AN204364

IEC60730 CLASS B SELF-TEST LIBRARY

Author: ACEH
Associated Part Family: 8-FX Family
Associated Code Examples: "None"
Related Application Notes: "None"

This application note describes how to use and implement the library functions provided. It will first show the requirement of IEC60730 Class B, and then explain how it can be implemented. At last an example is given to show to how to integrate test functions into a Real system

Contents

1 Introduction

1.1 Definitions, Acronyms and Abbreviations

API Application Programming Interface

STL Self-Test Library

1.2 Reference Documents

[1]. IEC 60730-1 Edition3.2, 2007

[2]. F2MC-8FX MB95330H Series hardware manual (CM26-10126-1E)

[3]. F2MC-8FX MB95330H Series data sheet (DS07–12629–1E)

[4]. F2MC-8FX MB95630H Series hardware manual (MB95F636H-MN702-00009-1v0-E)

[5]. F2MC-8FX MB95630H Series data sheet (MB95F636H-DS702-00009-1v0-E)

[6]. FM3 MB9B500 Series IEC60730 CLASS B SELF-TEST LIBRARY APPLICATION NOTE (MCU-AN-510002-E-10)

2 IEC60730 CLASS B REQUIREMENT

2.1 About IEC60730

The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). All home appliances intended for European Union market must today comply with the international Electrotechnical Committee's standard IEC60730 from October 2007 onwards.

Annex H of IEC60730 specifically defines requirements for 'controls using software'. Annex H Software requirements for Class B controllers, 'control functions intended to prevent unsafe operation'. Manufacturers of home appliances must therefore consider how their products will be designed to comply with these requirements. In Annex H, the software-related Standard items are classified by Class A, B or C.

- Class A: control functions which are not intended to be relied upon for the safety of the equipment, such as humidity controls, lighting controls and timers.
- Class B: software that includes code intended to prevent hazards if a fault, other than a software fault, occurs in the appliance, such as thermal cut-outs and door locks for laundry equipment.
- Class C: software that includes code intended to prevent hazards without the use of other protective devices, such as thermal cut-outs for closed water heater systems.

2.2 IEC60730 STL Test Items

The specification defined in IEC60730 requires controls with functions classified as software class B or C shall use measures to avoid and control software-related faults/errors in safety-related data and safety-related segments of the software. This means the software must use test method to detect faults internal and external of the microcontroller.

New 8FX IEC60730 self-test library (STL) focuses on software Class B requirement for MB95330H series MCU and MB95630H series MCU, which covers most IEC60730 requirements listed in the standard. For Class B controllers, below table lists elements that must be tested, method to be adapted and definitions to be implemented as summary of Annex H table H.11.12.7

Table 2-1. Character Set Quick Reference

Component	Fault/error	Acceptable measures	Definitions	In STL
1. CPU				
1.1 Register	Stuck at	Static memory test logical	H.2.19.6	YES
1.3 Program counter	Stuck at	monitoring of the program sequence	H.2.18.10.2	YES
2. Interrupt handling & execution	No or too frequent interrupt	Time-slot monitoring	H.2.18.10.4	YES
3. Clock	Clock failure	Frequency monitoring	H.2.18.10.1	YES
4. Memory	All single	Word protection with multi-bits	H.2.19.8.1	YES
4.1 Invariable memory	bit faults	multiple checksum redundancy	H.2.19.3.2	YES
4.2 Variable memory	DC fault	Periodic static memory test	H.2.19.6	YES
4.3 Addressing	Stuck at	Watchdog timer reset window	H.2.19.18.2	YES
5. Internal data path	Stuck at			NO
6. External communication	Hamming distance 3	Word protection with multi-bit redundancy	H.2.19.4.2	NO
7. Input/output periphery		Input comparison	H.2.18.8	YES
7.1 Digital I/O	Function error	Output verification	H.2.18.12	YES
7.2 A/D	Function error	Input comparison	H.2.18.8	YES

Notes:

- The address test can be partly covered by test method of invariable and variable memory test. E.g. the error that two cells are mapped to a same address can be identified when doing invariable memory test with CRC test.
- Internal data path is only tested when using external memory.
- The external communication test is not involved in this STL. But external communication data can be tested with similar method of invariable memory test

3 IEC60730 CLASS B STL OVERVIEW

As shown in following figure, the STL block diagram includes CPU, Interrupt, Clock, Memory, and Input/output periphery modules. It shows file structure and software APIs in the STL. The STL is coded by mixed C and assembly language.

For the user application there are preconfigured functions available to initialize and run the tests for the first time. There are also two function execute tests (SelfTests_Sync) like CPU, Memory and I/O Tests and check the result of CPU independent tests (SelfTests_Async) like Program Counter, Clock and Interrupt tests.

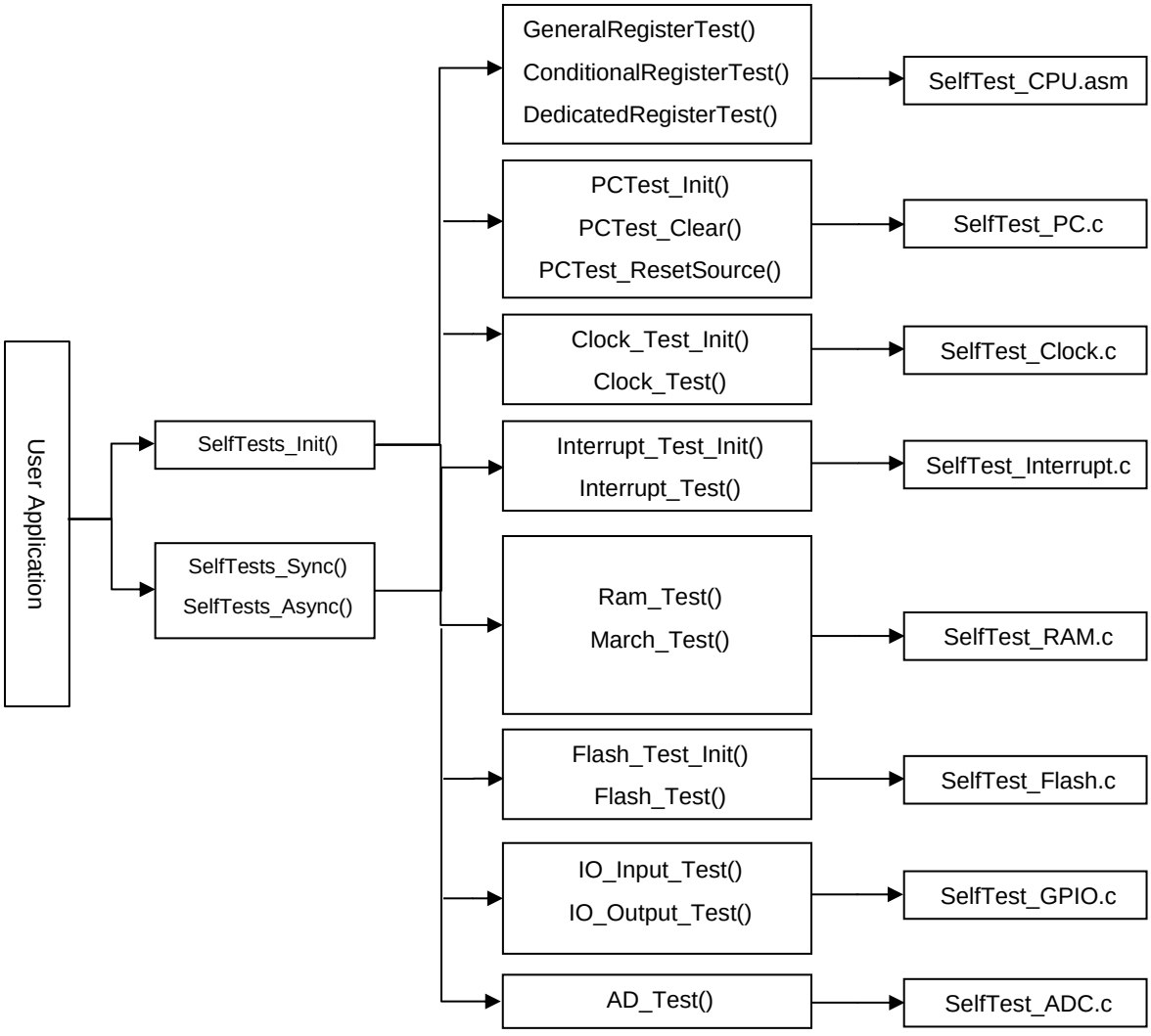

Figure 1: IEC60730 Class B STL Block Diagram

4 IEC60730 CLASS B STL API

4.1 CPU Test

4.1.1 Register Test

New-8FX MCU has 13 core registers, which can be read and written. These registers need to be tested.

Register Name	Bits test
R0~R7	[7:0]
A (Accumulator)	[15:0]
T (Temporary accumulator)	[15:0]
IX (Index register)	[15:0]
EP (Extra pointer)	[15:0]
SP (Stack pointer)	[15:0]

Table 4-1: New 8FX MCU Register List

4.1.1.1 Test Description

As shown at Table H.11.12.7 in IEC60730-1 file, registers must be checked for “stuck-at error”, a checkerboard method is used to implement register test, which is an effective method to detect stuck-at error.

Because this test is destructive, it should be called at startup file when system resets. Assembly is used to implement register test due to access to registers directly.

The test is split in four separate functions:

- GeneralRegisterTest() for registers R0 to R7
- DedicatedRegisterTest() for the special registers
- ConditionalRegisterTest() for the Condition registers
- DirectBankPointerTest() for testing the remapping functionality

All test will return and unsigned char with the result of the test. All functions are in SelfTest_CPUTest.asm.

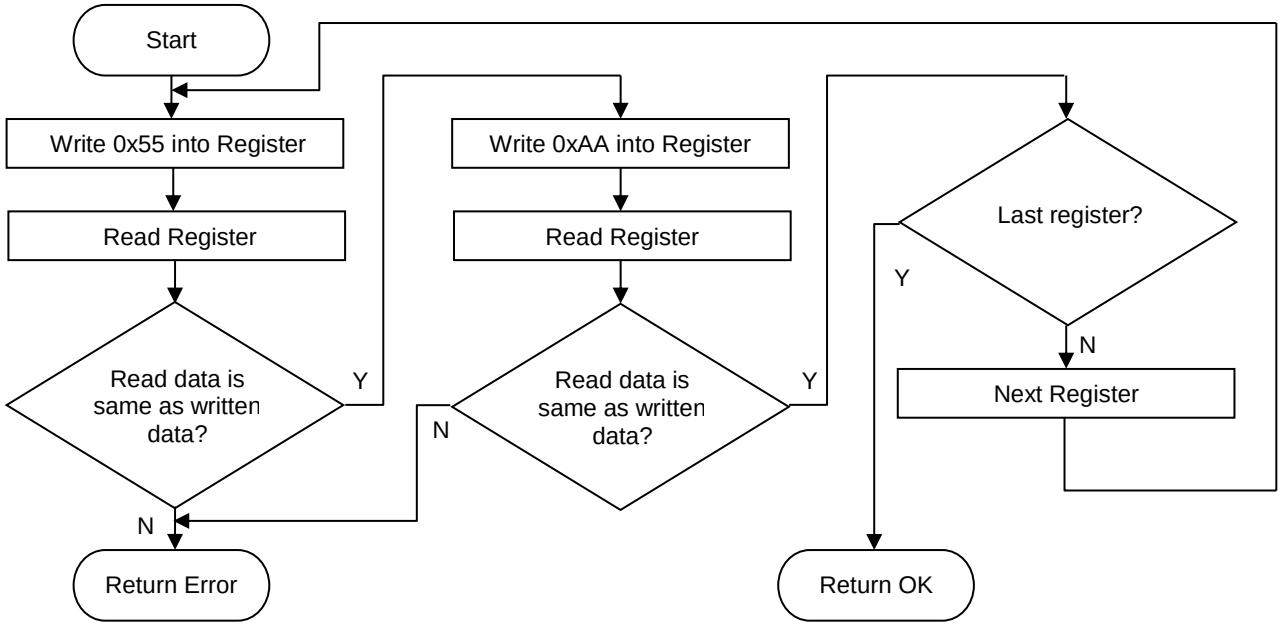

Figure 2: GeneralRegisterTest() flowchart

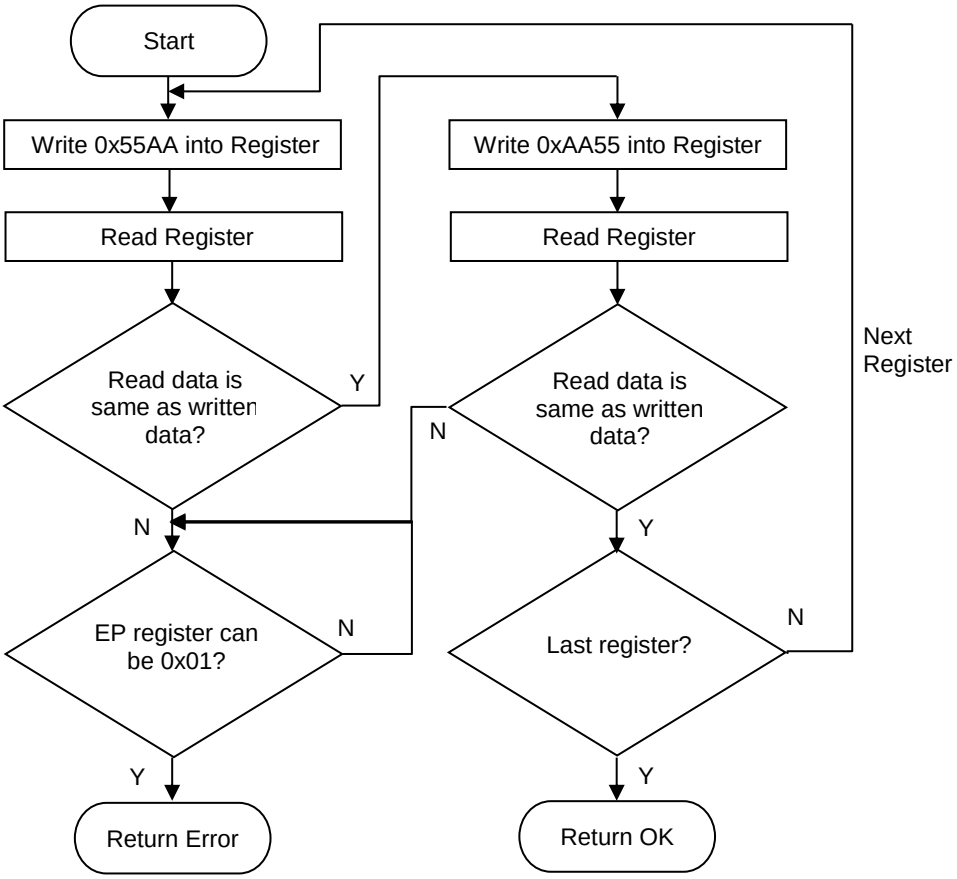

Figure 3: DedicatedRegisterTest() flowchart

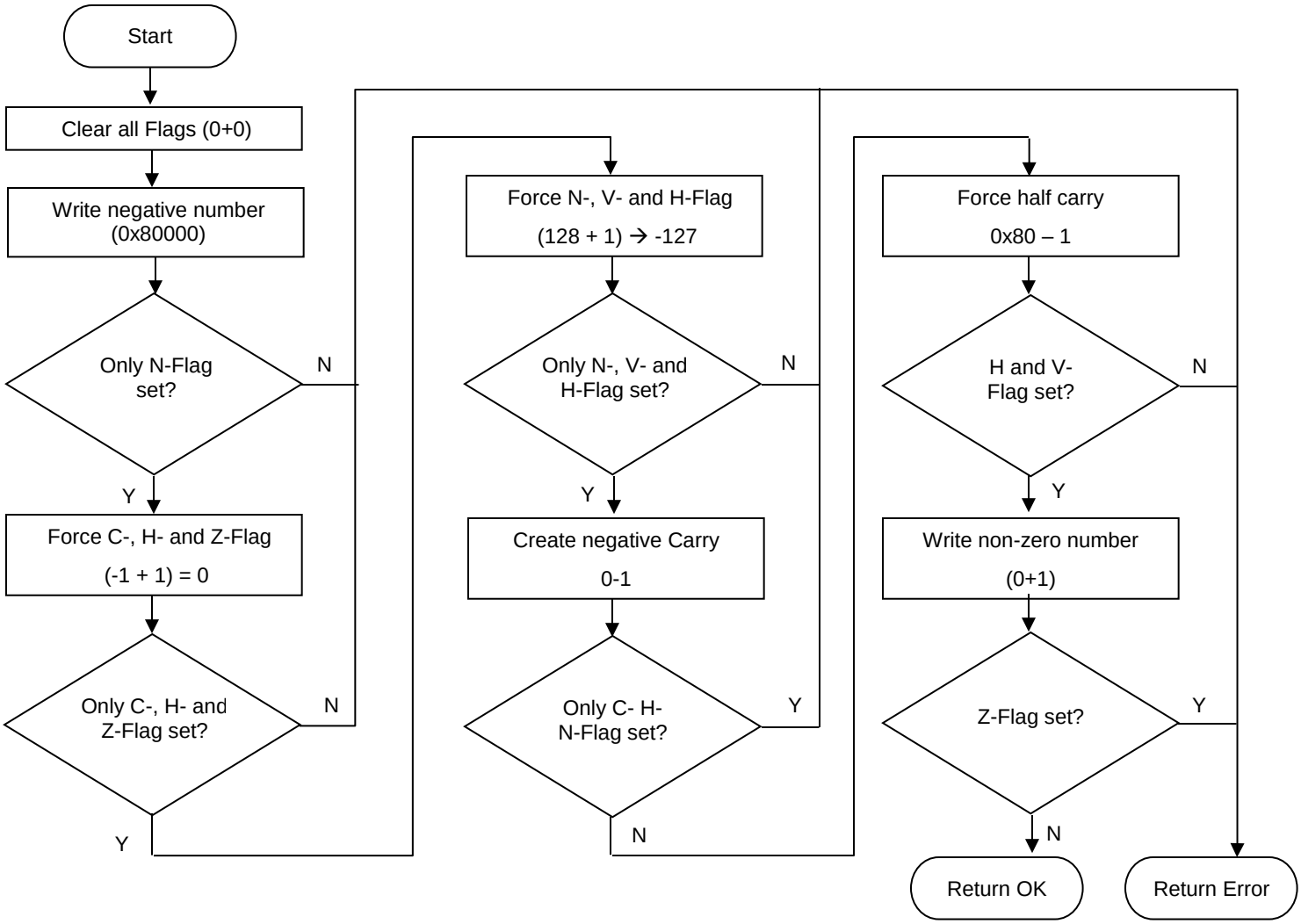

Figure 4: ConditionRegisterTest() flowchart

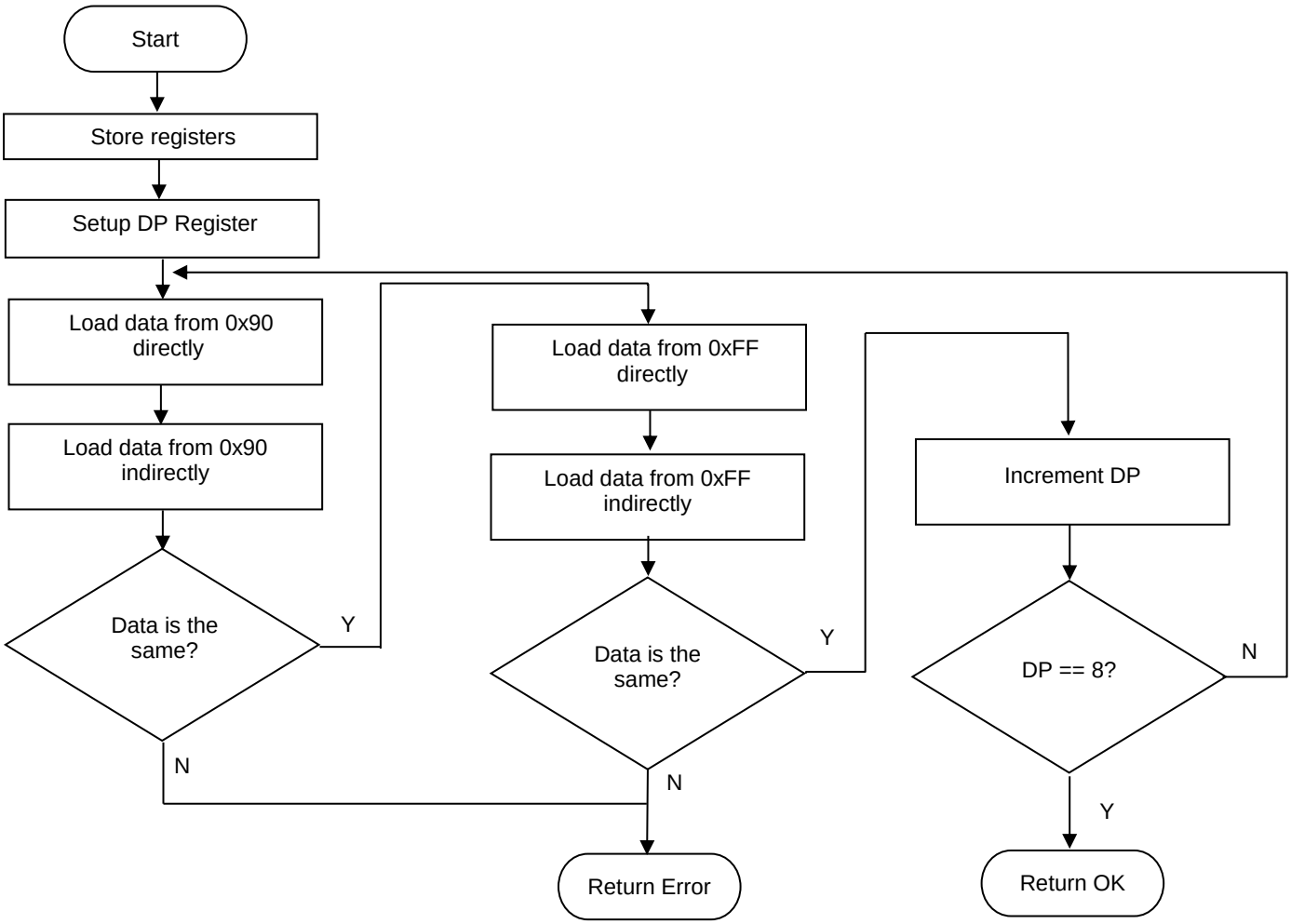

Figure 5: DirectBankPointerTest() flowchart

4.1.1.2 API Definition

GeneralRegisterTest

Name	Parameters	Return
GeneralRegisterTest	registerID (0 to 31)	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

GeneralRegisterTest() perform write and read back tests of the general purpose CPU registers R0-R7, two different test patterns are written and read back to and from registers by assembly.

The Set of General Registers is selected by the parameter registerID.

The test returns ‘0’ in case no error was detected and ‘1’ if an error was detected.

DedicatedRegisterTest

Name	Parameters	Return
DedicatedRegisterTest	None	0: TEST_NORMAL !0: TEST_FUNC_ERROR

Description:

DedicatedRegisterTest()perform write and read back tests of the special CPU registers (A, T, IX, EP, SP), two different test patterns are written and read back to and from registers by assembly.

The test returns ‘0’ in case no error was detected and non-‘0’ if an error was detected.

ConditionalRegisterTest

Name	Parameters	Return
ConditionalRegisterTest	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

ConditionalRegisterTest() perform data load, arithmetic and compare operations to check the functionality of all condition flags present in the Condition Code Register (CCR).

DirectBankPointertest

Name	Parameters	Return
DirectBankPointertest	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

DirectBankPointertest () perform data load operations from the direct bank pointer (DP) and checks if the data is valid.

4.1.2 CPU PC Test

4.1.2.1 Test Description

In IEC60730-1 file, PC must be checked for “stuck-at error”. PC test makes use of the watchdog timer clocked by an independent clock source (sub-CR). The watchdog timer will be started with PCTest_Init() and has to be constantly reset with the function PCTest_Clear(). If the watchdog timer is not reset in time, the device will be reset. In case of a PC error it is assumed that the program will not return to its destination address and reset the watchdog timer in time.

The watchdog timer clock is supervised by the clock test.

At startup the reset-cause can be checked with the function PCTest_ResetSource(). Depending on the definitions in the header file SelfTestPC.h, the reset-cause is handled as error or normal behavior.

Note: To enable the watchdog while debugging the entry “Watchdog” in „Settings → Debug Environment → Debug Environment → Chip“ has to be enabled.

4.1.2.2 API Definition

PCTest_Init

Name	Parameters	Return
PCTest_Init	None	None

Description:

PCTest_Init() starts the watchdog timer.

PCTest_Clear

Name	Parameters	Return
PCTest_Clear	None	None

Description:

PCTest_Clear() resets the watchdog timer.

PCTest_ResetSource

Name	Parameters	Return
PCTest_ResetSource	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

PCTest_ResetSource() is used to check the reset source register at startup. Returns error if an unexpected reset happened.

4.2 Interrupt Test

4.2.1 Test Description

To meet Class B requirement, interrupt must be checked for “incorrect frequency”. This test is a task which is highly system dependent and therefore the STL can only contribute the wrap up handle, which checks that a number of specific interrupts occurred at least and at most a predefined number of times.

It is assumed that Interrupt_Test (interrupt test function) is called periodically, e.g. triggered by a timer or constantly in the main loop. Each specific interrupt handler which is to be supervised, must decrement a dedicated global variable (freq), InterruptTest() compares that variable to predefined upper and lower bounds, returns an error, if the limits are exceeded.

The Interrupt Test Flow Chart is shown as following figure.

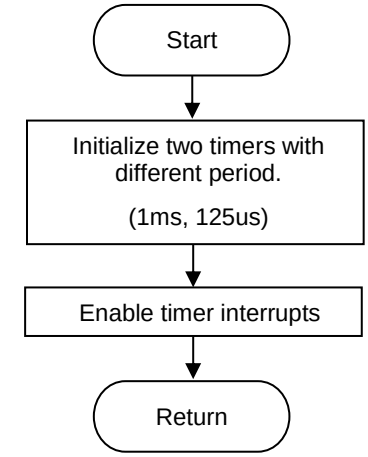

Figure 7: Interrupt Test Init flowchart

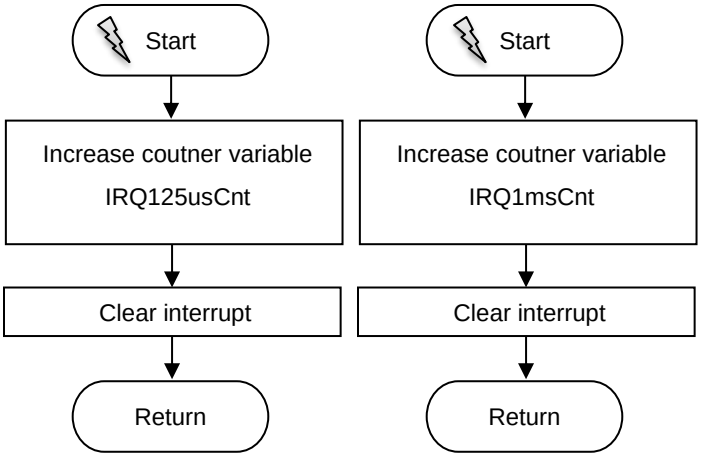

Figure 8: Interrupt handler flowchart

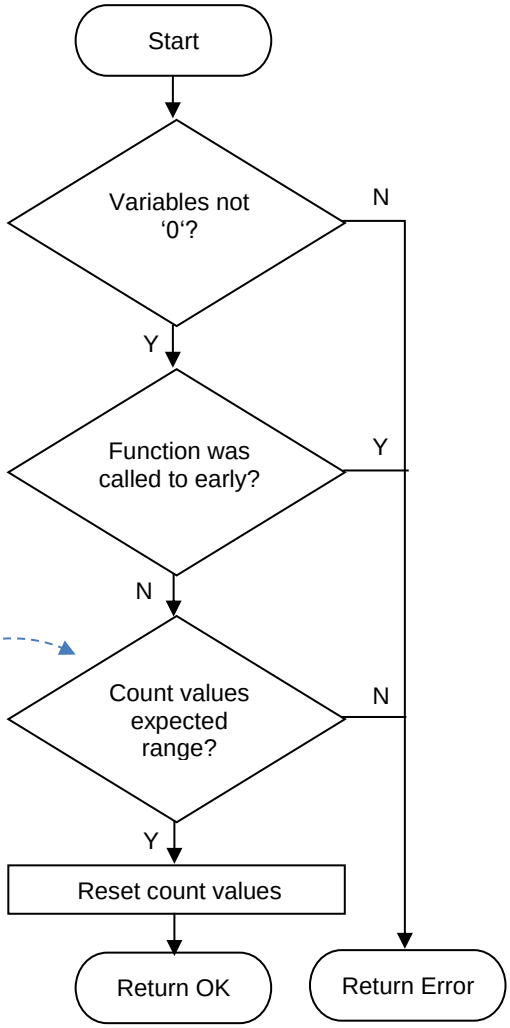

Figure 6: Interrupt Test flowchart

4.2.2 API Definition

Interrupt_Test_Init

Name	Parameters	Return
Interrupt_Test_Init	None	None

Description:

Interrupt_Test_Init() initialize the two timers with different periods and enable interrupts.

Interrupt_Test

Name	Parameters	Return
Interrupt_Test	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

Interrupt_Test() checks whether the number of interrupts that occurred is within the predefined range. It has to be called periodically by the main application.

4.3 Clock Test

4.3.1 Test Description

To meet Class B requirement, CPU clock must be checked for “wrong frequency”. This requires a second independent clock for comparison.

New-8FX MCU have integrated a Watch Prescaler counter which can be sourced either by the internal Sub-CR Oscillator (100 kHz) or an external Sub Clock Oscillator (32.768 kHz).

The Main Clock is samples by a Reload Timer and the count value is compared by a periodic triggered Watch Prescaler Counter Interrupt. If the two clocks differ more than the allowed clock tolerances an global variable is set that has to be checked by the Clock_Test() function.

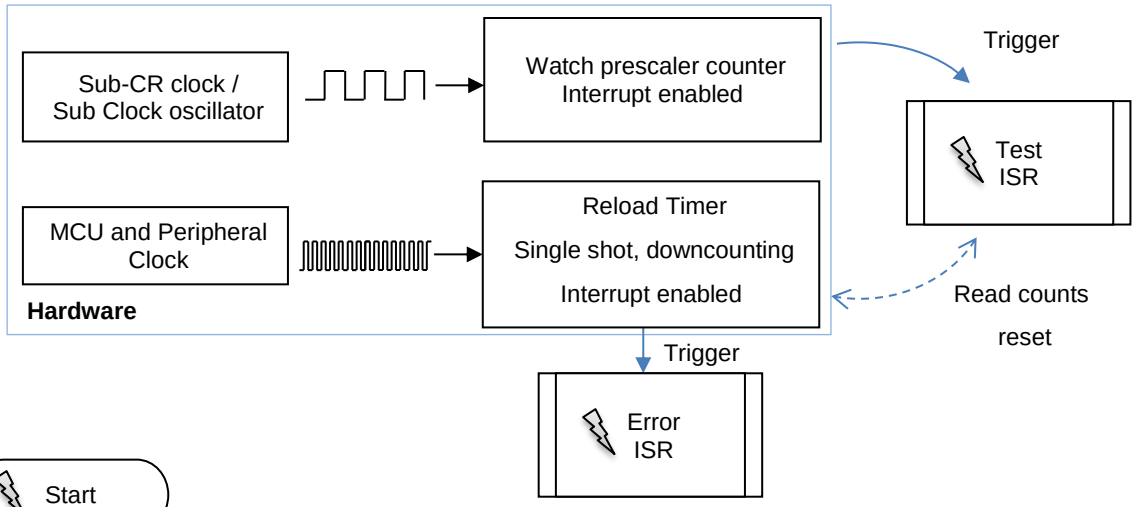

Figure 9: Clock Test block diagram

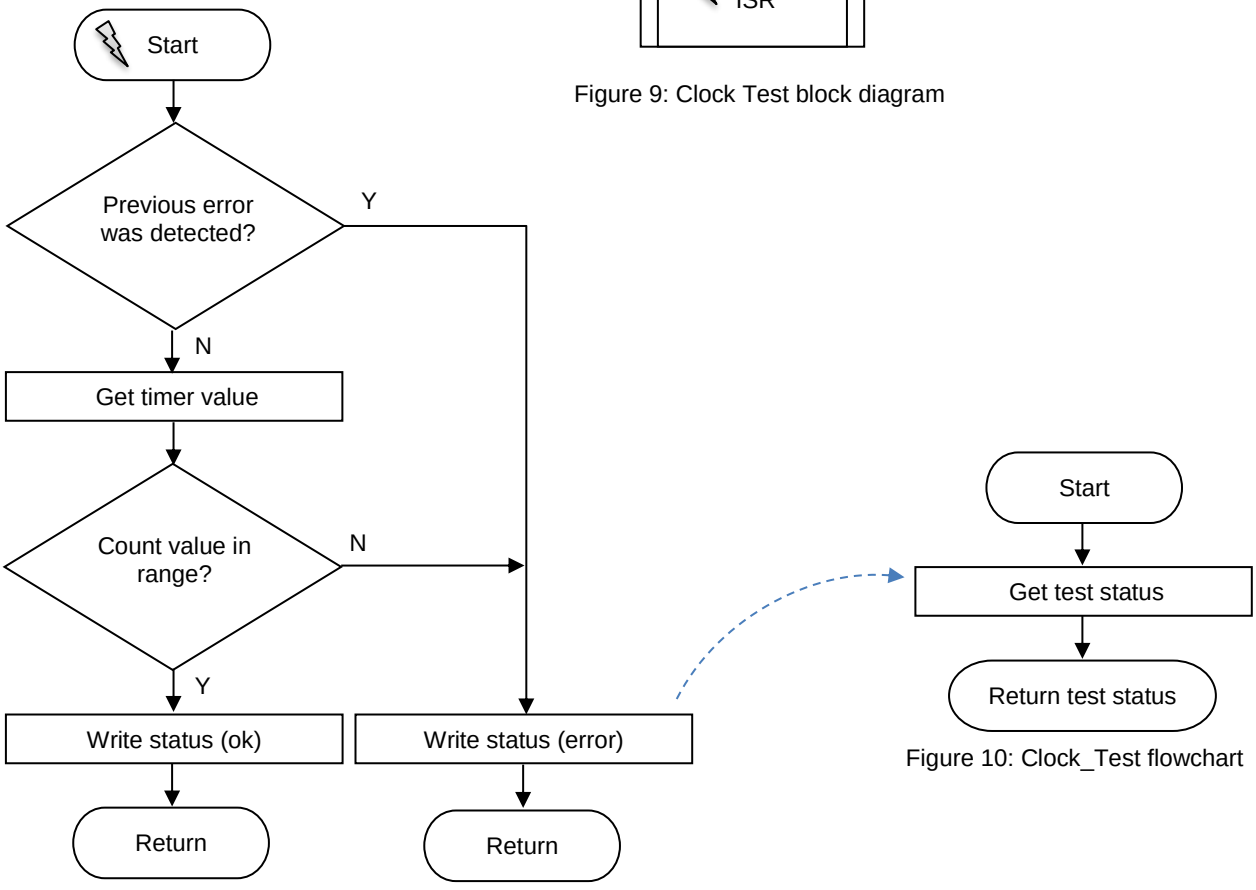

Figure 10: Clock_Test flowchart

Figure 11: Watch Prescaler Interrupt Handler flowchart

4.3.2 API Definition

Clock_Test

Name	Parameters	Return
Clock_Test	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

Clock_Test() has to be periodically called by the application, for example in the main loop. The function checks and returns the result of the Watch prescaler interrupt.

Clock_Test_Init

Name	Parameters	Return
Clock_Test_Init	None	None

Description:

Clock_Test_Init() Set clock test corresponding variable and flag (upper/lower frequency value).this API should be called at system initialization before starting clock test. The variable value (upper/lower frequency value) should be set according to accuracy of the clock sources.

4.4 Invariable Memory Test (ROM)

The Flash size can be configured according to different products in SelfTest_Flash.h. The CRC (Cyclic Redundancy Check) is an error detection system. The CRC code is a remainder after an input data string is divided by the pre-defined generator polynomial, assuming the input data string is a high order polynomial. Ordinarily, a data string is suffixed by a CRC code when being sent, and the received data is divided by a generator polynomial as described above. If the received data is dividable, it is judged to be correct. In this module, the generator polynomials are fixed to the numeric values for this mode.

- CCITT CRC16 generator with polynomial: 0x1021.
- CRC-16-IBM generator with polynomial 0xA001

4.4.1 Test Description

To meet Class B requirement, Flash test must be checked for “single bit fault”. This test can be implemented as CRC16 test.

This test can be implemented at startup procedure to test whole code area. Sample source code generates expected CRC code by API. But we recommend using expected CRC code which is calculated in advance. In this case, it is necessary to exclude pre-calculated CRC code area from target code area for CRC check.

Because CRC16 is limited to 32kb (4kB) the flash is checked in chunks of 4kB and a CRC value is stored for each chunk.

The Invariable Memory Test Flow Chart is shown as following figure.

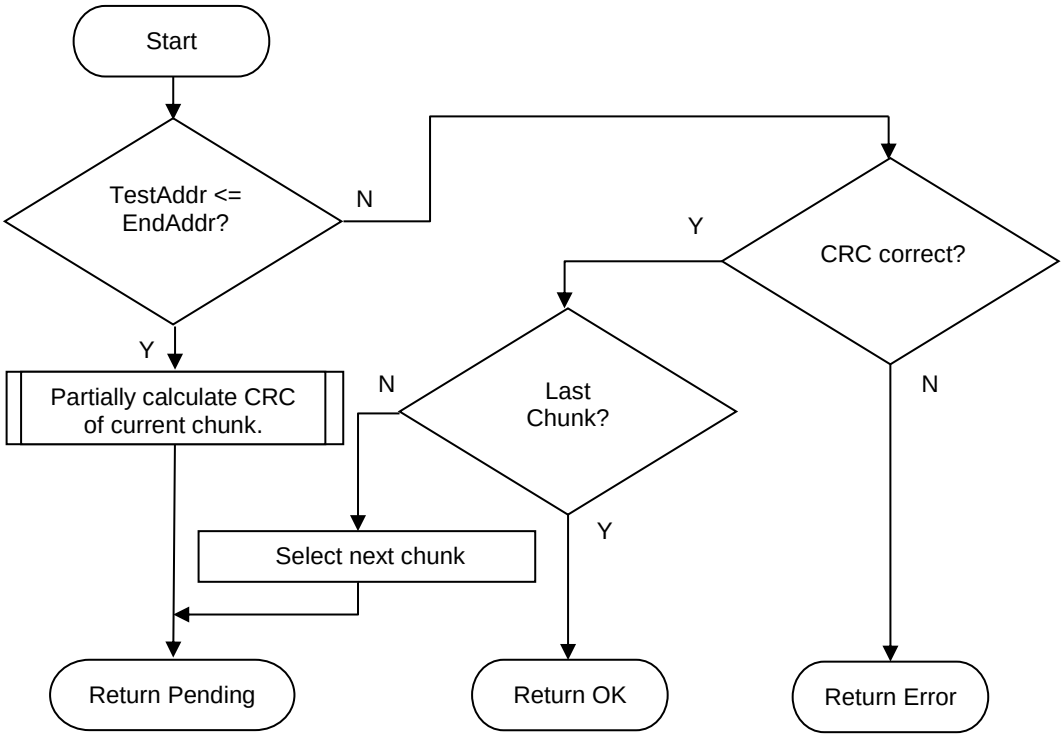

Figure 12: ROM Test flowchart

The test includes two CRC16 implementations, to choose from. Both implementations have a hamming distance of 4, but differ in execution speed and the amount of memory allocated for the pre-calculated CRC table.

The CRC algorithm being used can be selected through the define *CRC_Mode* in SelfTest_Flash.h.

First CRC16 implementation (Mode0) – fast, high memory footprint:

```
unsigned int calc_crc16( unsigned char  *start_adr, unsigned int size, unsigned int
crcdat)
{
    unsigned int crc16 = crcdat;

    while(size--)
    {
        if((start_adr < (unsigned char *)(Flash_Test_CRC_ADDR)) || \
           (start_adr > (unsigned char *)(Flash_Test_CRC_ADDR + \
           (Flash_Test_SectorCount * 2) + 1)))
        {
            crc16 = crctab[(crc16 ^ *(start_adr++)) & 0xff] ^ (crc16 >> 8);
        }
        else
        {
            start_adr++;
        }
    }
    return (unsigned int)crc16;
}

// CRC-16-IBM polynom 0xA001
// Hamming distance: 4 @ 4kB
// Detects odd bit errors
static const unsigned int crctab[256] = {
        0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
        0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
        0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
        0x0A00, 0xCAC1, 0xCB81, 0x0B40, 0xC901, 0x09C0, 0x0880, 0xC841,
        0xD801, 0x18C0, 0x1980, 0xD941, 0x1B00, 0xDBC1, 0xDA81, 0x1A40,
        0x1E00, 0xDEC1, 0xDF81, 0x1F40, 0xDD01, 0x1DC0, 0x1C80, 0xDC41,
        0x1400, 0xD4C1, 0xD581, 0x1540, 0xD701, 0x17C0, 0x1680, 0xD641,
        0xD201, 0x12C0, 0x1380, 0xD341, 0x1100, 0xD1C1, 0xD081, 0x1040,
        0xF001, 0x30C0, 0x3180, 0xF141, 0x3300, 0xF3C1, 0xF281, 0x3240,
        0x3600, 0xF6C1, 0xF781, 0x3740, 0xF501, 0x35C0, 0x3480, 0xF441,
        0x3C00, 0xFCC1, 0xFD81, 0x3D40, 0xFF01, 0x3FC0, 0x3E80, 0xFE41,
        0xFA01, 0x3AC0, 0x3B80, 0xFB41, 0x3900, 0xF9C1, 0xF881, 0x3840,
        0x2800, 0xE8C1, 0xE981, 0x2940, 0xEB01, 0x2BC0, 0x2A80, 0xEA41,
        0xEE01, 0x2EC0, 0x2F80, 0xEF41, 0x2D00, 0xEDC1, 0xEC81, 0x2C40,
        0xE401, 0x24C0, 0x2580, 0xE541, 0x2700, 0xE7C1, 0xE681, 0x2640,
        0x2200, 0xE2C1, 0xE381, 0x2340, 0xE101, 0x21C0, 0x2080, 0xE041,
        0xA001, 0x60C0, 0x6180, 0xA141, 0x6300, 0xA3C1, 0xA281, 0x6240,
        0x6600, 0xA6C1, 0xA781, 0x6740, 0xA501, 0x65C0, 0x6480, 0xA441,
         0x6C00, 0xACC1, 0xAD81, 0x6D40, 0xAF01, 0x6FC0, 0x6E80, 0xAE41,
        0xAA01, 0x6AC0, 0x6B80, 0xAB41, 0x6900, 0xA9C1, 0xA881, 0x6840,
```

```
        0x7800, 0xB8C1, 0xB981, 0x7940, 0xBB01, 0x7BC0, 0x7A80, 0xBA41,
        0xBE01, 0x7EC0, 0x7F80, 0xBF41, 0x7D00, 0xBDC1, 0xBC81, 0x7C40,
        0xB401, 0x74C0, 0x7580, 0xB541, 0x7700, 0xB7C1, 0xB681, 0x7640,
        0x7200, 0xB2C1, 0xB381, 0x7340, 0xB101, 0x71C0, 0x7080, 0xB041,
        0x5000, 0x90C1, 0x9181, 0x5140, 0x9301, 0x53C0, 0x5280, 0x9241,
        0x9601, 0x56C0, 0x5780, 0x9741, 0x5500, 0x95C1, 0x9481, 0x5440,
        0x9C01, 0x5CC0, 0x5D80, 0x9D41, 0x5F00, 0x9FC1, 0x9E81, 0x5E40,
        0x5A00, 0x9AC1, 0x9B81, 0x5B40, 0x9901, 0x59C0, 0x5880, 0x9841,
        0x8801, 0x48C0, 0x4980, 0x8941, 0x4B00, 0x8BC1, 0x8A81, 0x4A40,
        0x4E00, 0x8EC1, 0x8F81, 0x4F40, 0x8D01, 0x4DC0, 0x4C80, 0x8C41,
        0x4400, 0x84C1, 0x8581, 0x4540, 0x8701, 0x47C0, 0x4680, 0x8641,
        0x8201, 0x42C0, 0x4380, 0x8341, 0x4100, 0x81C1, 0x8081, 0x4040
};
```

Second CRC16 implementation (Mode1) – slow but small memory footprint:

```
unsigned int calc_crc16(unsigned char *p_data, unsigned int size, unsigned int crcdat)
{
    unsigned char temp;
    unsigned int crc = crcdat;
    while(size-- != 0)
    {
        temp = ((unsigned char)(crc/256))/16;
        crc <<= 4;
        crc ^= crc_table[temp ^ (*p_data/16)];
        temp = ((unsigned char)(crc/256))/16;
        crc <<= 4;
        crc ^= crc_table[temp^(*p_data & 0x0F)];
        p_data ++;
    }

    return crc;
}

// CRC-CCITT 16 polynom 0x1021
// Hamming distance: 4 @ 4kB
// Detects odd bit errors
const unsigned int crc_table[16]={
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50A5, 0x60C6, 0x70E7,
    0x8108, 0x9129, 0xA14A, 0xB16B, 0xC18C, 0xD1AD, 0xE1CE, 0xF1EF,
};
```

4.4.2 API Definition

Flash_Test_Init

Name	Parameters	Return
Flash_Test_Init	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR 2: TEST_FUNC_PENDING

Description:

Flash_Test_Init() tests the complete Flash for bit faults, should be used for the initial flash test at start-up. The function can be configured by setting CRC_Write_Flash to write the first calculated CRC value to the flash.

Returns Error if the previous generated CRC value differs and Pending if the CRC calculation is not yet completed.

Flash_Test

Name	Parameters	Return
Flash_Test	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR 2: TEST_FUNC_PENDING

Description:

Flash_Test() checks the flash by calculating the CRC values in small iterations. Returns Error if the CRC values do not match and Pending if the CRC calculation is not yet completed.

4.5 Variable Memory Test (RAM)

To meet Class B requirement, RAM test must be checked for “DC fault”. A checkerboard algorithm and a marching C- are provided for the RAM test.

In addition, Macros are provided to load and store variables and their inverted value.

4.5.1 Checkerboard Test

The test will restore the RAM to the previous state, so it can be executed parallel to the application, but because the RAM is modified during the test interrupts will be disabled. The tests are designed to check the RAM in chunks, to reduce the time the main application is stopped. The test returns pending, as long as the RAM is not tested completely, but instantly returns an error if detected.

Operation	Description
⇑ W_55AA	Write 0x55AA to the RAM area to be tested
⇑ R_55AA	Read back the previous written 0x55AA
⇑ W_AA55	Write 0xAA55 to the RAM area to be tested
⇑ R_AA55	Read back the previous written 0xAA55
⇑ W_FF00	Write 0xFF00 to the RAM area to be tested
⇑ R_FF00	Read back the previous written 0xFF00
⇑ W_00FF	Write 0x00FF to the RAM area to be tested
⇑ R_00FF	Read back the previous written 0x00FF

Table 4-2: Checkerboard pattern

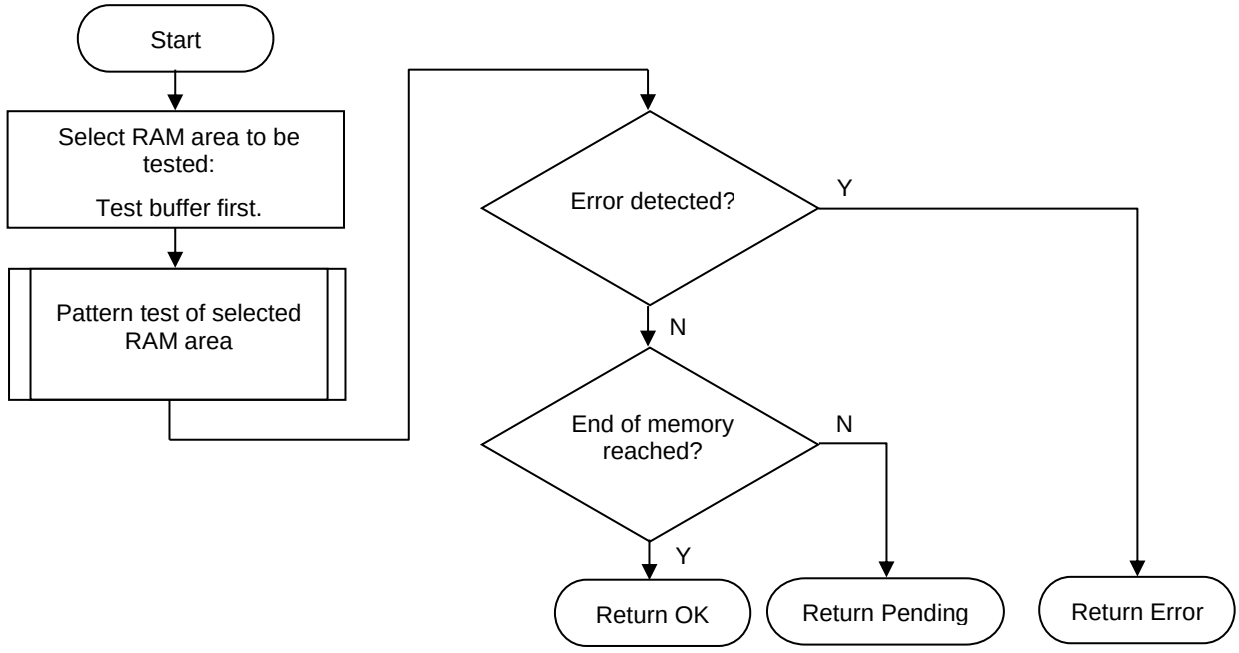

Figure 13: Checkerboard Test flowchart 1/2

Figure 14: RAM Test flow chart

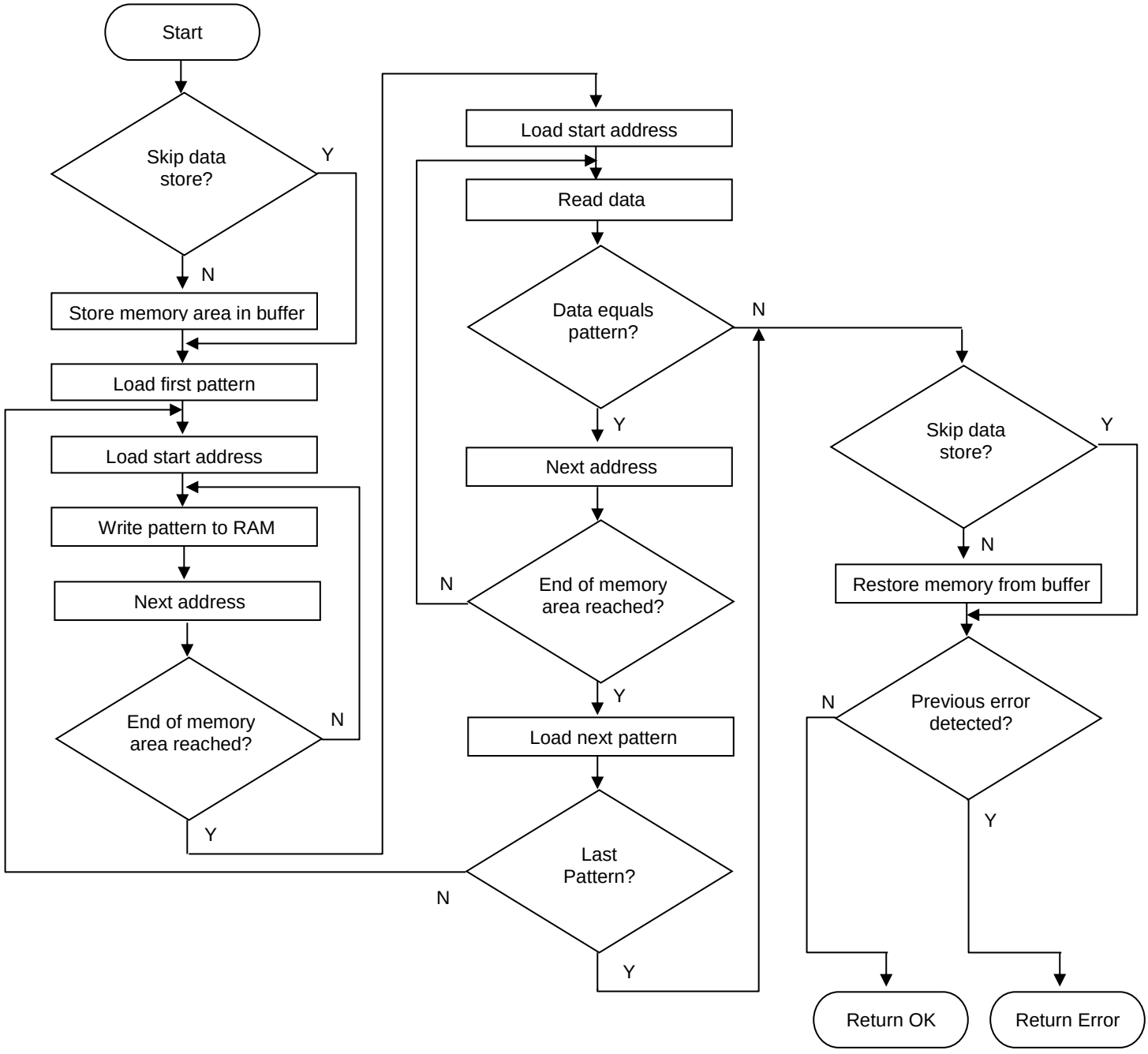

Figure 15: Checkerboard Test flowchart 2/2

4.5.1.1 API Definition

Ram_Test

Name	Parameters	Return
Ram_Test	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR 2: TEST_FUNC_PENDING

Description:

Ram_Test() tests a chunk of memory as defined in SelfTest_RAM.h (RAM_TEST_BUFFER_SIZE) with the PATTERN_CHECK() function. The Buffer will be tested first, to prevent false write back.

To test the complete RAM the function has to be called repeatedly.

While the RAM isn't completely tested, the function returns TEST_FUNC_PENDING and if the complete RAM was tested return TEST_NORMAL. In case of an error or TEST_FUNC_ERROR is returned.

Ram_Test_All

Name	Parameters	Return
Ram_Test_All	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

Ram_Test_All() a while loop that executes Ram_Test() until the complete memory is tested or an error occurred.

PATTERN_CHECK

Name	Parameters	Return
PATTERN_CHECK	unsigned char *start_adr unsigned int size unsigned int skip_store	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

PATTERN_CHECK() checks the specified RAM area with the patterns defined in SelfTest_RAM.h. Up to four different pattern can be defined (or deselected with 0x01). The function is able to restore the memory after the test, but for testing the buffer this can be deselected by setting skip_store to '1'.

4.5.1.2 Definitions

Name	Value	Note
RAM_TEST_BUFFER_SIZE*	0x50	Number of bytes to be checked per cycle
RAM_TEST_START_ADDRESS*	Start address of RAM test	RAM test starts form the set address. User can set the optional value.
RAM_TEST_END_ADDRESS*	End address of RAM test	RAM test tests to the set address. It includes the set address on this #define. User can set the optional value.
RAM_TEST_BUFFER_ADR*	Address of the Buffer	Buffer used to temporarily store the RAM data.

*definitions shared between RAM tests

Table 4-3: Pattern RAM Test Definitions

4.5.2 Extended Marching C-/X test

In addition to the checkerboard test an Extended Marching C- test is provided. The checkerboard test is executed faster, but does not detect the same level of memory faults as Marching C- test does.

Operation	Description	
⇑ W0	Write 0 to the RAM area to be tested	
⇑ R0, ⇑ W1, ⇑ R1	Read back the previous written 0, Write 1 to the RAM area to be tested and Read back 1	
⇑ R1, ⇑ W0	Read back the previous written 1, Write 0 to the RAM area to be tested	Excluded in March X test.
⇓ R0, ⇓ W1	Read back the previous written 0, Write 1 to the RAM area to be tested	
⇓ R1, ⇓ W0	Read back the previous written 1, Write 0 to the RAM area to be tested	
⇓ R0	Read back the previous written 0	

Table 4-4: Extended March C-Pattern

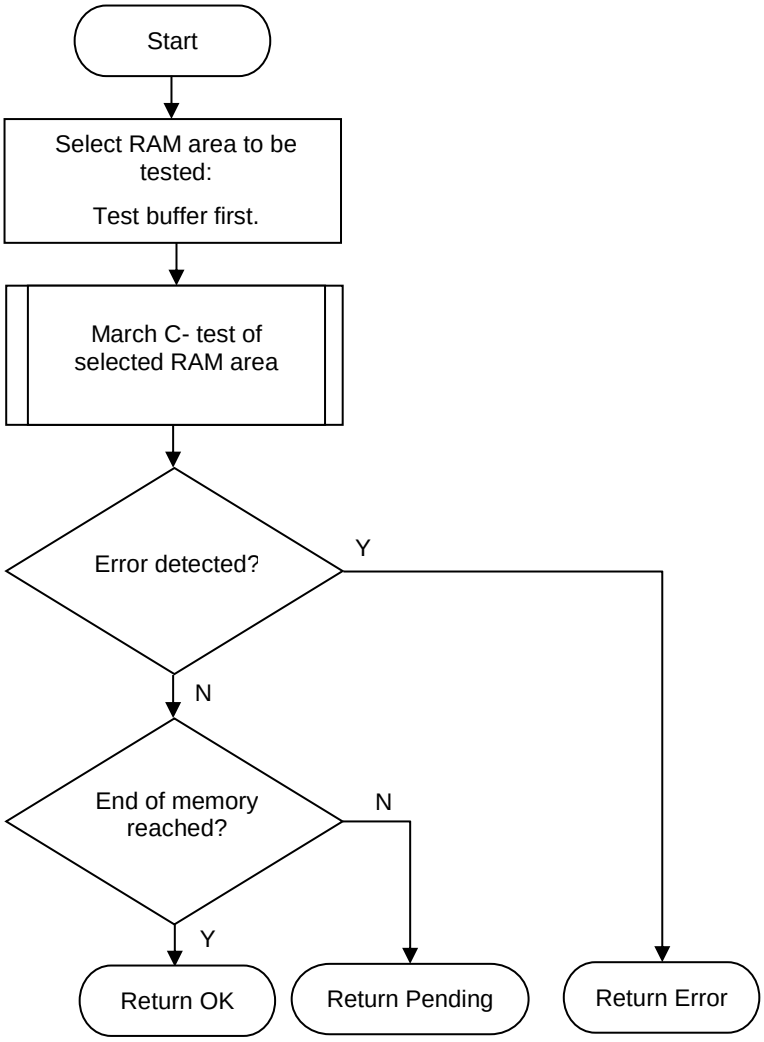

Figure 16 : March C flowchart

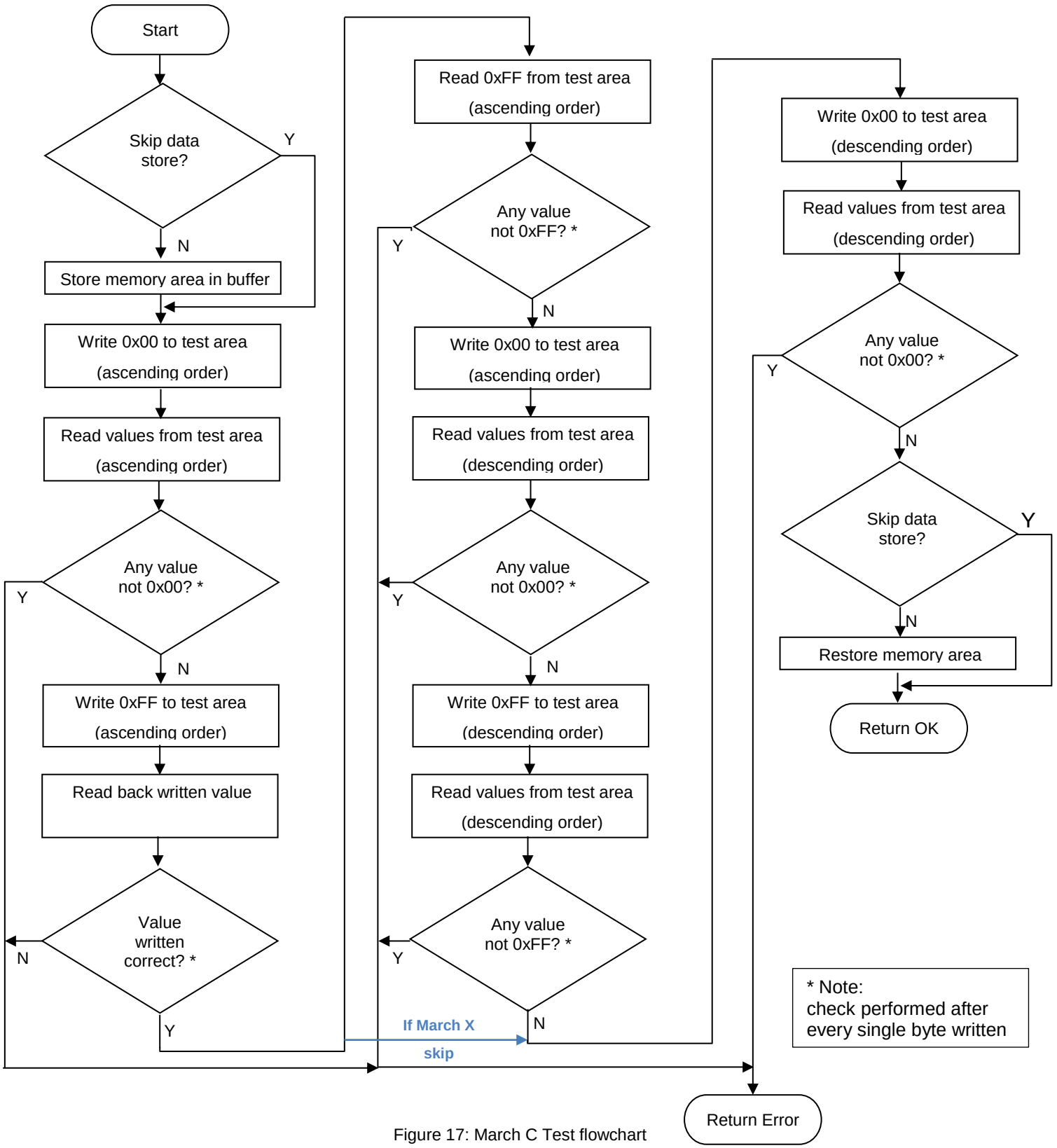

Figure 17: March C Test flowchart

4.5.2.1 API Definition

Ram_March_Test

Name	Parameters	Return
Ram_March_Test	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR 2: TEST_FUNC_PENDING

Description:

Ram_March_Test() tests a chunk of memory as defined in SelfTest_RAM.h (RAM_TEST_BUFFER_SIZE) with the PATTERN_CHECK() function. The Buffer will be tested first, to prevent false write back.

To test the complete RAM, the function has to be called repeatedly.

While the RAM isn't completely tested, the function returns TEST_FUNC_PENDING and if the complete RAM was tested return TEST_NORMAL. In case of an error or TEST_FUNC_ERROR is returned.

Ram_March_Test_All

Name	Parameters	Return
Ram_March_Test_All	None	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

Ram_March_Test_All() a while loop that executes Ram_March_Test() until the complete memory is tested or an error occurred.

MARCH_C_TEST

Name	Parameters	Return
MARCH_C_TEST	unsigned char *start_adr unsigned int size unsigned int skip_store unsigned char marchX	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

MARCH_C_TEST() checks the specified RAM area with the patterns defined in SelfTest_RAM.h. Up to four different pattern can be defined (or deselected with 0x01). The function is able to restore the memory after the test, but for testing the buffer this can be deselected by setting skip_store to '1'.

A MarchX test (only execute first two and last two Tests) can be selected by setting parameter marchX to other than '0'.

4.5.2.2 Definitions

Name	Value	Note
RAM_TEST_BUFFER_SIZE	0x50	Number of bytes to be checked per cycle
RAM_TEST_START_ADDRESS	Start address of RAM test	RAM test starts form the set address. User can set the optional value.
RAM_TEST_END_ADDRESS	End address of RAM test	RAM test tests to the set address. It includes the set address on this #define. User can set the optional value.
RAM_TEST_BUFFER_ADR	Address of the Buffer	Buffer used to temporarily store the RAM data.

*definitions shared between RAM tests

4.5.3 Mailbox for special variables

A Mailbox system is implemented to share special variables and add an additional layer of security to the software.

The variables in the mailbox are stored as complementary values and special functions are implemented to validate them.

The variables are stored as part of a structure that combines both the regular and inverted variable, so they are stored next to each other in RAM.

Declaration-, Write- and Read commands are provided in form of macros.

To validate the variables a function Mailbox_check() has to be executed. It will check a set amount of data at a time.

4.5.3.1 Functions

Mailbox_Init

Name	Parameters	Return
Mailbox_Init	none	none

Description:

Initialize the mailbox by setting the complete Mailbox memory to 0xFF to cause an error if a value in the mailbox is not set before testing.

Mailbox_check

Name	Parameters	Return
Mailbox_check	none	0: TEST_NORMAL 1: TEST_FUNC_ERROR 2: TEST_FUNC_PENDING

Description:

Check the Mailbox that all values and their complementary entries match.

Per execution MAILBOX_CHECK_SIZE variables will be tested, until completion TEST_FUNC_PENDING and on completion TEST_NORMAL will be returned. In case of an error TEST_FUNC_ERROR is returned.

4.5.3.2 Macros

Name	Value
Mailbox_LONG (Var)	Declare variable Var and its complementary value Var_n as long data type.
Mailbox_ULONG (Var)	Declare variable Var and its complementary value Var_n as unsinged long data type.
Mailbox_SET(Var, Data)	Set value Data to variable Var and the complementary value to Var_n.
Mailbox_GET(Var)	Read value of variable Var
Mailbox_CHECK(Var)	Check variable: return 0 if valid, 1 if not.

4.5.3.3 Definitions

Name	Value	Note
MAILBOX_CHECK_SIZE	3	Number of variables to check per execution
MAILBOX_SIZE	n*8	Size of the mailbox. Because only 32bit (4byte) values and another inverted 32bit value are used per entry in the mailbox, it has to be a multiple of 8.
MAILBOX_END	0x0440	End address of the mailbox.
MAILBOX_START	MAILBOX_END - MAILBOX_SIZE	Start address of the mailbox. Calculated by end address - size.

4.6 IO Test

4.6.1 Test Description

To meet Class B requirement, GPIO must be check for “Function error”. So function test is implemented for both input and output function. The IO Output Test Flow Chart is shown as following figure. Actually, GPIO has following characters.

- When output mode, reading the PDR register returns the PDR register value.
- When input mode, reading the PDR register returns the pin value.

That is to say, when output mode, the pin value may not accord with PDR register value.

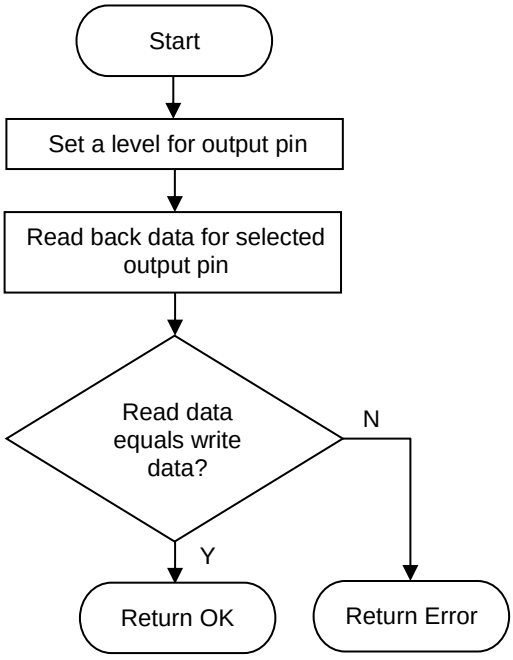

Figure 18: Output test

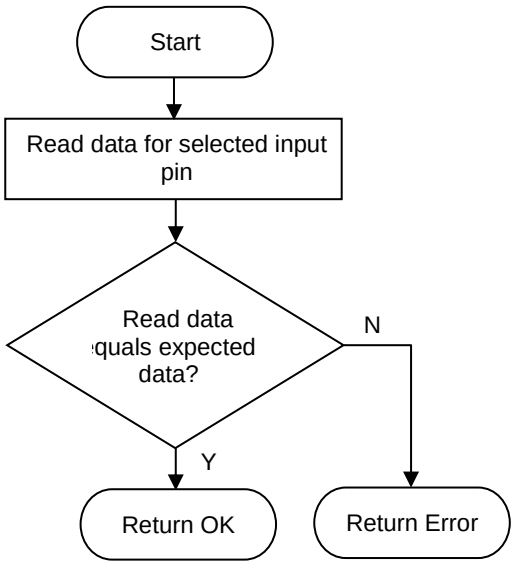

Figure 19: Input test

4.6.2 API Definition

IO_Input_Test

Name	Parameters	Return
IO_Input_Test	port: port address (corresponding PDR register address) bit: bit number pull: pull register address value: expected pin level	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

IO_Input_Test () can check if selected IO input value is same with expected value. It should be done before user code initialization.

IO_Output_Test

Name	Parameters	Return
IO_Output_Test	port: port address (corresponding PDR register address) bit: bit number value: output level	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

IO_Output_Test() can check if output value is correct. It should be done before user code initialization.

Note: The Output Test can’t ensure that a short to VCC is present, as the device does not have an internal pull-down resistor. Thus in case of a short to VCC and forcing the pin to GND the pin will be overloaded.

4.7 A/D Test

4.7.1 Test Description

To meet Class B requirement, A/D must be check for “Function error”. This test samples A/D signal from selected A/D channels and check if the A/D convert values are in the expected ranges. The A/D Test Flow Chart is shown as following figure.

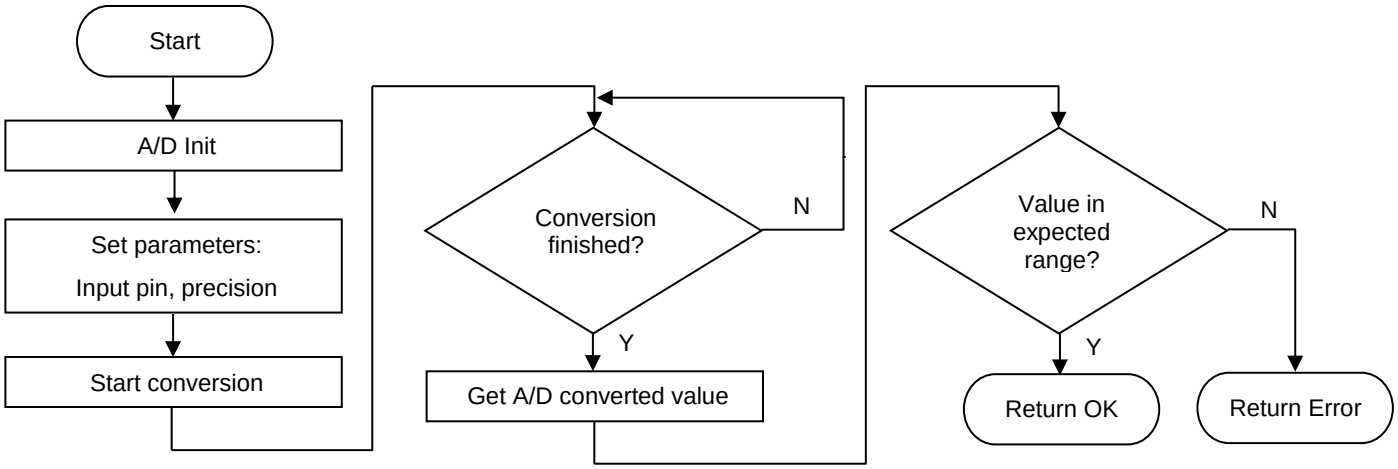

Figure 20: ADC Test flowchart

4.7.2 API Definition

Name	Parameters	Return
AD_test	ad_type: 8/10-bit precision - AD_TYPE_8BIT - AD_TYPE_10BIT pin_num: A/D channel number value_low: expected lower limit value_upper: expected upper limit	0: TEST_NORMAL 1: TEST_FUNC_ERROR

Description:

AD_test() samples A/D signal from selected A/D channel and check if the A/D convert value is within an expected range. If it cannot match the range Error is returned.

5 Usage Example of STL

The project shows how to integrate the IEC60730 STL into a real system.

5.1 Project Structure

All class B STL routines are designed for periodic execution and placed in the IEC60730 folder. The component to be tested is included in the file names as follows SelfTest_[component].c/h.

Interrupt handler used for the Interrupt and Clock Test are places in vectors.c.

In addition, functions for flash programming are provided in the helper folder to store the CRC values of the ROM test in flash.

5.1.1 Startup Self-Test

At startup the function PCTest_ResetSource() should be executed to determine if there was an unexpected reset event and stop further execution. Afterwards, register and complete Memory should be tested to ensure basic operation.

5.1.2 Periodic Test Initialization

Interrupt and Clock test should be initialized after the initial Self-Test completed.

Appendix A.

A.1 List of figures:

A.2 List of tables:

A.3 Testing:

Notes:

- As the flash test calculates a checksum for the flash, all breakpoints have to be set before this is happening, as they change the value being read of the statement they are put at.
- To enable the watchdog while debugging the entry “Watchdog” in „Setup → Debug Environment → Debug Environment → Chip“ has to be enabled
- CPU tests modify the CPU flags, data pointer and working registers thus they will disable all interrupts.

RAM test:

Set a breakpoint when the patterns are written inside SelfTestRAM.c and modify the values of the RAM. Further execution should detect the error and the test should fail.

Flash test:

The flash calculates a CRC checksum for the whole flash. To Test the functionality, you can:

- use a breakpoint after the initial CRC was calculated (see above notes).
- set a breakpoint in SelfTestFlash.c to modify the data received from flash before it is used in the calculation.
- use a function in your code to write to the flash and modify the flash directly. Flash functions are provided.

CPU test:

General register test:

- The general purpose registers test is encapsulated in itself, for testing a breakpoint has to be set inside the function and the working register values have to be modified to trigger the error flag.

conditional register test:

- The conditional register test is encapsulated in itself, for testing a breakpoint has to be set inside the function and the register values have to be modified to force different status flags and trigger the error flag.

Dedicated register test:

- The dedicated register test is encapsulated in itself, for testing a breakpoint has to be set inside the function and the register values have to be modified to trigger the error flag.

Direct bank pointer test:

- The direct bank pointer test is encapsulated in itself, for testing a breakpoint has to be set inside the function and the register values have to be modified to trigger the error flag. This can be done either by modifying the direct pointer (DP), Memory data or return values inside the working registers.

Program Counter test:

- To test the program, the watchdog counter is used and resets the device if the program gets stuck. For testing a delay can be added to the program by stopping program execution through a breakpoint. The watchdog has to be running during debugging, see above notes.

Interrupt handling & execution

- Can be tested by disabling one of the interrupt sources or changing the interrupt frequency, so the interrupt is triggered too less/often.

Clock test:

Comparison of the Main RC Oscillator and either sub clock or sub RC oscillator. Testing can be done by:

- changing the Main CR oscillator frequency in “mcuconfig.h” (define MAIN_CR_CLOCK).
- disabling one of the clocks during runtime.
- in case of the external sub clock by disconnecting the oscillator or changing its frequency.

Digital I/O test:

- Input: The test expects a specified value on the input pin, for testing the opposite value can be provided on the pin.
- Output: hard to test without overloading the MCU pin.

ADC test:

- The ADC test checks if an analog value is in a predefined range. For testing force the external voltage to a different level.

A.4 Timing analysis:

All Tests done at 8MHz CPU clock if not stated different within the test description or Remarks. Timing measured by signaling function start and end via GPIO. The start is signaled before calling the function, the end after the function returned from the call. Tests that are performed in multiple passed are executed in a loop and time is measured before and after that loop. This will add a small but negligible error to the measured timing due to the additional overhead by setting/clearing the GPIO and processing the loop.

Each test performed at least ten times, Typ value is the average time of the measurements, Min and Max the longest and shortest execution times (If there was any variation).

ID	Name	Time				Description
		Min	Typ	Max		
CPU Test – Register Tests						
T1.0	General Register Test function runtime		88		µs	single pass, one register set
T1.1	General Register Test function runtime		3175		µs	complete pass loop of 32 register sets
T1.2	Dedicated Register Test function runtime		70		µs	
T1.3	Conditional Register Test function runtime		33.3		µs	
T1.4	Direct Bank Pointer Test function runtime		400		µs	
CPU Test – PC Test						
T2.0	PC Test Init function runtime		N/A*			
T2.1	PC Test Clear function runtime		N/A*			
T2.2	PC Test Reset Source function runtime		N/A*			
Interrupt Test						
T3.0	Test Init function runtime		38.5		µs	
T3.1	ISR runtime		41		µs	
T3.2	Interrupt Test function runtime		316		µs	OK – ISR ratio 1:8
T3.3	Interrupt Test function runtime		312		µs	ERROR – ISR ratio 1:1
T3.4	Interrupt Test function runtime		N/A*			ERROR – ISR counts 0
Clock Test						
T4.0	Clock Test Init function runtime		30		µs	
T4.1	Clock Test function runtime		N/A		µs	
Invariable Memory Test (ROM) - CRC Mode 0						
T5.0	Flash Test Init function runtime		233.5		ms	
T5.1	Flash Test function runtime		240		ms	single pass 32 byte data size
T5.2	Flash Test function runtime		1208		ms	test area: 0x8000 – 0xBFFF 32 byte data size

Invariable Memory Test (ROM) - CRC Mode 1						
T6.0	Flash Test Init function runtime		1960		ms	
T6.1	Flash Test function runtime		1970		ms	single pass 32 byte data size
T6.2	Flash Test function runtime		98600		ms	test area: 0x8000 – 0xBFFF 32 byte data size
Variable Memory Test (RAM) – Checkerboard						
T7.0	Checkerboard Test function runtime		0.695	2.24**	ms	Single pass 16 byte data size
T7.1	Checkerboard Test All function runtime		26.2		ms	RAM start address 0x210 RAM end address 0x48F 16 byte data size
Variable Memory Test (RAM) – MARCH C						
T8.0	MARCH C Test function runtime		0.75	2.65**	ms	Single pass 16 byte data size
T8.1	MARCH C Test All function runtime		30.1		ms	RAM start address 0x210 RAM end address 0x48F 16 byte data size
Variable Memory Test (RAM) – MARCH X						
T9.0	MARCH X Test function runtime		0.59	1.66**	ms	Single pass 16 byte data size
T9.1	MARCH X Test All function runtime		22		ms	RAM start address 0x210 RAM end address 0x48F 16 byte data size
Variable Memory Test (RAM) – Mailbox						
T10.0	Mailbox Init function runtime		N/A*			Mailbox size 30
T10.1	Mailbox Check dataset function runtime		43		µs	Check size 3

* Time measured less than 10µs, to high error due to the measurement method.

** Occurs when the RAM test buffer is checked. Occurs whenever the RAM test start from the beginning.

Document History

Document Title: AN204364 - IEC60730 CLASS B SELF-TEST LIBRARY

Document Number: 002-04364

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**		MIQI	06/13/2012	Initial release
*A	5849853	MIQI	05/12/2017	Converted Spansion Application Note “AN702-00006-1v0-E” to Cypress format
*B	6955819	ACEH	08/28/2020	Complete Update of Self-test Library Update of description: - Added DirectBankPointerTest - Updated RAM test with improved Checkerboard test and March-C/X test - Added Mailbox system to store variables - Added description for Testing the functionality - Added timing data for the functions

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709