AN3181
Application note

Guidelines for obtaining IEC60335 Class B certification in an STM8 application

Introduction

Safety plays an increasingly important role in electronics hardware and software applications. Safety requirements and suggestions are specified at a world wide level, with international standards defined by IEC (International Electro Technical Commission) and require certification proving compliance to the standards and showing system robustness.

The main purpose of this STM8 family application note and the associated software is to ease and to speed up software development and certification processes for applications subject to these requirements and certifications.

The software packages are separated for STM8S and STM8L families, but as they use the same principles the description is a common one. The main differences between packages concerns hardware architectural differences. They are mentioned in the present document and summarized in *Appendix A: Main differences of STM8S and STM8L packages*.

These optimized packages are independent from other firmware libraries published by ST. The only part which is included from firmware libraries are family headers and type files like ***stm8xxx.h*** and ***stm8xxx_type.h*** where all names of the registers are defined, bit masks and basic constants are listed. To optimize code size and speed up code execution, the rest of the library is not included.

Projects were prepared under STVD visual development tool and compiled by Cosmic compiler. Both tools are available for free on ST and Cosmic web sites.

Reference documents:

- **AN1015** software techniques improving EMC
- **AN1709** EMS design guide
- **AN2860** EMC guidelines for STM8S

Contents

List of tables

List of figures

1 Compliance with IEC and VDE standards

IEC is a world wide recognized authority on international standards for a vast range of safety issues concerning electrical, electronic and related technologies. One such safety standard is IEC 60335-1 which covers household electronic appliances safety and security following a hardware and software failure.

The basic principle here is that the appliance must remain safe in case of any component failure. If safety relies on an electronic component, it must remain safe after two consecutive failures. The two failures can be either:

- electronic component failure plus mechanical failure
- two electronic failures

A microcontroller is a specific software-driven electronic component. An appliance driven by a microcontroller must stay in a safe mode after either of the two above conditions. The microcontroller is either in reset or not operating properly. The software aspect becomes important only when a second applied fault occurs and only in the case where the safety of the device depends on the software. Safety standards are divided into three types of safety classes from this point of view, depending on application purpose:

- **Class A**: Safety does not rely on software
- **Class B**: software prevents unsafe operation
- **Class C**: software is intended to prevent special hazards

This application note and associated ST software packages cover the group B specification only.

The IEC standard compliance for microcontrollers relates both to hardware and software. A list of microcontroller components needing evaluation for compliance is found in IEC 60335-1 Annex T which refers to IEC60730 Annex H. The components for certification can be divided into two groups: microcontroller specific and application specific (see *Table 1: MCU components to be tested for Class B compliance*).

Microcontroller specific parts are related only to the microcontroller architecture and can be made generic (core self diagnostic, variable and invariable memories integrity checking, clock system tests). Application specific parts rely on customer application structure and must be defined and developed by user (communication, I/Os software control, interrupts, analog inputs and outputs). The ST solution consists only of the microcontroller specific tests. They rely on the powerful STM8 MCU hardware features such as two independent watchdogs (WDG) or clock sources.

It is important to note that there are several other peer recognized bodies concerning electronic safety standards besides IEC, such as VDE in Germany, IET in the United Kingdom and the IEEE in the United States. In addition, an increasing number of product safety standards are being harmonized to international safety standards. For example, the UL 60335-1, the DSA 60335-1 and the EN 60335-1 are all documents which are based on the IEC 60335-1. VDE also includes a Testing and Certification Institute which is a pioneer of software safety inspection. This is a registered National Certification Body (NCB) for Germany. The main purpose of this testing house is to offer standards compliance and quality testing services for electrical appliance manufacturers. The STM8 FW in this package has been VDE certified.

Table 1. MCU components to be tested for Class B compliance

Group	Components to be tested
Micro specific	CPU registers
	CPU program counter
	Clock
	Fixed and variable memory spaces
	Internal addressing
	Internal data path
Application specific	Interrupt handling
	External communication and addressing (if any)
	Timing
	I/O periphery
	ADC and DAC
	Analog multiplexer

1.1 VDE certified tests included in ST firmware library

STM8 firmware library packages include the following microcontroller specific software modules, which have also been certified by VDE:

- CPU registers test
- clock monitoring
- RAM functional check
- FLASH checksum integrity check
- watchdog self-test
- stack overflow monitoring

An overview of the methods used for these MCU specific tests is given in MCU parts which must be tested under Class B compliance. They are described in more detail in the next chapters. The last two are not explicitly required by the standard, but they improve overall fault coverage and as such are implemented in the ST solution.

Table 2. Overview of methods used in microcontroller specific tests of associated ST firmware library

Components to be verified	Method used
CPU registers	A, X, Y registers, flags and stack pointer functional tests are done at startup. Flags are not tested during run test. Stack is permanently tested for overflow and underflow. Stuck is performed at fault detection.
Program counters	Two different WDGs running on two independent clock sources can reset the device when the program counter is lost. Window WDG (running at core frequency) performs time slot monitoring(1). Independent WDG (running at low speed frequency, impossible to disable once enabled) must be serviced at regular intervals. Program control flow is monitored by specific software routine (see *Section 2.1.3*).
Addressing and data path	RAM functional and FLASH integrity tests, stack overflow (specific pattern is written at the end of stack space and checked for corruption in regular intervals).
Clock	Two independent internal frequencies are used for the dedicated timer clock and they are verified by reciprocal comparison. One frequency is fed into the dedicated timer while the other gates it.
Fixed memory	Full memory16-bit CRC checksum test done at startup. Partial memory test is repeated during run mode (block by block).
Variable memory	March C- (March X optionally) full memory test done at startup. Partial memory test is repeated during run mode (block by block). Word protection with double inverse redundancy is used for safety critical (Class B) variables. Class A variable space, stack and unused space are not tested during run mode

1. Window WDG feature is not available at STM8L10x devices.

The user can include a part or all of these VDE certified software modules into his project. If the modules stay unchanged and are integrated in accordance with the guidelines provided in this application note, development time and costs to have end-application certification would be significantly reduced.

1.2 Application specific tests

The user should be aware that the following are also required for Class B certification but are not included in the ST firmware library:

- Analog: ADC/DAC & multiplexer
- I/Os
- Interrupts and external communication
- Timing
- External addressing

1.2.1 ADC/DAC

Analog components depend on device's application and peripheral capabilities. Used pins should be checked at correct intervals. Free analog pins can be used for checking user analog reference points. Internal references should be checked, too.

1.2.2 I/Os

Class B tests must detect any malfunction on digital I/Os. This could be covered by plausibility checks together with some other application parts (e.g. change in an analog signal from temperature sensor when heating/cooling digital control is switched on/off).

1.2.3 Interrupts and external communication

Application interrupts occurrence and external communications can be checked by different methods. One of them could be a control using a set of incremental counters where every interrupt or communication event increments a specific counter. The values in the counters are then verified at given time intervals by cross-checking against some other independent time base.

1.2.4 Timing

Timing could be verified by ensuring that the application routines execution times are correct and that there are no unexpected delays. A cross-check with a different time base could be done. Timing control is strictly dependent on the application.

1.2.5 External addressing

External addressing is not used with STM8 microcontrollers.

2 Class B software package

This section highlights the basic common principles used in ST software solution. The workspace organization under STVD tool is described together with its configuration and debugging capabilities.

2.1 Basic software principles

The basic software methods and common principles used for all the tests included in the ST firmware library are described in detail at this section.

2.1.1 Fail Safe mode

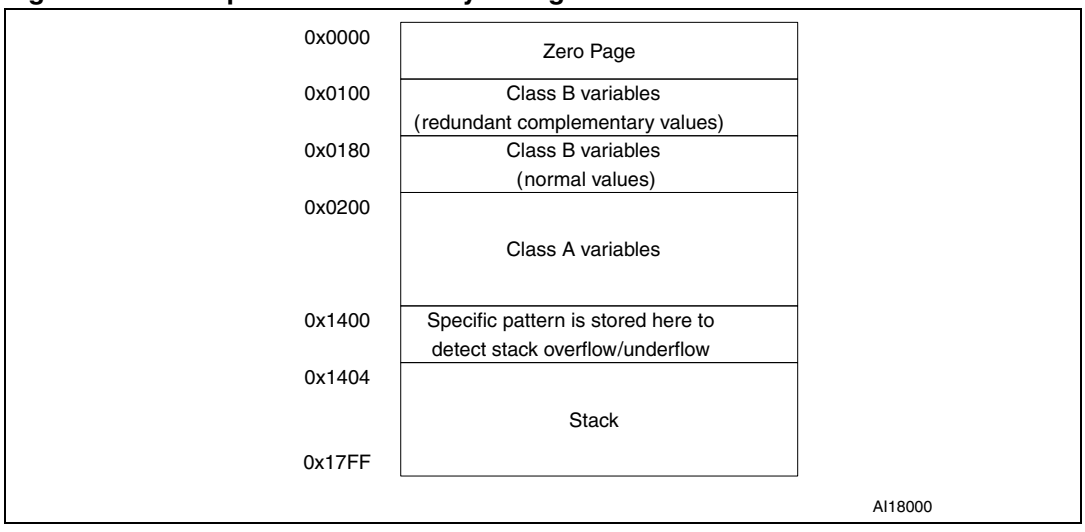
FailSafe() routine is called (defined in ***stm8x_stl_startup.c*** file) at any fail detection. The program stays in a never ending loop waiting for WDG reset. Except for some debugging features, the routine is almost empty. When editing this routine, the user must remember to include procedures necessary to keep the application in a safe state.

If the user wants to recognize the error raised, the debug or verbose mode described in *Section 2.3: Package configuring and debugging* can be used. In debug mode the independent WDG is refreshed inside the never ending loop to prevent resetting the microcontroller when a failure occurs.

2.1.2 Class B variables

Class B variables are dedicated variables defined by the user as critical to the application. They are always stored as a pair of complementary values in two separate RAM areas. Both normal and redundant values are always placed into non-adjacent memory locations. Partial transparent RAM March C- or March X test is performed permanently on these two regions through the system interrupt routines in run mode. The integrity of the pair is compared before the value is used. If any value stored is corrupted, ***FailSafe()*** routine is called. An example of RAM configuration is shown in *Figure 1*. The user can adapt the RAM space allocation according to application needs and with respect to device hardware capability.

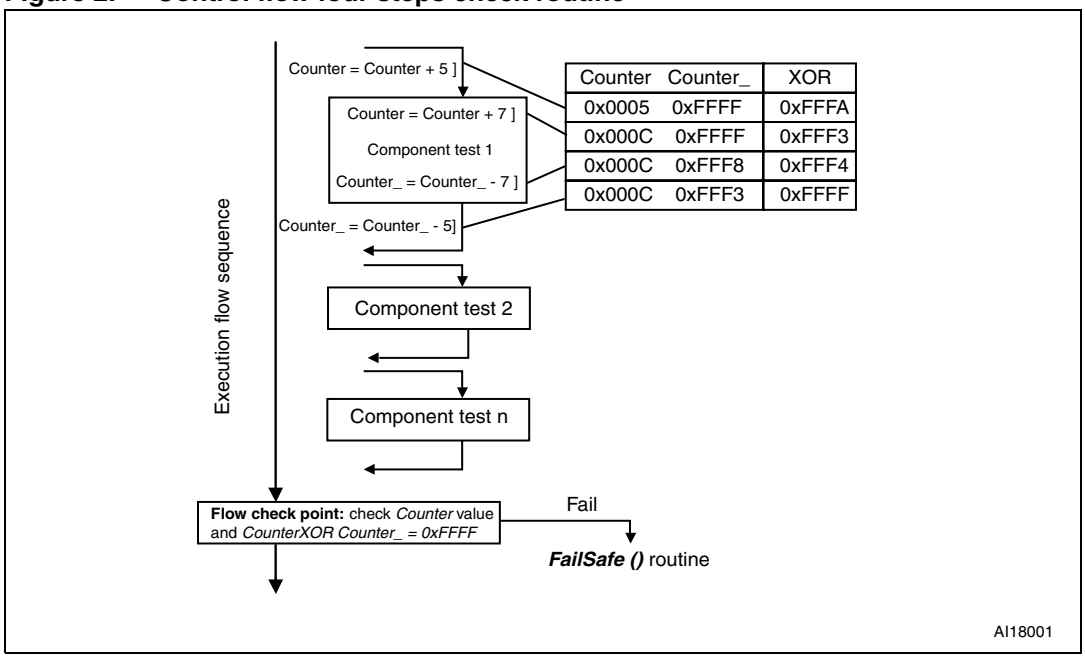
Figure 1. Example of RAM memory configuration for STM8S20x



2.1.3 Class B flow control

A specific method is used to test and check program execution flow. Unique user defined numbers are assigned for every software execution block. These unique numbers are stored in two complementary counters. When a block is executed, a symmetrical four step change of the counter pair content is done (add or subtract the label values). The first two steps check if the block is correctly called from main flow level (processed just before calling and just after return from the called procedure). The next two steps check if the block is correctly completed (processed just after enter and just before return from the procedure). An example is given in *Figure 2* where a routine performing a component test is called in the control flow sequence and the four step checking service is shown. This method decreases CPU load as the check is done by counting one member of the complementary counter pair only. As there is always the same number of call/return and entry/exit points, the value stored in the counter pair must be always complementary after any block is processed. Several flow check points are placed subsequently in the code flow. Counter integrity is always checked here and any unexpected value results in calling the ***FailSafe()*** routine.

Figure 2. Control flow four steps check routine



1. For this example, the unique number for Component test 1 is “5” and for the procedure itself it is defined as “7”. The counters are initialized to 0x0000 and 0xFFFF. The table in the upper right corner in *Figure 2* shows how the counters are changed in four steps. Also shown is their complementary value after the last step (return from procedure) is done.

2.2 Package organization

This section describes how the ST solution is organized.

2.2.1 Projects and workspaces included in the package

ST provides three separate product dependent workspaces for STM8 family. STVD workspaces IEC6035 (stored in IEC6035.stw files) include predefined projects for corresponding STM8 family members.

Workspace for STM8S family includes:

- iec60335_207_208 (stm8s207_208.stp) for STM8S20x devices
- iec60335_105 (stm8s105.stp) for STM8S105 devices
- iec60335_103 (stm8s103.stp) for STM8S103 devices
- iec60335_903 (stm8s903.stp) for STM8S903 devices

Workspace for STM8L15x family includes:

- iec60335_15x (stm8l_15x.stp) for STM8L15x devices

Workspace for STM8L101 family includes:

- iec60335_101 (stm8l_101.stp) for STM8L101 devices

2.2.2 Linker file settings

Each project covers all settings and includes needs for both compilation and linking processes. Also included is the STVD tool for automatic linker file generation. Some specific segment settings, sections under checksum computation (*-ck* parameter) and some special segments are added:

- *checksum -ik* (collects descriptors of all segments with CRC computation)
- *CLASS_B* (defines space for Class B variables under RAM run check)
- *STACK_BOTTOM* (space for 4 byte pattern used for stack space limit)

The user can modify these settings but this must be done carefully as some addresses are already defined for testing routines (See Class B configuration section). Changing some settings can influence test results.

When the version of STVD tool does not support including the vector table content into the checksum computation and the user wants to protect this part of the code, the proper linker script file must be modified by placing the vector table as follows:

The line: “+seg .const -b 0x8000 -k”

must be replaced by: “+seg .const -b 0x8000 -k -ck”.

Note: Previously, “__ckdend__” symbol definition needed to be to included into linker script file for some older versions of Cosmic compiler. There is no longer any need to use it.

2.2.3 Application demonstration example

A short example of user application is attached in each project ***main.c*** file with respect to the package integration criteria (see *Chapter 3: Class B solution structure*). Except for STM8L_10x, the examples were written to run on the corresponding STM8 evaluation boards (STM8S/128-EVAL and STM8L1526-EVAL) as Class B package demonstration

firmware. They use on board LCD and LED diodes to display current versus initial master clock frequency measurement changes. Display or LED outputs can be disabled in ***main.h*** file. The main loop also performs initialization and calling all run mode tests at ordinary intervals.

2.3 Package configuring and debugging

A functional part of the package may need to be changed, suspended, excluded or included. For example the microcontroller type might need to be changed or there could be a non-volatile memory space limitation. Modifications are also required to debug the user application. This section describes how the ST solution can be configured, modified and debugged.

2.3.1 Configuration control

Software configuration is done at two levels. The first one is automatic and consists of configuring the project for a given microcontroller. This is done by selecting the corresponding device project. The second one is done through user configuration. All user defined configuration settings are collected in the Class B configuration file ***stm8x_stl_param.h***. The user must be very careful when modifying this file. Most of the package functional blocks are under conditional compilation controlled by a predefined set of constants in this file.

It is possible to disable some part(s) of the library by putting definitions in comments. This is useful e.g. disable Flash CRC when break points are inserted to debug the code. The full Class B package, even optimized and with disabled verbose messages, takes about 3.5KB of code memory. Disabling some library parts could be required for microcontrollers where the memory space is too small for application needs or when the tested parts are not included in the application (e.g. HSE testing).

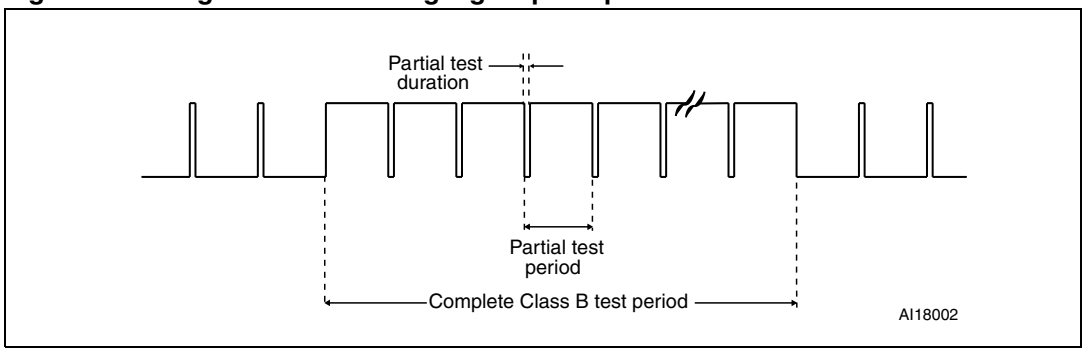
2.3.2 Verbose diagnostic mode

STM8S UART1 (UART2 for STM8105, UART for STM8L15x) interface is used to output text messages in verbose mode. This mode is useful in debug phase as the interface can be connected to an external terminal (the line setting is 115200 Bd, no parity, 8 bit data, 1 stop bit). However the text messages consume too much code space in this mode. Different levels of verbose mode can be enabled or disabled by the user through definition of constants in the Class B configuration file ***stm8x_stl_param.h***. For example, verbose mode could be limited to startup, run mode or fail case only.

LEDs toggling is another way to verify a Class B event. First LED toggles at each begin and end of run mode checks and with every successful finish of program memory test. Next LED toggles with each begin and end of RAM partial check called on at each tick interrupt and with every successful finish of the RAM memory test. LEDs slow toggling signals the main processes and quick pulses modulated on the slow signal correspond to the length of partial services, in other words, the loading time (see *Figure 3*). LED control can be disabled by user in Class B configuration file ***stm8x_stl_param.h***.

Note: Verbose mode via UART line is not used at STM8L10x package. Error codes are passed to FailSafe() routine instead.

Figure 3. Diagnostic LED timing signal principle



2.3.3 Debugging the package

While debugging the package, it is useful to disable:

- reset in ***FailSafe()*** routine by servicing independent WDG,
- all program memory CRC checksum tests when using breaks in the code to prevent program memory checksum error occurrence
- window WDG to prevent improper service out of the time slot window dedicated to refresh[(a)].
- control flow monitoring (mostly done automatically by changing values of constants when some tests are omitted)

At the debugging phase it may be useful to enable:

- verbose diagnostic mode to watch messages at UART terminal
- LEDs toggling to see basic process flow

a.Window WDG feature is not available at STM8L10x devices

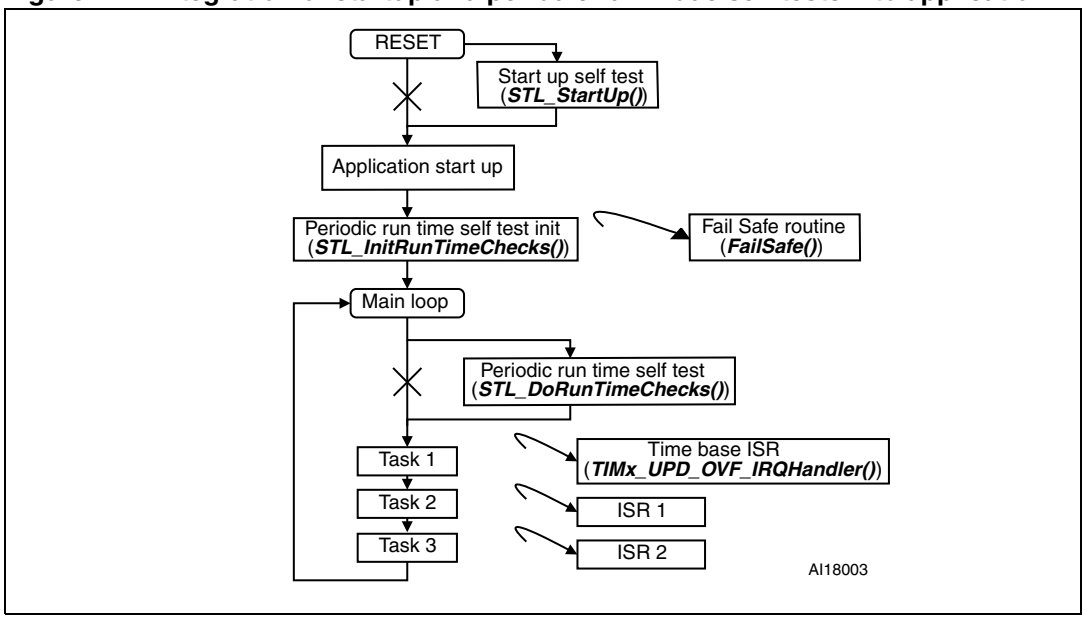
3 Class B solution structure

3.1 Integrating software into user application

Class B routines are divided into two main processes: periodic run mode self tests and start-up tests. The periodic run mode test must be initialized by the set-up block before it is applied. All the blocks are checked by sufficient flow control checked at a number of flow check points (see *Section 2.1.3: Class B flow control*). All class B variables are kept redundantly in a pair of control registers stored in the Class B variable space defined by user (see *Section 2.1.2: Class B variables*). This variable space is split into two separate RAM regions which are permanently undergoing the transparent test as a part of run mode tests.

Figure 4: Integration of startup and periodic run mode self tests into application shows the basic principle of how to integrate the Class B software package into user software. The reset vector should be forced by the user to ***STL_StartUp()*** routine which collects all system startup self tests. If they pass successfully, then the standard ***_stext()*** start-up routine is performed. While the application is running, periodic tests must be executed at regular intervals. To ensure this, the user must initialize these tests by calling the initialization routine ***STL_InitRunTimeChecks()*** before entering main loop and then inserting a periodical call of ***STL_DoRunTimeChecks()*** at main level. For best results, this should be inside the main loop. TIM4 (or TIM6 for some devices), which is configured during initialization routine to generate periodic system interrupts, provides the time base for all the tests. Short partial transparent RAM March C- or March X check is performed at each interrupt tick. If any self test fails, ***FailSafe()*** routine is called.

Figure 4. Integration of startup and periodic run mode self tests into application



3.2 Detailed description of startup self tests

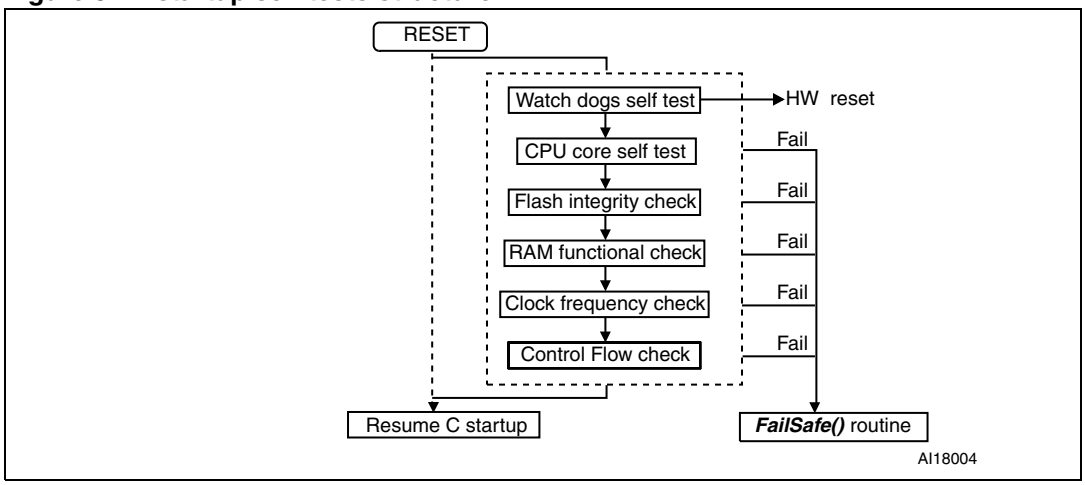
The startup self test is forced during initialization phase as the earliest checking process after resetting the microcontroller (see *Figure 4*) and before standard application startup

routine. The user must force the reset vector to the first address in ***STL_StartUp()*** routine. The startup tests block structure is shown in *Figure 5* and includes the following self tests:

- Watch dogs startup test
- CPU startup test
- Flash complete checksum test
- Full RAM March C-/X test
- Clock startup test
- Control flow checks

These blocks are described in more details in the next chapters. User can control including them as usual in the configuration file ***stm8_stl_param.h***.

Figure 5. startup self tests structure

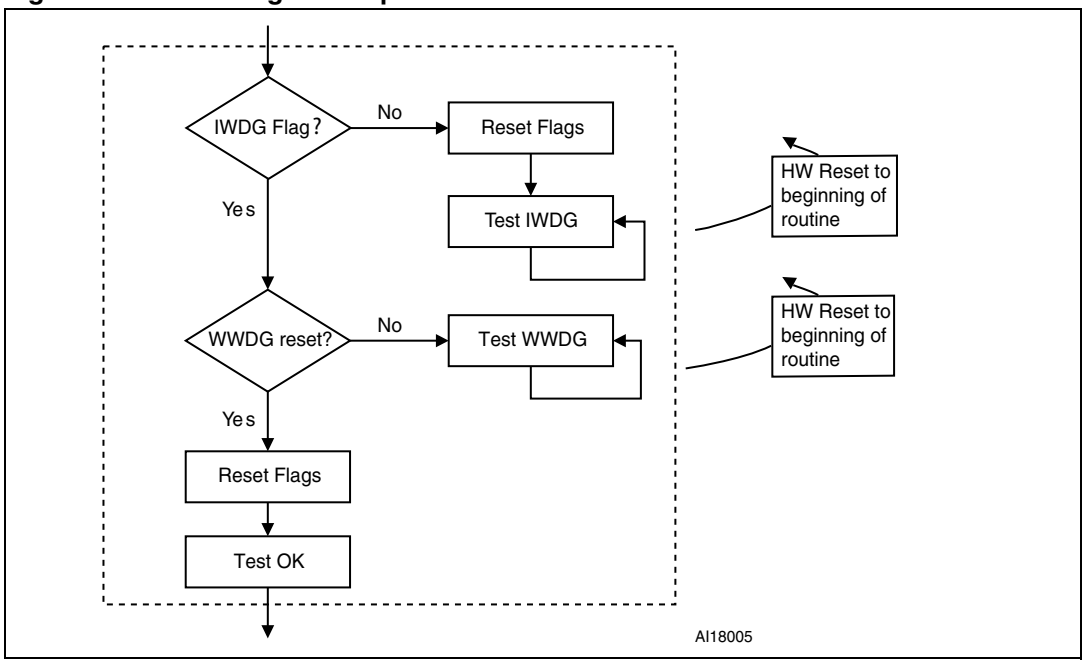


3.2.1 Watchdog startup self test

The first startup self test is to enter WDG self test routine. The program passes through this routine three times. First the routine checks the reset source in reset status register. If Independent watchdog (IWDG) is not flagged in reset status register, then reset sources is cleared and IWDG test begins. The IWDG is set to the shortest period and the microcontroller is reset by hardware and IWDG flag is set (see *Figure 6*). The routine comes back to the beginning and check WDG flags. When IWDG is recognized, it looks for WWDG flag. If WWDG is not flagged, the test continues a second time with window watchdog (WWDG) test. Again the microcontroller is reset by hardware and WWDG flag is set. When both WDG flags are set in the reset status register, the test is assumed as finished and both flags are cleared. WWDG is not present at STM8L10x devices, so WWDG test is not present.

User must carefully set both IWDG and WWDG periods. Time periods and window refresh parameters must be set according to the time base interval because refresh is done at the successful end of the periodical run mode test in main loop.

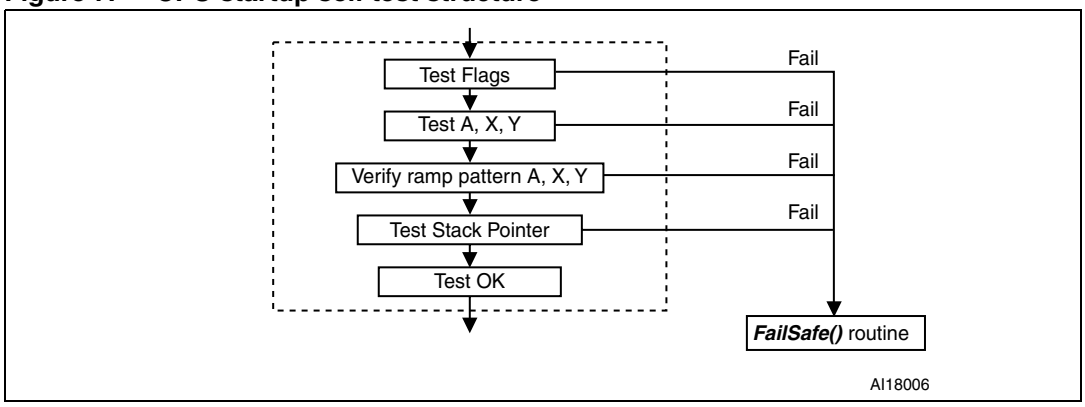
Figure 6. Watchdogs startup self test structure



3.2.2 CPU startup self test

Core flags, registers and stack pointer are tested for functionality. In case of any error, ***FailSafe()*** routine is called.

Figure 7. CPU startup self test structure



3.2.3 Flash complete checksum self test

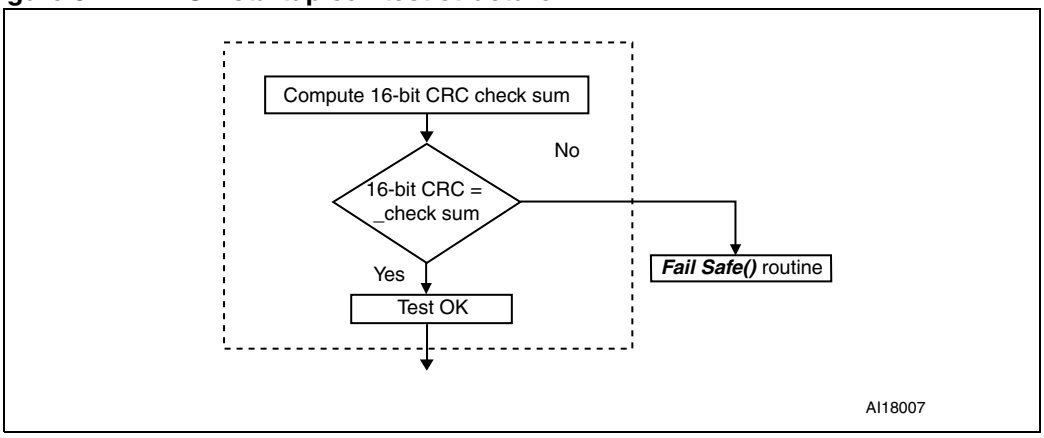
The CRC checksum computation is performed over the entire Flash memory space announced by linker checksum structure. The resulting computation is compared with the linker result. ***FailSafe()*** routine is called if there is an error.

The user actually has a choice of three different methods for calculating the CRC (see *Figure 8*) over the code memory content in the project.

- An 8-bit check sum calculation over the 16-bit address space can be used for checking up to 64kB of memory code, the simplest method.
- A 16-bit check sum calculation over the 16-bit address space can be used for checking up to 64kB of the memory code; this method is more precise and is the default method.
- a 16-bit check sum calculation over all the possible address space can be used when the code memory exceed 64kB, 24-bit addressing must be used; this way uses the most code space and is the most time consuming.

The STM8 FW includes six separate sources files defined for each of these methods. There is one pair for each method: one used for startup and one for run time. The user should include the correct source file pair into the project.

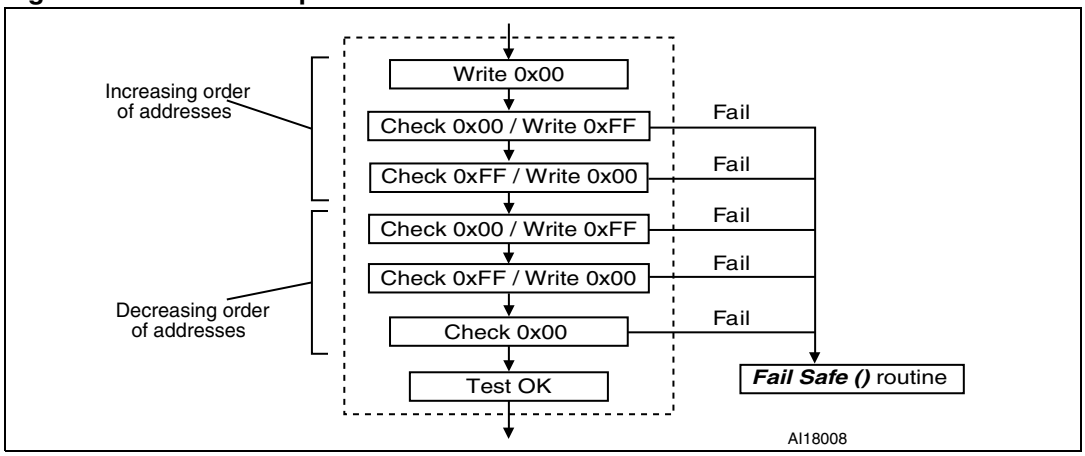
Figure 8. FLASH startup self test structure



3.2.4 Full RAM March C-/X self test

The whole RAM space is alternately filled and checked simultaneously by zero and 0xFF pattern in six loops, either by March C- or March X algorithms. March X algorithm is faster as the two middle steps are skipped over. The first three loops are performed in incremental order of addresses the last three in decremented order. In case of error, ***FailSafe()*** routine is called.

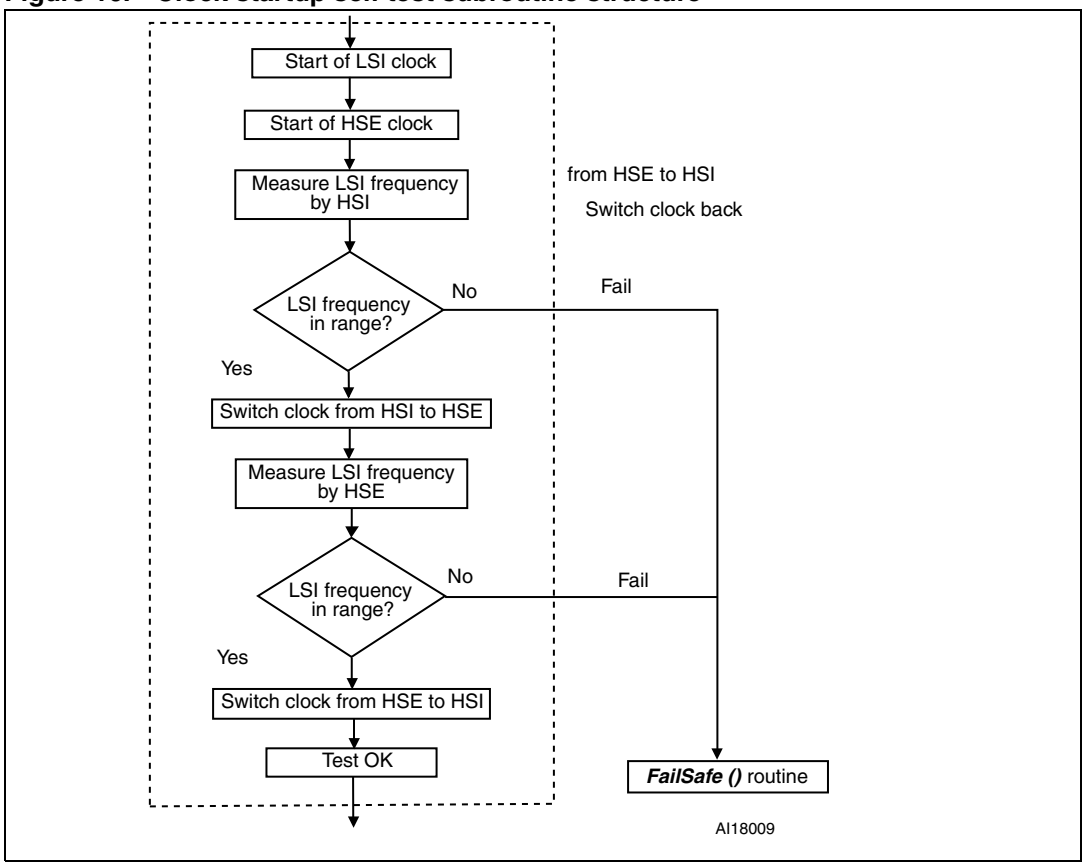
Figure 9. RAM startup self test structure



3.2.5 Clock startup self test

When the test begins, first the low speed internal clock (LSI) source is started, then the high speed external source (HSE). The CPU is running on the high speed internal source (HSI). HSI is switched onto a dedicated timer input, mainly TIM3, but TIM1 or TIM2 can also be used on some devices. The timer is gated by the LSI source. The number of gated ticks is compared and if it falls outside of the predefined interval (more than +/- 25% from nominal value) an HSI fail is signaled and ***FailSafe()*** routine is called. CPU clock continues with HSI, default source. If there is no error, the test continues with the next step and HSE is checked. HSE is switched on as the new CPU clock source and input into the dedicated timer. The same measurement and check of gated ticks are repeated but with HSE feeding the timer. If the measure falls out of the interval, CPU clock is immediately turned back to HSI and HSE fail is signaled with a ***FailSafe()*** routine being called. Otherwise, the test returns OK. CPU clock is switched back to default HSI source after the test is finished. All the parts of this test are under conditional compilation control, so the user can skip over some test e.g. HSE test when no external oscillator is used.

Figure 10. Clock startup self test subroutine structure



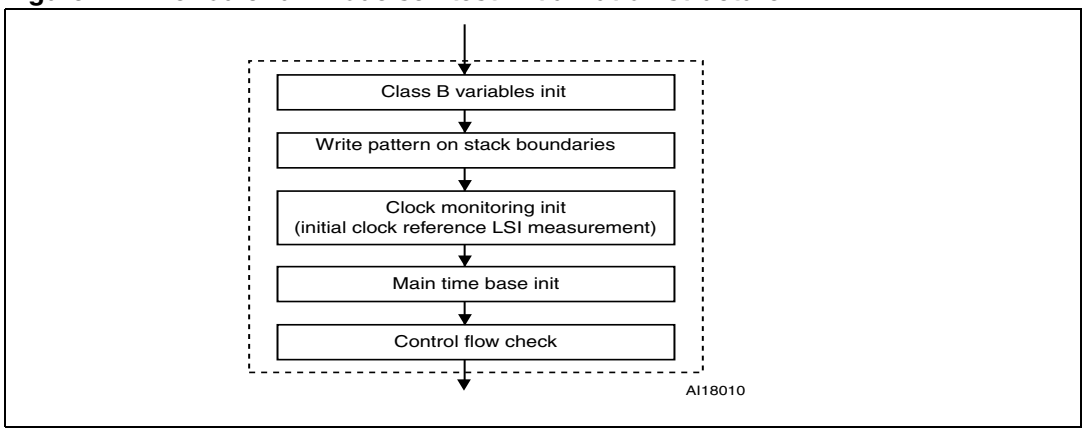
1. Low speed external (LSE) clock source can be started/checked at the beginning of the test and measured feeding the dedicated timer on STM8L15x devices. They can be inserted into the beginning of this routine.
2. High speed external (HSE) tests are skipped on STM8L10x devices.
3. LSI frequency is different for STM8S and STM8L devices. That is why the four consequent LSI periods are gated at STM8S devices (LSI=128 kHz) and only one period is used for measure on STM8L devices (LSI=38 kHz).

3.3 Periodic run mode self tests initialization

Run time self tests must be initialized just before the program enters the main loop performing the run mode self tests (refer to *Figure 4*). This must come just after start-up self tests have been executed and standard initialization done. The timing should be set to ensure that run mode tests are called properly and at regular intervals.

All class B variables must first be initialized. Zero and its complement value are stored in every class B variable complementary pair. The magic pattern is than stored at the top of stack space. Timer peripherals are configured for the tick interval measurement and master clock frequency measurement. Master frequency is gated by the LSI clock. The same method as for startup test is used. The resulting number of pulses is stored into the class B variable pair as an initial reference sample of master frequency measure.

Figure 11. Periodic run mode self test initialization structure



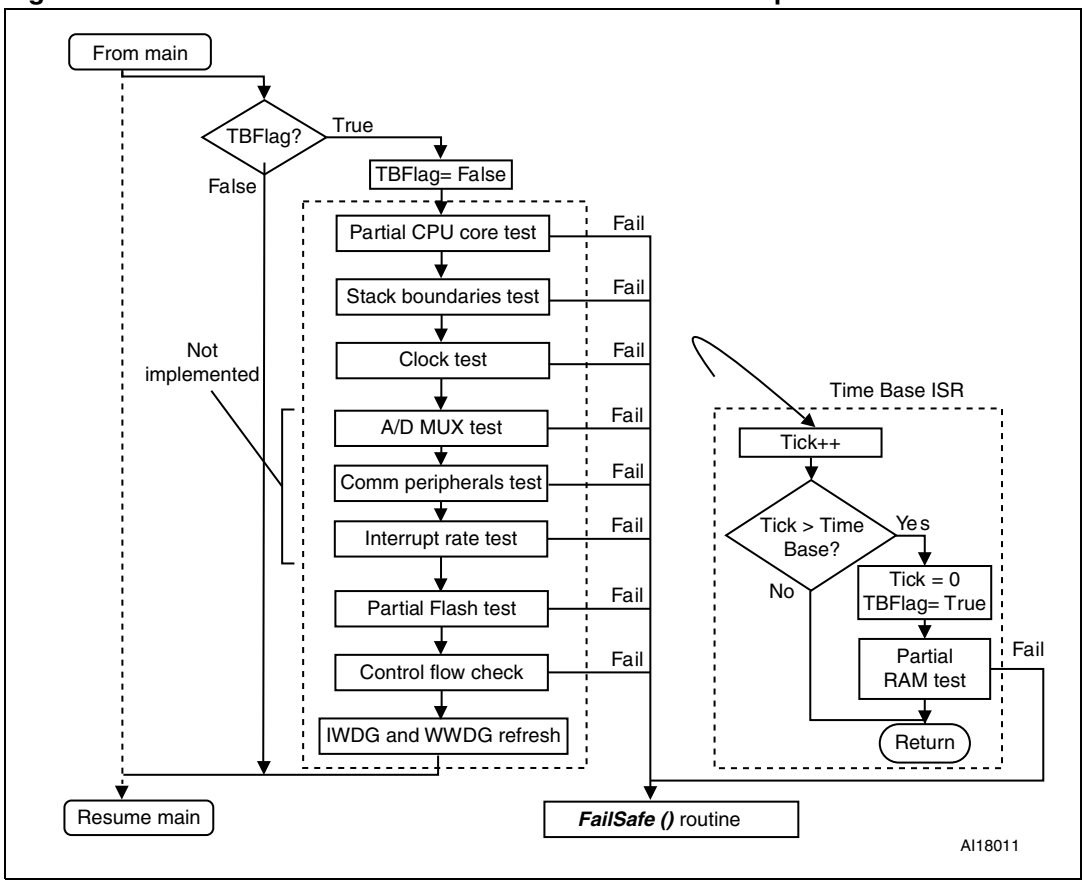
3.4 Detailed description of periodic run mode self tests

3.4.1 Run time self tests structure

Run time self tests are performed periodically, their period is based on time base interrupt settings. Before the first run, all tests must be initialized by run mode initialization routine (refer to *Figure 4*). All tests are performed in the main loop level except partial transparent RAM test, which is executed during the time base interrupt service. Some tests (analog, communication peripherals and application interrupts) are not automatically included. Depending on device capability and application needs, the user could implement them. The following is the list of the run mode self tests:

- CPU core partial run mode test
- Stack boundaries overflow test
- Clock run mode test
- AD MUX self test (not implemented)
- Interrupt rate test (not implemented)
- communication peripherals test (not implemented)
- Flash partial CRC test including evaluation of the complete test
- IWDG and WWDG refresh
- Partial transparent RAM March C-/X test (under system interrupt scope)

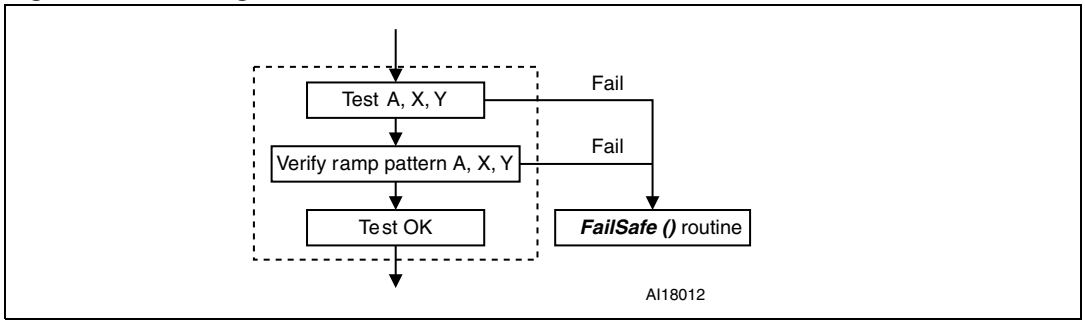
Figure 12. Periodic run mode self test and time base interrupt service structure



3.4.2 CPU light run mode self test

CPU core run mode self test is a simplified startup test where the flags and stack pointer are not tested. In case of error, ***FailSafe()*** routine is called.

Figure 13. CPU light run mode self test structure

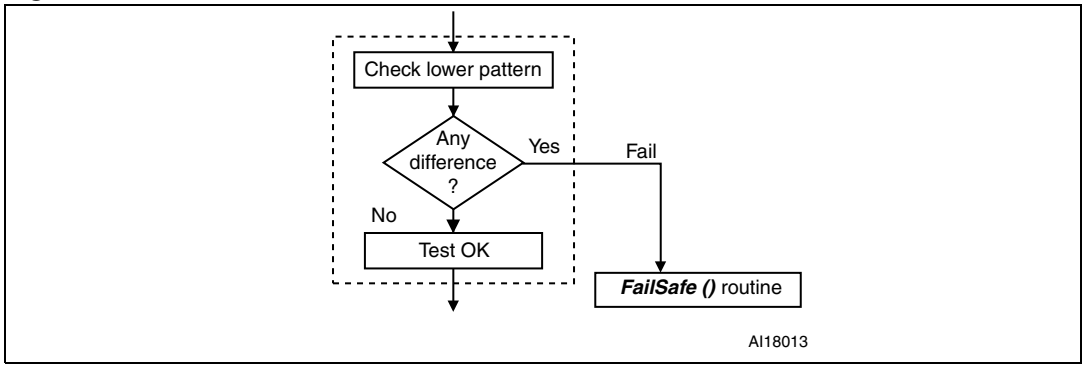


3.4.3 Stack boundaries run mode test

The magic pattern stored at the top of the stack is checked here. In case the original pattern is corrupted, ***FailSafe()*** routine is called. The pattern is placed at the lowest address dedicated for stack area.Overflow and underflow are detected as the stack pointer rolls over

inside this range in both cases. This area differs among the devices. The user must respect the dedicated stack area when the pattern location is changed.

Figure 14. Stack overflow run mode test structure



3.4.4 Clock run mode self test

The current master clock frequency selected by user application is gated by LSI internal clock source. The resulting number of pulses is compared with the initial master frequency reference sample measure which was stored during initial run mode self tests. When there is more than +/-25% difference found, ***FailSafe()*** routine is called. Occasional over captured measure is ignored. It can appear when a longer user interrupt service corrupts current measurement.

Figure 15. Clock run mode self test structure

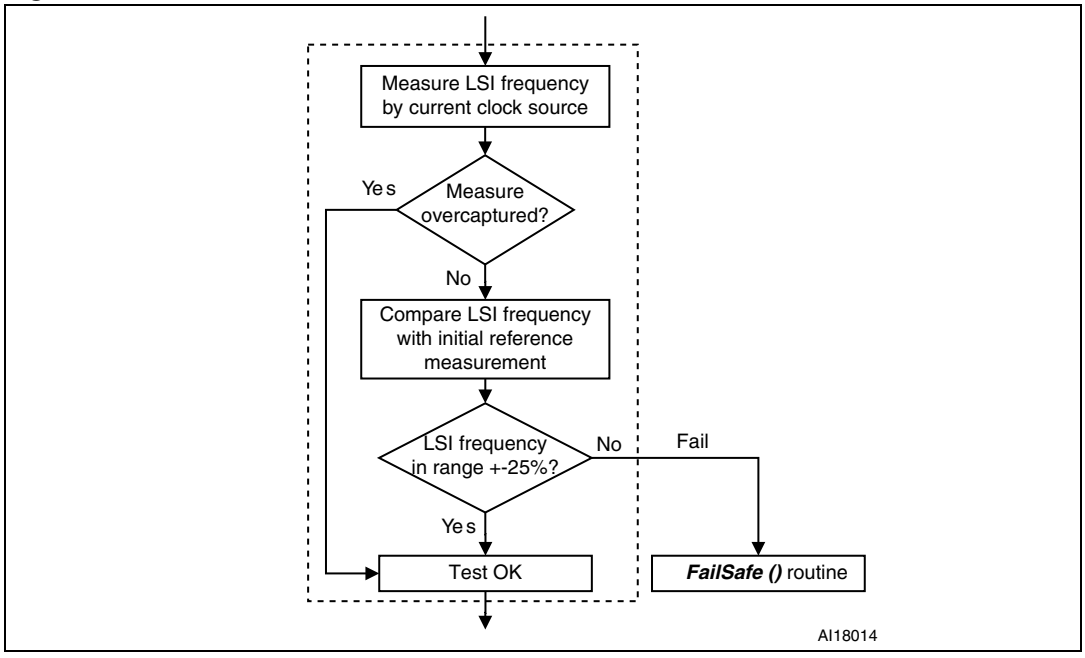
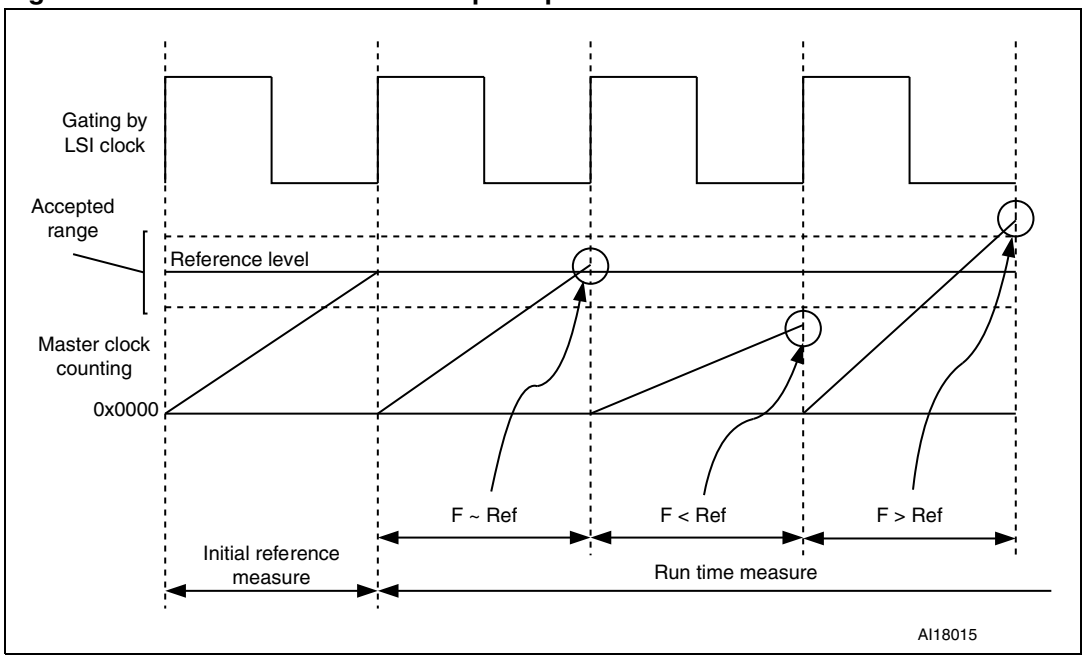


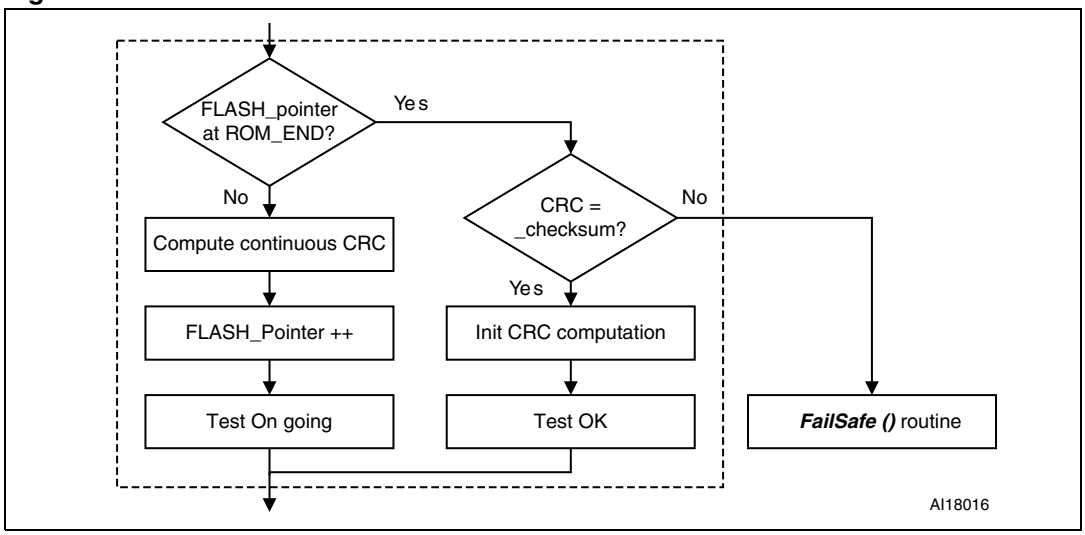
Figure 16. Clock run mode self test principle



3.4.5 Partial FLASH CRC run mode self test

Partial 16-bit CRC checksum over the Flash memory block is performed in every step. The boundaries are given in the segment table created by the linker. When the last block is reached the CRC checksum is compared with the value stored by linker at the last record in the segment table. In case of a difference, ***FailSafe()*** routine is called, otherwise a new computation cycle is initialized.

Figure 17. Partial FLASH CRC run mode self test structure



1. *For more details about CRC calculation, please refer to Section 3.2.3: Flash complete checksum self test.*

3.4.6 Watchdog service in run mode test

If the run mode service block is correctly completed, then both window and independent WDGs are refreshed at the last step just before returning to the main loop. To correctly refresh WDGs, the user must ensure calling ***STL_DoRunTimeChecks()*** routine (see *Figure 12*) at corresponding intervals in order to be able to properly react to the time base flag change.

Only one WDG refresh inside the main loop is necessary and contributes to overall efficiency. There should be no other WDG refresh except the one in ***STL_DoRunTimeChecks()*** routine. Sometimes it is necessary to refresh WDGs at initialization phase, too. In this instance, the refresh should be outside of any software infinite loop.

3.4.7 Partial RAM run mode self test

Partial transparent RAM test is performed inside the time base interrupt service. The test covers the part of the RAM containing the class B variables. One block of six bytes is tested at every test step. To guarantee coupling fault coverage, consecutive test steps are performed on memory blocks with an overlapping of two adjacent bytes. During the first phase, the block content is stored in the storage buffer. The next phase is to perform the marching destructive tests on all the bytes in the tested RAM block. Then, the final step is to restore the original content. March X algorithm is faster as two middle marching steps are skipped over (see *Table 3*). The last test sequence step is to perform a marching test on the storage buffer itself, again with the next two additional bytes to cover coupling faults. Then the whole test is re-initialized and it begins again. In case any fault is detected, ***FailSafe()*** routine is called.

Figure 18. Partial RAM run mode self test structure

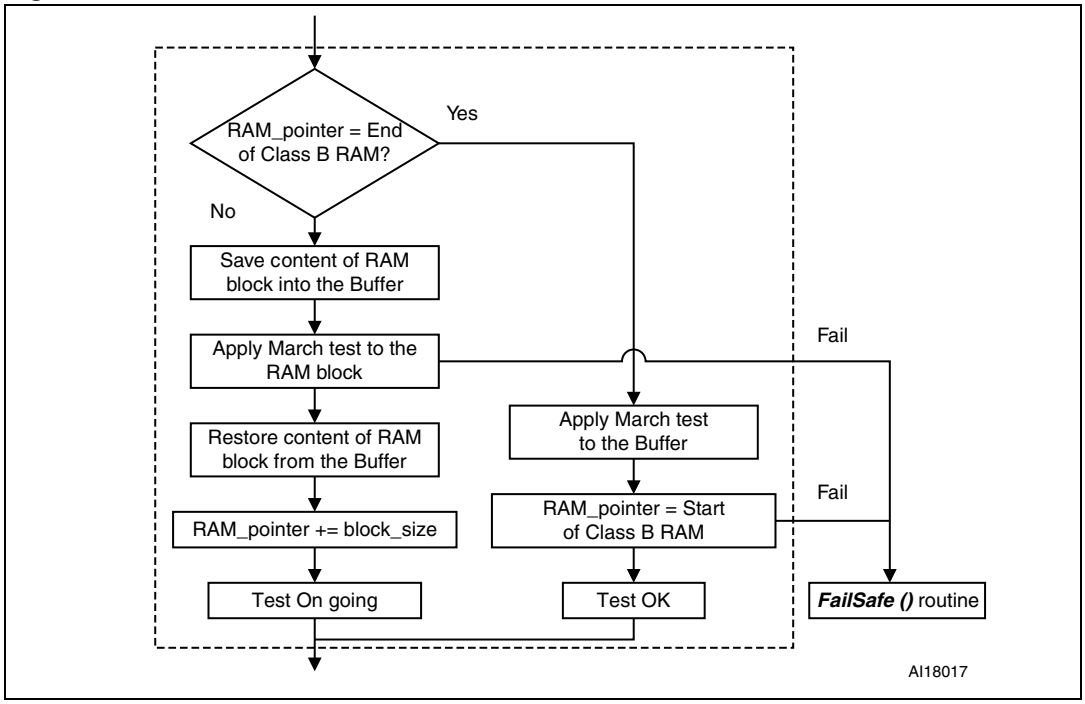


Figure 19. Fault coupling principle used at partial RAM run mode self test

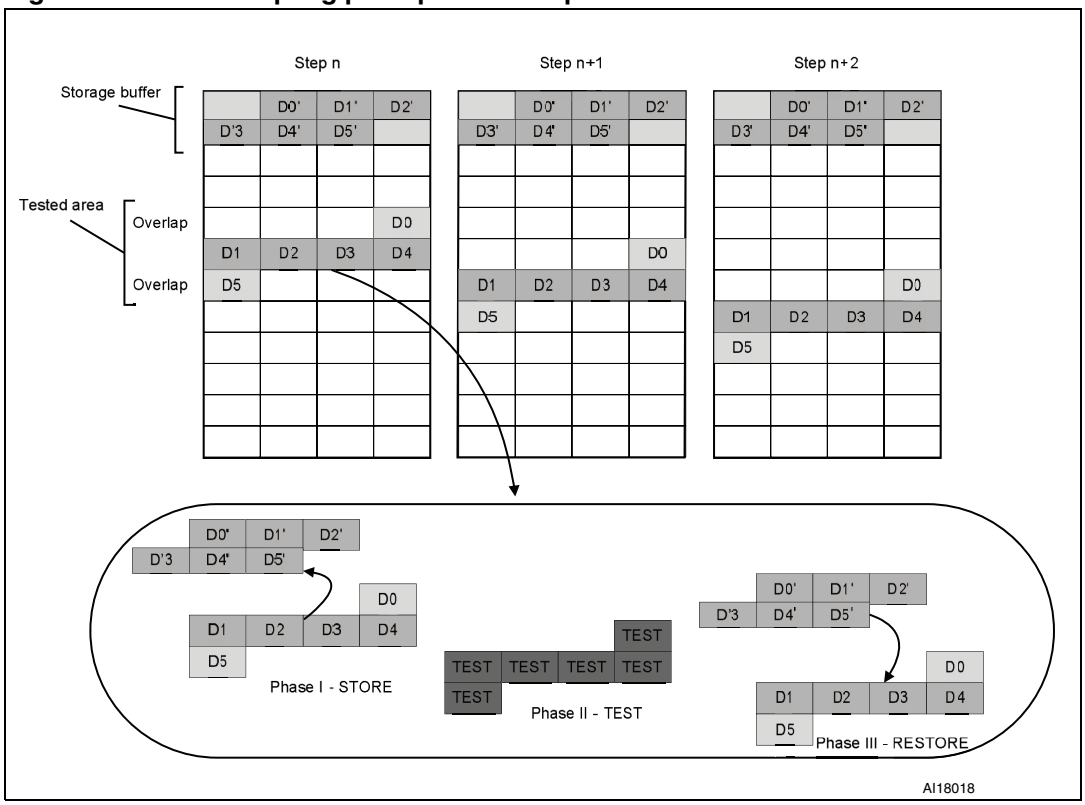


Table 3. March C- phases at RAM partial test

March phase	Partial bytes test over the block	Address order
Initial	Write 0x00 pattern	Increasing
1	Test 0x00 pattern, write 0xFF pattern	Increasing
2(1)	Test 0xFF pattern, write 0x00 pattern	Increasing
3(1)	Test 0x00 pattern, write 0xFF pattern	Decreasing
4	Test 0xFF pattern, write 0x00 pattern	Decreasing
5	Test 0x00 pattern	Decreasing

1. Steps 2 and 3 are skipped over when March X algorithm is used.

Appendix A Main differences of STM8S and STM8L packages

Table 4. Main differences of STM8 Class B optimized software packages

	STM8Sxxx	STM8L15x	STM8L10x
optional UART verbose mode	YES	YES	NO(1)
Demo mode with optional LCD screen	YES	YES	NO
WWDOG test	YES	YES	NO
High speed external HSE clock test	YES	YES	NO
Low speed external LSE clock test	NO	YES	NO
LSI frequency / number of LSI periods used at clock tests (LSI measurement)	128 kHz / 4	38 kHz / 1	38 kHz / 1
Stack space at the top of RAM [bytes]	1024 / 512(2)	513	513

1. Errors are manifested through to ***FailSafe()*** routine, see *Section 2.3.2: Verbose diagnostic mode*.
2. For STM8S Access Line devices (STM8S10x, STM8S903).

Revision history

Table 5. Revision history

Date	Revision	Description of changes
01-Jun-2010	1	Initial release