life.augmented

AN3307
Application note

Guidelines for obtaining IEC 60335 Class B certification for any STM32 application

Introduction

The role of safety has become very important for electronics applications. The level of safety requirements for components used in electronic designs is steadily increasing. The manufacturers of electronic devices include many new technical solutions in the design of new components. Software techniques for improving safety are continuously being developed.

The current safety recommendations and requirements are specified at worldwide level by recognized international standards bodies such as IEC (International Electrotechnical Commission) and come under the compliance, verification and certification process of testing houses and authorities like VDE (Association for Electrical, Electronic and Information Technologies). The certification process is closely associated with the ElectroMagnetic Compatibility (EMC) tests when the robustness of the system against the noise emission and the noise sensitivity is tested for the compliance with international standards.

The main purpose of this application note and its associated software[a] (STM32-CLASSB-SPL) is to facilitate and accelerate the user software development and the certification processes for appliances which are subject to these requirements and certifications and are based on some of the ST 32-bit family of microcontrollers.

There are three packages certified by VDE for the 32-bit ST microcontrollers covering the STM32F0xx, STM32F1xx and STM32F3xx microcontroller families, referred to in this document by STM32F. All the certified packages are based on dedicated CMSIS and Standard peripheral libraries and all of them use similar principles described by this document, with focus on the main differences. While using the hardware and firmware compatibility between ST families, these packages can also be adapted for some other ST 32-bit microcontrollers not yet certified (for example, STM32F2xx, STM32F4xx, STM32Lxx or STM32Wxx families)[b].

---

a. All the Class B firmware packages are available upon request only, contact the nearest ST sales office.

b. STMicroelectronics is developing derivative firmware supporting new products step by step. Contact the local ST sales office to obtain the latest information about available packages and plans for their certification.

Contents

1 Package variation overview 6

2 Main differences between STM32F packages and their possible modifications 7

3 Compliance with IEC and VDE standards 11

3.1 Generic tests included in the STM32Fxxx firmware library 12

3.2 Application-specific tests not included in the ST firmware library 14

4 Class B software package 15

4.1 Basic software principles used 15

4.1.1 Fail Safe mode 15

4.1.2 Class B variables 15

4.1.3 Class B flow control 16

4.2 Package organization 17

4.2.1 Projects included in the package 17

4.2.2 Tool-specific integration of the library 18

4.2.3 Application demonstration example 18

4.3 Package configuration and debugging 19

4.3.1 Configuration control 19

4.3.2 Verbose diagnostic mode 20

4.3.3 Debugging the package 21

5 Class B solution structure 22

5.1 Integrating the software into the user application 22

5.2 Description of startup self tests 23

5.2.1 CPU startup self-test 24

5.2.2 Watch dog startup self-test 25

5.2.3 Flash complete checksum self-test 26

5.2.4 Full RAM March C-/X self-test 26

5.2.5 Clock startup self-test 28

5.3 Periodic runtime self-test initialization 30

5.4 Description of periodic runtime self-tests 31

5.4.1 Runtime self-test structure 31

5.4.2 CPU light runtime self-test 32
5.4.3 Stack boundaries runtime test 32
5.4.4 Clock runtime self-test 33
5.4.5 Partial Flash CRC runtime self-test 34
5.4.6 Watchdog service in runtime test 34
5.4.7 Partial RAM runtime self-test 35

6 Revision history 37

List of tables

Table 1. MCU parts which must be tested for Class B compliance 12
Table 2. Overview of methods used in the micro-specific tests provided with this application note . 13
Table 3. Physical order of RAM addresses organized into blocks of 16 words 26
Table 4. March C- phases in RAM partial test 36
Table 5. Revision history 37

List of figures

Figure 1. Example of RAM memory configuration 16
Figure 2. Control flow four-step checking principle 17
Figure 3. Diagnostic LED timing signal principle 19
Figure 4. STM32 demo hyper terminal output window in verbose mode 21
Figure 5. Integration of startup and periodic runtime self-tests into the application 22
Figure 6. Startup self test structure 23
Figure 7. CPU startup self-test structure 24
Figure 8. Watchdog startup self-test structure 25
Figure 9. Flash memory startup self-test structure 26
Figure 10. RAM startup self-test structure 27
Figure 11. Scramble test - detailed structure of a single check and fill procedure 27
Figure 12. Clock startup self-test subroutine structure 28
Figure 13. Clock frequency measurement principle 29
Figure 14. Periodic runtime self-test initialization structure 30
Figure 15. Periodic runtime self-test and timebase interrupt service structure 31
Figure 16. CPU light runtime self-test structure 32
Figure 17. Stack overflow runtime test structure 32
Figure 18. Clock runtime self-test structure 33
Figure 19. Partial Flash CRC runtime self-test structure 34
Figure 20. Partial RAM runtime self-test structure 35
Figure 21. Fault coupling principle used at partial RAM runtime self-test 36

1 Package variation overview

The STM32F0xx package is based on the Cortex Microcontroller Software Interface Standard (CMSIS) and the STM32F0xx Standard peripheral library V1.0.0 published by ST. It supports the STM32F050xx microcontrollers where the Flash memory density ranges between 32 and 64 Kbytes.

The STM32F10x package is based on the CMSIS and the STM32F10x Standard peripheral library V3.5.0 published by ST. Available configurations support the following devices in the STM32F10x family in range of this standard peripheral library revision:

- Low, medium and high density **Performance line** devices
- Low, medium and high density **Access line** devices
- Low, medium and high density **Value Line** devices

The STM32F3xx package is based on the CMSIS, STM32F30x and STM32F37x standard peripheral libraries V0.1.0 published by ST. It supports the STM32F30x and STM32F37x microcontrollers where the Flash memory density ranges between 64 and 256 Kbytes.

These associated firmware packages are available upon request only. Please contact the local ST sales office for more information.

Two projects have been prepared and tested for each package under IAR/EWARM version 6.40 and KEIL®/MDK-ARM™ version 4.50 environment and toolchains.

For more EMC information please refer to the following related application notes:

- *Software techniques for improving microcontroller EMC performance* application note (AN1015)
- *EMC design guide for ST microcontrollers* application note (AN1709)

For more detailed information about the cyclic redundancy check calculation (CRC) please refer to the following related application note:

- *Using the CRC peripheral in STM32 family* application note (AN4187)

2 Main differences between STM32F packages and their possible modifications

This document describes mainly the basic STM32F1xx package; as the other packages are its derivatives. The user can find some slight differences mainly due to the hardware deviations, compilers and debugging tools incompatibilities. Main differences are focused on in this section. The user can find a complete list and detailed information about *STL* files included in documentation **.chm* file, which is extracted and attached for each package.

CPU Tests

The register R13 is renamed to MSP (Main stack pointer) at assembly source files of the STM32F3xx package. With the STM32F0xx package, all the instructions in the tests loading immediate 32-bit constant operands are replaced by instructions loading constants placed at code memory, due to restricted instruction set of ARM® Cortex®-M0 core.

Clock tests and time base interval measurement

Instead of Real-time Clock (RTC) counter used in the STM32F1xx package, a sub-second register is used at STM32F0xx and STM32F3xx packages for clock measurement due to different implementation of RTC registers. The clock system measurement technique differs slightly due to this fact, in these packages. Internal timers are used for cross-check frequency measurement when this feature is available mainly to determine harmonic or sub-harmonic frequencies if the system clock is provided by the external crystal.

SRAM tests

Some ST microcontrollers feature a built-in word protection with single bit redundancy (hardware parity check) applied on CCM RAM or on part of SRAM, at least. This hardware protection is one of the acceptable methods required by the IEC 60335 and IEC 60730 Class B standards. It reduces software testing overhead significantly. The applied software SRAM March test can be removed from both startup and runtime test sets in this case. It can be used optionally for testing the parts of RAM which are not covered by the hardware parity check, or as a supplementary testing method. The diagnostic coverage of single bit redundancy protection is limited to detect specific range of errors; so additional checks can be required by the certification authority at some safety critical cases. An example could be a combination of hardware parity protection and software March test which can be an acceptable test of variable memory satisfying even more severe Class C criteria. The reliability of hardware parity check can be increased, too, by other suitable methods like using double inverse storage of safety critical information at physically separated areas (e.g. to prevent a corruption caused by radiation or EMI which typically attacks a limited physical memory locality).

The hardware parity check is an optional feature. When it is enabled, an unintentional parity error event can be detected during the execution of startup self-test procedure due to not initialized parity system (e.g. case of local variables if they are allocated and read out while using different data access to memory).

To prevent this, it is suggested to initialize the concerned RAM space before calling startup self-test procedure. A simple loop inserted into startup assembly code can solve the problem and initialize parity system at dedicated RAM area:

```
; Program starts here after reset
;-----------------------------------------------------------------------
Reset_Handler
; Parity system initialization has to be performed here prior to the
;    startup self-test procedure
;-----------------------------------------------------------------------
; r0 is used as a pointer to RAM,
; r1 keeps end address of the area
;-----------------------------------------------------------------------
;At every step of the loop, the 32-byte block (r2-r9) is copied to RAM
;    starting from address kept at r0, r0 is then increased by 32
; the loop body is performed while r0<r1
                  LDR                R0, =RAM_block_begin
                  ADD                R1, R0, #RAM_block_size
RAM_init_loop
                  STMIA              R0!, {R2-R9}
                  CMP                R0, R1
                  BLT                RAM_init_loop
; RAM is initialized now, program can continue by startup self-test
                  LDR                R0, =STL_StartUp
                  BLX                R0
```

Note: *The real content of the registers copied by STMIA instruction does not care because the purpose of this loop is to initialize the parity system. The content of the RAM will be initialized later by the compiler standard startup procedure anyway. The RAM_block_begin and RAM_block_size constants' setting should correspond with the number of data copied by STMIA instruction to prevent any undefined memory access.*

If the initial software March test is performed over a RAM area dedicated for stack, it destroys the stack content where the return address of the test routine is stored. That is why the return address must be saved before the test starts and restored back to the stack after the test finishes. The return address is kept at dedicated register while the test is running. The store and restore handling depends on compiler implementation and it can differ for different level of optimization settings. In case of any such change the user has to check, how the return address is saved into stack and perform adequate store and restore of its value correctly back before return from test is performed and the address is popped out (follow intrinsic operations at begin and end of `STL_FullRamMarchC()` routine at *stm32fxxx_STLFullRamMc.c* file).

Flash memory tests

While the IAR compiler can be set to include a 32-bit CRC result compatible with STMicroelectronics hardware directly into code (see IAR documentation), the CRC calculation is not fully supported by KEIL compiler for the STM32 microcontrollers yet. That is why the result provided by STM32's internal CRC generator cannot be used directly for

such invariable memory checking at KEIL project. The users have to store the checksum computation result into final source code manually by modifying and proper placement of CHECKSUM area at the end of Flash memory (see end of *startup_stm32fxxx.s* file and linker file settings). The result of a CRC calculation can be obtained by an external tool or it can be read at debug phase from begin of RAM (see invariable memory check part in *stm32fxxx_STLstartup.c* file) where the result is stored temporarily by initial test. The CHECKSUM area itself must be excluded from the range of CRC calculation. Other possible testing method could be to compute a 32-bit checksum by the CRC generator in the initial phase and validate its result by using 16-bit software CRC computation in parallel. Specific SW procedures for 16-bit CRC calculation must be provided and called for proper testing at this case during startup tests. The validated 32-bit CRC value can be then saved as a reference for comparing all the subsequent runtime checks. This method decreases the CPU load at run time phase significantly, as the CRC computation is done via a hardware CRC generator exclusively. Please follow the latest updates from providers of compilers to implement the latest and most sufficient methods.

The Flash memory can be tested while the DMA is used for feeding the CRC block, too. In comparison with software test, the CPU load decreases significantly when the DMA is applied but the speed of the test itself does not change so much as the DMA needs few cycles to read and transfer data anyway. The test even can slow down when the DMA services some other transfers in parallel. Moreover some additional DMA configuration is required at initialization of the test. The user can find some detailed information about using the DMA at CRC calculation at the dedicated application note, *AN4187: Using the CRC peripheral in STM32 family*.

Startup and system initialization

There are differences at the initial system configuration and setup of debug and diagnostic utilities (e.g. recognizing reset cause) respecting hardware deviations. The startup file is modified to force the very first flow to begin of startup test set.

Parameters

All the configuration parameters are centralized into one file *stm32fxxx_STLparam.h*. The configuration differences respect mainly different sizes of tested areas, different compilers, slight deviations of control flow, etc. Be careful when modifying an initial or run time test of possible corruption of control flow. In this case, the values summarized at complementary control flow counters can differ from the constants defined for comparison at flow check points. To prevent any control flow error, the user must change the definition of these constants in an adequate way.

Files structure

The structure of directories of all the packages is nearly the same for all the packages. It follows common structure used for ST standard peripheral drivers with added extra directory for self-test files. There is an additional subdirectory at Libraries directory at STM32F3xx package where two different directories are applied for STM32F30x and STM32F37x standard peripheral drivers, respecting both subfamilies differences. The dedicated group of standard drivers is included in dependency on project setting. The number of included self-test library header files is reduced at STM32F0xx package, as most of headers associated with their source files stay simply empty and they follow a formal structure only. The function of *stm32xxx_STLlib.h* header including all these associated headers is replaced by common header *stm32f0xx_STL_param.h* here, which is included by all the sources in this package instead of *stm32xxx_STLlib.h,* as it is used in the other packages.

Verbose and diagnostic modes

The verbose messages can slightly vary between the packages but they are mostly self-explanatory. The users can easily find the calling place and condition at the source code from which the message is published. Signals provided on GPIO LEDs can be used for basic check if the application runs properly, as well as for the measurement of some basic timing intervals. Be sure the verbose messages increase the processing load at execution phase and significantly increase an overhead of code memory capacity, especially.

Duration of the tests

The duration of the initial functional self-test performed at startup phase can vary slightly depending on CPU clock frequency and size of memory areas tested. The differences between duration of periodical run time self-tests are negligible and depend mainly on the CPU clock. Both software RAM transparent and partial Flash tests last about 20 µs with a default block size setting (at CPU clock about 72 MHz).

The overall test RAM or Flash cycle duration depends on:

- The size of the memory area under check,
- The size of the block checked at one step and
- The frequency of the partial test steps.

For instance, a test of the RAM area collecting 32 pairs of word Class B variables needs 16 partial steps if 4 words are tested by one step (in fact, 6 words are tested at every step if just one word overlap is configured at both the highest and the lowest address - follow the transparent RAM test configuration).

the test of 64 Kbytes of Flash memory can be done at 1024 partial steps when the block size is set to default 16 words. This means an overall duration of 10 s if the period of steps is 10 ms.

At any critical case when a such long interval for Flash testing is not acceptable, the program memory can be divided into few sectors and the test can be reduced to be performed above one selected sector just keeping the safety functions and data currently used.

The toggling of some LEDs can be used to measure the timing performance of the firmware. Be sure any control of the verbose messages used at debugging is disabled as well as the initialization of evaluation board display when doing any such measurement.

3 Compliance with IEC and VDE standards

The IEC (International Electrotechnical Commission) is a non-profit and non-governmental authority recognized worldwide for preparing and publishing international standards for a vast range of electrical, electronic and related technologies. IEC standards are focused mainly on safety and performance, the environment, electrical energy efficiency and its renewable capabilities. The IEC cooperates closely with ISO (International Organization for Standardization) and ITU (International Telecommunication Union). Their standards define not only the recommendations for hardware, but also for software solutions. Standards are divided into a number of safety classes depending on the purpose of the application.

The other bodies that are recognized worldwide in the field of electronic standard organizations are VDE in Germany, IET in the United Kingdom and the IEEE in the United States. The VDE association 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 standard compliance and quality testing services to manufacturers of electrical appliances.

One of the pivotal IEC standards is the IEC 60335-1 norm, which covers safety and security of household electronic appliances destined for domestic and similar environment. Appliances incorporating electronic circuits are subject to component failure tests.

The basic principle is that the appliance must remain safe in the case of any component failure. The microcontroller is an electronic component just like any other from this point of view. If safety relies on an electronic component, it must remain safe after two consecutive faults. This means the appliance must stay safe with one hardware failure and with the microcontroller not operating (under reset or not operating properly).

If the safety depends on software, the software is taken into account with the second applied failure. The conditions required for software are defined precisely in Annex Q of the IEC 60335-1 norm. Three classes of appliances are defined here:

- **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 the associated ST software package covers the group B specification. Appliances under group C need some other special requirements such as dual microcontroller operation, which is outside the scope of this document.

Class B compliance aspects for microcontrollers are related both to hardware and software. A list of microcontroller parts under compliance is evaluated at IEC 60335-1 Annex T which refers to IEC 60730 Annex H. This list can be divided into two groups - micro-specific and application-specific items. See *Table 1*.

While application-specific parts rely on customer application structure and must be defined and developed by the users (communication, I/O control, interrupts, analog inputs and outputs), micro-specific parts are related purely to the micro structure and can be made generic (core self-diagnostic, volatile and non-volatile memory integrity checking, clock system tests). This group of micro-specific tests is the focus of the ST solution based on powerful hardware features of STM32 MCU such as dual independent watchdogs or clock sources.

Table 1. MCU parts which must be tested for Class B compliance

Group	Components to be tested
Micro-specific	CPU registers
	CPU program counter
	Clock
	Volatile & non-volatile memories
	Internal addressing (and external memory addressing if any)
	Internal data path
Application-specific	Interrupt handling
	External communication
	Timing
	I/O peripherals
	Analog A/D and D/A
	Analog multiplexer

3.1 Generic tests included in the STM32Fxxx firmware library

The certified by VDE STM32Fxxx firmware library packages are composed of the following micro-specific software modules:

- CPU register 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 *Table 2* and they are described in more detail in the following sections. The last two items from the list above are not explicitly asked for by the norm, but they improve overall fault coverage.

Table 2. Overview of methods used in the micro-specific tests provided with this application note

Components to be verified	Method used
CPU registers	Functional test of all registers and flags including R13 (stack pointer), R14 (link register) and PSP (Process stack pointer) is done at startup. In the runtime test R13, R14, PSP and flags are not tested. The stack pointer is tested for underflow and overflow, the link register is tested by PC monitoring. If any error is found, the software jumps directly to the Fail Safe routine.
Program counter	Two different watchdogs driven by two independent clock sources can reset the device when the program counter is lost. The Window watchdog, driven by the main oscillator, performs time slot monitoring while the Independent one, driven by the low speed internal RC oscillator, is impossible to disable once enabled. Both watchdogs must be serviced at regular intervals. Program control flow is additionally monitored using a specific software method.
Addressing and data path	This is tested indirectly by RAM functional and Flash integrity tests, stack overflow (a specific pattern is written at a low boundary of stack space and checked for corruption at regular intervals) and underflow (a second pattern is written at a high boundary if it is not at the RAM end). In addition, a bus error exception vector is fetched by the CPU if a memory access fault occurs.
Clock	A reciprocal method of comparing two independent clock sources is used while clocking two timers. Wrong main frequency (harmonic/sub harmonic) can be detected using the external low speed 32 kHz oscillator as a timebase.
Non-volatile memory	A 16-bit CRC software checksum test of the entire memory is done at startup and a partial memory test is repeated at runtime (block by block). Optionally the built in hardware for fast 32-bit CRC calculation can be used.
Volatile memory	A March C- (or optionally March X) full memory test is done at startup and a partial memory test is repeated at runtime (block by block), scrambled order of physical addresses in RAM is respected in tests for optimal coverage of coupling faults, word protection with double redundancy (inverse values stored in non adjacent memory space) is used for safety critical Class B variables, Class A variable space, stack and unused space are not tested at runtime.

The users can include a part or all of these software modules certified by VDE into the project. If they stay unchanged and are integrated in accordance with the guidelines, the time and costs needed for a certified end-application would be significantly reduced.

Note: If minor changes have to be made to the modules, it is suggested that the user keeps careful track of all of these changes in order to inform the certification authority of all modifications made to the certified routines.

3.2 Application-specific tests not included in the ST firmware library

The users have to focus on all the remaining tests that cover application-specific MCU parts that are not included in the ST firmware library:

- Testing the analog blocks (AD/DA, multiplexer)
- Testing the digital I/O
- External Addressing
- External communication
- Timing and interrupts

The tests for the analog part depend on the application and peripheral capability of the device. The pins used should be checked for correct analog intervals (by checking that the measured voltages correspond to the real analog values) and free analog pins can be used to read some reference voltages in conjunction with testing the analog multiplexers if they are used in the application. The internal reference voltage should also be checked. Some STM32 devices feature two (and some even three) independent ADC blocks. It make sense to perform conversions on the same channel using two different ADC blocks for security reasons.

Class B tests must also detect any malfunction of the digital I/Os. This could be covered by plausibility checks together with some other application parts (for example, the change in an analog signal from a temperature sensor should be checked when the heating/cooling digital control is switched on/off). Selected port bits can be locked by applying the correct lock sequence to the lock bit in the GPIOx_LCKR register to prevent unexpected changes to the port configuration. The reconfiguration is only possible at the next reset sequence in this case. In addition, the bit banding feature can be used for atomic manipulation of the SRAM and peripheral registers.

Application interrupt occurrences and external communication flows should also be checked. Different methods can be used: one method can use a set of incremental counters with specific counters incremented by each interrupt or communication event. The values in the counters are then checked periodically at given time intervals to make a cross-check with an independent timebase. The number of events performed within the last period should correspond to the application requirements. The configuration lock feature can be used to secure the timer register settings with three levels controlled by the TIMx_BDTR register.

The data exchange at communication sessions should be checked while including a redundant information into the data packets. Parity, sync signals, CRC check sums, block repetition or protocol numbering can be used for this purpose. Robust application software protocol stacks like TCP/IP give higher level of protection, if necessary.

4 Class B software package

This section highlights the basic common principles used in the ST software solution. The workspace organization is described as well as configuration and debugging capabilities. Any differences between the two supported development environments (IAR's EWARM and Keil's MDK-ARM) are highlighted.

4.1 Basic software principles used

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

4.1.1 Fail Safe mode

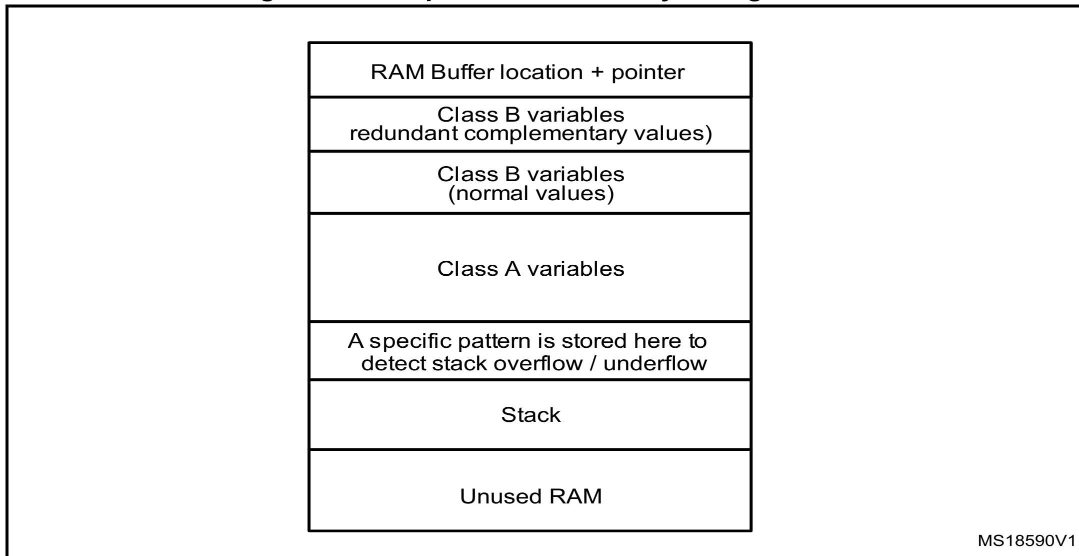
If any failure is detected, the ***FailSafePOR()*** routine is called (defined in ***stm32fxxx_STLstartup.c*** file). The program stays in an endless loop waiting for a watchdog reset. Apart from some debugging features the routine is almost empty. The users have to provide the content to the routine and ensure that it executes the right actions to keep the application in a safe state.

The debug or verbose mode described in *Section 4.3: Package configuration and debugging* can be used to identify which error has occurred.

4.1.2 Class B variables

Each class B variable is stored as a pair of two complementary values in two separate RAM regions. Both normal and redundant complementary values are always placed in non adjacent memory locations. A partial transparent RAM March C- or March X test is performed continuously on these RAM areas by means of an interrupt subroutine. The pair is compared for integrity each time before the value is used. If any value stored in the pair is corrupted, Fail Safe mode is invoked. An example of RAM configuration can be seen in *Figure 1*. The users can adapt the RAM space allocation according to the application needs and the hardware capabilities of the device.

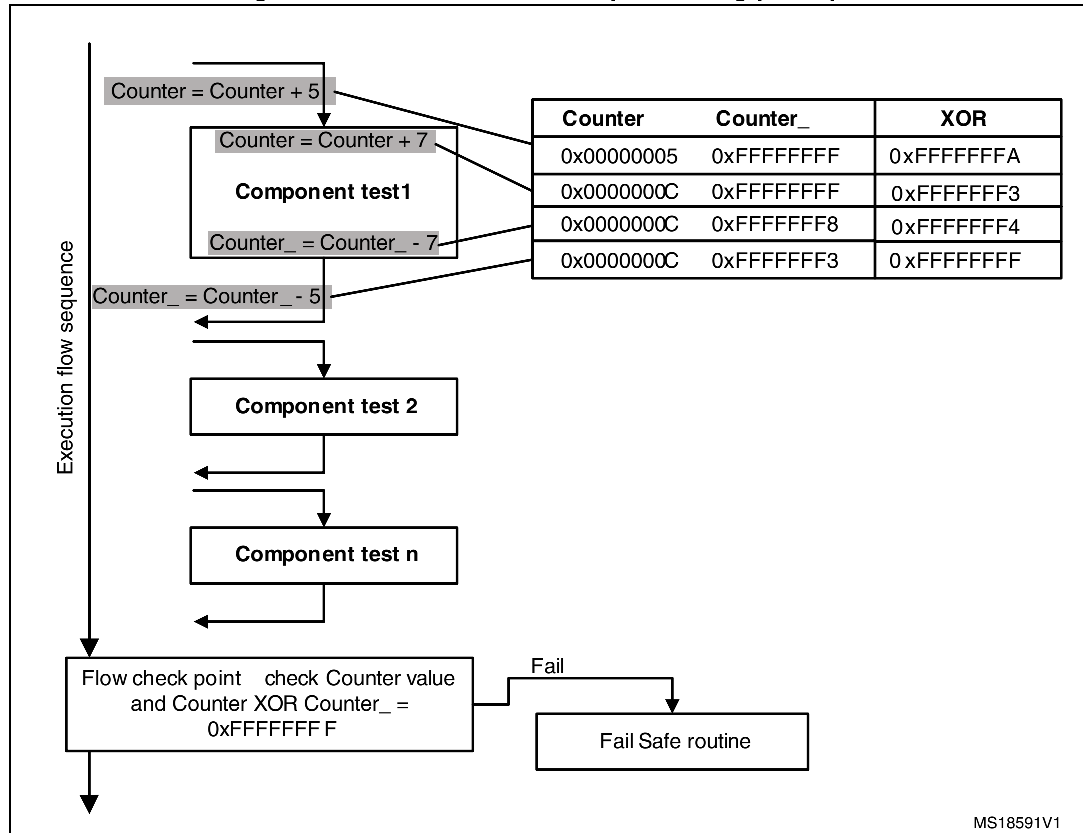
Figure 1. Example of RAM memory configuration



4.1.3 Class B flow control

A specific software method is used to check if all parts of the test flow are successfully passed. Unique labels (constant numbers) are defined for identifying all key points (blocks with component tests) in the code flow in order to make sure no block is skipped and all of them are performed as expected. The unique labels are processed in two complementary counters to comply with class B variable criteria. The main principle is a symmetrical four-step change of the counter pair content (adding or subtracting the unique label values) each time any significant testing block is processed. Two of the steps check if the block is correctly called from the 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 inside the called procedure (processed just after entry 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 checked flow sequence and the four-step checking service is shown. This method decreases the load on the CPU as all these points are always checked 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 values stored in the counter pair must be always complementary ones after any block is passed completely. Several execution flow check points are evaluated and placed in the code flow where the integrity of the counter pair is checked. If the counters are not complementary or they do not contain the expected values at any of these checkpoints, the Fail Safe routine is called.

Figure 2. Control flow four-step checking principle



Note: The unique number for the calling point for Component test 1 at the main level is defined as 5, and for the procedure itself it is defined as 7 in this example. The initial value of the counters are set to 0 and 0xFFFFFFFF for simplicity. The table in the upper right corner of Figure 2 shows how the counters are changed in four steps and their complementary state after the last step of the checking policy (return from procedure) is done.

4.2 Package organization

This section describes how the ST solution is organized.

4.2.1 Projects included in the package

As well as the standard firmware and self-test libraries, the installation package includes two projects for STM32 family - IAR's EWARM and Keil's MDK-ARM. The corresponding *Project.eww* or *Project.uvproj* project file must be configured for a specific STM32Fxxx family device before compilation.

The required linker files are predefined and the defined symbols for concrete configuration are already declared in the preprocessor sections. For information on certain exceptions and special cases, refer to *Section 4.2.2: Tool-specific integration of the library*.

4.2.2 Tool-specific integration of the library

Startup file

It is recommended to reuse the following compiler-specific files:

- *startup_STM32fxxx.s* file where regular reset handler should be replaced by STL_StartUp function address and checksum storage area should be defined (in case of MDK-ARM compiler)
- **.icf* (for IAR) and **.sct* (for KEIL) linker script files where all the memory regions including Class B specific ones are defined
- The workspace can also be used as an example (CRC generation is enabled in the linker/ processing tab of project options with IAR).
- Self-test startup routines do not alter or disable the compiler's standard C startup files, variables and stack/heap which are initialized in the usual way.

Defining new variables and memory sizes

A double storage in CLASS_B_RAM and CLASS_B_RAM_REV is necessary to ensure the redundancy of the safety critical data (Class B). All other variables defined without any particular attributes are considered as Class A variables and are not checked during the transparent RAM test.

Class A and Class B variable region sizes can be modified in the linker configuration file. New Class B variables must be declared in the ***stm32fxxx_STLclassBvar.h*** header file, with the following syntax for the IAR development environment:

```
__no_init EXTERN uint32_t MyClassBvar @ "CLASS_B_RAM";
__no_init EXTERN uint32_t MyClassBvarInv @ "CLASS_B_RAM_REV";
```

and with the following syntax for the KEIL development environment:

```
EXTERN uint32_t MyClassBvar __attribute__((section("CLASS_B_RAM"), zero_init));
EXTERN uint32_t MyClassBvarInv __attribute__((section("CLASS_B_RAM_REV"), zero_init));
```

Note: The start and end addresses of RAM/ROM regions are not exported when using the KEIL environment. These address modifications must be edited in the stm32fxxx_STLparam.h file, which contains addresses and constants required to do the RAM and ROM (Flash) tests.

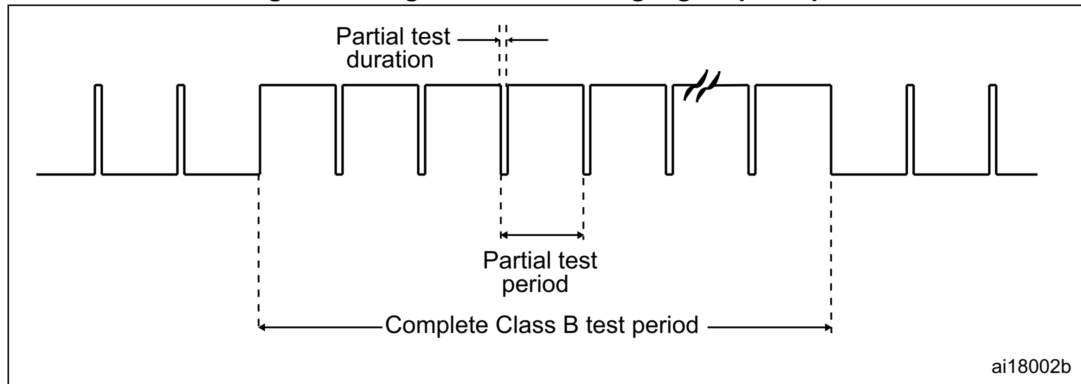
4.2.3 Application demonstration example

A short demonstration example of a user application is attached at *main.c* file related to the dedicated project (see Section 5: Class B solution structure). It provides an example of how the Class B routines can be integrated into an application-specific solution and demonstration software controlling hardware of dedicated evaluation board.

The demonstration software uses the following hardware:

- On board LCD screen to display basic status messages
- Outputs to drive LEDs indicating that testing routines are executed properly. Dedicated LEDs are toggled once the partial RAM or Flash memory test is completed and at every beginning and end of Class B testing procedures. These LED signals can be used to measure duration and frequency of the tests. Some of the LEDs indicate partial intervals of initial startup test, too. Please, follow the source code for more details.

Figure 3. Diagnostic LED timing signal principle



4.3 Package configuration and debugging

Sometimes a functional part of the package should be suspended, excluded or included. The reason could be that certain tests are not required for a given application context. Some modifications can also useful when the package has to be debugged. This section describes how the ST solution can be configured, modified and debugged.

4.3.1 Configuration control

The configuration of the software is done at two basic levels. One configuration level is given by the project settings, where the differences between the STM32 family members is the factor. This part is mainly done automatically by proper configuration of the project. The other configuration level is the user settings. All the user configuration settings are centralized in the Class B configuration file ***stm32fxxx_STLparam.h***. A set of constants defined in this file control the conditional compilation of some functions. Adjustable constants must have specific settings to run all the tests properly (see note at *Section 4.2.1: Projects included in the package*).

Some runtime tests can be skipped depending on the end application. If the periodicity of the test is inherent (for example, for a washing machine, the users switch the application on or off when it is used), then power on tests are sufficient and transparent/runtime tests can be avoided. This point must be discussed with the chosen test institute case by case.

For a maximum robustness, the recommendation is to enable the independent watchdog using the hardware option bytes, and start the window watchdog as soon as possible in the main routine once the application development is close to the end. This is not done by default in the STM32 self-test library demonstration.

The stack overflow detection and watchdog self-check are not mandatory according to the EN/IEC60335-1 standard. They can be disabled or skipped if necessary.

It can help decrease the CPU load during runtime if 32-bit CRC checks are made using the STM32’s internal CRC generator (32-bit wide CRC computation using the standard 0x04C11DB7 polynomial). Today the KEIL compiler does not support CRC format directly, but it can be used in conjunction with 16-bit CRC. One possible method could be to compute a 32-bit checksum in the initial phase and validate it using 16-bit CRC computation compared with the linker result. The validated 32-bit CRC value can be then saved as a reference for comparing all the subsequent runtime checks.

The CRC generation is not supported in the KEIL environment; so calling the CRC checking routines without a proper setting of CHECKSUM area at startup file will cause the application to reset continuously (as if a failure was detected).

Examples:

1. To disable the CRC check at startup, modify the output logic control used to evaluate the test result (assuming the computed CRC is different from REF_CRC16):
   In the *stm32fxxx_STLstartup.c* file modify:
   *if(STL_crc16(CRC_INIT,(u8 *)ROM_START, ROM_SIZE) != REF_CRC16)*
   into:
   *if(STL_crc16(CRC_INIT,(u8 *)ROM_START, ROM_SIZE) == REF_CRC16)*
2. To replace 16-CRC by 32-bit internal CRC computation, modify *stm32fxxx_STLmain.c* file:
   RomTest = STL_crc16Run();
   by
   RomTest = STL_crc32Run();
   Then comment the following #defines in file stm32f10x_STLparam.h starting from line 129:
   #define DELTA_MAIN ....
   #define LAST_DELTA_MAIN ....
   #define FULL_FLASH_CHECKED ...
   and uncomment the corresponding #defines for 32-bit CRC computation. In this case only the CRC32 algorithm control flow will be checked correctly.

4.3.2 Verbose diagnostic mode

The dedicated USART Tx serial peripheral line is used in verbose mode as a standard output for Class B status text messages. This mode is useful in the debug phase when the line can be monitored by an external terminal (the line setting is 115200 Bd, no parity, 8-bit data, 1 stop bit). The verbose mode is enabled by default and can be disabled during startup and/or runtime by commenting lines with the STL_VERBOSE_POR and STL_VERBOSE defined in the Class B configuration file ***stm32x_stl_param.h***. Each successful finishing of SRAM and Flash test at run time is signalled by printing a specific character '#' or '*' at the terminal window. The verbose messages can differ slightly between packages. *Figure 4* shows an example of verbose mode.

Figure 4. STM32 demo hyper terminal output window in verbose mode

```
Start-up CPU Test OK
Pin reset
... Power-on or software reset, testing IWDG ...

 *******  Self Test Library Init  *******
Start-up CPU Test OK
Pin reset
IWDG reset
... IWDG reset from test or application, testing WWDG

 *******  Self Test Library Init  *******
Start-up CPU Test OK
Pin reset
IWDG reset
WWDG reset
... WWDG reset, WDG test completed ...
Start-up FLASH 16-bit CRC OK
Control Flow Checkpoint 1 OK
 Full RAM Test OK
Ref 32-bit CRC OK
Clock frequency OK
Control Flow Checkpoint 2 OK
... main routine starts ...
RamPntr= 144
```

Connected 00:00:06 | ANSI | 115200 8-N-1 | SCROLL | CAPS | NUM | Capture | Print echo

4.3.3 Debugging the package

If any of the self-test routines fail, an MCU reset is triggered in the FailSafePOR function defined at *stm32xxx_STLstartup.c* file. This makes the debugging of the application difficult, and can cause the debugger to lose the execution context.

While debugging the package it is useful to disable:

- The call to the NVIC_GenerateSystemReset() macro in the FailSafePOR() routine to prevent losing execution context when resetting the micro
- The control flow monitoring when adding or removing self-test routines, in particular run-time self-diagnostics
- All the program memory CRC checksum tests when using software breaks in the code to prevent program memory check sum error occurrence
- The window watch dog to prevent improper service out of the time slot window dedicated for its refresh.

In the debugging phase it may be useful to enable the verbose diagnostic mode to watch Class B status text messages via the USART terminal.

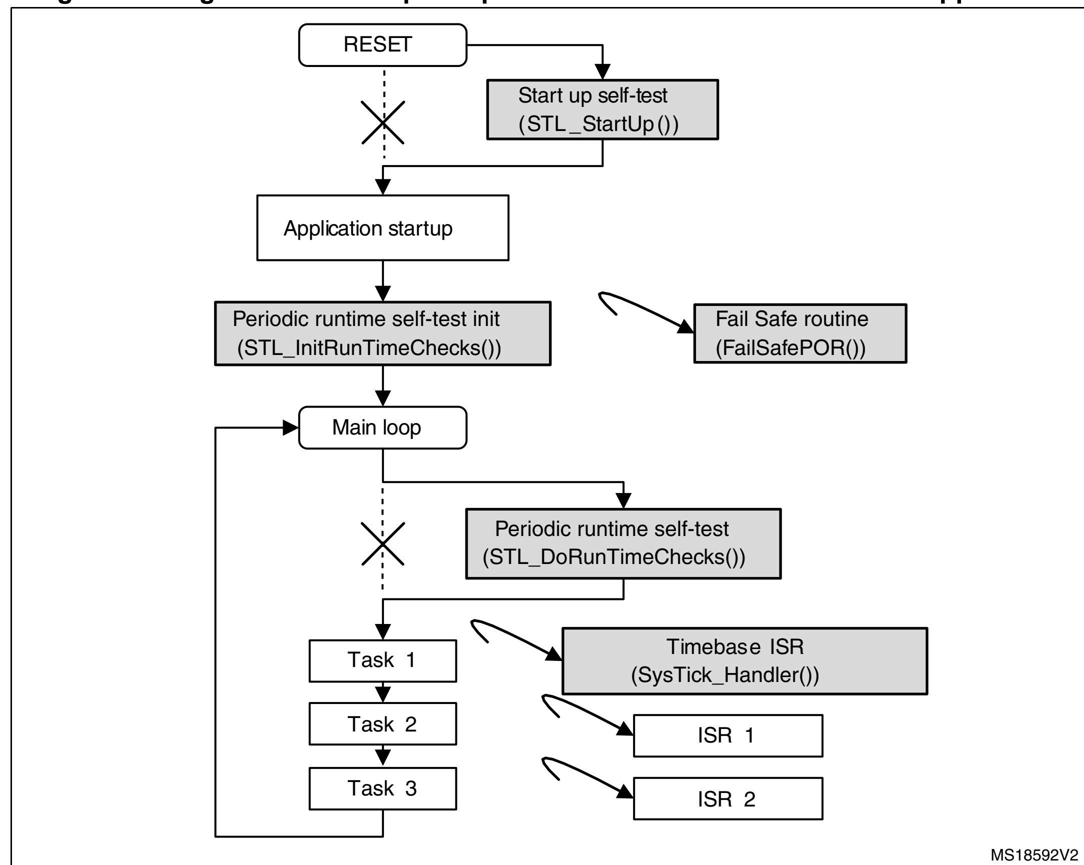
5 Class B solution structure

5.1 Integrating the software into the user application

The Class B routines are divided into two main processes: startup and periodic runtime self tests. The periodic runtime test must be initialized by a set-up block before it is applied. All the processes are covered by adequate caller-called controls checked at a number of flow check points. All the class B variables are kept redundantly in duplicate control registers stored in user-defined Class B variable space. This space is split into two separate RAM regions which are under permanent control of a transparent test which is part of the runtime tests.

Figure 5 shows the basic principle of how to integrate the Class B software package into a user software solution. The reset vector should be forced by the user to jump to the ***STL_StartUp()*** procedure which contains all the system self tests to be performed at startup. If all these tests pass sucessfully then the standard startup procedure ***__iar_program_start()*** for IAR or ***Reset_Handler()*** for the KEIL solution is performed.

Figure 5. Integration of startup and periodic runtime self-tests into the application



While the application is running, periodic tests are performed at certain intervals. To run them, the users must execute an initialization block by calling ***STL_InitRunTimeChecks()*** routine in the initialization phase and then insert a periodic call to ***STL_DoRunTimeChecks()*** at the main level, preferably in the main loop. Periodic system interrupts are configured by initializing the SysTick clock frequency and the main time base

measurement in the initial routine. The RTC is also reset and synchronized to allow measurement to be performed in the SysTick interrupt service routine. The ***SysTick_Handler()*** interrupt service counts ticks and performs short partial transparent RAM March C or March X checks at correct intervals (20 ms) when a flag (*TimeBaseFlag*) is set to synchronize the rest of the runtime checks called from the main level. The ***FailSafePOR()*** routine itself is described in *Section 4.1.1: Fail Safe mode*.

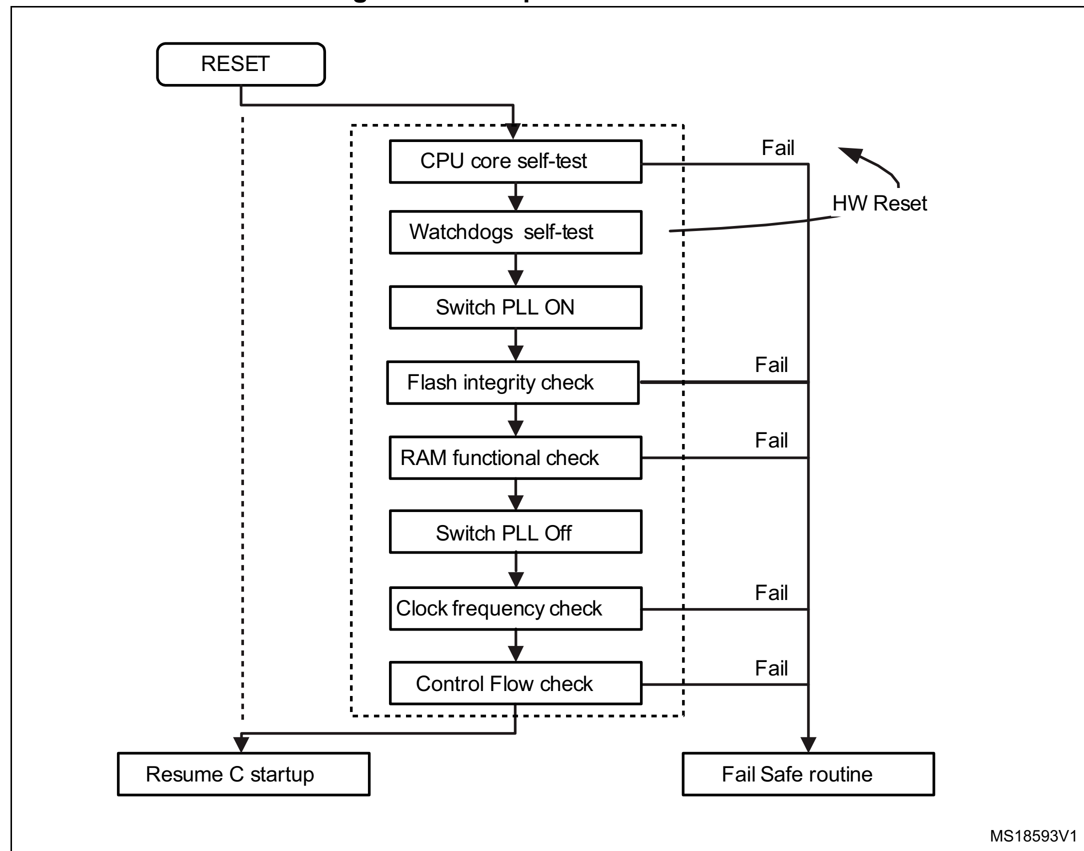
5.2 Description of startup self tests

The startup self-tests should be run during the initialization phase as the first check performed after resetting the micro (see *Figure 5*). This is done by forcing the reset vector to the address the start of the ***STL_StartUp()*** routine. The structure of the block of startup test structure is shown in *Figure 6* and it includes the following self tests:

- CPU startup test
- Watchdog startup tests
- Flash complete checksum test
- Full RAM March C/X test
- Clock startup test
- Control flow check

These blocks are described in more detail further in this section.

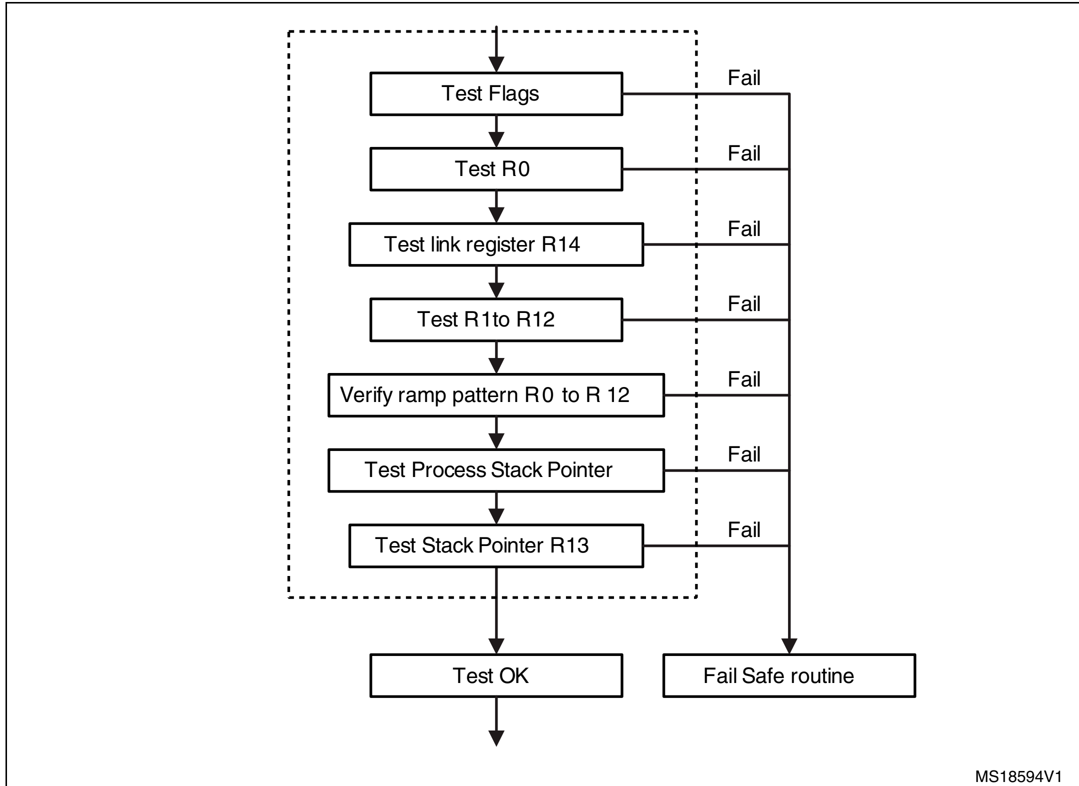
Figure 6. Startup self test structure



5.2.1 CPU startup self-test

The CPU startup self-test tests the core flags, registers and stack pointers for correct functionality. If any error is found, it jumps directly to the Fail Safe routine. The source files are written in assembly, and there are two different files for the IAR and KEIL development environments. The structure is given in *Figure 7*.

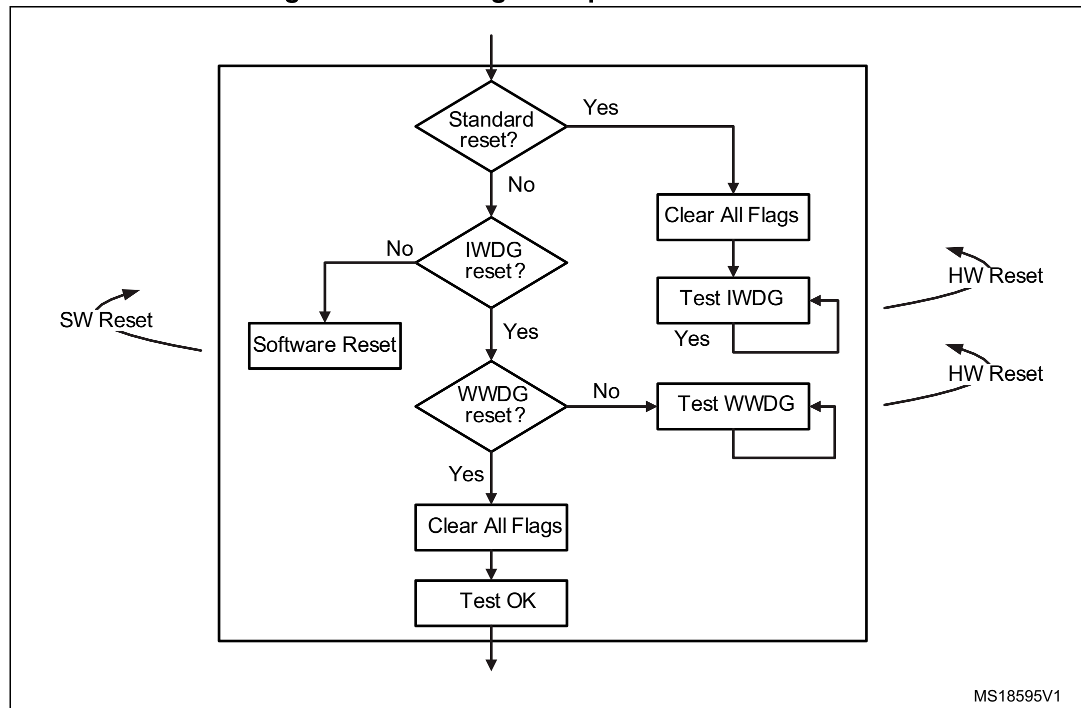
Figure 7. CPU startup self-test structure



5.2.2 Watch dog startup self-test

The structure of the test is based on the reset status register content which registers all the previous reset causes by setting different flags (see *Figure 8*).

Figure 8. Watchdog startup self-test structure



The standard reset condition (power on, low power, software or external pin flag indicates the previous reset cause) is assumed at the beginning of the watchdog test. All the flags are cleared, the IWDG timeout is set to the shortest period so an IWDG reset should then occur. After the next reset, the IWDG flag should be set and recognized as the sole reset cause. The test can then continue with the WWDG test. When both flags are set in reset status register, the test is considered as completed and all the flags in the reset status register are cleared again.

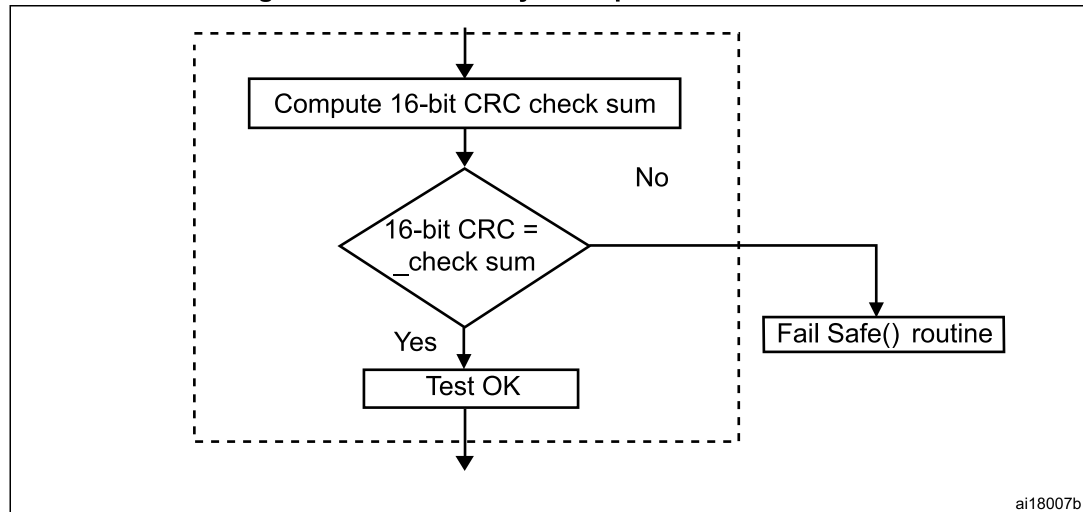
The users must take care to set both IWDG and WWDG periods properly. Their periods and the refresh window parameter must be set in accordance with the time base interval[a] because a normal refresh is performed at the successful end of the periodic runtime test in the main loop.

a. The system tick interrupt service routine indicates a defined time base interval via a dedicated time base flag. The run-time test is started at the main level periodically at this interval. As the watchdog control is the last step of successfully finished run-time tests (and it should be the only place where the watchdog is refreshed in the main loop) the time base interval must be set in correlation with the watchdog timeout and vice versa. The watchdog refresh period must be shorter than the watchdog timeout to prevent a reset of the CPU. Refer to the flowchart in *Figure 15*.

5.2.3 Flash complete checksum self-test

The CRC checksum computation is performed on the entire Flash space defined in the linker checksum structure. The result of the computation is compared with that of the linker. If they differ, the test fails. Refer to *Section 4.3.3: Debugging the package* for additional comments on CRC procedures.

Figure 9. Flash memory startup self-test structure



5.2.4 Full RAM March C-/X self-test

The entire RAM space is alternately checked and filled word by word with background patterns (value 0x00000000) and inverse background patterns (value 0xFFFFFFFF) in six loops as shown in *Figure 10*. The first three loops are performed in incremental order of addresses, and the last three in decremented reverse order. The order of tested addresses is scrambled as it respects the physical order of addresses to prevent and recognize any cross-talk between physically adjacent memory cells. The scramble principle is shown in *Table 3*. The basic physical unit is a pattern (a row) covering a block of 16 words. The numbers in the table cells represent logical addresses, while their order in the table represents the physical layout. Bold frames highlight the places where the logical order is scrambled. Any address scrambling is not present at new ST products. The March test of RAM is simplified significantly for them.

Some new products have implemented a hardware word protection with single bit redundancy (hardware parity check) applied on CCM RAM or on part of SRAM at least. The applied software SRAM March test can be removed completely or partially from both startup and runtime test sets for this case. The software test can be used only for testing of those parts of RAM which are not covered by the hardware parity check or as a supplementary testing method.

Table 3. Physical order of RAM addresses organized into blocks of 16 words

	Physical order of addresses---->															
Rows -->	0	1	***3***	***2***	4	5	7	6	8	9	11	10	12	13	***15***	***14***
	16	*17*	***19***	***18***	20	21	23	22	24	25	27	26	28	29	***31***	***30***
	32	*33*	***35***	***34***	36	37	39	38	40	-	-	-	-	-	-	-

The algorithm of the entire test and a detailed description one of its scramble loops are given in *Figure 10* and *Figure 11*. If an error is detected, the test is interrupted and a fail result is returned. The simplified March X algorithm can also be used. It is faster, as two middle marching steps are skipped.

Figure 10. RAM startup self-test structure

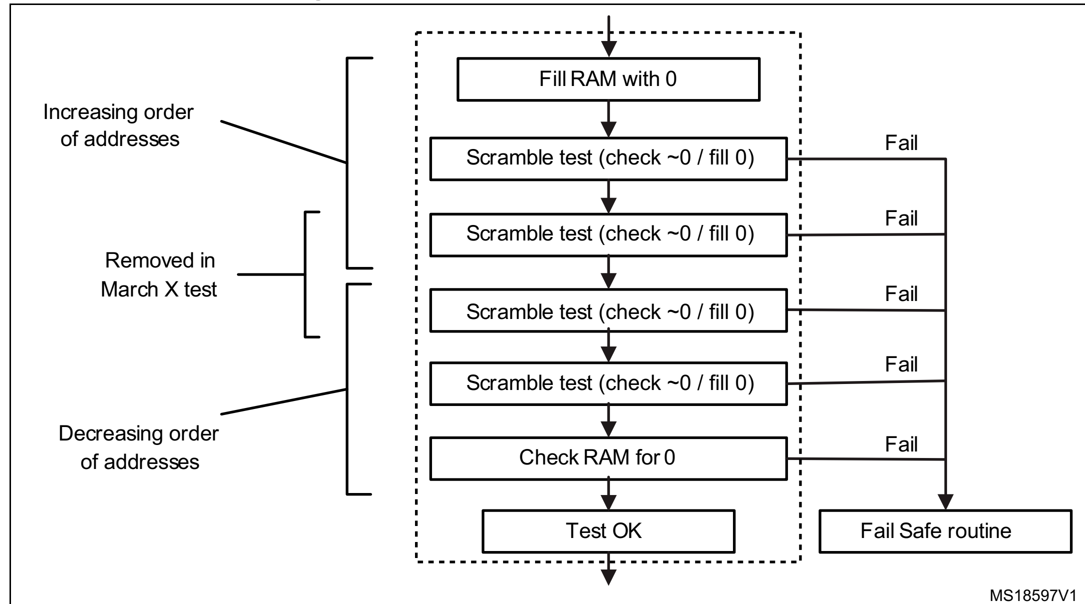
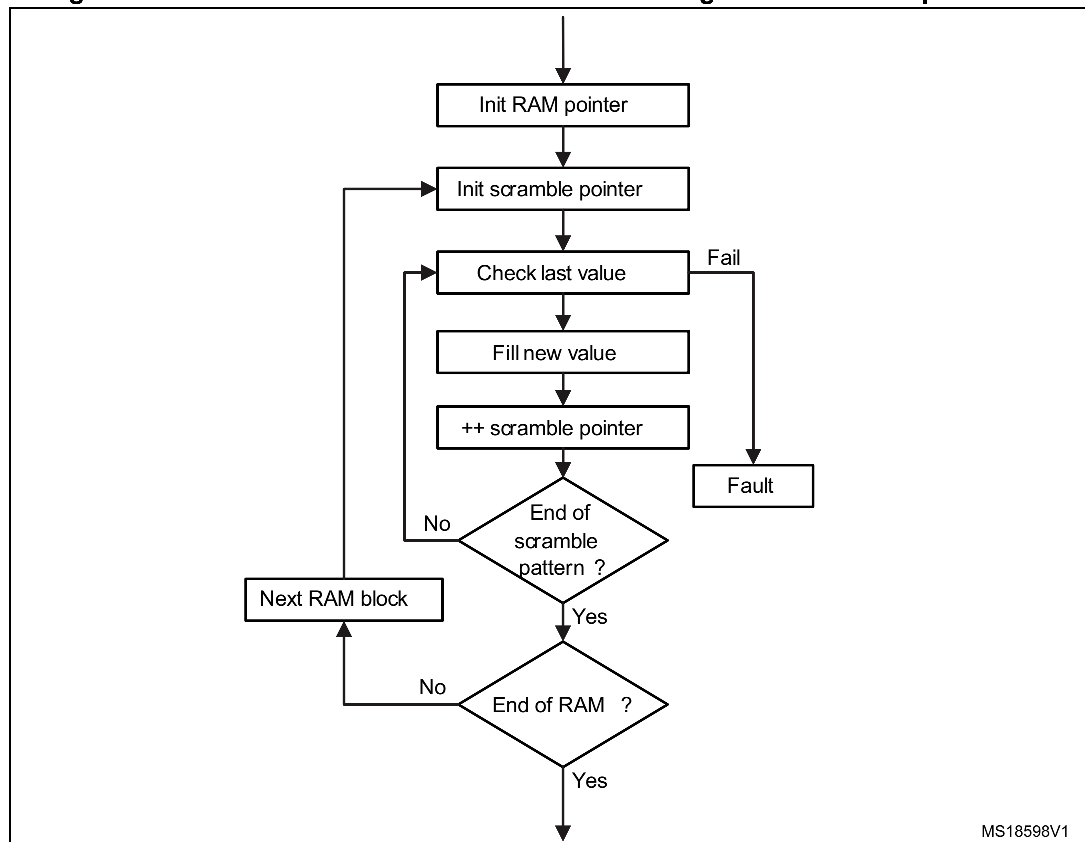


Figure 11. Scramble test - detailed structure of a single check and fill procedure

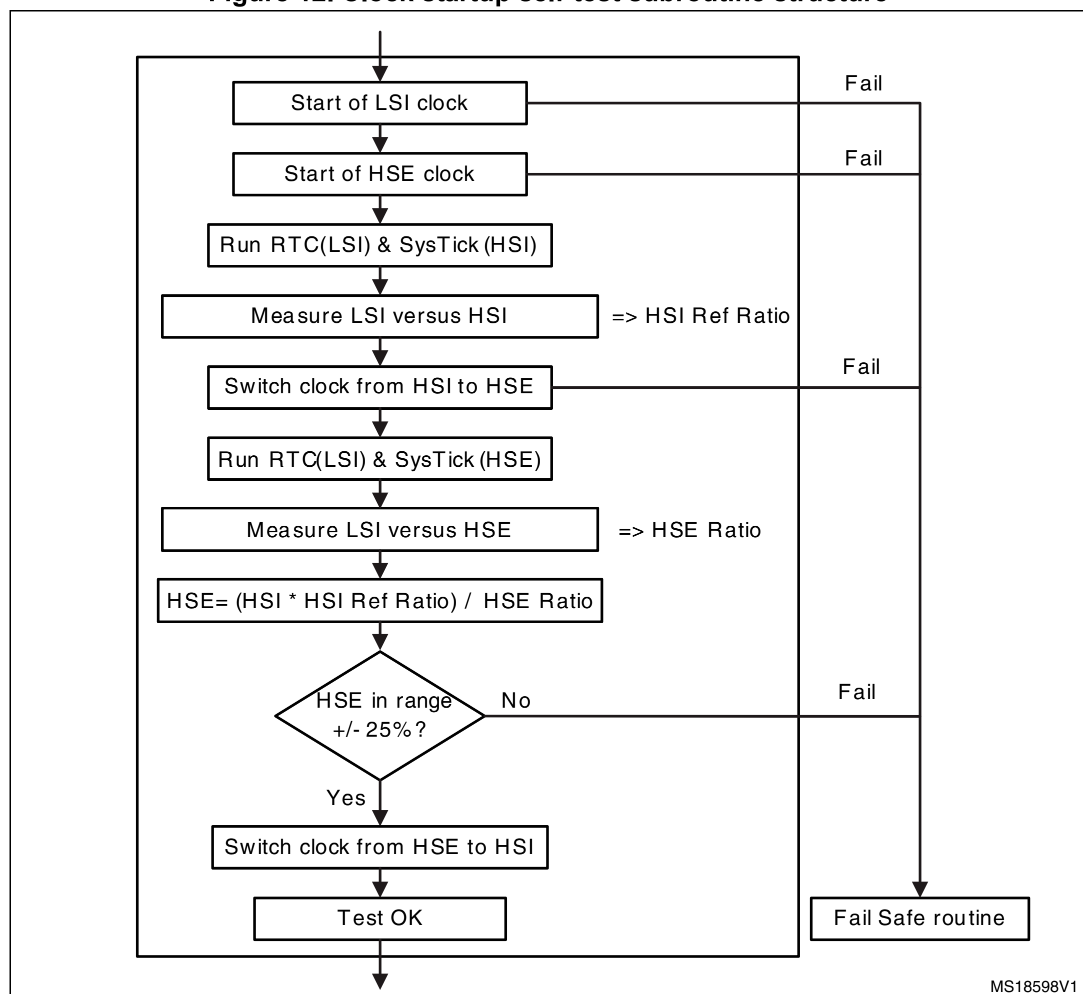


Note: The RAM test is byte oriented but the more precise bit March testing algorithm can be used, too. This method is more time and code consuming.

5.2.5 Clock startup self-test

The flow chart diagram of the test is shown in *Figure 12*. It expresses the principle of the method, graphically. Initially the internal low speed clock (LSI) source is started. The external high speed source (HSE) is started in the next step. The CPU is still running on the internal high speed clock source (HSI). The system timer (SysTick) is started with the HSI clock source while the RTC system runs on the LSI clock source. One complete underflow cycle of the SysTick timer is performed. The number of LSI periods counted in that cycle gives the HSI ratio value (number of HSI pulses within the interval corresponding to one LSI period). This initial HSI ratio measurement is stored as a reference for the next runtime measurements.

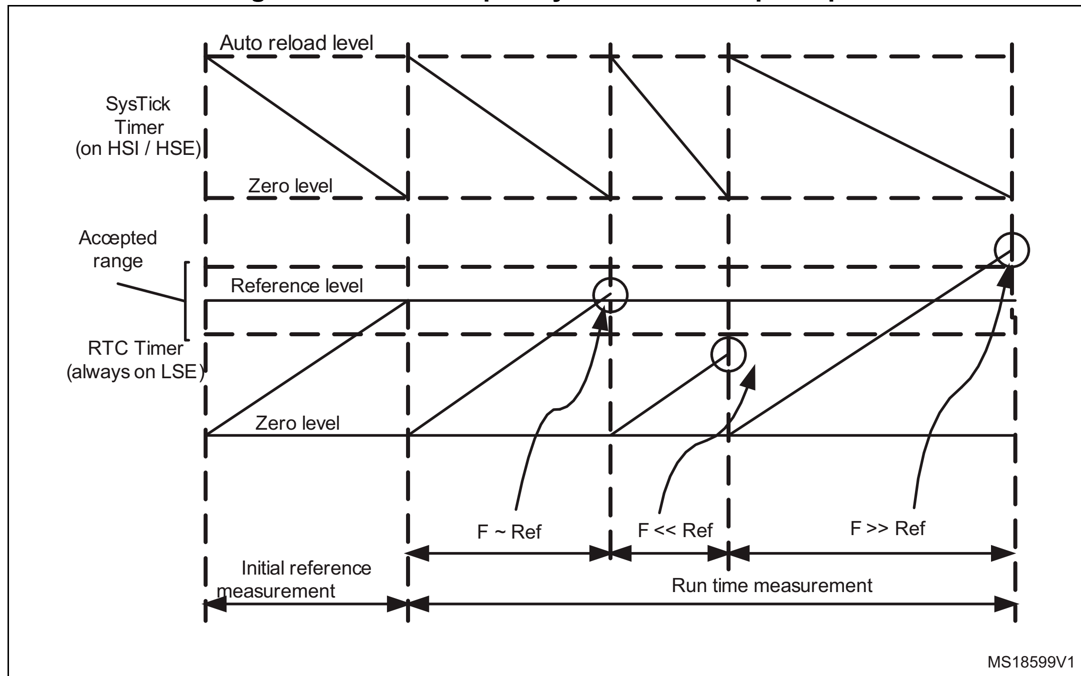
Figure 12. Clock startup self-test subroutine structure



In the next step, the external HSE clock is switched as the new clock source for the SysTick timer. Another complete underflow cycle of SysTick timer is performed. The number of LSI periods counted in this cycle gives the HSE ratio (number of HSE pulses in the interval corresponding to one LSI period). The HSE value is then computed from the known HSI value and both HSI and HSE ratios measured. If the HSE value deviates from the expected

interval (more than +/- 25% of its nominal value), the CPU clock source is immediately switched back to HSI, and HSE fail status is returned. Otherwise the test returns the OK status. In either case, the CPU clock is switched back to the default HSI source after the test is finished.

Figure 13. Clock frequency measurement principle

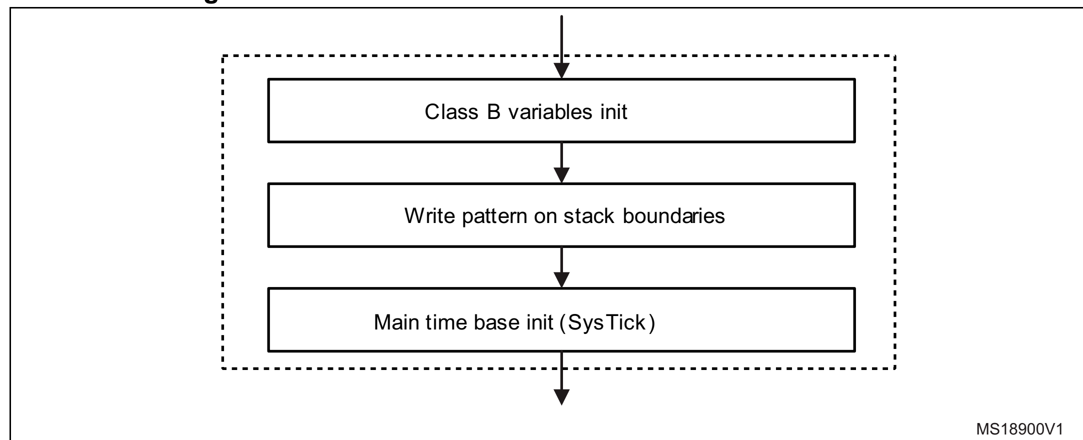


5.3 Periodic runtime self-test initialization

Assuming that the startup self-tests passed sucessfully and the standard initialization has been done, then the runtime self-tests package must be initialized just before the program enters the main loop performing regular calling of the runtime self-tests (refer to *Figure 5*). The timing should be set properly to ensure the procedures of the runtime tests are called at the necessary intervals.

At first all the class B variables are initialized. Zero and its complement value are stored into every class B variable complementary pair. The magic pattern is then stored at the top of the space separated for the stack. The timer peripherals are configured for the tick interval measurement and master clock frequency measurement. The same method as the startup test is used.

Figure 14. Periodic runtime self-test initialization structure



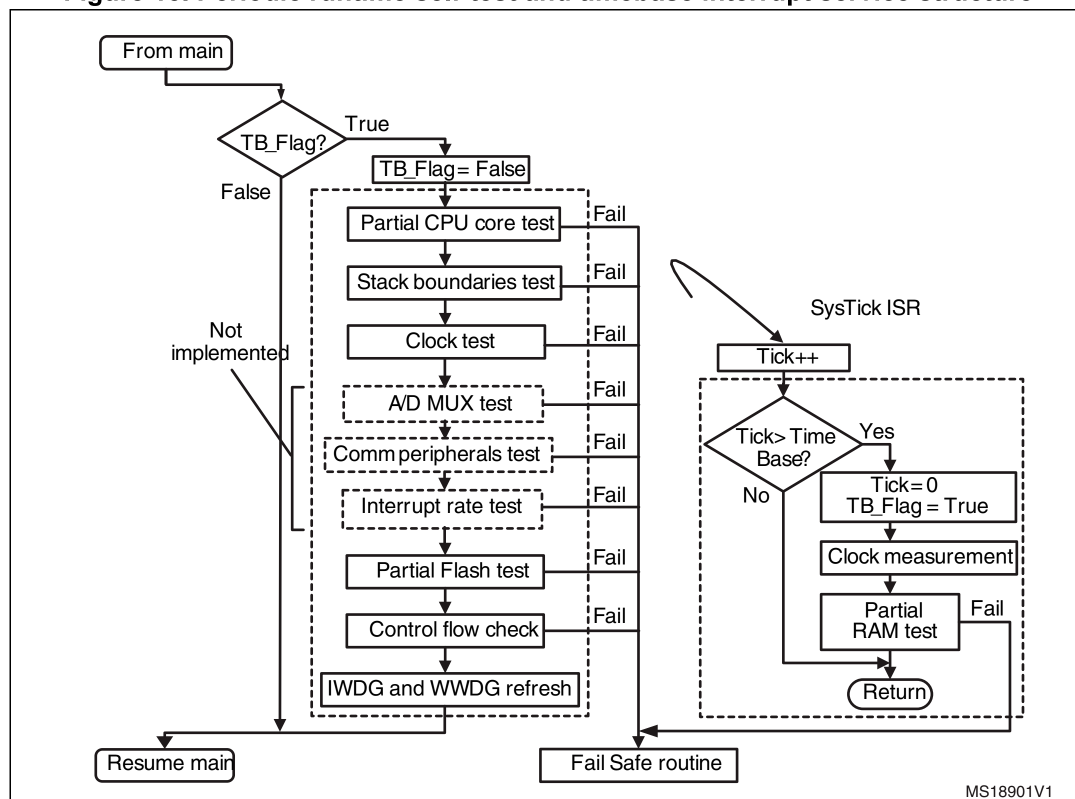
5.4 Description of periodic runtime self-tests

5.4.1 Runtime self-test structure

The Runtime self-test is a block of tests which is performed periodically at the main loop level. The execution period of the block is based on timebase interrupt settings. Before the first run, all the tests included must be initialized by the runtime initialization phase block (refer to *Figure 5*). Most of these tests are performed at the main loop level. Only the partial transparent RAM test and the backup of the latest clock measurement results are performed in the SysTick timer interrupt service routine at proper configurable intervals, signaled by TimeBaseFlag settings. It includes the following self-tests:

- CPU core partial runtime test
- Stack boundaries overflow test
- Clock runtime 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
- Independent and window watchdog refresh
- Partial transparent RAM March C-/X test (under system interrupt scope)

Figure 15. Periodic runtime self-test and timebase interrupt service structure

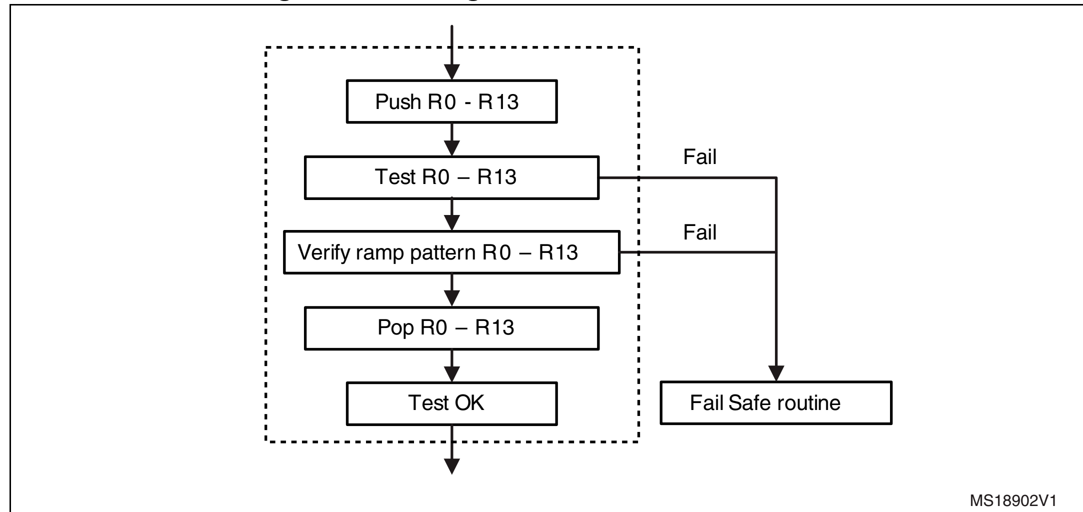


Note: The tests of the analog part, communication peripherals and application interrupts are not included and their implementation depends on the specific microcontroller device capabilities and user application needs.

5.4.2 CPU light runtime self-test

The light runtime CPU core self-tests is a simplified version of the runtime test described in *Section 5.4.1: Runtime self-test structure*. The flags and stack pointers are not tested. If an error code is returned the Fail Save procedure is called.

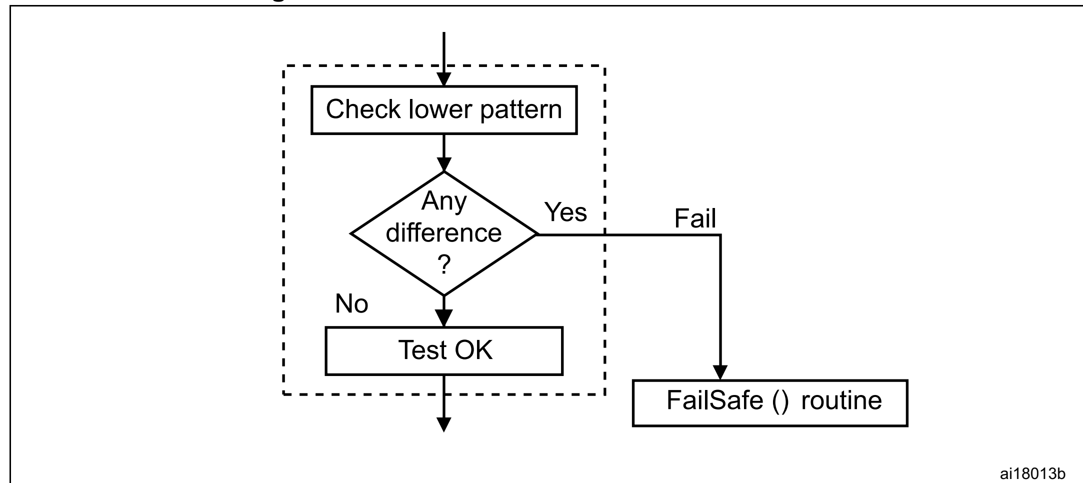
Figure 16. CPU light runtime self-test structure



5.4.3 Stack boundaries runtime test

This test checks the magic pattern stored at the top of the space reserved for the stack. If the original pattern is corrupted, the Fail Safe routine is called. The pattern is placed at the lowest address reserved for the stack area. It can detect both overflow and underflow as the stack pointer rolls over this range in either case. The stack area differs depending on the specific microcontroller device type. Be careful of the stack area limits when the location of the pattern is changed.

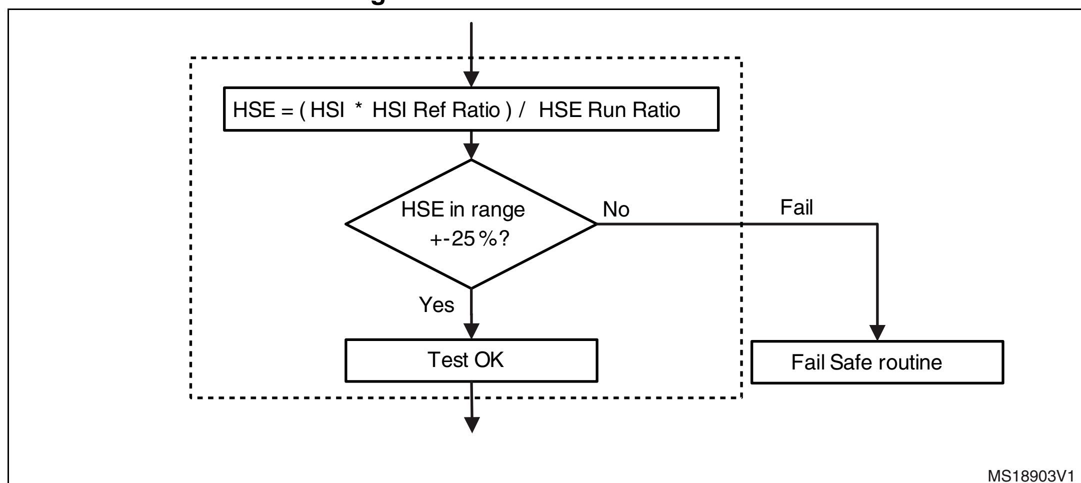
Figure 17. Stack overflow runtime test structure



5.4.4 Clock runtime self-test

The clock runtime self-test uses a similar procedure to the one used in the startup self-test. A few underflow cycles of the SysTick timer are performed at background processing level. The number of LSI periods counted during this period by the RTC timer gives the runtime HSE ratio (number of HSE pulses within the interval corresponding to the LSI period) which is stored as a Class B variable by the STL_MeasurePeriod() function called by the SysTick interrupt service routine. The HSE value is then computed at the main processing level in the clock run self-test. The inputs are the same as in the startup test: known HSI reference ratio value (stored in the initial startup test phase) and the currently measured HSE ratio. If the computed HSE value deviates from the expected interval (more than +/- 25% of its nominal value) the CPU clock source is immediately switched back to HSI, and HSE fail status is returned. Otherwise the test returns OK status.

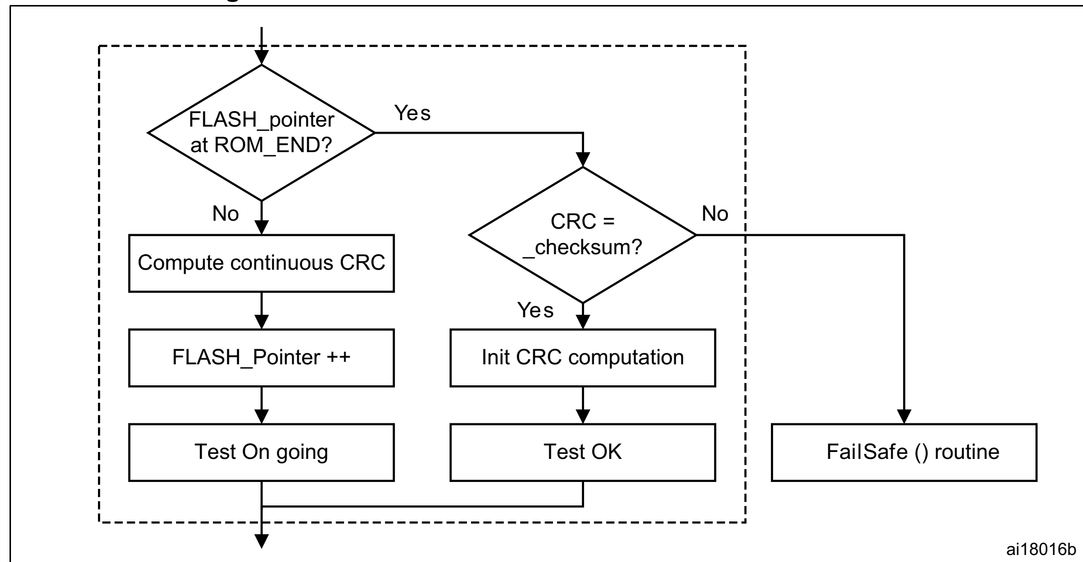
Figure 18. Clock runtime self-test structure



5.4.5 Partial Flash CRC runtime self-test

The partial 16-bit CRC checksum of the block in Flash is performed at each step. The boundaries are given by the structure stored by the linker. When the last block is reached, the CRC checksum is compared with the value stored by the linker. If a difference is found, the Fail Safe routine is called, otherwise a new computation cycle is initialized. Refer also to *Section 4.3.3: Debugging the package* for additional comments on CRC procedures.

Figure 19. Partial Flash CRC runtime self-test structure



5.4.6 Watchdog service in runtime test

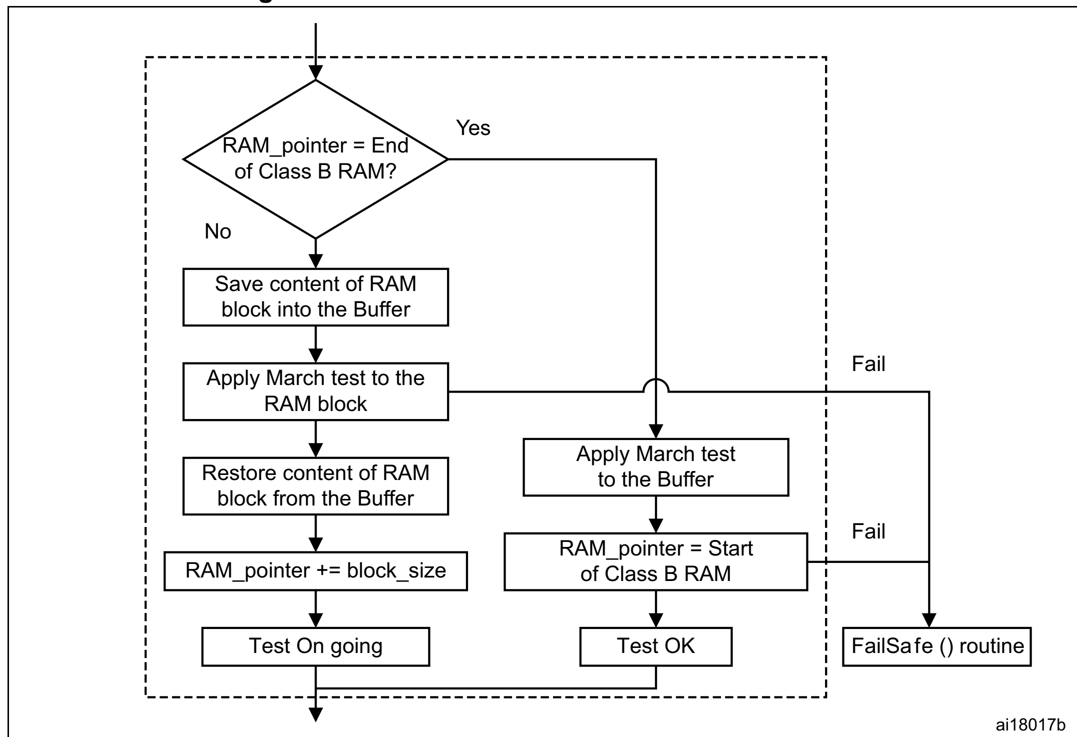
If the runtime service block is successfully completed, the window and independent watchdogs should both be refreshed as a last step, just before returning to the main loop. For the watchdogs to be refreshed correctly, a proper timing of the call to the runtime block is essential. The period when the block should be called is signaled internally by a time base flag which is tested at the beginning of the ***STL_DoRunTimeChecks()*** routine (see *Figure 15*). The users must ensure not to miss calling this procedure at the main level to be able to react properly to any change in the time base flag and consequently refresh the watchdogs at the correct intervals.

To use the watchdogs efficiently, it is important to keep the structure of the application with only one refresh placed in the main loop. There should be no other watchdog refreshes except the one in the ***STL_DoRunTimeChecks()*** routine. Sometimes it may also be necessary to refresh watchdogs in the initialization phase of the flow. In this case, the refresh should be outside any software infinite loop. Ideally it should only be put in a straightforward part of the code.

5.4.7 Partial RAM runtime self-test

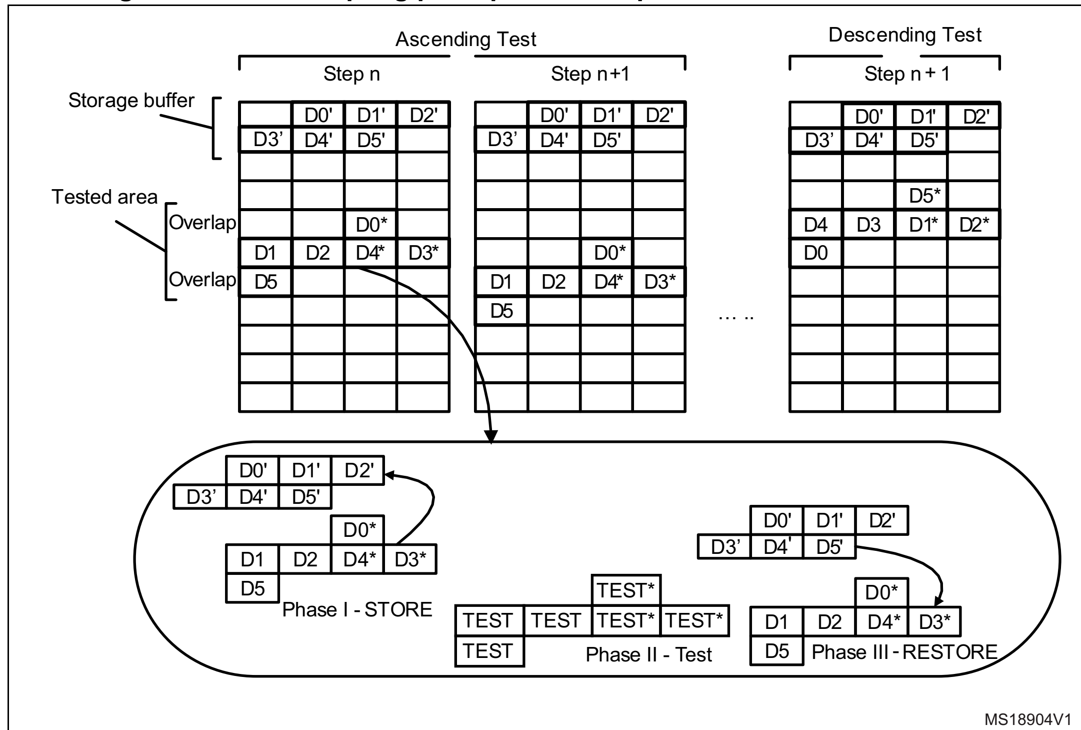
The partial transparent RAM test is performed inside the timebase interrupt service routine. The test covers the part of the RAM allocated to class B variables. One block of six bytes is tested in each step of the test. To guarantee coupling fault coverage, the blocks of memory are overlapped by two side bytes from one test step to the other. The order of testing the bytes in the block is scrambled with respect to the physical order of addresses in memory. For more details see *Section 5.2.4: Full RAM March C-/X self-test*.

Figure 20. Partial RAM runtime self-test structure



In the first phase, the content of the block is stored in the storage buffer. Marching destructive tests are performed over all the bytes of the tested RAM block in the next phase. Then, in the final phase of this step, the original content is restored from the storage buffer. The March X algorithm is faster because two middle marching steps are skipped. As a last step of the test sequence, the storage buffer itself is tested by a marching test. The buffer is tested together with the next two additional bytes to cover coupling faults in the buffer itself. Then the whole test is reinitialized and it starts from the beginning. If any fault is detected, the Fail Safe routine is called.

Figure 21. Fault coupling principle used at partial RAM runtime self-test



Note: The scrambling of physical addresses is respected in both ascending and descending tests, if applicable.

Table 4. March C- phases in 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	Test 0xFF pattern, write 0x00 pattern	Increasing
3	Test 0x00 pattern, write 0xFF pattern	Decreasing
4	Test 0xFF pattern, write 0x00 pattern	Decreasing
5	Test 0x00 pattern	Decreasing

Note: Steps 2 and 3 are skipped when the March X algorithm is used.

6 Revision history

Table 5. Revision history

Date	Revision	Description of changes
09-Dec-2010	1	Initial release
27-May-2011	2	Added the following text to cover page: The associated firmware package is available upon request only. Please contact your local ST sales office for more information.
29-Oct-2012	3	Updated document throughout to add support of STM32F0 and STM32F3 devices. Modified the *Introduction*. Added *Section 1: Package variation overview* and *Section 2: Main differences between STM32F packages and their possible modifications*. Modified *Section 4.2: Package organization*, *Section 4.3.1: Configuration control* and *Section 4.3.2: Verbose diagnostic mode*. Removed Appendix A.
26-Apr-2013	4	Modified title and removed Table 1. Applicable products and note. Updated *Introduction*, *Package variation overview*, and *Section 2: Main differences between STM32F packages and their possible modifications*. Modified *Clock tests and time base interval measurement*, *SRAM tests* and *Flash memory tests*. Added *Duration of the tests*. Updated *Section 4.2.3: Application demonstration example* and *Section 5.2.4: Full RAM March C-/X self-test*. Modified *Section 5.4.1: Runtime self-test structure* and note after *Figure 21: Fault coupling principle used at partial RAM runtime self-test*.
30-Mar-2016	5	Updated cover adding STM32-CLASSB-SPL Root Part Number. Updated *Figure 1: Example of RAM memory configuration*.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST's terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2016 STMicroelectronics – All rights reserved