life.augmented

AN3181 Application note

Guidelines for obtaining UL/CSA/IEC 60730-1/60335-1 Class B certification in any STM8 application

Introduction

The role of safety is more and more important in electronic applications. The level of safety requirements for components used in electronic designs is steadily increasing and 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, and also the associated standards related to safety requirements for hardware and software are in continuous development.

The current safety recommendations and requirements are specified in world wide recognized standards issued by IEC (International Electrotechnical Commission), UL (Underwriters Laboratories) and CSA (Canadian Standards Association) authorities, and come under compliance, verification and certification process by institutions like TUV and VDE (mostly operating in Europe), UL and CSA (targeting mainly US and Canadian markets).

The main purpose of this application note and of its associated software (STM8-SafeCLASSB) is to facilitate and accelerate user software development and certification processes for applications (based on STM8 microcontrollers) that are subject to these requirements and certifications.

The certified package is provided for:

- Mainstream STM8S and automotive STM8A high, medium and low density devices
- Ultra-low-power medium-density STM8L and STM8AL devices
- Ultra-low-power low density STM8L, STM8AL and STM8TL touch-sensing devices.

Due to limited memory capacity of most of 8-bit devices all these packages are optimized and independent from other firmware libraries published by ST. Proper headers *stm8xxx_it.h* and *stm8xxx_type.h* from ST standard peripheral libraries are included only to keep consistency of names of registers, bit masks, interrupt vectors and constants defined there. Optimized code reduces program memory overhead and increases the code execution speed.

All certified packages use similar principles, described in this document with focus on the main differences. Provided hardware and firmware compatibility within the STM8 sub-class is ensured, the user can easily adapt the projects included in the package to all the STM8 microcontrollers.

The STL package is pre-certified for the methodology and the used techniques. The provided examples show how to integrate the STL package in the application, however the final implementation and functionality has to be always verified by the certification body at the application level.

Contents

1 **Package overview** 6

2 **Package structure overview** 8

3 **Main differences from the product point of view** 11

3.1 Clock system test 13

3.2 RAM test 13

3.3 Flash memory integrity test 13

3.4 Start-up and system initialization 14

3.5 Firmware configuration 14

4 **Compliance with IEC, UL and CSA standards** 16

4.1 Generic tests included in STL firmware package 18

4.2 Application specific tests 20

4.2.1 Analog signals 20

4.2.2 Digital I/Os 21

4.2.3 Interrupts and external communication 21

4.2.4 Timing and program flow 21

4.2.5 External addressing 21

4.3 Safety life cycle 22

5 **Class B software package** 24

5.1 Common software principles 24

5.1.1 Fail Safe mode 24

5.1.2 Class B variables 24

5.1.3 Class B flow control 26

5.2 Firmware package structure 28

5.2.1 Projects and workspaces included in the package 28

5.2.2 Tools and other specifc controls of the library 28

5.2.3 Application examples 29

5.3 Package configuring and debugging 29

5.3.1 Configuration control 29

5.3.2 Verbose diagnostic mode 30

5.3.3 Debugging the package 30

6 Class B solution integration 32

6.1 Integrating software into user application 32

6.2 Detailed description of startup self tests 32

6.2.1 Watchdog startup self test 33

6.2.2 CPU startup self test 34

6.2.3 Flash memory complete checksum self test 34

6.2.4 Full RAM March C-/X self test 35

6.2.5 Clock startup self test 36

6.3 Periodic run mode self tests initialization 37

6.4 Detailed description of periodic run mode self tests 38

6.4.1 Run time self tests structure 38

6.4.2 CPU light run mode self test 39

6.4.3 Stack boundaries run mode test 39

6.4.4 Clock run mode self test 40

6.4.5 Partial Flash memory CRC run mode self test 41

6.4.6 Watchdog service in run mode test 42

6.4.7 Partial RAM run mode self test 42

Appendix A STM8 Class B firmware package variations 44

Appendix B List of verbose messages and codes reported at Fail Safe mode entry45

Revision history 46

List of tables

Table 1. Projects related to STM8-SafeCLASSB 8
Table 2. FW organization 8
Table 3. Overview of common STL procedures 9
Table 4. Overview of common tool specific STL procedures 10
Table 5. Integration support files 10
Table 6. STM8 compatibility aspects 12
Table 7. Specific compiler configurations 14
Table 8. STL specific configurations 14
Table 9. MCU parts that must be tested under Class B compliance 18
Table 10. Methods used in micro specific tests of associated ST package 19
Table 11. March C- phases at RAM partial test 43
Table 12. STM8 Class B firmware packages 44
Table 13. Verbose messages and unique codes reported at Fail Safe mode entry 45
Table 14. Revision history 46

List of figures

Figure 1. Example of RAM memory configuration 25
Figure 2. Control flow four steps check routine 27
Figure 3. Diagnostic LED timing signal principle 30
Figure 4. Integration of startup and periodic run mode self tests into application 32
Figure 5. startup self tests structure. 33
Figure 6. Watchdogs startup self test structure 34
Figure 7. CPU startup self test structure 34
Figure 8. Flash memory startup self test structure. 35
Figure 9. RAM startup self test structure 36
Figure 10. Clock startup self test subroutine structure. 37
Figure 11. Periodic run mode self test initialization structure. 38
Figure 12. Periodic run mode self test and time base interrupt service structure 39
Figure 13. CPU light run mode self test structure 39
Figure 14. Stack overflow run mode test structure 40
Figure 15. Clock run mode self test structure 40
Figure 16. Clock run mode self test principle. 41
Figure 17. Partial Flash memory CRC run mode self test structure. 41
Figure 18. Partial RAM run mode self test structure). 42
Figure 19. Fault coupling principle used in partial RAM run mode self test 43

1 Package overview

Available configurations of **STM8S/A package** support the following devices:

- STM8S high-density Performance line devices, with 32 to128 Kbytes Flash memory (like STM8S20x)
- STM8S medium- and low-density Access line devices, with 4 to 32 Kbytes Flash memory (like STM8S105x, STM8S103x)
- STM8S high-, medium- and low-density Value Line devices, with 4 to 64 Kbytes Flash memory (like STM8S007x, STM8S005x, STM8S003x)
- STM8S low-density Application specific devices, with 8 Kbytes Flash memory (like STM8S903x)
- STM8A high-density CAN line devices, with 32 to 128 Kbytes Flash memory (like STM8AF5xxx)
- STM8A high and medium density Standard line devices, with 8 to 128 Kbytes Flash memory (like (STM8AF6xxx)

Available configurations of medium-density **STM8L/AL package** support the following ultra-low-power devices:

- STM8L medium density standard devices, with 4 to 64 Kbytes Flash memory (like STM8L15xx)
- STM8L medium-density Value Line devices, with 4 to 64 Kbytes Flash memory (like STM8L05xx)
- STM8AL medium-density devices, with up to 32 Kbytes Flash memory (like STM8AL31xx, STM8AL3Lxx)

Available configurations of low-density **STM8L/AL/TL package** support the following ultra-low-power devices:

- STM8L low density standard devices, with up to 8 Kbytes Flash memory (like STM8L10xx)
- STM8AL low density devices, with up to 8 Kbytes Flash memory (like STM8AL30xx)
- STM8TL low density devices, with up to 16 Kbytes Flash memory and touch sensing interface (like STM8TL53xxx)

All these firmware packages are available on *www.st.com*.

The STM8 microcontrollers are based on a proprietary advanced 8-bit architecture core, performing up to 20 MIPS at 24 MHz.

Two projects have been prepared and tested for each package, using the following environment and toolchains:

1. IAR Embedded Workbench® for STM8 IDE (EWSTM8™) with IAR C/C++ Compiler™ version 3.10.1
2. ST Visual Develop (STVD) version 4.3.11 with Cosmic STM8 C compiler 32 K version 4.4.6

For more info on ElectroMagnetic Compatibility (EMC) refer to the following application notes, available on *www.st.com*:

AN1015, Software techniques for improving microcontroller EMC performance

- AN1709, EMC design guide
- AN2860, EMC guidelines for STM8S

2 Package structure overview

The projects included in this FW package are summarized in *Table 1*.

Table 1. Projects related to STM8-SafeCLASSB

Name	Revision	RPN	Evaluation board	Sub-family
STM8AS128_EVAL	2.0.0	STM8S208MBT6B	STM8/128-EVAL MB631 Rev D	STM8S/AF 128K
STM8S_Discovery	2.0.0	STM8S105C6T6	STM8S-Discovery MB867 Rev A	STM8S/AF 32K
STM8SVLDiscovery	2.0.0	STM8S003K3T6	STM8S-VLDiscovery MB1008 Rev A	STM8S/AL 8K
STM8L1528_EVAL	2.0.0	STM8L152M8T6	STM8L 1528 EVAL MB904 Rev B	STM8L/AL 64K/32K
STM8L101_Discovery	2.0.0	STM8L101K3T6	STM8L1-DB MB709 Rev A	STM8L/AL/TL 8K

All the projects are based on common self-test procedures collecting the principal tests and methods. These common C-language source files are kept in the Middleware directory.

Part of the methods are tool specific, but common to the whole family. Most of them are written in Assembler for specific compilers. Those files are located in a single place of the IAR and Cosmic directories of the pilot STM8AS128_EVAL project, and included by all the other projects, from these dedicated directories.

BSP directory collects specific drivers for evaluation boards (used for demonstrations) exclusively to control the board hardware (like displays, serial channels and LEDs). These drivers are dedicated to debug only, and are out of the certification scope.

The structure and content of the package is described in detail in *Table 2*.

Table 2. FW organization[1]

<table>
<tr><th colspan="2">Directory</th><th>Comments</th></tr>
<tr><td rowspan="2">Drivers/BSP</td><td>STM8128_EVAL</td><td rowspan="2">Evaluation board specific drivers</td></tr>
<tr><td>STM8L1528_EVAL</td></tr>
<tr><td rowspan="2">Middleware/ SM8_SelTest_Library</td><td>inc</td><td rowspan="2">Common STL procedures</td></tr>
<tr><td>src</td></tr>
<tr><td rowspan="4">Projects/STM8AS128_EVAL
Projects/STM8S_Discovery
Projects/STM8SVLDiscovery
Projects/STM8L1528_EVAL
Projects/STM8L101_Discovery</td><td>inc</td><td rowspan="2">Product integration example and product specific STL procedures</td></tr>
<tr><td>src</td></tr>
<tr><td>Cosmic</td><td rowspan="2">Tool specific STL procedures[2], product and tools specific configurations</td></tr>
<tr><td>IAR</td></tr>
</table>

1. Bold type is used to indicate the procedures that are the main focus of the certification.
2. Common tool specific procedures are collected exclusively for pilot STM8AS128_EVAL project directory.

The included projects for specific STM8 products and dedicated evaluation boards have been prepared and tested under two environments and tool chains:

- IAR™-EWSTM8 version 3.10.1
- Cosmic version 4.4.6 - STVD version 4.3.11

The detailed structure of these projects and the list of files collecting the common and specific STL procedures are summarized in *Table 3* and *Table 4*, respectively.

Table 3. Overview of common STL procedures

STL	Common STL procedures	
	File	**Description**
Start-up test	stm8_stl_startup.c	Start-up STL flow control
	stm8_stl_clockstart.c	Clock system initial test
Run time test	stm8_stl_main.c	Run time STL flow control
	stm8_stl_crcrun.c	Partial Flash test
	stm8_stl_clockrun.c	Partial clock test
	stm8_stl_transpRam.c	Partial RAM test
Headers	stm8_stl_classB_var.h	Definition of Class B variables
	stm8_stl_lib.h	Overall STL includes control
	stm8_stl_startup.h	Initial process STL header
	stm8_stl_main.h	Run time process STL header
	stm8_stl_param.h	STL configuration file
	stm8_stl_clockstart.h	Start-up clock test header
	stm8_stl_clockrun.h	Run time clock test header
	stm8_stl_cpu.h	CPU test header
	stm8_stl_crc16Run.h	Flash test header
	stm8_stl_fullRam_Mc.h	Start-up RAM test header
	stm8_stl_transpRam.h	Run time RAM test header

Table 4. Overview of common tool specific STL procedures

STL	Compiler	Common STL procedures	
		File	Description
Source	Cosmic	_classb_cksumXXX.s	Start-up CRC calculation
		_block_cksumXXX.s	Run time CRC calculation
		stm8_stl_cpustart_CSMC.s	Start-up CPU test
		stm8_stl_cpurun_CSMC.s	Run time CPU test
		stm8_stl_fullRam_CSMC.s	Start-up RAM test
	IAR	stm8_stl_cpustart_IAR.asm	Start-up CPU test
		stm8_stl_cpurun_IAR.asm	Run time CPU test
		stm8_stl_fullRam_IAR.asm	Start-up RAM test
		stm8_stl_crc16_IAR.c	Start-up CRC calculation

Additional supporting files used in the examples are listed in *Table 5*.

Table 5. Integration support files

File	Description
cstartup.s	C start-up for IAR™ compiler
stm8_interrupt_vector.c	Interrupt vector table for Cosmic
main.c	Main flow of the example source
stm8*xxx*_it.c	STL Interrupts, clock measurement processing and configuration procedures
main.h	Main flow header
stm8*xxx*.h	Product specific header
stm8*xxx*_it.c	Product specific ISR header

3 Main differences from the product point of view

The user can find some small differences, mainly due to product hardware configuration, and to incompatibilities of compilers and debugging tools.

The main differences are described in this section, they are due mainly to compatibility aspects between different STM8 products, summarized in *Table 6*.

Table 6. STM8 compatibility aspects

Feature	STM8S207/208 STM8AF (128K)	STM8S105/005 STM8AF (32K)	STM8S103/903/003 (8K)	STM8L15x/16x/05x/06x STM8AL (32K and 64K)	STM8L101 STM8AL/TL5x (8K)
Core	STM8 - Proprietary				
Technology [nm]	130			130[1]	
Frequency [MHz]	24	16			
Performance [DMIPS]	20	10	10	16[2]	
Flash memory density [KB]	128	32	8	64 / 32	8
RAM density [KB]	6	2	1	4	1.5
Data EEPROM [bytes]	2048	640	128	2048	-
ECC on Non-Volatile Memory[3]	Yes				
Window watchdog	Yes				No[4]
Stack HW roll-over limit at RAM end [bytes]	1024	513			513[5]
Clock system	HSE-24 HSI-16 LSI ~128 kHz	HSE-16 HSI-16 LSI ~128 kHz	HSI-16 LSI ~128 kHz	HSE-16 LSE-32,768 kHz HSI-16, LSI ~38 kHz	HSI-16 LSI ~38 kHz
Clock cross reference measurement	TIM3/Ch1	TIM3/Ch1	TIM1/Ch1	TIM2/Ch1	TIM2/Ch1

1. Low power technology.
2. CISC MIPS.
3. Both embedded Flash and EEPROM feature internal single bit correction, hidden to the user.
4. WWDG is available only on STM8STL5x devices.
5. Stack is not limited for STM8TL5x devices; there is rollover (due to over/underflow) only if the stack overlaps the 4 KB.

3.1 Clock system test

Internal timers are used to cross-check frequency measurements. This method is required to determine harmonic or sub-harmonic frequencies when the system clock is provided by an external crystal (or ceramic resonator, if applicable), or to detect any significant discrepancy in the application timing. Different product dependent timers are dedicated to perform such cross check measurements.

The initial configuration of the specific timers is slightly different, while dedicated interrupt vectors are used for the measurement in dependency of the available timers on a given device.

CSS clock security feature is enabled for HSE quartz clock by default. The user has to ensure proper setting of the HSECNT (HSE oscillator stabilization time) parameter in the option bytes, thus ensuring sufficient time for the oscillator proper start after reset, and thus preventing premature action of the CSS system.

If the system clock doesn't use the HSE quartz clock, the user can set up the clock measurement HSI vs. LSI commenting out the parameter STL_INCL_HSECSS in the stm8_stl_param.h file, or adapting the clock measurement to be based on another reliable clock source (e.g. line power frequency) to satisfy the standard requirements for the clock monitoring.

In any case, if the cross check measurement depends upon the RC clock (HSI or LSI), the user has to consider the accuracy of this clock source over the whole temperature range. This is necessary to prevent any false clock failure detection, especially when the unit under self-test operates over a wide temperature range. The user can apply an adaptable clock test algorithm while monitoring the trend of the ambient temperature, or consider a more accurate source to be taken as a clock reference.

3.2 RAM test

By default, a lighter Marching X algorithm is applied during run time instead of the Marching C- one applied at start-up test of volatile memory. It depends on STL_RUN_USE_MARCHX symbol defined in the stm8_stl_param.h file. If this symbol is defined, two middle marching steps are skipped and not implemented during the transparent run time test. Optionally, user can apply Marching C- test at run time, too, by commenting out this parameter definition.

The test range both at startup and during run time has to be customized according to the product volatile memory capacity, by proper setting of constants in the linker or in the library configuration file.

3.3 Flash memory integrity test

Different methods can be applied, depending on the used compiler. Cosmic uses a lighter CRC test, which is faster but its algorithm doesn’t fully correspond to CRC standard. Moreover Cosmic has defined different procedures for different memory models. Consequently the user has to verify the applied model and if 8-bit or 16-bit CRC pattern is used.

IAR™ uses a standard procedure, where the CRC calculation is simulated by SW or uses a look-up table. The second method is considerably faster, but needs a significant part of code.

When a slower method is used and/or large memory area is tested, the user has to split the memory testing in segments, and take care about the handling of watchdogs between them.

3.4 Start-up and system initialization

Standard product start-up file dedicated to IAR™ is modified to call set of start-up tests at the very first program execution. Reset vector in the interrupt vector table for Cosmic is modified to force the program flow to startup-tests after the application reset.

3.5 Firmware configuration

All the STL configuration parameters and constants used in the STL code written at C-level are collected into one file, stm8_stl_param.h. Configuration differences are mainly related to different size of tested areas, different compilers and to small deviations of the control flow.

The specific compiler and the STL configuration options are summarized, respectively, in *Table 7* and *Table 8*.

Projects for Cosmic compiler have been tested with Short Stack model, disabled optimizations and enforced functions prototyping for C-compiler, while those for IAR compiler have bee tested with standard C-language conformance with IAR extensions enabled, C99 dialect and low level optimizations.

Table 7. Specific compiler configurations

Feature	Where it acts	Target
Device	Compilation parameters	Set up proper peripheral configuration and physical ranges of the embedded memories (e.g. STM8S208 or STM8L15X_MD).
Optimization	Compiler configuration	Some higher optimization settings may cause unexpected removal or corruption of testing procedures.
Memories ranges	Linker file, Compiler configuration, STL parameters	Set ranges tested in volatile and non-volatile memories both at startup and during run time.
Check sum calculation	Compiler configuration, Project configuration, STL parameters	Set type of program Flash memory integrity check (8-bit, far addressing, fast or slow method) and include proper source files supporting the selected calculation.

Table 8. STL specific configurations

Feature	Where it acts	Target
Set test	STL parameters	Remove unused tests (e.g. external quart clock if not applied) as, by default, all the tests are included.
Debug diagnostic	STL and Compilation parameter	Select included diagnostic functions (e.g. DEBUG, STL_VERBOSE, EVAL_BOARD_CONTROL) and limit some STL functions when debugging the library (e.g. watchdogs or Flash memory check sum evaluation).

Table 8. STL specific configurations (continued)

Feature	Where it acts	Target
Safety variables	stm8_tl_ClassB_var.h header file	Definition of safety critical variables keeping redundant information stored in the area under permanent transparent testing during run time.
RAM transparent test	STL parameter	Control algorithm during RAM transparent test during run time (enables lighter and faster March X method).

For more detailed description of the FW structure configuration and integration aspects see *Section 5: Class B software package*.

4 Compliance with IEC, UL and CSA standards

IEC (International Electro technical Commission) is a not-for-profit and non-governmental world wide recognized authority 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 the ISO (International Organization for Standardization) and the ITU (International Telecommunication Union). Their standards define not only the recommendations for hardware but as well for software solutions divided into a number of safety classes in dependency of the purpose of the application.

Other world wide recognized bodies in the field of electronic standards are TUV or VDE in Germany, IET in the United Kingdom and the IEEE, UL or CSA in the United States and Canada. Beyond providing expertise during standard development process, they act as testing, inspection, consultancy, auditing, education and certification bodies. Most of them target global market access but are primarily recognized and registered as a local National Certification Bodies (NCB) or National Recognized Testing Labs (NRTL). The main purpose of these institutions is to offer standards compliance and quality testing services to manufacturers of electrical appliances.

Due to globalization process, most of manufacturers push for harmonization of national standards. This is contrary to the efforts of many governments, still protecting smaller local producers by building administrative barriers to prevent easy local market access from abroad. As a matter of fact, most of the standards are well harmonized, with negligible differences. This makes the certification process easier, and any cooperation with locally recognized bodies is fruitful.

The pivotal IEC standards are IEC 60730-1 and IEC 60335-1, well harmonized with UL/CSA 60730-1 and UL/CSA 60335-1 starting from their 4th edition (previous UL/CSA editions use references to UL1998 norm in addition). They cover safety and security of household electronic appliances for domestic and similar environment.

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

The conditions required are defined in detail in Annexes Q and R of the IEC 60335-1 norm and Annex H of the IEC 60730-1 norm.

Three classes are defined by the 60730-1 standard:

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

For programmable electronic component applying a safety protection function, the 60335-1 standard requires incorporation of software measures to control fault /error conditions specified in tables R.1 and R.2, based on Table H.11.12.7 of the 60730-1 standard:

- Table R.1 summarizes general conditions comparable with requirements given for Class B level in Table H.11.12.7.
- Table R.2 summarizes specific conditions comparable with requirements for Class C level of the 60730-1 standard, for particular constructions to address specific hazards.

Similarly, if software is used for functional purposes only, the R.1 and R.2 requirements are not applicable.

The scope of this Application note and associated STL package is Class B specification in the sense of 60730-1 standard and of the respective conditions, summarized in Table R.1 of the 60335-1 standard.

If safety depends on Class B level software, the code must prevent hazards if another fault occurs in the appliance. The self test software is taken into account after a failure. An accidental software fault occurring during a safety critical routine will not necessarily result into an hazard thanks to another applied redundant software procedure or hardware protection function. This is not a case of much more severe Class C level, where fault at a safety critical software results in a hazard due to lack of next protection mechanisms.

Appliances complying with Class C specification in the sense of the 60730-1 standard and of the respective conditions summarized in Table R.2 of the 60335-1 standard are outside the scope of this document as they need more robust testing and usually lead to some specific HW redundancy solutions like dual microcontroller operation. In this case, user should use product dedicated safety manuals and apply the methods described there.

Class B compliance aspects for microcontrollers are related both to hardware and software. The compliant parts can be divided into two groups, i.e. micro specific and application specific items, as exemplified in *Table 9*.

While application specific parts rely on customer application structure and must be defined and developed by user (communication, IO control, interrupts, analog inputs and outputs) micro specific parts are related purely to the micro structure and can be generic (core self diagnostic, volatile and non-volatile memories integrity checking, clock system tests). This group of micro specific tests is the focus of the ST solution, based on powerful hardware features of STM8 MCUs, such as dual independent watchdogs or clock system monitoring.

Table 9. MCU parts that must be tested under Class B compliance

Group	Component to be tested according to the standard
Microcontroller specific	CPU registers
	CPU program counter
	System clock
	Invariable and variable memories
	Internal addressing (and External if any)
	Internal data path
Application specific	Interrupt handling
	External communication
	Timing
	I/O periphery
	Analog A/D and D/A
	Analog multiplexer

4.1 Generic tests included in STL firmware package

The certified STM8 STL firmware package is composed by the following micro specific software modules:

- CPU registers test
- System clock monitoring
- RAM functional check
- Flash integrity check
- Watchdog self test
- Stack overflow monitoring.

Note: The last two items from the upper list are not explicitly requested by the norm, but they improve overall fault coverage and partially cover some specific required testing (e.g internal addressing, data path, timing etc.).

An overview of the methods used for the MCU-specific tests (described in deeper detail in the following sections) is given in *Table 10*.

User can include a part or all of the certified SW modules into his project. If they aren't changed and are integrated according with these guidelines the time and costs needed to get a certified end-application will be significantly reduced.

When tests are removed user should consider side effects because any not applied component test could play a significant role at indirect testing of other components as well.

Table 10. Methods used in micro specific tests of associated ST package

Components to be verified	Method used	IEC/UL 60730 references		
		Table	Items	Applied methods
CPU registers	Functional test A, X and Y registers, flags and stack pointer are performed at startup. In the run time flags are not tested. Stack pointer is tested for overflow and underflow. If any error is found, the software jumps directly to the Fail Safe routine.	H.1	1.1	H.2.16.5 H.2.16.6 H.2.19.6
Program counters	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(1), while the Independent one (driven by lthe ow speed internal RC oscillator) cannot be disabled once enabled. Both watchdogs must be serviced at regular intervals. Program control flow is additionally monitored by specific software method (see Section 3.1.3: Class B flow control).	H.1	1.3	H.2.18.10.2 H.2.18.10.4
Addressing and data path	This is tested indirectly by RAM functional and Flash memory 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).	H.1	4.3, 5 and 5.2	H.1.5.1 H.1.5.2 (indirect testing)
Clock	Two independent internal frequencies are used for the dedicated timer clock and they are verified by reciprocal comparison. One frequency is fed to the dedicated timer while the other gates it.	H.1	3	H.2.18.10.1 H.2.18.10.4
Non-volatile memory	A 16-bit CRC software checksum test of the entire memory is carried out at startup and a partial memory test is repeated at runtime (block by block).	H.1	4.1	H.2.19.4.1 H.2.19.8.1
Variable memory space	March C- (or, optionally, March X) full memory test is performed at startup. Partial memory test is repeated during run time (block by block). Word protection with double inverse redundancy (inverse values stored in nonadjacent memory space) is used for safety critical Class B variables. Class A variable space, stack and unused space are not tested during run time.	H.1.	4.2	H.2.19.6.2 H2.19.8.2

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

The applied tests are primarily dedicated to detect permanent faults (to cover faults under so called d.c. fault model). Detection of transient faults by any software testing is always limited, because of the relatively long repetition period of testing (in comparison with any HW methods with permanent checking capability), and can be covered partially with indirect routes.

Note: In case of minor changes to the modules, the user should keep track of all of them, placing clear explanation commentaries in the source files and informing the certification authorities of the differences vs. the certified routines.

4.2 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 and multiplexer
- Digital I/Os
- Interrupts and external communication
- Timing and program flow
- External addressing

4.2.1 Analog signals

Measured values should be checked for plausibility and verified by measurements performed by other redundant channels, while free channels can be used to read some reference voltages in conjunction with testing of analog multiplexers used in the application. The internal reference voltage should also be checked. Multiple acquisition at one channel or comparison of redundant channels, followed by averaging operations, can be applied.

ADC input signal disconnection

Can be tested by using Schmitt triggers and pull-ups. It is recommended that triggers are disabled on the GPIO pin used for ADC analog input. In this case the GPIO digital input signal cannot be read from the pin, as it always returns 0. To test the analog signal disconnection the Schmitt triggers can be enabled temporary on the analog input pin. This enables the digital input functionality on the tested GPIO. Activation of pull-up resistor at this pin can be used for testing analog source signal disconnection from the pin. User can read the digital input value on the GPIO with activated pull-up and compare it with the analog value measured by ADC on the pin.

Other free pins with DAC functionality (or GPIO output functionality) can be used for analog signal injection into tested ADC input. Monitoring the ADC input allows the user to detect the analog signal disconnection from ADC input channel pin.

Routing interface can be used on some STM8L devices for internal connection between analog pins. By routing interface it is possible to connect two pins in parallel and then perform ADC measurement on those two pins on independent channels.

Testing of internal reference voltage and temperature sensor

ADCs can measure the internal reference voltage and/or internal temperature sensor (on some STM8L devices). Functionality of the internal reference voltage and of the internal temperature sensor can be based on the measured ratio between those two voltages, by checking that the ratio is within the allowed range.

Additional redundant testing (for internal reference voltage and internal temperature sensor functionality) can be performed on system where the V_{DD} voltage is known.

ADC clock testing

Measurement of the ADC conversion time (by timers) can be used to test the independent ADC clock functionality.

DAC output functionality (STM8L only)

Free ADC channel can be used to check if the DAC output channel is working correctly. The Routing interface can be used for connection between ADC input channel and DAC output channel.

Comparator functionality

Comparator inputs can be used for comparison between known voltage and DAC output voltage or internal reference voltage.

Analog signal disconnection can be tested by pull-down or pull-up activation on tested pin, and compariso of this signal with DAC voltage as reference on another comparator input.

ADC/DAC

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

4.2.2 Digital I/Os

Class B tests must detect any malfunction on digital I/Os. This can 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).

4.2.3 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.

Data exchange during communication sessions should be checked, including 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. Periodicity and effective occurrence of the communication events together with protocol error signals has to be permanently checked.

4.2.4 Timing and program flow

Timing and program flow can be verified by ensuring that the application routines execution times and order are both correct and complete, and that there are no unexpected delays. A cross-check with a different time base can be performed too. Timing control is strictly dependent on the application.

4.2.5 External addressing

External addressing is not used by STM8 microcontrollers.

4.3 Safety life cycle

Development and maintenance of FW are provided with respect to requirements of UL/IEC 60730-1 concerning prevention of systematic errors focused mainly in Section H.11.12.3. All the associated processes follow the ST internal policy to ensure they have the required level of quality.

Application of these internal rules and the compliance with the recognized standards are target of regular inspections and audits carried out by recognized external inspection bodies.

The following phases are involved:

Specification of safety requirements

The main target was pointed by internal planning to provide set of generic modules independent on user application to be easily integrated into user firmware targeting compliance with UL/IEC 60730-1 and UL/IEC 60335-1 standards. Used solutions and methods reviewed by certification authority speed up the user development and certification processes.

Architecture planning

The STL packet structure is the result of a long experience with repeatedly certified FW, where both optimized and standard peripheral libraries based modules were integrated into these sets of self tests in the past dedicated for each sub product separately. Main goal of the new FW has been to collect and integrate most of the safety critical routines into common sources to be shared and reused by all the members of the family overall.

Such common architecture is considerably safer from a systematic point of view, involves easier maintenance and integration of the solution when migrating either between existing or into new products. The same structures are applied by many customers in many different configurations, so their feedback is absolutely significant and helps to efficiently address weaknesses, if any.

Planning the modules

The testing methods of modules comes from proved solutions used at the original FW. Some methods were optimized to speed up the test period and so minimize limitation of the process safety time at the final application applying these self testing methods, provided mostly by software.

Coding

Coding is based on principles defined by internal ST policy, respecting widely recognized international standards of coding, proven verification tools and compilers.

Emphasis is put on performing very simple, single and transparent thread structure of code, calling step by step the defined set of testing functions while using simplified and clear inputs and outputs.

The process flow is secured by specific flow control mechanism and synchronized with system tick interrupts service providing specific particular background transparent testing. Hardware watchdogs service is provided exclusively once the full set of partial checking procedures is successfully completed.

Testing modules

Modules have been tested for functionality on different products, with different development tools. Details can be found in the following sections and in the specific test documentation dedicated to certification authorities (Test report).

Modules integration testing

Modules integration has been tested in several examples dedicated to different products using different development tools, focusing on proper timing measurements, code control flow, stack usage and other methods. Again, details can be found in the following sections and in the test documentation.

Maintenance

For the FW maintenance ST uses feedback from customers (including preliminary beta testers) processed according to standard internal processes. New upgrades are published at regular intervals or when some significant bugs are identified. All the versions are published with proper documentation describing the solution and its integration aspects. Differences between upgrades, applied modifications and known limitations are described in associated Release Notes included in the package.

Specific tools are used to support proper SW revision numbering, source files and the associated documentation archiving.

All the FW and documentation are available to ST customers directly from *www.st.com,* or on request, made to local supporting centers.

5 Class B software package

This section highlights the basic common principles used in ST software solution.

The workspace organization is described together with its configuration and debugging capabilities. The differences between the supported development environments (IAR™ EWARM, STVD Cosmic) are also addressed.

5.1 Common 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.

5.1.1 Fail Safe mode

FailSafe() routine is called (defined in ***stm8_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 5.3* 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.

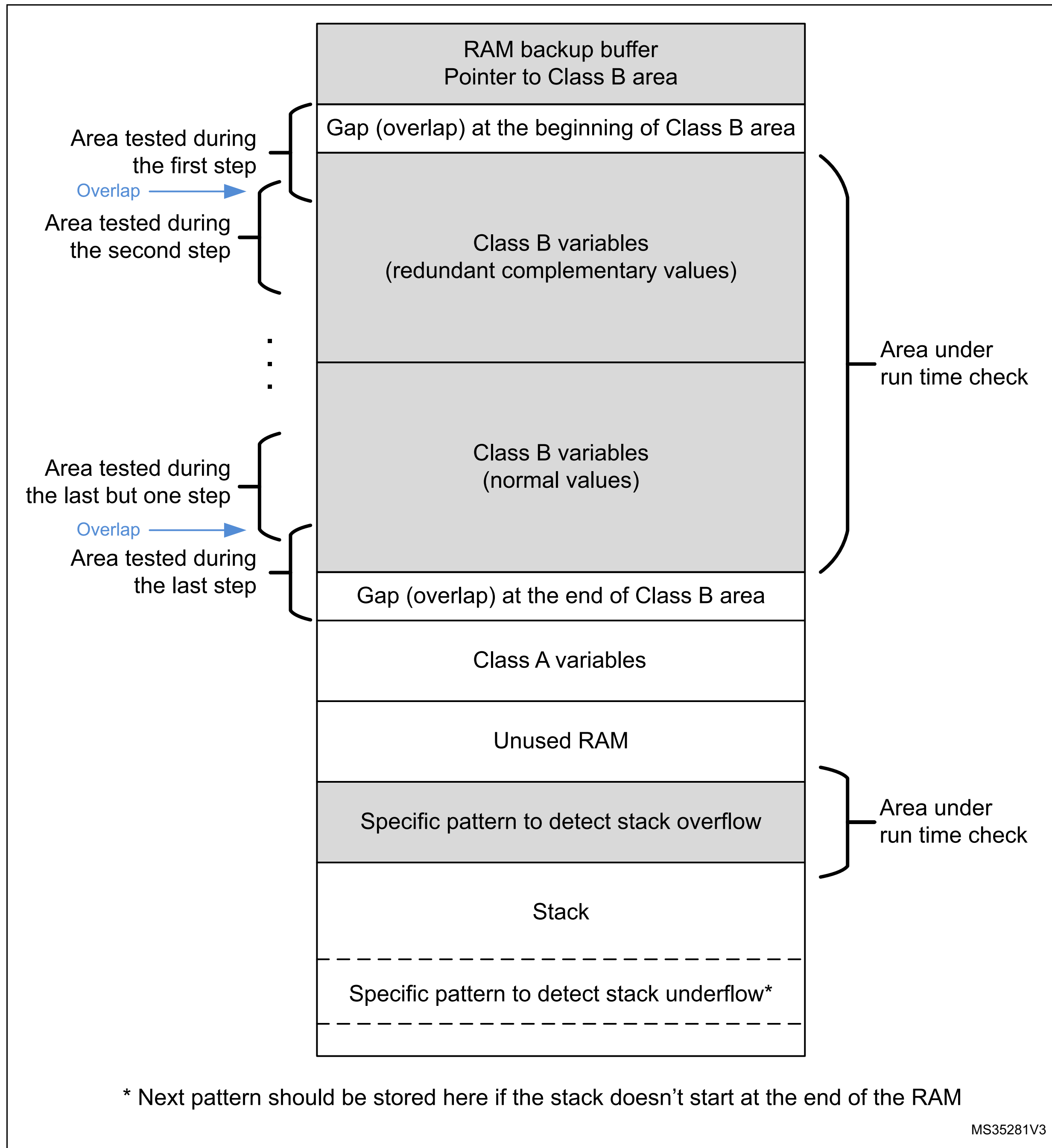
Even when debug mode is not applied, the user can detect the cause of the entry from the unique code passed by the caller as a parameter, and thus adjust the flow.

The complete list of error codes is provided in *Appendix B: List of verbose messages and codes reported at Fail Safe mode entry*.

5.1.2 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



Note: In the integration examples associated with the firmware package, all the Class B variables and buck-up buffers used for temporal storage of the RAM area tested during run time are allocated in a portion of Page0 accessed by short single byte addressing mode to speed up the testing procedures.

For a better consistency of the run time test, both class B regions are merged in a single compact memory location. The user should align the size of the tested area to multiple single transparent steps while respecting the overlay used for the first and the last step of the test.

If the area is not aligned, the not aligned bytes at the end of the area are tested together with the following redundant bytes completing the block size in the last step of the test, while the next additional one is included for the block overlay.

This is why the user has to allocate dummy gaps at the beginning and at the end of the area dedicated to Class B variables. The size of these gaps corresponds to the applied overlay of the single tested block (set to a single byte by default), with occasional number of redundant bytes making the area aligned with multiples of the steps.

Backup buffer and run time pointer to Class B area have to be allocated out of the area dedicated to Class B variables, in a specific location, tested separately each time the overall Class B area test cycle is completed.

When the full dedicated space for stack is applied at the end of the RAM area defined by the stack HW roll over limit, it is advisable to place a common specific pattern for detection of both overflow and underflow at the end of the RAM, while initializing the stack pointer in the first address just below this pattern.

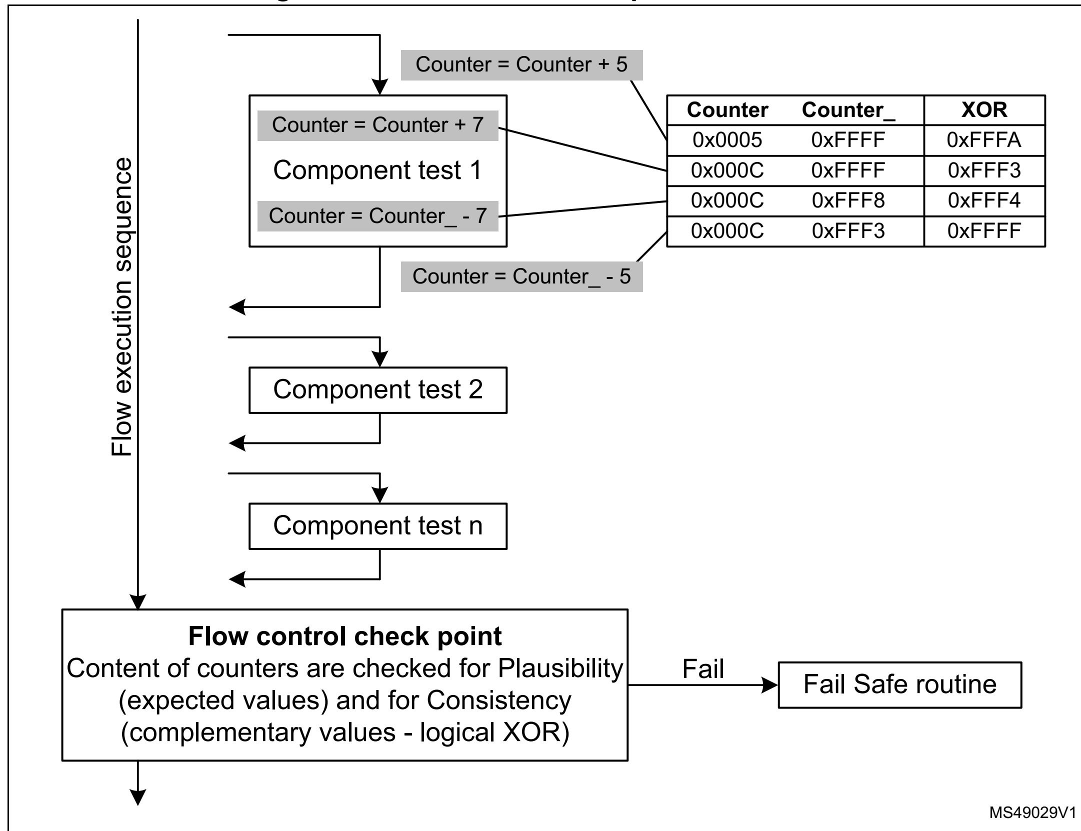
5.1.3 Class B flow control

Program flow control is a method highly recommended by the standards, because it's an efficient way of ensuring that all specific parts of code are correctly executed and passed.

A specific software method is used for this check. Unique labels (constant numbers) are defined for identifying all key points (blocks with component tests) in the code flow in order to make sure that no block is skipped and that all the flow is executed as expected. The unique labels are processed in two complementary counters complying with class B variable criteria. The main principle is a symmetrical four steps change of the counter pair content (adding or subtracting the unique label values) each time any significant testing block is processed. Two of these check steps are placed outside the called block at caller (main flow) level. This ensures that 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 are performed inside the called procedure to ensure that 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 controlled flow sequence and the four-step checking service is shown. This method decreases the load on CPU as all these points are always checked by counting one member of the complementary counter pair only. Because there is always the same number of call/return and entry/exit points, the values stored in the counter pair after each block is passed completely must be always complementary ones. 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 if they do not contain the expected values at any of these checkpoints, the Fail Safe routine is called.

Figure 2. Control flow four steps check routine



Counter	Counter_	XOR
0x0005	0xFFFF	0xFFFA
0x000C	0xFFFF	0xFFF3
0x000C	0xFFF8	0xFFF4
0x000C	0xFFF3	0xFFFF

1. For this example, the unique number for Component test 1 caller 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 completed.

5.2 Firmware package structure

This section describes how the ST solution is organized.

5.2.1 Projects and workspaces included in the package

Two projects have been prepared for each sub-family representative using different toolchains. The project for IAR C/C++ Compiler™ is done under IAR EWSTM8™ environment, while the project for Cosmic C-compiler is done as ST Visual Develop (STVD) project workspace. The corresponding Project.eww or Project.stw project file must be configured for specific STM8 device settings, and adequate workspace configuration before any compilation must be available.

5.2.2 Tools and other specifc controls of the library

Each project covers all settings and includes needs for both compilation and linking processes. User has to check the symbols defined for preprocessor at project settings as these symbols configure the project main structure and used features. The user can make detailed configurations at library parametric header file (see *Section 5.3*). Predefined linker script files **.icf* are included in IAR™ projects for different microcontrollers. Default linker script files generated by STVD projects are overwritten and replaced at the project directories dedicated for Cosmic solution.

Reset handler must be modified to perform *STL_StartUp* function after reset before standard C compiler initialization starts. Therefore startup assembly file *csstartup.s* (IAR™) must be modified in that way. For Cosmic, the reset handler reference is modified in the *stm8_interrupt_vector.c* file.

Caution: Be careful when invariable memory checksum setting is configured for the project, as it differs from compiler to compiler.
IAR C/C++ Compiler™ uses project option setting applied to its specific checksum card window of linker. The user has to ensure proper setting of the memory area under check, and avoid calculation running over the checksum pattern itself.
Some problems can raise during debug, implementation of break points or code optimization, too. All can lead to different memory content and corrupt the check sum calculation result. That is why the checksum result is ignored during debug, but verified at release time.
Cosmic C-compiler requires to define all the segments under checksum computation (by adding *-ck* parameter to each of them) and specific *checksum* segment keeping the checksum descriptor table must be included into the project and allocated at invariable memory additionally with applied *-ik* parameter.

Specific segments for Class B variables and checking stack overflow must be added into project and allocated at variable memory, too. Double storage in CLASS_B_RAM and CLASS_B_RAM_REV separated segments 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 their storage area is not checked during the transparent RAM test. The size and allocation of these segments can be modified in the project linker script file.

A new Class B variable must be declared as a complementary pair of two variables allocated at different segments by definition placed into proper part of the **stm32fxxx_STLclassBvar.h** header file.

The following syntax is used for the compilers:

Cosmic

```
EXTERN @near u16 MyClassBVariable;
....
EXTERN @near u16 MyClassBVariableInv;
```

IAR

```
EXTERN __near u16 MyClassBVariable @ "CLASS_B_RAM";
.....
EXTERN __near u16 MyClassBVariableInv @ "CLASS_B_RAM_REV";
```

5.2.3 Application examples

A short example of the self-test package integration in an application is attached in each project ***main.c*** file with respect to the package integration criteria (see *Section 6: Class B solution integration*). Except for some low-density STM8S devices, the examples are written to run on the corresponding STM8 evaluation boards (STM8S/128-EVAL and STM8L1528-EVAL) as Class B package demonstration firmware. They use on board LCD and LED diodes to display current versus expected master clock frequency measurement changes. Display or LED outputs can be disabled and enabled by editing the set of code conditional compilation defines in the project option dedicated to the compiler preprocessor. The main loop also performs initialization and calling all run mode tests at ordinary intervals.

5.3 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.

5.3.1 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 while adapting its preprocessor user-defined options. 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 compilation parameters to control their inclusion in comments. The full Class B package, even optimized and with disabled verbose messages, takes about 3.5 Kbytes of code memory. Disabling some library parts can be required for microcontrollers where the memory space is too small for the application needs, or when the tested parts are not included in the application (e.g. HSE testing).

The user must be careful, when modifying the initial / run time test flow, of possible corruption of the implemented procedures control flow. In this case, the values summarized in the 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 definition of these constants in an adequate way.

5.3.2 Verbose diagnostic mode

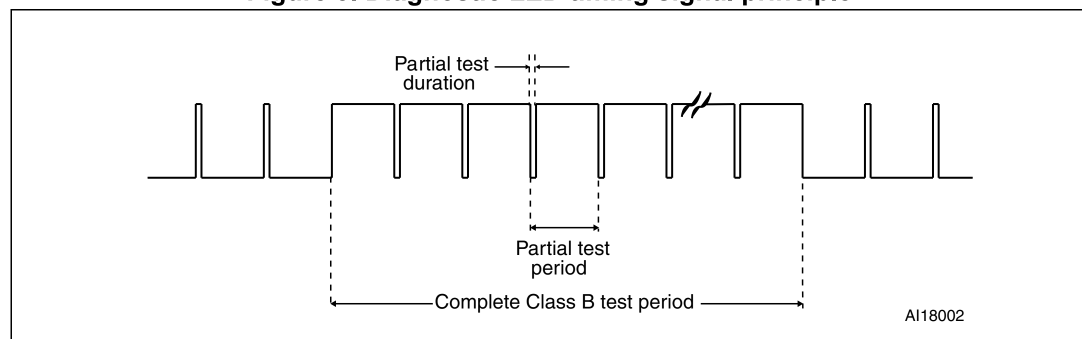
The Tx pin of dedicated UART interface is used to output text messages in verbose mode. This mode is useful in debug phase as the interface can be connected to an external terminal (the line setting is 115200 Bd, no parity, 8-bit data, 1 stop bit). However the text messages take too much code space in 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.

The complete list of verbose messages is provided in *Appendix B: List of verbose messages and codes reported at Fail Safe mode entry*.

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, that is, the loading time (see *Figure 3*). LED control can be disabled by the user in Class B configuration file ***stm8x_stl_param.h***.

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

Figure 3. Diagnostic LED timing signal principle



5.3.3 Debugging the package

While debugging the package, it is useful to disable:

- Reset in ***FailSafe()*** routine by servicing independent WDG,
- Verification of the program memory integrity test result 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 in STM8L10x devices.*

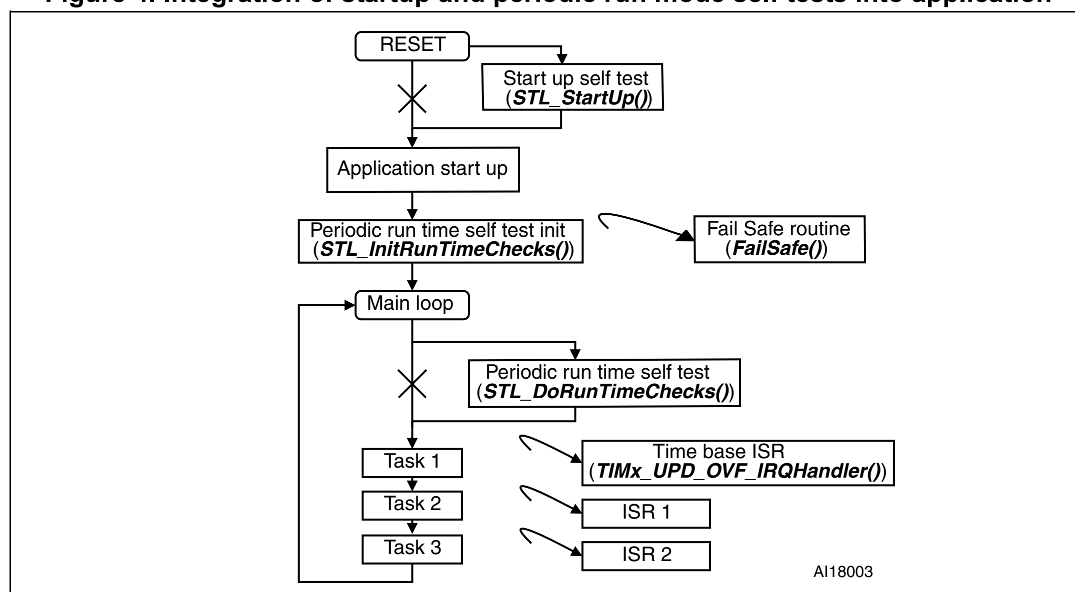
6 Class B solution integration

6.1 Integrating software into user application

Class B routines are divided into two main processes: periodic run mode self tests and startup 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 5.1.3: 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 5.1.2: 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 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 initialization procedure of C compiler 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



6.2 Detailed description of startup self tests

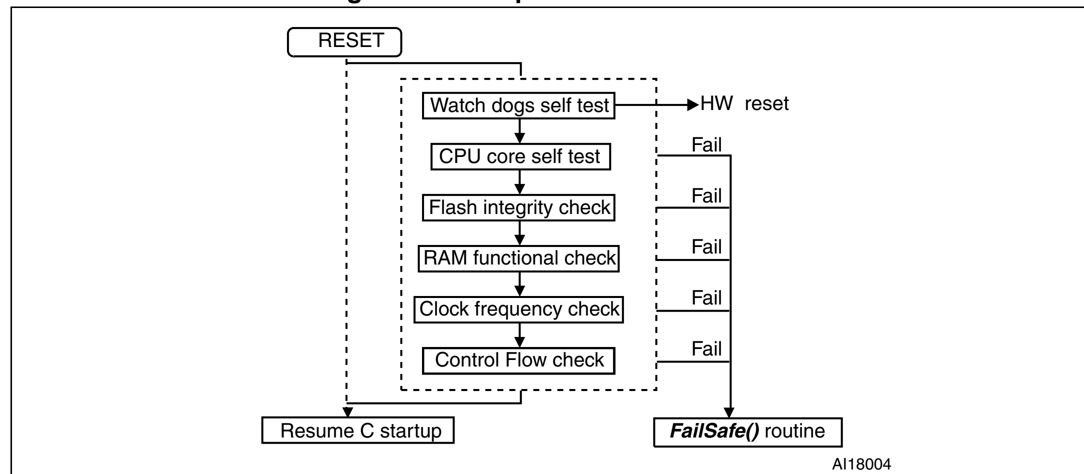
The startup self test is forced during initialization phase as the earliest checking process after resetting the microcontroller (see *Figure 4*) and before standard application startup routine. The user must force the reset vector to the first address in ***STL_StartUp()*** routine.

The startup tests block structure is shown in *Figure 5* and includes the following self tests:

- Watch dogs startup test
- CPU startup test
- Flash memory 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

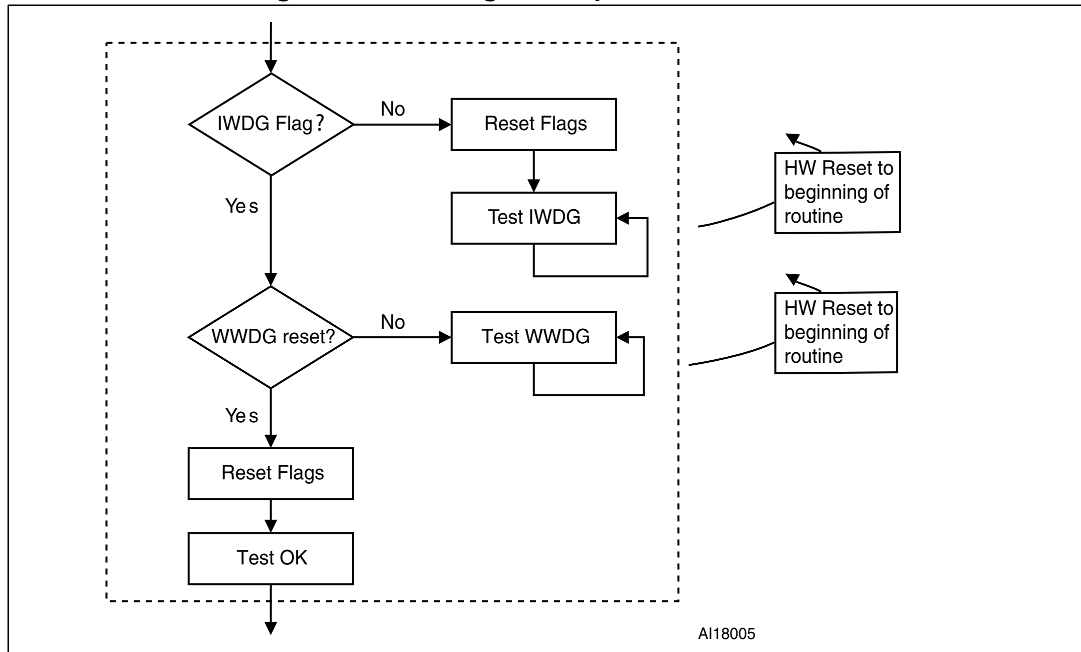


6.2.1 Watchdog startup self test

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.

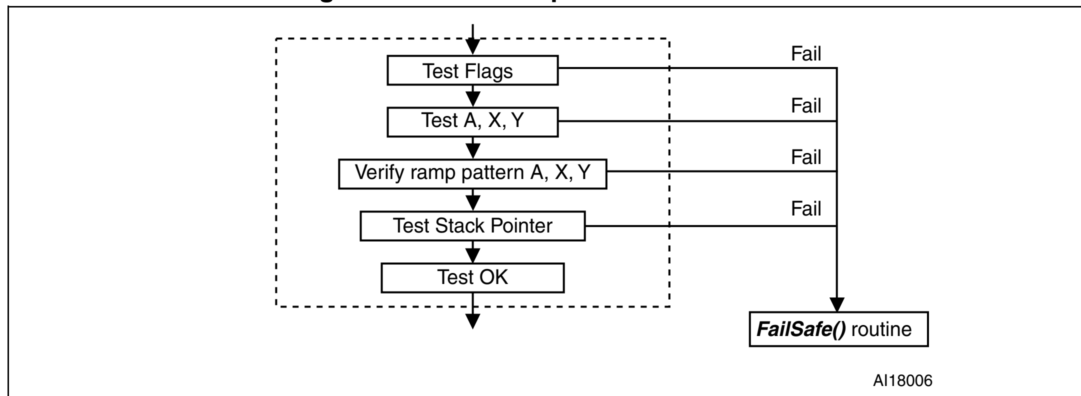
Figure 6. Watchdogs startup self test structure



6.2.2 CPU startup self test

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

Figure 7. CPU startup self test structure



6.2.3 Flash memory complete checksum self test

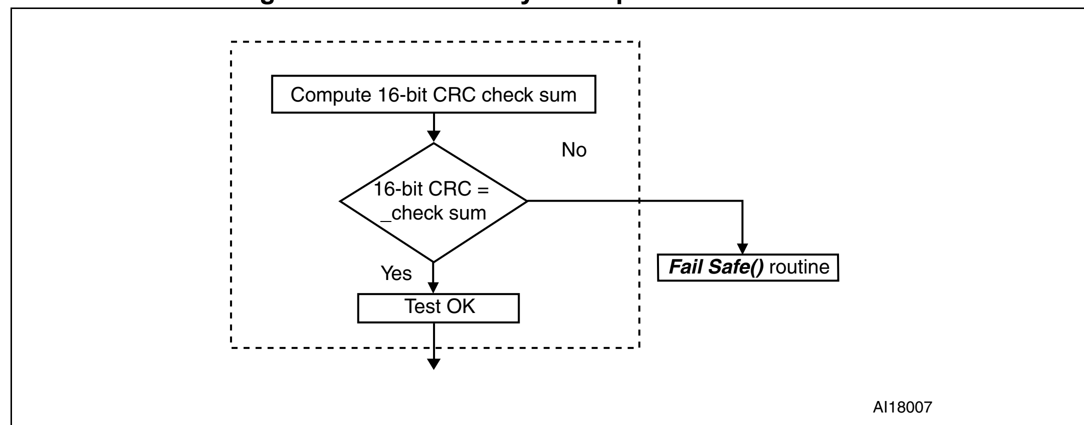
The CRC checksum computation is performed over the entire Flash memory space indicated by linker checksum structure. The resulting computation is compared with the linker result. ***FailSafe()*** routine is called if there is an error.

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 64 Kbytes of memory code, this is the simplest method.
- A 16-bit check sum calculation over the 16-bit address space can be used for checking up to 64 Kbytes 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 exceeds 64 Kbytes, 24-bit addressing must be used; this way uses the larger code space and is the most time-consuming.

The STM8 firmware includes six separate source 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 memory startup self test structure



6.2.4 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. 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

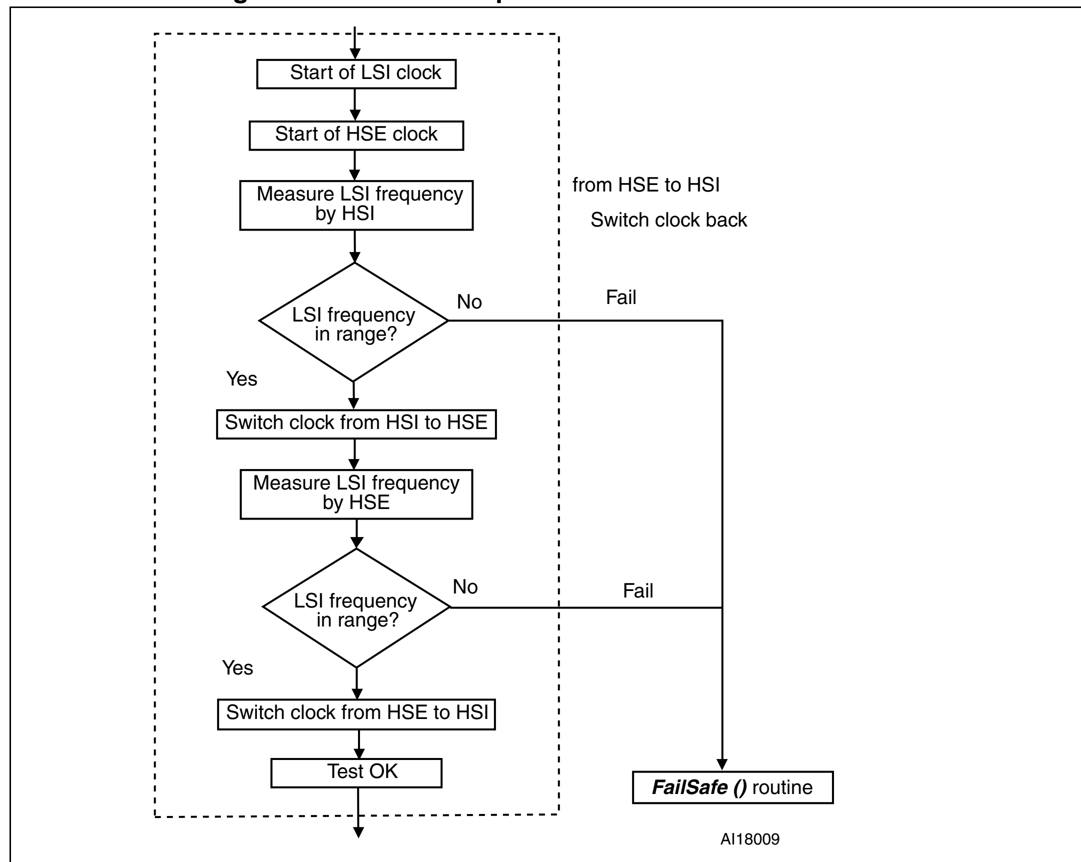
6.2.5 Clock startup self test

When the test begins, the low speed internal clock (LSI) source is started first, followed by 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 in 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 some tests, e.g. HSE test when no external oscillator is used.

Figure 10. Clock startup self test subroutine structure



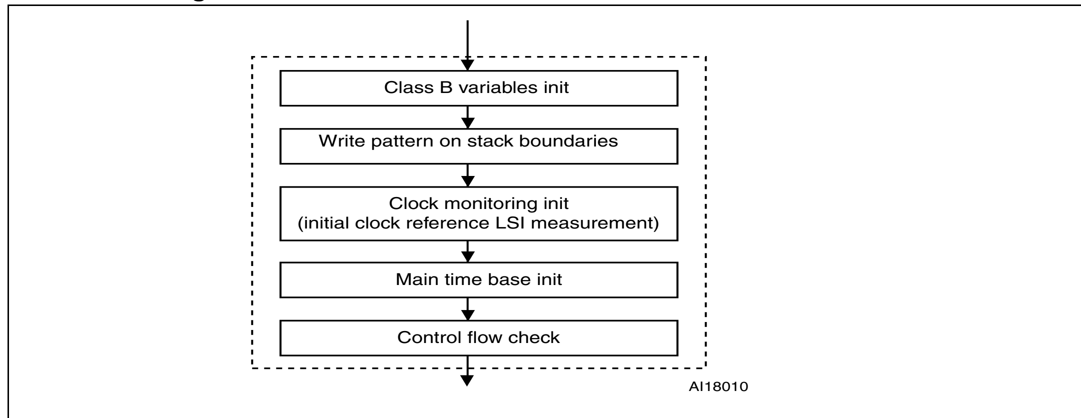
1. Low speed external (LSE) clock source can be started/checked at the beginning of the test and measured feeding the dedicated timer on STM8L15x devices. They can be inserted at the beginning of this routine.
2. High speed external (HSE) tests are skipped on STM8L10x devices.
3. LSI frequency is different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

6.3 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 so as 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



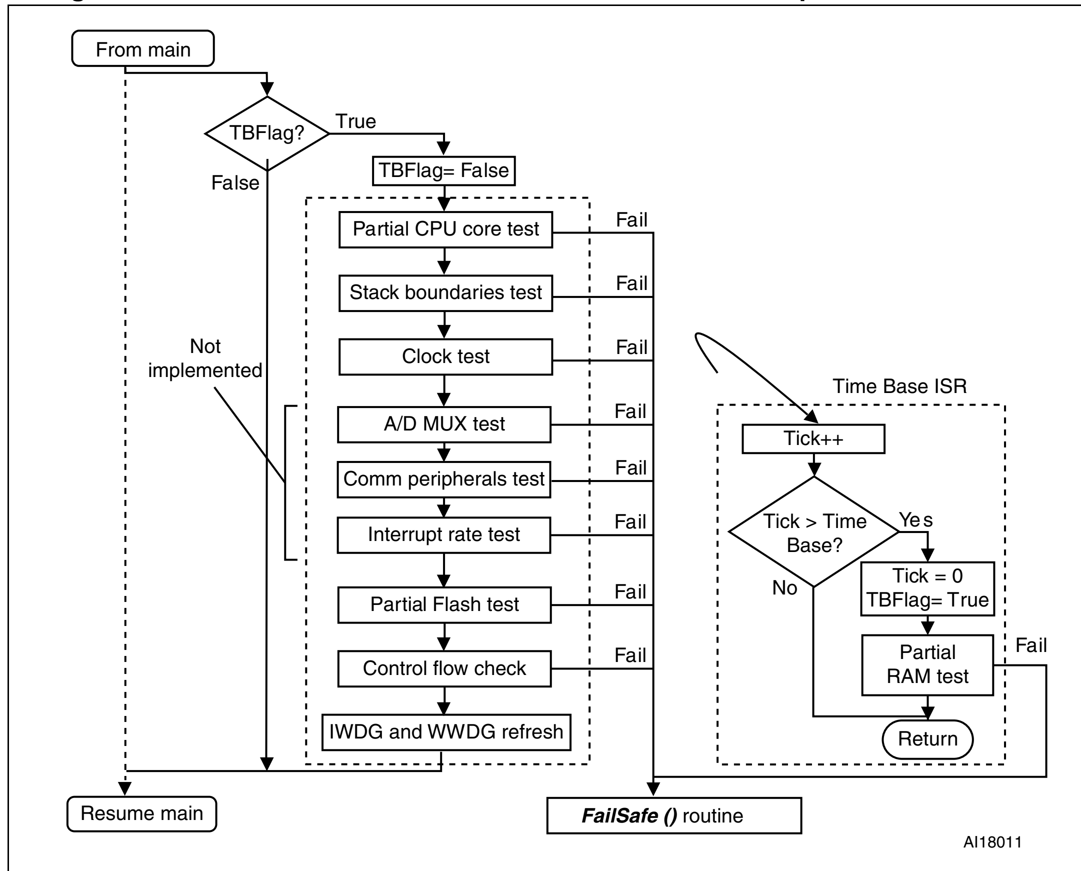
6.4 Detailed description of periodic run mode self tests

6.4.1 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 can 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 memory partial CRC test including evaluation of the complete test
- IWDG and WWDG refresh
- Partial transparent RAM March C-/X test (under system interrupt scope)

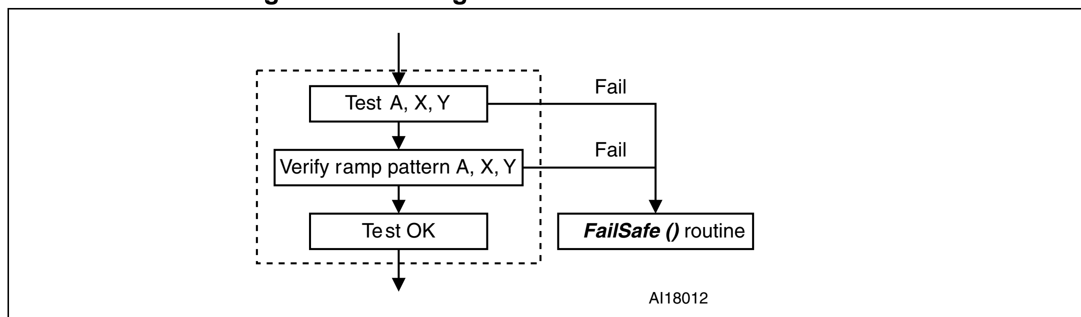
Figure 12. Periodic run mode self test and time base interrupt service structure



6.4.2 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

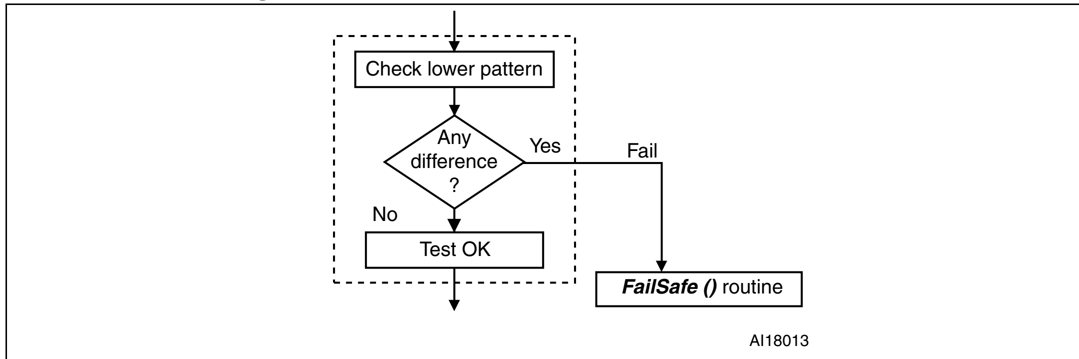


6.4.3 Stack boundaries run mode test

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



6.4.4 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 stored during initial run mode self tests. When the difference is larger than +/-25% ***FailSafe()*** routine is called. Occasional over captured measures are ignored (they can appear when a longer user interrupt service corrupts current measurement.

Figure 15. Clock run mode self test structure

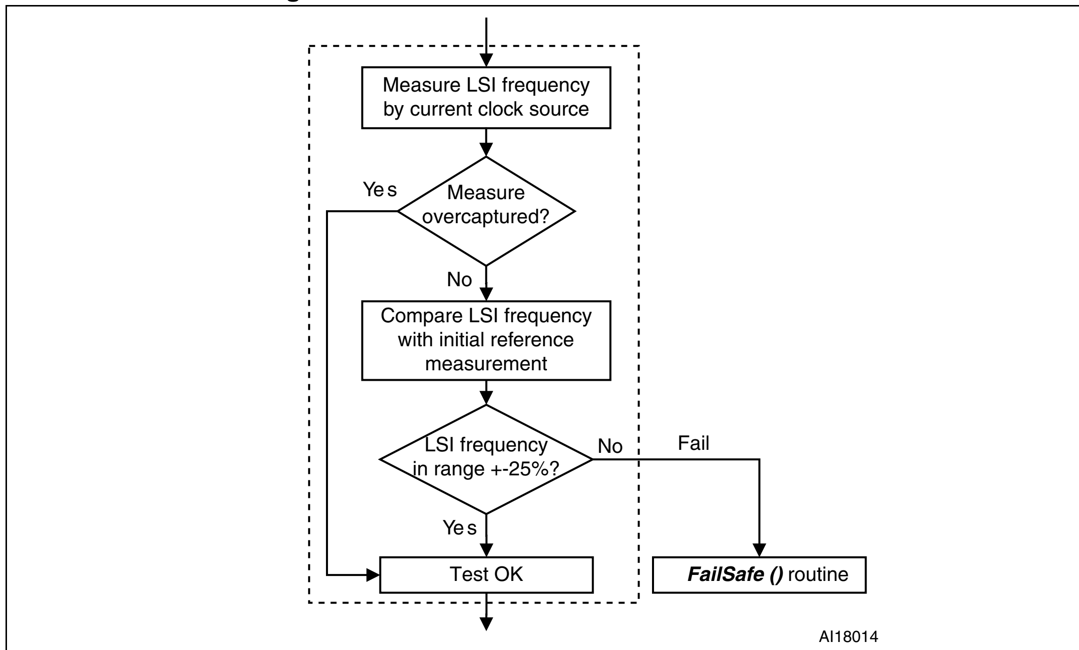
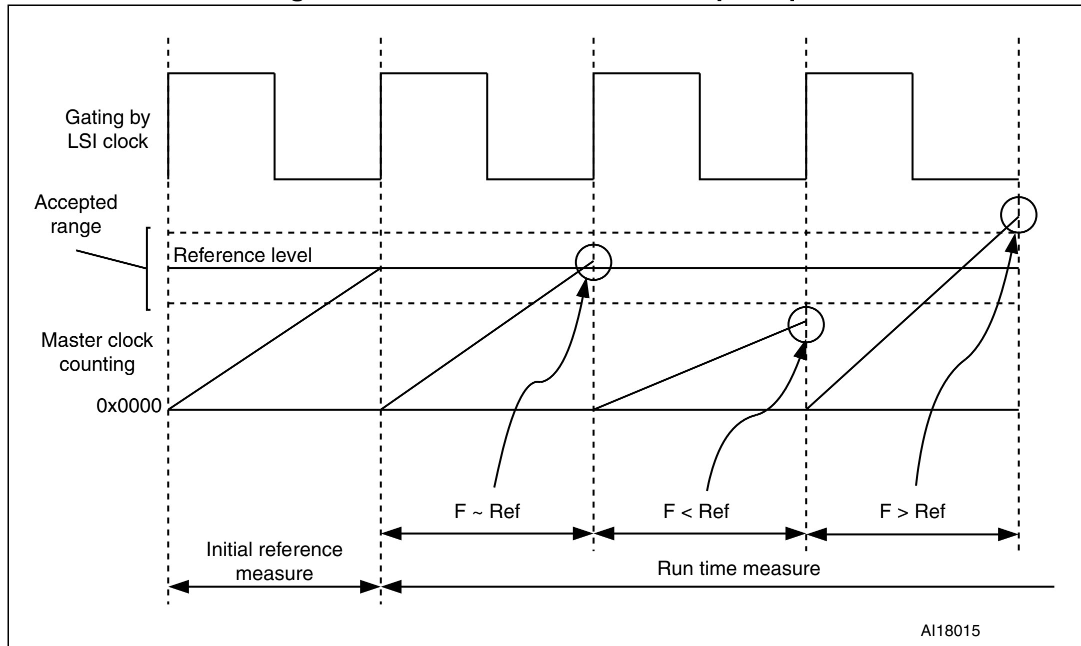


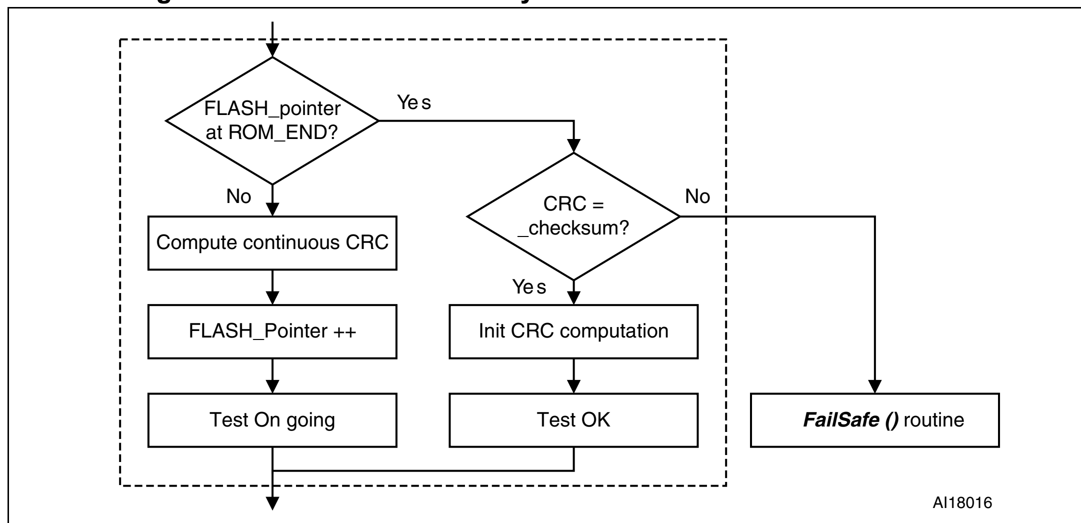
Figure 16. Clock run mode self test principle



6.4.5 Partial Flash memory 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 memory CRC run mode self test structure



1. *For more details about CRC calculation, refer to Section 6.2.3: Flash memory complete checksum self test.*

6.4.6 Watchdog service in run mode test

If the run mode service block is correctly completed, then both window and independent watchdogs (WDGs) are refreshed at the last step just before returning to the main loop. To correctly refresh watchdogs, 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.

6.4.7 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 11*). 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

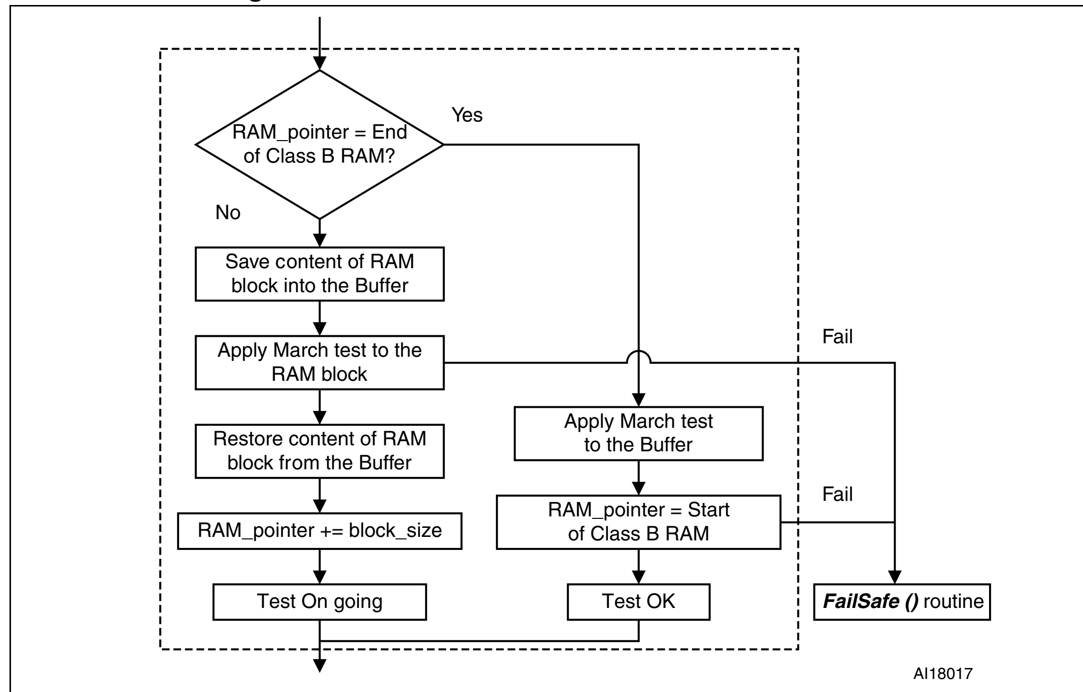


Figure 19. Fault coupling principle used in partial RAM run mode self test

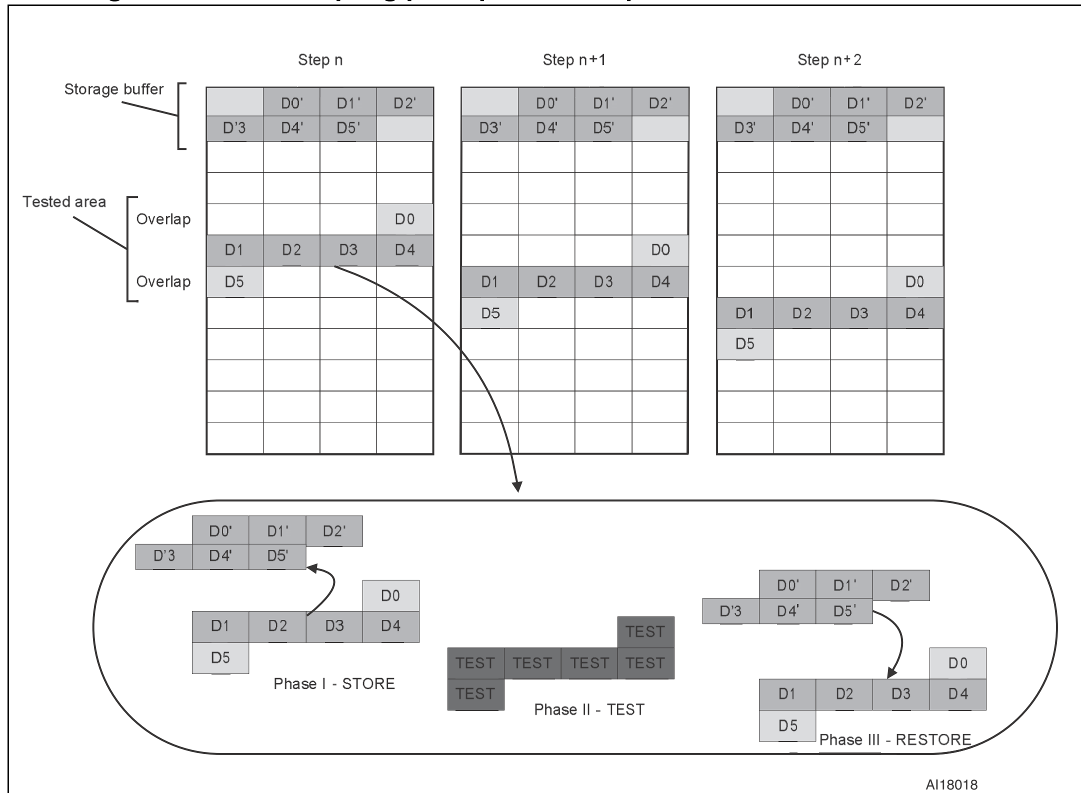


Table 11. March C- phases at RAM partial test

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 when March X algorithm is used.

Appendix A STM8 Class B firmware package variations

Table 12. STM8 Class B firmware packages

Test	STM8S/A package	Medium density STM8L/AL package	Low density STM8L/AL/TL package
Optional UART verbose mode	Yes	Yes	No(1)
Demo mode with optional LCD screen	Yes	Yes	No
Window watchdog test	Yes	Yes	No(2)
High speed external (HSE) clock test	Yes	Yes	No
Low speed internal (LSI) frequency / Number of LSI periods used for clock tests (measurement)	128 kHz / 4	38 kHz / 1	38 kHz / 1
Stack space at the top of RAM [bytes]	1024 / 512(3)	513	513(4)

1. Errors are signaled by error codes passed to *FailSafe()* routine.
2. Window watchdog is available on STM8TL5x devices only.
3. The stack is limited for some STM8S Access line devices and STM8A devices with up to 32 Kbytes non-volatile memory; for correct values see the corresponding datasheets.
4. Stack is not limited for STM8TL5x devices; there is rollover (due to over/underflow) only if the stack overlaps the 4 Kbytes.

Appendix B List of verbose messages and codes reported at Fail Safe mode entry

Table 13. Verbose messages and unique codes reported at Fail Safe mode entry

STL block	Tested module	Error code (hex)	Verbose message
Start up	CPU	00	Start-up CPU Test Failure
	Flash	01	FLASH 8/16-bit CRC Error at Start-up
	Flow control	02	Control Flow Error (Start up 1)
	RAM	03	RAM Test Failure
	Clock	04	HSI clock source failure
		05	LSI start-up failure
		06	HSE start-up failure
		07	Clock switch failure
		08	EXT clock source failure
		09	Abnormal Clock Test routine termination (POR)
	Flow control	0A	Control Flow Error (Start-up 2)
Run time	Clock	10	Abnormal Clock Test routine termination (main init)
	Flow control	11	Control Flow Error (Main init)
	CPU	12	Run Time CPU Test Failure
	Stack control	13	Stack overflow
	Clock	14	EXT clock frequency error (clock test)
		15	System clock frequency error (clock test)
		16	Class B variable error (clock test)
		17	Abnormal Clock test routine termination (main)
	Flash	18	Run-time FLASH CRC Error
	Flow control	19	Control Flow Error (main loop, Flash CRC)
		1A	Control Flow Error (main loop, Flash CRC on-going)
		1B	Control Flow Error (main loop)
	Variable integrity	1C	Class B variable error (time base check)
IT system	Clock	20	Clock Source failure (Clock Security System)
	RAM	21	RAM Error (March Run-time check)
	Flow control	22	Control Flow Error (March Run-time check)
		23	Control Flow Error (ISR)
	Variable integrity	24	Class B Error on Tick Counter

Revision history

Table 14. Revision history

Date	Revision	Description of changes
01-Jun-2010	1	Initial release
29-Nov-2012	2	Modified *Introduction* and document throughout to include all new members of the STM8 family. Added *Section 1: Package overview* and *Table 1: Applicable products*. Modified *Table 3: Overview of methods used in micro-specific tests* and *Appendix A: STM8 Class B firmware package variations*.
07-Mar-2016	3	Updated document title, *Introduction* and *Section 4: Compliance with IEC, UL and CSA standards*. Removed former *Table 1: Applicable products*. Minor text changes across the whole document.
26-Feb-2018	4	Updated document title. Updated *Introduction*, *Section 1: Package overview*, *Section 4: Compliance with IEC, UL and CSA standards*, *Section 4.1: Generic tests included in STL firmware package*, *Section 4.2: Application specific tests*, *Section 4.2.1: Analog signals*, *Section 4.2.2: Digital I/Os*, *Section 4.2.3: Interrupts and external communication*, *Section 4.2.4: Timing and program flow*, *Section 4.3: Safety life cycle*, *Section 5: Class B software package*, *Section 5.1: Common software principles*, *Section 5.1.1: Fail Safe mode*, *Section 5.1.2: Class B variables*, *Section 5.1.3: Class B flow control*, *Section 5.2: Firmware package structure*, *Section 5.2.1: Projects and workspaces included in the package*, *Section 5.2.2: Tools and other specifc controls of the library*, *Section 5.2.3: Application examples*, *Section 5.3.1: Configuration control*, *Section 5.3.3: Debugging the package* and *Section 6: Class B solution integration*. Added *Section 2: Package structure overview*, *Section 3: Main differences from the product point of view* and their subsections. Added *Appendix B: List of verbose messages and codes reported at Fail Safe mode entry*. Updated title of *Table 4: Overview of common tool specific STL procedures*, *Table 6: STM8 compatibility aspects*, *Table 10: Methods used in micro specific tests of associated ST package* and *Table 12: STM8 Class B firmware packages*. Updated *Figure 1: Example of RAM memory configuration* and *Figure 2: Control flow four steps check routine*.

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.

© 2018 STMicroelectronics – All rights reserved