults at the even bits and stuck-at 1 faults at the odd bits in the data words.

March test 2 tests in decreasing addressing order the stuck-at 0 faults at the odd bits and the stuck-at 1 faults in the even bits. It also tests the retention of the charged cells when loaded with the dbg pattern. It also takes the direct coupling in account.

March test 3 and 4 test in increasing order the inversion of March tests 0 and 1, where March test 5 does the same for test 2.

March tests 6, 7 and 8 are testing in increasing and decreasing addressing order the direct coupling more extensively.

AN11208

Application note **Rev. 1.1 — 22 January 2014** **46 of 76**
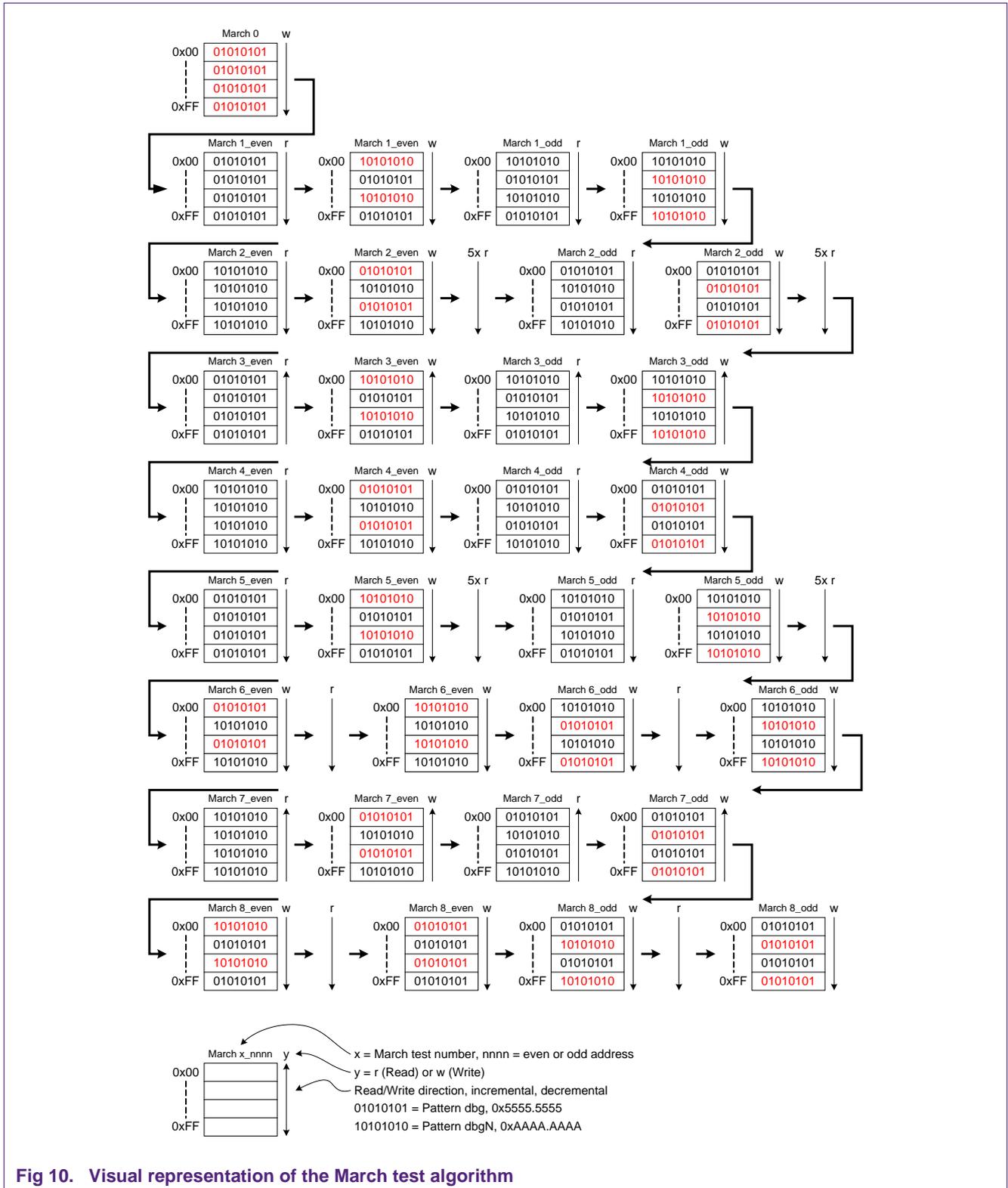
Fig 10. Visual representation of the March test algorithm

AN11208

Application note

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2014. All rights reserved.

Rev. 1.1 — 22 January 2014

47 of 76

4.7.2 Test usage

4.7.2.1 IEC60335_B_RAMTest.h

File name	Function prototyping
IEC60335_B_RAMTestBIST.h	**extern type_testResult** IEC60335_RAMtest (**UINT32** startAddrs, **UINT32** length); **extern type_testResult** IEC60335_RAMtest_BIST (**UINT32** startAddrs, **UINT32** length);

The IEC60335_B_RAMTestBIST.h file prototypes all the functions needed for executing the March BIST test on a selected range of RAM. The two functions prototyped are used for implementation of the RAM test in the user code. IEC60335_RAMtest_BIST is a predefined function simplifying the implementation. These functions will be described in detail in the following chapter.

The PATTERN (0x5555.5555) used during the March tests on the RAM is defined within the IEC60335_B_Config.h file (explained in more detail in section 4.8.3). The inversion of the defined pattern is generated while running the test.

Important notification:

- The PATTERN definition is the best pattern to be used for testing the NXP ARM Cortex-M0 family RAM and therefore **should not** be changed.

4.7.2.2 IEC60335_B_RAMTest.c

File name	Functions
IEC60335_B_RAMTest.c	**type_testResult** IEC60335_marchIncr (**UINT32** startAddrs, **UINT32** length, **UINT32** *pntr, **UINT32** pat, **UINT8** rd_cntr, **UINT8** wr_cntr)
	type_testResult IEC60335_marchDecr (**UINT32** startAddrs, **UINT32** length, **UINT32** *pntr, **UINT32** pat, **UINT8** rd_cntr, **UINT8** wr_cntr)
	type_testResult IEC60335_RAMtest (**UINT32** startAddrs, **UINT32** length)
	type_testResult IEC60335_RAMtest_BIST (**UINT32** startAddrs, **UINT32** length)

IEC60335_B_RAMTest.c contains the functions for executing the March RAM BIST test. The functions will be explained in detail in the following paragraphs.

Function:

type_testResult IEC60335_marchIncr
(
UINT32 startAddrs,
UINT32 length,
UINT32 *pntr,
UINT32 pat,
UINT8 rd_cntr,
UINT8 wr_cntr

)

Purpose:

This function takes care of the incrementing March tests. It will do the write and read operations to the memory range that is tested.

Input variables:

UINT32 startAddrs

Defines the start address of the memory range to be tested

UINT32 length

Defines the length of the memory range to be tested

UINT32 *pntr

Pointer to the current address tested.

UINT32 pat

Contains the pattern that will be written to the address tested.

UINT8 rd_cntr

With this variable the number of read cycles of the tested memory range can be defined.

UINT8 wr_cntr

With this variable the number of write cycles of the tested memory range can be defined.

Return value:

IEC60335_testPassed

IEC60335_testFailed

Function:

type_testResult IEC60335_marchDecr
(
UINT32 startAddrs,
UINT32 length,
UINT32 *pntr,
UINT32 pat,
UINT8 rd_cntr,
UINT8 wr_cntr

)

Purpose:

This function takes care of the decrementing March tests. It will do the write and read operations to the memory range that is tested. Testing will start at startAddrs + length counting down to startAddrs.

Input variables:

UINT32 startAddrs

Defines the start address of the memory range to be tested. It points to the **lowest** address.

UINT32 length

Defines the length of the memory range to be tested

UINT32 *pntr

Pointer to the current address tested.

UINT32 pat

Contains the pattern that will be written to the address tested.

UINT8 rd_cntr

With this variable the number of read cycles of the tested memory range can be defined.

UINT8 wr_cntr

With this variable the number of write cycles of the tested memory range can be defined.

Return value:

IEC60335_testPassed

IEC60335_testFailed

Function:

type_testResult IEC60335_RAMtest
(
UINT32 startAddrs,
UINT32 length

)

Purpose:

This function executes sequentially the nine March tests. The user can use this function to execute a RAM test on a defined memory range.

Input variables:

UINT32 startAddrs

Defines the start address of the memory range to be tested. It points to the **lowest** address.

UINT32 length

Defines the length of the memory range to be tested

Return value:

IEC60335_testPassed

IEC60335_testFailed

AN11208

Application note **Rev. 1.1 — 22 January 2014** **52 of 76**

Function:

type_testResult IEC60335_RAMtest_BIST
(
UINT32 startAddrs,
UINT32 length

)

Purpose:

This function executes sequentially the nine March tests in BIST-mode. The user can use this function to execute a RAM test on a defined memory range.

Input variables:

UINT32 startAddrs

Defines the start address of the memory range to be tested. It points to the **lowest** address.

UINT32 length

Defines the length of the memory range to be tested

Return value:

IEC60335_testPassed

IEC60335_testFailed

4.7.2.3 IEC60335_B_ RAMTestPOST_nnn.asm

File name	Function prototyping
IEC60335_B_RAMTestPOST_nnn.asm[1]	**_RAMTestPOST**
	Included Definitions
	IEC60335_BOTTOM_RAM_POST_MARCH
	IEC60335_TOP_RAM_POST_MARCH

This file contains the POST testing routing of the volatile (Flash) memory image used by the application. It gives the user access to the RAM POST.

The routine is made available by means of the **_RAMtestPOST** assembly label.

The POST test will perform the full March test suite described in section 4.7.2.

To configure the ram POST test, the user needs to define the memory range over which the test will be performed. This is specified in a target specific configuration header file (example: LPC1227_TargetConfig.h), which the user needs to include within the global library IEC60335_B_Config.h file (explained in more detail in section 4.8.3).

The actual range of memory being tested is determined by the symbols included from the target specific configuration header file. These symbols are called:

- IEC60335_BOTTOM_ROM_POST_MARCH: this defines the start address of the memory being tested.
- IEC60335_TOP_ROM_POST_MARCH: this defines the limit of the memory range being tested, i.e. is equal to the sum of the start address plus the size of the volatile memory.

Example: for the LPC1227, which has 8 kB of RAM memory, IEC60335_TOP_ROM_POST_MARCH would be set to 0x1000 0000 + 0x2000 = 0x1000 2000.

Note: these symbols might be automatically determined by means of the symbols exported by the tool chain used for generating (compiling, assembling, linking) the application code, as shown in the example applications provided with the library.

Important file or function notifications:

- The **RAMtestPOST** function must be executed prior to the branch to main. It should also execute in Privileged Thread mode.
- After test execution, and all included tests pass, the variable type_testResult RamPostTestStatus will be set to IEC60335_testPassed = 1
- The variable RamPostTestStatus needs to be defined by the user and can be located in any software module which is part of the application code, as long as its scope is made visible (so it cannot be declared as a C *static* variable). This status variable should preferably be defined in a dedicated module, which the user could place in a specific section of the device RAM memory, according to its application requirements.
- The variable RamPostTestStatus should be defined as being "not initialised", to prevent its value being changed by the application initialisation code before reaching

main , so that the POST test result is preserved. This is tool chain specific and left to the user.

- In case of failure during test execution, the variable `type_testResult RamPostTestStatus` will be set to IEC60335_testPassed = 0
- In case of failure, the function will behave in the following way:
  - (default) The CPU will be kept in a safe state, executing an infinite loop.

    This behavior can be overridden by the user application, by re-defining the function `_ramPostTestFailureHook` in an assembly module included within the application code.
  - (application specific) The function `_ramPostTestFailureHook` is an optional assembly function, located in a module included in the user application.

This allows the system to perform different or additional recovery actions than the one described in point 1 above.

4.8 Secure Data Storage (5.1)

4.8.1 Test description

The Library delivers mechanisms to safely use critical data.

Critical data could be variables used in important calculations or structures of configuration data. Such data must be checked before usage.

There are two ways to handle critical data:

1. Intended for native data types, such as `UINT32`, `INT16` or `float`.

2. Intended to be used for complex data types such as structures or unions.

For the native data types, there are defined structures, wherein the variable will be saved, together with its complement. To handle these structures, there are defined function-like macros for initialisation, writing, reading and checking such a variable. Function-like macros are used because they are type-independent.

Fig 11. The read macro

Fig 12. The write macro

AN11208

Application note

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2014. All rights reserved.

Rev. 1.1 — 22 January 2014

56 of 76

Fig 13. The check macro
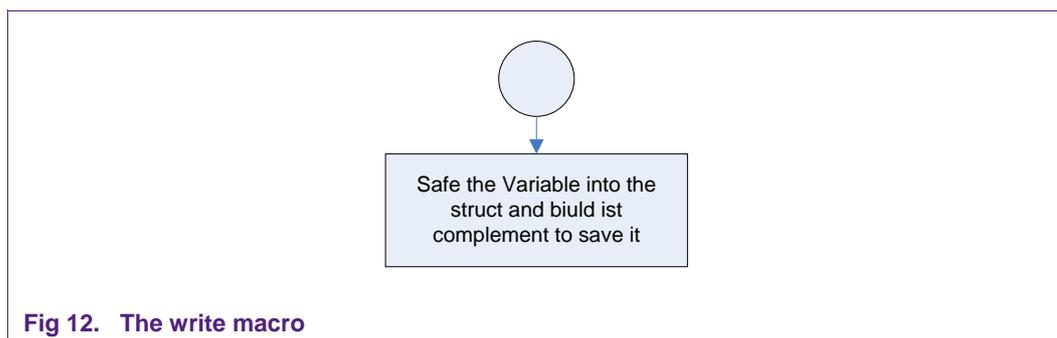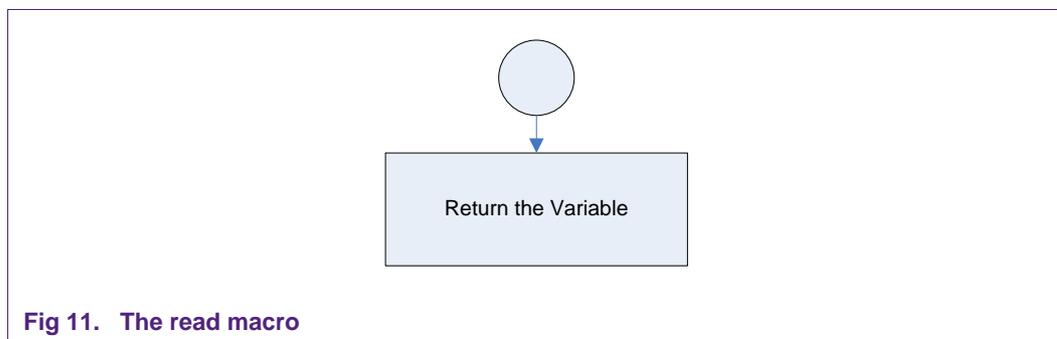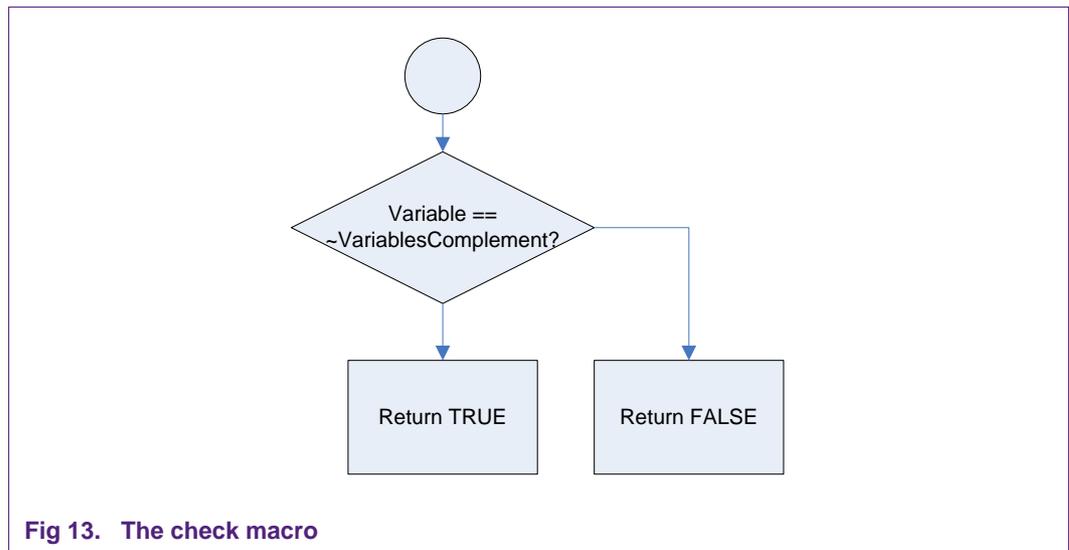
For initialization, there is a special macro to ease the usage. It is a function-like macro intended for use with global critical data that is declared outside functions.

```
#define IEC60335_CriticalDataInitialise(value)            \
     {value, ~value}
}
```

To initialise critical data inside of functions, use the write macro.

4.8.2 Usage

For elementary data types, structures and function macros are defined. To use such a single critical elementary data type, a suitable structure must be defined and initialised with default values.

System malfunction must be prevented by checking each critical variable before using it. If the content of this variable changes, the write macro will handle the recalculation of the mirror inside the structure.

There is also a possibility to instantiate complex data types.

All critical variables can be placed into a special section of the RAM; the test can be solved with a call of the RAM test function, pointing to the content containing the critical data. This allows for a couple of possibilities to check the correctness of your critical data content.

Use the macro `IEC60335_CriticalDataInitialise` to initialise a new instance of a critical variable.

If you instance a critical variable without initialising immediately with a value, you must initialise it with the function `IEC60335_CriticalDataWrite`. The macro `IEC60335_CriticalDataInitialise` will only work on initialising within the line which declares the new instance.

AN11208

Application note **Rev. 1.1 — 22 January 2014** **58 of 76**

4.8.2.1 IEC60335_B_SecureDataStorage.h

File name	Type definitions
IEC60335_B_SecureDataStorage.h	**typedef struct** tag_secured_FLOAT64 { FLOAT64 data; FLOAT64 mirror; } type_secured_FLOAT64;
	typedef struct tag_secured_FLOAT32 { FLOAT32 data; FLOAT32 mirror; } type_secured_FLOAT32;
	typedef struct tag_secured_UINT64 { **UINT64** data; **UINT64** mirror; } type_secured_UINT64;
	typedef struct tag_secured_UINT32 { **UINT32** data; **UINT32** mirror; } type_secured_UINT32;
	typedef struct tag_secured_INT32 { **INT32** data; **INT32** mirror; } type_secured_INT32;
	typedef struct tag_secured_UINT16 { **UINT16** data; **UINT16** mirror; } type_secured_UINT16;
	typedef struct tag_secured_INT16 { **INT16** data; **INT16** mirror; } type_secured_INT16;
	typedef struct tag_secured_UINT8 { **UINT8** data; **UINT8** mirror; } type_secured_UINT8;
	typedef struct tag_secured_INT8 { **INT8** data; **INT8** mirror; } type_secured_INT8;
	Macro definition
	IEC60335_CriticalDataCheck(criticalVar)
	IEC60335_CriticalDataRead(criticalVar)
	IEC60335_CriticalDataWrite(criticalVar, value)
	IEC60335_CriticalDataInitialise(value)

AN11208

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2014. All rights reserved.

Application note **Rev. 1.1 — 22 January 2014** **59 of 76**

4.8.3 IEC60335_B_Config.h

File name	Definitions
`IEC60335_B_Config.h`	**`SIZE32K`** **`SIZE64K`** **`SIZE128K`** **`SIZE256K`** **`SIZE512K`** **`CCITT_ALGO`** **`CRC16_ALGO`** **`CRC32_ALGO`** **`MISRHW_ALGO`** **`PATTERN`** **`FMSSTART_ADDR`** **`FMSSTOP_ADDR`** **`FMSW0_ADDR`** **`FMSW1_ADDR`** **`FMSW2_ADDR`** **`FMSW3_ADDR`** **`FMSTAT_ADDR`** **`FMSTATCLR_ADDR`** **`CRC_MODE_ADDR`** **`CRC_SEED_ADDR`** **`CRC_SUM_ADDR`** **`CRC_WR_DATA_ADDR`**
Included files (depending on target build rule)	
`LPC1114_TargetConfig.h` `LPC1227_TargetConfig.h`	

This header file is a global configuration file for the library, which:

- Predefines some global values which are independent of the target device:

1	SIZE32K	0x00007FFF
2	SIZE64K	0x0000FFFF
3	SIZE128K	0x0001FFFF
4	SIZE256K	0x0003FFFF
5	SIZE512K	0x0007FFFF
6	CCITT_ALGO	1
7	CRC16_ALGO	2
8	CRC32_ALGO	3
9	MISRHW_ALGO	4
10	PATTERN	0x55555555

Note: the definitions for items 1 to 10 should never be changed or overridden by the user configuration file

AN11208

Application note **Rev. 1.1 — 22 January 2014** **60 of 76**

- Predefines some macros used to define hardware addresses relative to the CRC and MISR engines on the actual LPC12xx and LPC11xx family of microcontrollers:

11	FMSSTART_ADDR	TARGET_FMSSTART EQU 0x4003C020
12	FMSSTOP_ADDR	TARGET_FMSSTOP EQU 0x4003C024
13	FMSW0_ADDR	TARGET_FMSW0 EQU 0x4003C02C
14	FMSW1_ADDR	TARGET_FMSW1 EQU 0x4003C030
15	FMSW2_ADDR	TARGET_FMSW2 EQU 0x4003C034
16	FMSW3_ADDR	TARGET_FMSW3 EQU 0x4003C038
17	FMSTAT_ADDR	TARGET_FMSTAT EQU 0x4003CFE0
18	FMSTATCLR_ADDR	TARGET_FMSTATCLR EQU 0x4003CFE8
19	CRC_MODE_ADDR	TARGET_CRC_MODE_ADDR EQU 0x50070000
20	CRC_SEED_ADDR	TARGET_CRC_SEED_ADDR EQU 0x50070004
21	CRC_SUM_ADDR	TARGET_CRC_SUM_ADDR EQU 0x50070008
22	CRC_WR_DATA_ADDR	TARGET_WD_DATA_ADDR EQU 0x50070008

The table above shows the definition for the IAR and ARM compilers, an alternative for the GNU compiler is provided in the header file as well.

Note: the definitions for items 11 to 22 should not be changed or overridden by the user configuration file, for the actual family of LPC11x microcontrollers.

It shall be verified by the user and eventually adapted, for more recent devices (in respect to the library release date), according to the hardware address values given in the specific device datasheet.

- Includes a target specific header file, which provides the mandatory target specific configuration parameters in order to use the library functions on the specific device. Within the example code shipping with the library, two template files are provided for LPC1227 and LPC1114 (LPC1227_TargetConfig.h and LPC1114_TargetConfig.h). The user may adapt the settings within those files, to change the symbol definitions for a different target or configuration.

AN11208

Application note **Rev. 1.1 — 22 January 2014** **61 of 76**

4.8.4 LPC1xxx_TargetConfig.h

This type of file is included by the library global header file IEC60335_B_Config.h.

Within this file, target specific options need to be defined in order to configure the library.

The following symbols need to be defined:

Flash POST test

- `IEC60335_BOTTOM_ROM_POST`
- `IEC60335_TOP_ROM_POST`
- `SELECTED_CRC_TYPE`
- `CRC_SIGNATURE_ADDR`
- `MISR_SIGNATURE_ADDR`

RAM POST test

- `IEC60335_TOP_RAM_POST_MARCH`
- `IEC60335_BOTTOM_RAM_POST_MARCH`

Those symbols can be either hard coded or determined from the tool chain configuration (compiler, assembler, linker) being used. Resolving the symbols in numeric values is tool chain dependant and left to the user.

The examples provided with the library demonstrate how those symbols can be defined.

Modifying the settings above from within this header file does not change the functionality of the library but determines only the parameters which the specific tests will use.

The user needs to carefully specify the test parameters according to the requirements specified in the relative sections of this document.

For a description of the symbols, please refer to the relevant test descriptions within this document.

Important file or function notifications:

CRC_SIGNATURE_ADDR and MISR_SIGNATURE_ADDR both need to be defined, independent of the target.

However CRC_SIGNATURE_ADDR will be effectively used only for the POST CRC algorithms, whereas MISR_SIGNATURE_ADDR will be effectively used only for the POST MISR algorithms.

5. Tested peripheral detailed description

5.1 CPU, the Cortex-M0

The processor or central processing unit (CPU) of the NXP Cortex-M0 microcontrollers uses the ARM Cortex-M0 version r0p0 core, which is an implementation of the ARMv6-M architecture, developed by ARM Ltd.

The Cortex-M0 processor is built on a high-performance processor core, with a 3-stage pipeline von Neumann architecture, making it ideal for demanding embedded applications. The processor is extensively optimized for low power and area, and delivers exceptional power efficiency through its efficient instruction set, providing high-end processing hardware including either:

- a single-cycle multiplier, in designs optimized for high performance
- a 32-cycle multiplier, in designs optimized for low area.

The Cortex-M0 processor implements the ARMv6-M architecture that implements the ARMv6-M Thumb instruction set, including Thumb-2 technology. This provides the exceptional performance expected of a modern 32-bit architecture, with a higher code density than other 8-bit and 16-bit microcontrollers.

The Cortex-M0 processor closely integrates a configurable *Nested Vectored Interrupt Controlle*r (NVIC), to deliver industry-leading interrupt performance. The NVIC:

- includes a *non-maskable interrupt* (NMI)
- provides:
  - a zero-jitter interrupt option
  - four interrupt priority levels.

The tight integration of the processor core and NVIC provides fast execution of *interrupt service routines* (ISRs), dramatically reducing the interrupt latency. This is achieved through the hardware stacking of registers, and the ability to abandon and restart load-multiple and store-multiple operations. Interrupt handlers do not require any assembler wrapper code, removing any code overhead from the ISRs. Tail-chaining optimization also significantly reduces the overhead when switching from one ISR to another.

To optimize low-power designs, the NVIC integrates with sleep mode. Optionally, sleep mode support can include a deep sleep function that enables the entire device to be rapidly powered down.
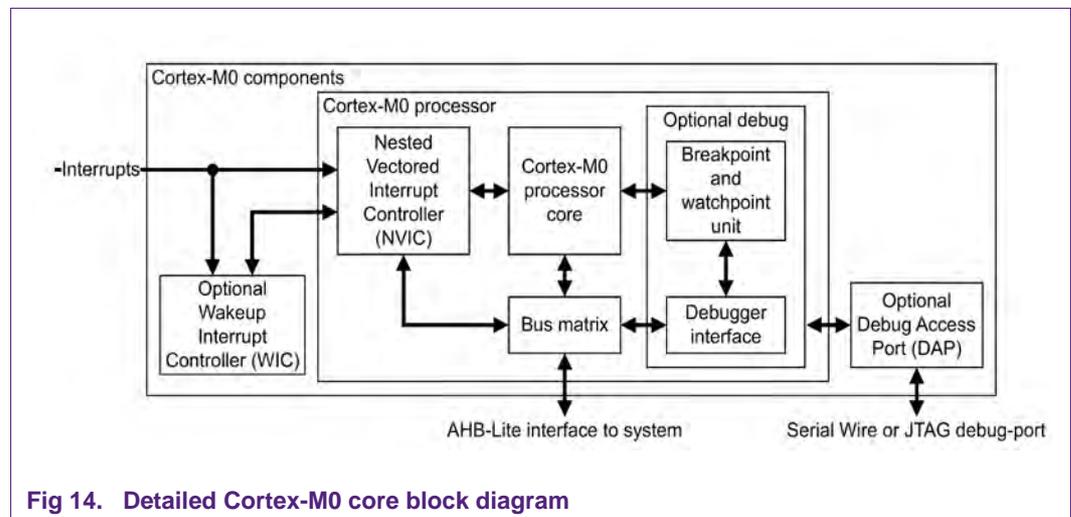
Fig 14. Detailed Cortex-M0 core block diagram

5.2 CPU registers and Program counter

Cortex-M0 r0p0 core [8] has 13 general-purpose registers [r0-r12], which can be divided by two sets of registers: the low and high registers. The low registers are accessible by all instructions that specify a general-purpose register and the high registers are only accessible by 32-bit instructions.

Besides the general-purpose registers, r13-r15 has some special functions. Register r13 is the Stack pointer, a banked alias for the SP_main and SP_process registers. The

AN11208
Application note

All information provided in this document is subject to legal disclaimers.

Rev. 1.1 — 22 January 2014

© NXP B.V. 2014. All rights reserved.

63 of 76

handler mode will always use the SP_main, but can be configured in Thread mode to use either SP_main or SP_process.

The Link register is located at r14, this register receives the address from the Program Counter (PC) when a *Branch and Link* (BL) or a *Branch and Link with Exchange* (BLX) instruction is executed. All other times r14 is a general-purpose register.

The last of the general registers is r15, the PC.

The processor core registers are:

Low registers	R0
	R1
	R2
	R3
	R4
	R5
	R6
	R7
	R8
	R9
High registers	R10
	R11
	R12

General purpose registers

Stack Pointer	SP (R13)
Link Register	LR (R14)
Program Counter	PC (R15)

PSP   MSP

PSR	Program Status Register
PRIMASK	Interrupt mask register
CONTROL	Control Register

Special registers
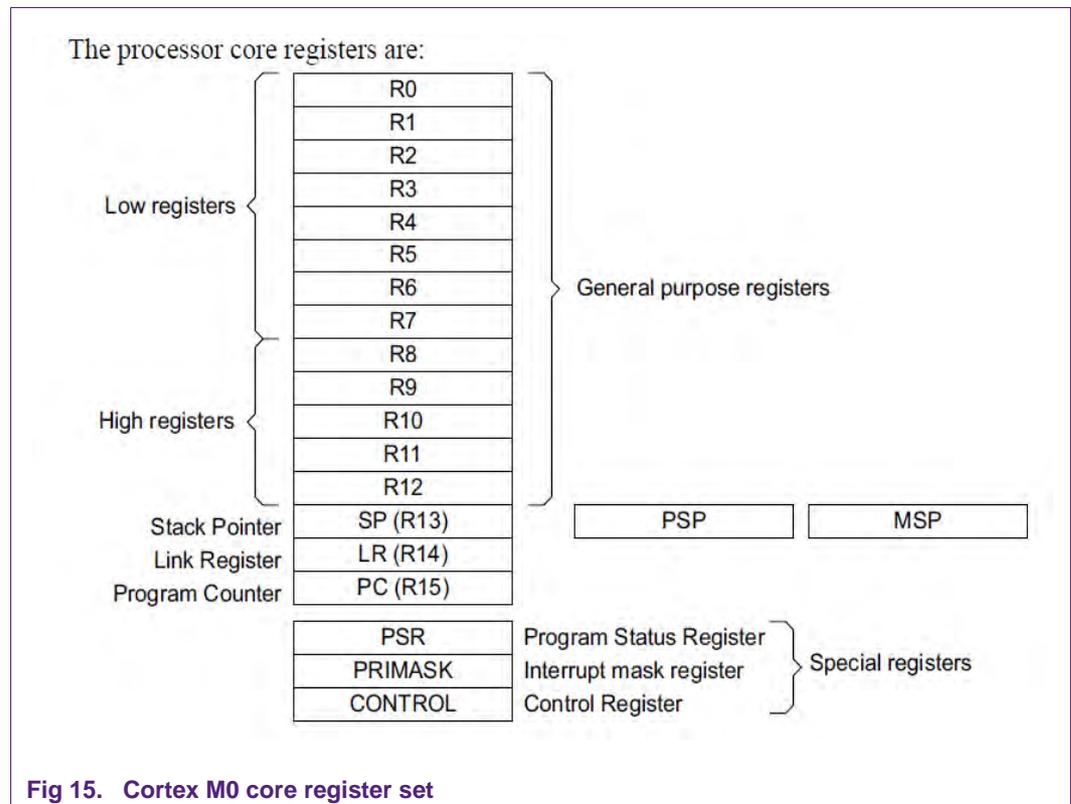
Fig 15. Cortex M0 core register set

The processor also has some status registers that can be divided in three categories at system level. These are the *Application Processor Status Register* (APSR), the *Interrupt Processor Status Register* (IPRS) and the *Execution Processor Status Register* (EPSR). For a detailed description see the *Cortex-M0 r0p0 Technical Reference Manual*[4].

AN11208

Application note **Rev. 1.1 — 22 January 2014** **64 of 76**

5.3 Interrupt handling and execution

The ARM Cortex-M0 core incorporates a Nested Interrupt Controller (NVIC) that is closely integrated with the core to achieve low latency interrupt processing. The NVIC of the NXP ARM Cortex-M0 families has the following features:

- An implementation-defined number of interrupts, in the range 1-32.
- Programmable priority levels of 0-192 in steps of 64 for each interrupt. A higher level corresponds to a lower priority, so level 0 is the highest interrupt priority.
- Level and pulse detection of interrupt signals.
- Interrupt tail-chaining.
- An external NMI.

For a detailed description of the NVIC controller see the Cortex-M0 r0p0 Technical Reference Manual, Chapter 4.2 "Nested Vectored Interrupt controller" [4].

For details on the usage of the NVIC in the NXP ARM Cortex-M0 families, see the *"Nested Vectored Interrupt Controller"* and the *"Cortex-M0 User Guide"* chapters in the product User Manual.

5.4 Clock domains

Note this chapter is only applicable for the NXP Cortex-M0 family members with an RTC.

There are three separate clock domains in the clock generation unit: the sysconfig domain, the watchdog timer domain and the Real time clock domain which is actually in the Power management Unit (PMU) domain.
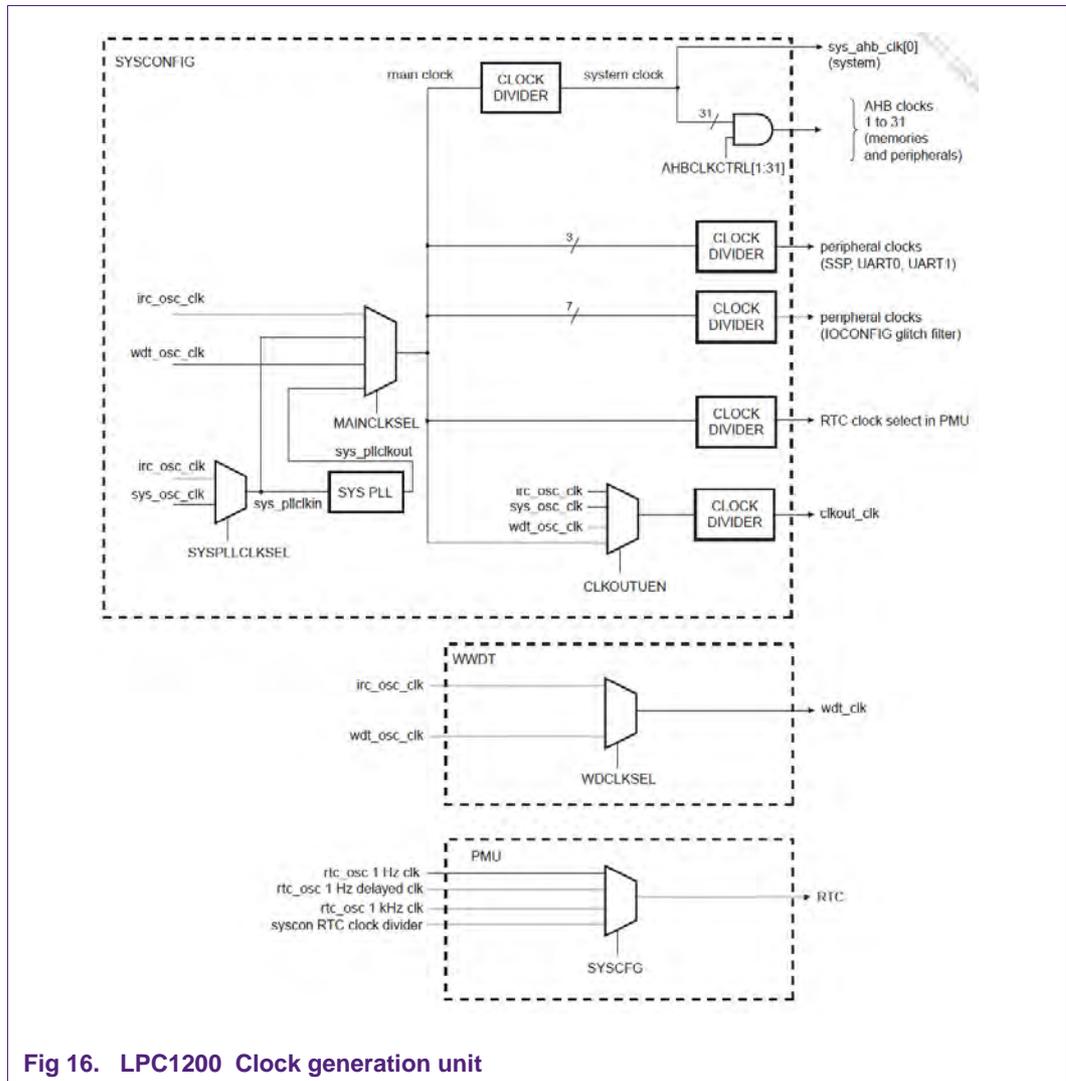
Fig 16. LPC1200 Clock generation unit

SYSCONFIG domain

The sysconfig domain creates the main clock sources needed, e.g. the core and the various peripherals.

AN11208

Application note

All information provided in this document is subject to legal disclaimers.

Rev. 1.1 — 22 January 2014

© NXP B.V. 2014. All rights reserved.

66 of 76

Watch dog domain

The watchdog domain gives the user the possibility to select either the 12 MHz internal RC (IRC) oscillator or the internal 400 kHz watchdog oscillator as clock source for the watchdog.

The RTC domain

The NXP Cortex-M0 family has an RTC sub-system that has a separate power domain, and is clocked by a dedicated 32 kHz ultra low power RTC oscillator.

Features:

- Dedicated 32 kHz ultra low power oscillator.
- Uses 1 Hz clock, delayed 1 Hz clock, 1 kHz clock, or peripheral RTC clock as inputs.
- Uses 1 Hz clock to count in one second intervals.
- 32-bit counter.
- Programmable 32-bit match/compare register.
- Software maskable interrupt when counter and match registers are identical.

AN11208

Application note **Rev. 1.1 — 22 January 2014** **67 of 76**

5.5 Memory

This chapter describes the memory in the NXP Cortex-M0 family. The memory size for both variable and invariable memory depends on the family member selected.

5.5.1 ARM Cortex-M0 Memory map

The ARM Cortex-M0 processor memory architecture is different from the traditional ARM processors.

The following new features are implemented:

- Predefined memory map
- Bit Band support
  - This feature provides atomic operations to bit data in memory and peripherals
- Big and little endian memory configuration support.

For detailed information on the ARM Cortex-M0 memory architecture and model please see the Cortex-M0 r0p0 Technical Reference Manual, Chapter 3.4 "Memory Model".

AN11208

Application note **Rev. 1.1 — 22 January 2014** **68 of 76**

```
                                         0xFFFFFFFF

              Device      511MB

                                         0xE0100000
         Private peripheral bus  1MB     0xE00FFFFF
                                         0xE0000000
                                         0xDFFFFFFF


           External device    1.0GB



                                         0xA0000000
                                         0x9FFFFFFF



            External RAM     1.0GB



                                         0x60000000
                                         0x5FFFFFFF
            Peripheral      0.5GB

                                         0x40000000
                                         0x3FFFFFFF

              SRAM         0.5GB

                                         0x20000000
                                         0x1FFFFFFF

               Code        0.5GB

                                         0x00000000
```

Fig 17. The Cortex-M0 predefined memory map

5.5.2 NXP Cortex-M0 memory map

The NXP Cortex-M0 family members offer a wide variety of memory sizes. These are all mapped according the ARM Cortex-M0 memory map. The invariable memory is placed in the low address range, starting at address 0x0000.0000, for all NXP Cortex-M0 family members. The invariable memories are placed in various regions.

5.5.3 Invariable memory (flash)

Depending on the NXP Cortex-M0 device, the flash ranges from 8 kB up to 64 kB. The invariable memory of the NXP Cortex-M0 family members are all mapped to the starting address 0x0000 0000.

5.5.3.1 Multiple Input Signature Register (MISR)

The flash module contains a built-in signature generator. This generator can produce a 128-bit signature placed in the Multiple Input Signature Register (MISR) from a user defined range of the flash memory. Typically, the flashed contents are verified against a calculated signature (e.g., during programming). Since the MISR is implemented in hardware and executed on the core clock frequency it is a faster method of creating a signature of the flash content for content verification.

As described in chapter 4.6.1.1 the algorithm used during the MISR calculation is known, therefore the signature can be calculated in advance and used for comparison.

Since the MISR is implemented in hardware, it must be tested for correct signature generation prior to usage. The algorithm used for the hardware is therefore also implemented in software.

AN11208

Application note **Rev. 1.1 — 22 January 2014** **70 of 76**

5.5.4 Variable memory

The NXP Cortex-M0 family has a variety of variable memory sizes starting from 2 kB.

The NXP Cortex-M0 devices have only one variable memory implemented, located in the code region of the ARM Cortex-M0 memory map, called the 'local SRAM'.

The local SRAM is placed in the invariable memory region; the code region. This allows a no latency fetch of the data and instructions in this SRAM region. It is even capable of pre-fetching. These two factors increase the performance of this SRAM region.
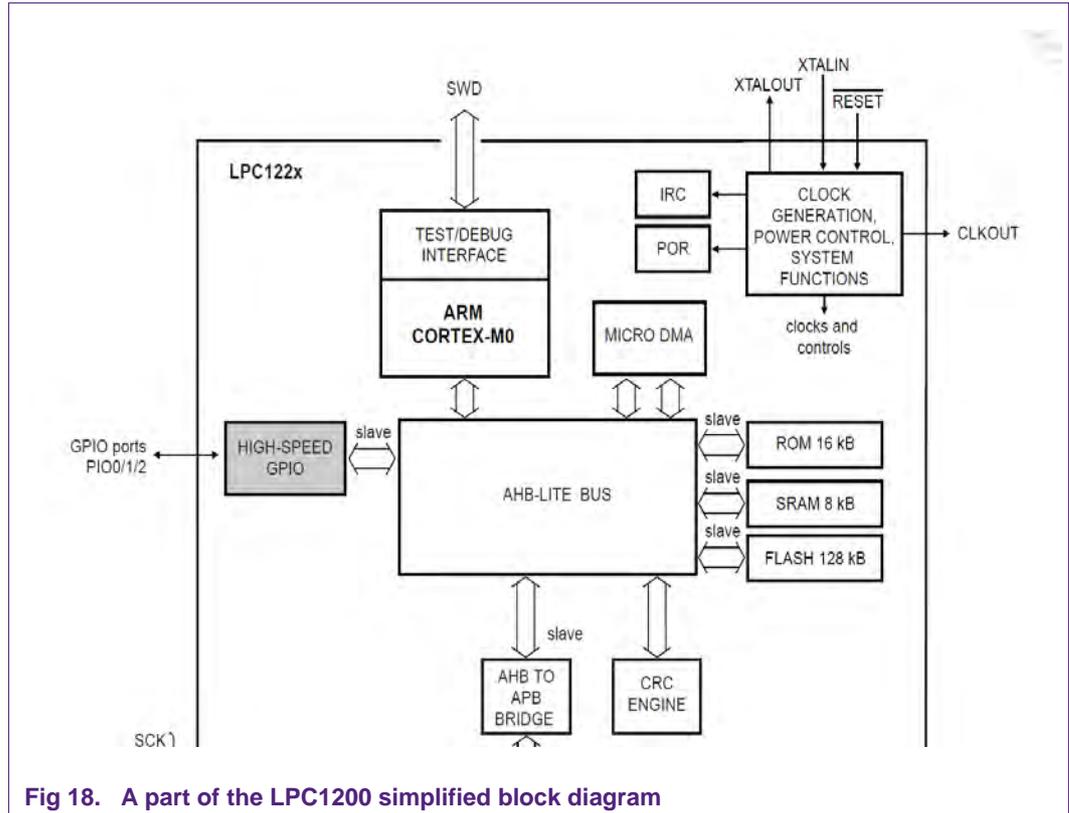
Fig 18.   A part of the LPC1200 simplified block diagram

AN11208

All information provided in this document is subject to legal disclaimers.

© NXP B.V. 2014. All rights reserved.

Application note

Rev. 1.1 — 22 January 2014

71 of 76

6. Reference list

[1] CEI/IEC 60335-1:2001+A1:2004+A2:2006, *Household and similar electrical appliances Safety*

[2] IEC 60730-1:1999+A1:2003+A2:2007(E), *Automatic electrical controls for household and similar use*

[3] The ARM website (http://www.arm.com)

[4] ARM Limited, Cortex-*M0 r0p0 Technical Reference Manual*, ARM DUI 0497A

[5] Yiu, Joseph, *The definitive guide to the ARM Cortex-M0,* 1st edition, Newnes

7. Legal information

7.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

7.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP

Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

7.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

8. List of figures

9. List of tables

10. Contents

Please be aware that important notices concerning this document and the product(s) described herein, have been included in the section 'Legal information'.