Introduction

The role of safety is more and more important in electronic applications. The level of safety requirements for components is steadily increasing and the manufacturers of electronic devices include many new technical solutions in their designs. Software techniques for improving safety are continuously being developed. The standards related to safety requirements for hardware and software are under continuous development as well.

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. Compliance, verification and certification are the focus of 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 the associated software X-CUBE-CLASSB is to facilitate and accelerate user software development and certification processes for applications based on STM32 32-bit Arm® Cortex® microcontrollers subject to these requirements and certifications.

The safety package (self test library, or STL) collects a set of common tests dedicated mainly to generic blocks of STM32 microcontrollers. The STL set is based on the unique STM32Cube interface with specific HAL (hardware abstraction layer) services and drivers published by ST for dedicated STM32 products. Differences are covered by product specific tests and added settings (e.g. CPU core, RAM design, clock control).

The user can include both the STL package and dedicated HAL drivers into a final customer project, together with additional product specific tests and settings. Implementation examples of the STL package are available for specific products of the mainstream STM32F0, STM32F1, STM32F3, STM32G0 and STM32G4, performance STM32F2, STM32F4, STM32F7 and STM32H7, low power STM32L0, STM32L1 and STM32L4 and wireless STM32WB Series. Specific projects (IAR™-EWARM, Keil® MDK-Arm® and Ac6 Eclipse™ or GCC STM32CubeIDE environment and toolchains) are included for each example, built upon a dedicated ST evaluation board.

The common part of STL package can be reused for any other microcontroller of the STM32 family due to the unique Cube interface to the HAL services.

The user has to understand that the STL package is pre-certified for methodology and techniques used. While the provided examples show how to integrate the STL package and the associated FW (HAL drivers) in the application, the final implementation and functionality always has to be verified by the certification body at the application level.

Note: STMicroelectronics develops derivative firmware supporting new products step by step. Contact your local ST sales office to get support and the latest information about available examples.
Contents

1       Reference documents .................................................. 6

2       Package variation overview ............................................. 7

3       Main differences between STL packages from product point of view 10
  3.1    CPU tests ................................................................. 12
  3.2    Clock tests and time base interval measurement .................. 12
  3.3    SRAM tests .............................................................. 12
  3.4    Flash memory integrity tests ........................................ 14
  3.5    Start-up and system initialization .................................. 15
  3.6    Firmware configuration parameters .................................. 16
  3.7    Firmware integration .................................................. 19
  3.8    HAL driver interface .................................................. 19
  3.9    Incompatibility with previous versions of the STL ............... 20

4       Compliance with IEC, UL and CSA standards ......................... 22
  4.1    Generic tests included in STL firmware package ................. 24
  4.2    Application specific tests not included in ST firmware self test library ... 26
    4.2.1  Analog signals ...................................................... 26
    4.2.2  Digital I/Os .......................................................... 27
    4.2.3  Interrupts ............................................................. 28
    4.2.4  Communication ....................................................... 28
  4.3    Safety life cycle ...................................................... 28

5       Class B software package ............................................... 30
  5.1    Common software principles used .................................... 30
    5.1.1  Fail safe mode ....................................................... 30
    5.1.2  Safety related variables and stack boundary control ........... 30
    5.1.3  Flow control procedure ............................................ 32
  5.2    Tool specific integration of the library ............................ 33
    5.2.1  Projects included in the package .................................. 33
    5.2.2  Start-up file .......................................................... 34
    5.2.3  Defining new safety variables and memory areas under check .... 34
5.2.4 Application implementation examples ........................................... 35
5.3 Execution timing measurement and control .................................. 36
5.4 Package configuration and debugging ........................................... 41
  5.4.1 Configuration control ................................................................. 41
  5.4.2 Verbose diagnostic mode ............................................................ 42
  5.4.3 Debugging the package ............................................................... 42

6   Class B solution structure ................................................................. 44
  6.1 Integration of the software into the user application ...................... 44
  6.2 Description of start-up self tests ................................................ 47
    6.2.1 CPU start-up self test ............................................................. 48
    6.2.2 Watchdog start-up self test .................................................... 49
    6.2.3 Flash memory complete check sum self test .................................. 50
    6.2.4 Full RAM March-C self test ..................................................... 50
    6.2.5 Clock start-up self test ............................................................ 51
    6.2.6 Control flow check ................................................................. 52
  6.3 Periodic run time self tests initialization .................................... 52
  6.4 Description of periodic run time self tests ................................... 53
    6.4.1 Run time self tests structure .................................................. 53
    6.4.2 CPU light run time self test .................................................... 54
    6.4.3 Stack boundaries runtime test ................................................ 55
    6.4.4 Clock run time self test .......................................................... 55
    6.4.5 Partial Flash CRC run time self test ....................................... 56
    6.4.6 Watchdog service in run time test ......................................... 57
    6.4.7 Partial RAM run time self test ................................................ 57

7   Revision history ................................................................................. 61
List of tables

Table 1. Overview of STL packages ......................................................... 7
Table 2. Organization of the FW structure .............................................. 7
Table 3. Used IDEs and toolchains ........................................................... 8
Table 4. Structure of the common STL packages ...................................... 8
Table 5. Structure of the product specific STL packages .............................. 9
Table 6. Integration support files .............................................................. 9
Table 7. Compatibility between different STM32 microcontrollers ............... 11
Table 8. How to manage compatibility aspects and configure STL package ............................ 17
Table 9. Overview of HAL drivers used by STL stack procedures ...................... 19
Table 10. MCU parts that must be tested under Class B compliance .................. 23
Table 11. Methods used in micro specific tests of associated ST package ....... 25
Table 12. Signals used for timing measurements ....................................... 39
Table 13. Comparison of results ............................................................. 40
Table 14. Possible conflicts of the STL processes with user SW .................... 45
Table 15. Physical order of RAM addresses organized into blocks of 16 words .......... 50
Table 16. March C phases at RAM partial test ......................................... 60
Table 17. Document revision history ....................................................... 61
List of figures

Figure 1. Example of RAM memory configuration .................................................... 31
Figure 2. Control flow four steps check principle ................................................... 33
Figure 3. Diagnostic LED timing signal principle ...................................................... 36
Figure 4. Typical test timing during start-up ............................................................. 37
Figure 5. Typical test timing during run time ............................................................ 38
Figure 6. STM32 demo hyper terminal output window in verbose mode .................. 42
Figure 7. Integration of start-up and periodic run time self tests into application .... 44
Figure 8. start-up self tests structure ................................................................. 47
Figure 9. CPU start-up self test structure .............................................................. 48
Figure 10. Watchdogs start-up self test structure .................................................... 49
Figure 11. Flash start-up self test structure ............................................................ 50
Figure 12. RAM start-up self test structure ............................................................ 51
Figure 13. Clock start-up self test subroutine structure .......................................... 52
Figure 14. Periodic run time self test initialization structure ..................................... 53
Figure 15. Periodic run time self test and time base interrupt service structure ........ 54
Figure 16. CPU light run time self test structure ..................................................... 54
Figure 17. Stack overflow run time test structure ................................................... 55
Figure 18. Clock run time self test structure ......................................................... 56
Figure 19. Partial Flash CRC run time self test structure ....................................... 56
Figure 20. Partial RAM run time self test structure ............................................... 58
Figure 21. Partial RAM run time self test - Fault coupling principle (no scrambling) .... 59
Figure 22. Partial RAM run time self tests - Fault coupling principle (with scrambling) .... 59
1 Reference documents

Several ST documents can be used when applying or modifying the STL stack or when developing a new one, and complete testing report can be provided upon request.

Specific safety manuals for STM32 products (based on Arm\(^a\) cores) are available or in preparation, where compliance aspects with other safety standards are provided. Application notes describing specific methods to control peripherals or to ensure system electromagnetic compatibility (EMC) against noise emission and noise sensitivity are available on www.st.com.

For more information about errors handling techniques refer to *Handling of soft errors in STM32 applications* (AN4750).

For more information on EMC refer to the following application notes:

- *Software techniques for improving microcontroller EMC performance* (AN1015)
- *EMC design guide* (AN1709)

For more detailed information about cyclic redundancy check calculation (CRC) refer to *Using the CRC peripheral in STM32 family* (AN4187).

The following safety manuals are available on www.st.com:

- UM1741 (for the F0 Series)
- UM1814 (for the F1 Series)
- UM1845 (for the F2 Series)
- UM1846 (for the F3 Series)
- UM1840 (for the F4 Series)
- UM2318 (for the F7 Series)
- UM2455 (for the G0 Series)
- UM2454 (for the G4 Series)
- UM2331 (for the H7 Series)
- UM2037 (for the L0 Series)
- UM1813 (for the L1 Series)
- UM2305 (for the L4 and L4+ Series)

The development of safety manuals for other Series is an ongoing process. Contact your local FAE or the nearest ST office to check for the availability of new documents.

\(a\). Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
2 Package variation overview

The STL packages and included HAL FW are summarized in Table 1.

Table 1. Overview of STL packages

<table>
<thead>
<tr>
<th>STM32 Series</th>
<th>HAL driver</th>
<th>CMSIS driver</th>
<th>Common STL stack Specific test(1)</th>
<th>Included projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>Rev. 1.5.0</td>
<td>Rev. 2.3.1</td>
<td></td>
<td>SMT32052B-EVAL</td>
</tr>
<tr>
<td>F1</td>
<td>Rev. 1.1.1</td>
<td>Rev. 4.2.0</td>
<td></td>
<td>STM3210C-EVAL</td>
</tr>
<tr>
<td>F2</td>
<td>Rev. 1.2.1</td>
<td>Rev. 2.2.0</td>
<td></td>
<td>STM322xG_EVAL</td>
</tr>
<tr>
<td>F3</td>
<td>Rev. 1.4.0</td>
<td>Rev. 2.3.1</td>
<td>Rev. 2.2.0</td>
<td>STMT324xG_EVAL</td>
</tr>
<tr>
<td>F4</td>
<td>Rev. 1.7.1</td>
<td>Rev. 2.6.1</td>
<td></td>
<td>STMT3273C-EVAL</td>
</tr>
<tr>
<td>F7</td>
<td>Rev. 1.2.2</td>
<td>Rev. 1.2.0</td>
<td></td>
<td>STMT32756G-EVAL</td>
</tr>
<tr>
<td>L0</td>
<td>Rev. 1.8.1</td>
<td>Rev. 1.7.1</td>
<td></td>
<td>STMT32L0xx_Nucleo</td>
</tr>
<tr>
<td>L1</td>
<td>Rev. 1.3.0</td>
<td>Rev. 2.2.1</td>
<td></td>
<td>STMT32L152D-EVAL</td>
</tr>
<tr>
<td>L4</td>
<td>Rev. 1.7.1</td>
<td>Rev. 1.3.1</td>
<td></td>
<td>STMT32L476G-EVAL</td>
</tr>
<tr>
<td>H7</td>
<td>Rev. 1.5.0</td>
<td></td>
<td>Rev. 2.3.0</td>
<td>STMT32734I-EVAL</td>
</tr>
<tr>
<td>G0</td>
<td>Rev. 1.2.0</td>
<td></td>
<td></td>
<td>STMT32G081B-EVAL</td>
</tr>
<tr>
<td>G4</td>
<td>Rev. 1.0.0</td>
<td></td>
<td></td>
<td>STMT32G474RE_NUCLEO</td>
</tr>
<tr>
<td>WB</td>
<td>Rev. 1.1.0</td>
<td></td>
<td></td>
<td>P-NUCLEO-WB55</td>
</tr>
</tbody>
</table>

1. There is no difference between Rev. 2.2.0 and Rev. 2.3.0. The projects associated with STL stack Rev 2.3.0 has been tested with the latest available revisions of the IDEs and compilers only. When user applies these tools on project associated with STL stack Rev 2.0, a small modification of the product specific STL source files is required, as described in Section 3.9.

The firmware uses a common structure of directories. It is based on available set of drivers either dedicated to a given product, or associated with specific HW development tools. Part of that is common with the whole STM32 family and ST debug support.

The basic structure is detailed in Table 2, where self test procedures and methods targeting the Class B requirements are collected under common STL stack and product specific STL stack directories. The remaining drivers are mostly application specific, and are subject to change or replacement in the final customer project, in accordance with user application HW.

Table 2. Organization of the FW structure

<table>
<thead>
<tr>
<th>Directory</th>
<th>Drivers</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drivers</td>
<td>BSP</td>
<td>Evaluation board specific drivers</td>
</tr>
<tr>
<td></td>
<td>CMSIS</td>
<td>Core specific drivers</td>
</tr>
<tr>
<td></td>
<td>HAL</td>
<td>Product specific peripheral drivers</td>
</tr>
<tr>
<td>Utilities</td>
<td>CPU, Fonts, Log</td>
<td>Common debug/development support</td>
</tr>
<tr>
<td>Middleware</td>
<td>Common STL stack</td>
<td>Common STM32 STL procedures</td>
</tr>
</tbody>
</table>
The included projects for specific STM32 products and dedicated evaluation boards have been prepared and tested with the environments and toolchains detailed in Table 3.

Table 3. Used IDEs and toolchains

<table>
<thead>
<tr>
<th>IDE</th>
<th>STL Rev. 2.2.0</th>
<th>STL Rev. 2.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAR™ EWARM</td>
<td>Rev. 7.80.4</td>
<td>Rev. 8.32.4</td>
</tr>
<tr>
<td>Keil® MDK-Arm®</td>
<td>Rev. 5.23</td>
<td>Rev. 5.27</td>
</tr>
<tr>
<td>Ac6 Eclipse™</td>
<td>Rev. 1.13.1</td>
<td>Rev. 1.17.0</td>
</tr>
<tr>
<td>STM32CubeIDE</td>
<td>-</td>
<td>Rev. 1.0.0</td>
</tr>
</tbody>
</table>

The detailed structure of these projects and the list of files included in the common and specific parts of STL stack are summarized in Table 4 and Table 5, respectively. Additional supporting files used in the examples are listed in Table 6.

Table 4. Structure of the common STL packages

<table>
<thead>
<tr>
<th>STL</th>
<th>Common STL stack source files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>File</td>
</tr>
<tr>
<td>Start-up test</td>
<td>stm32xx_STLstartup.c</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLclockstart.c</td>
</tr>
<tr>
<td>Run time test</td>
<td>stm32xx_STLmain.c</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLclockrun.c</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLcrc32Run.c</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLtranspRam.c</td>
</tr>
<tr>
<td>Headers</td>
<td>stm32xx_STLclassBvar.h</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLlib.h</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLstartup.h</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLmain.h</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLclock.h</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLcpu.h</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLcrc32.h</td>
</tr>
<tr>
<td></td>
<td>stm32xx_STLRam.h</td>
</tr>
</tbody>
</table>
### Table 5. Structure of the product specific STL packages

<table>
<thead>
<tr>
<th>STL</th>
<th>Product specific STL stack source and header files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Source files: start-up and run time CPU and RAM tests written in Assembler for IAR™, Keil® and Ac6也有各种不同版本的文件供选择。</td>
</tr>
<tr>
<td>Header</td>
<td>Header file: STL product specific configuration file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stm32xxxx_STLcpustartIAR.s</td>
<td>Start-up time CPU and RAM tests written in Assembler for IAR™, Keil® and Ac6。</td>
</tr>
<tr>
<td>stm32xxxx_STLcpurunIAR.s</td>
<td>Start-up time CPU and RAM tests written in Assembler for IAR™, Keil® and Ac6。</td>
</tr>
<tr>
<td>stm32xxxx_STLRamMcMxIAR.s</td>
<td>Start-up time CPU and RAM tests written in Assembler for IAR™, Keil® and Ac6。</td>
</tr>
<tr>
<td>stm32xxxx_STLcpustartKEIL.s</td>
<td>Start-up time CPU and RAM tests written in Assembler for IAR™, Keil® and Ac6。</td>
</tr>
<tr>
<td>stm32xxxx_STLcpurunKEIL.s</td>
<td>Start-up time CPU and RAM tests written in Assembler for IAR™, Keil® and Ac6。</td>
</tr>
<tr>
<td>stm32xxxx_STLRamMcMxKEIL.s</td>
<td>Start-up time CPU and RAM tests written in Assembler for IAR™, Keil® and Ac6。</td>
</tr>
<tr>
<td>stm32xxxx_STLcpustartGCC.s</td>
<td>Start-up time CPU and RAM tests written in Assembler for IAR™, Keil® and Ac6。</td>
</tr>
<tr>
<td>stm32xxxx_STLcpurunGCC.s</td>
<td>Start-up time CPU and RAM tests written in Assembler for IAR™, Keil® and Ac6。</td>
</tr>
<tr>
<td>stm32xxxx_STLRamMcMxGCC.s</td>
<td>Start-up time CPU and RAM tests written in Assembler for IAR™, Keil® and Ac6。</td>
</tr>
</tbody>
</table>

### Table 6. Integration support files

<table>
<thead>
<tr>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startup_stm32xxxxxIAR.s</td>
<td>C start-up for IAR™ compiler</td>
</tr>
<tr>
<td>startup_stm32xxxxxKEIL.s</td>
<td>C start-up for Arm® compiler</td>
</tr>
<tr>
<td>startup_stm32xxxxxGCC.s</td>
<td>C start-up for Ac6 compiler</td>
</tr>
<tr>
<td>main.c</td>
<td>Main flow of the example source</td>
</tr>
<tr>
<td>stm32xxxx_hal_msp.c</td>
<td>Application specific HAL drivers initialization</td>
</tr>
<tr>
<td>stm32xxxx_it.c</td>
<td>STL Interrupts, clock measurement processing and configuration procedures</td>
</tr>
<tr>
<td>main.h</td>
<td>Main flow header</td>
</tr>
<tr>
<td>stm32xxxx_hal_conf.h</td>
<td>HAL drivers configuration file</td>
</tr>
<tr>
<td>stm32xxxx_it.h</td>
<td>ISR header</td>
</tr>
</tbody>
</table>
Main differences between STL packages from product point of view

Users can find some small differences, mainly due to hardware deviations of the products, and to incompatibilities of compilers and debugging tools.

The main differences are due mainly to compatibility aspects between different STM32 products, all based on Arm® cores.

These differences, summarized in Table 7, are described in this section.
## Table 7. Compatibility between different STM32 microcontrollers

<table>
<thead>
<tr>
<th>STM32 Series</th>
<th>Mainstream</th>
<th>Performance</th>
<th>Low-power</th>
<th>Wireless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F0</td>
<td>F1</td>
<td>F3</td>
<td>G0</td>
</tr>
<tr>
<td>Arm® core</td>
<td>Cortex® M0</td>
<td>Cortex® M3</td>
<td>Cortex® M4</td>
<td>Cortex® M0+</td>
</tr>
<tr>
<td>Technology</td>
<td>180 nm</td>
<td>180 nm</td>
<td>90 nm</td>
<td>90 nm</td>
</tr>
<tr>
<td>Frequency</td>
<td>48 MHz</td>
<td>24-72 MHz</td>
<td>120 MHz</td>
<td>64 MHz</td>
</tr>
<tr>
<td>Performance</td>
<td>38 DMIPS</td>
<td>61 DMIPS</td>
<td>150 DMIPS</td>
<td>59 DMIPS</td>
</tr>
<tr>
<td>Flash memory</td>
<td>16-128 KB</td>
<td>16-1024 KB</td>
<td>128-1024 KB</td>
<td>16-512 KB</td>
</tr>
<tr>
<td>ECC on Flash memory</td>
<td>No</td>
<td>No</td>
<td>Yes(1)</td>
<td>Yes</td>
</tr>
<tr>
<td>CRC configurable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>RAM</td>
<td>4-16 KB</td>
<td>4-96 KB</td>
<td>64-128 KB</td>
<td>8-128 KB</td>
</tr>
<tr>
<td>RAM parity(2)/scrambling</td>
<td>Yes/Yes</td>
<td>No/Yes</td>
<td>No/No</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Auxiliary RAM</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Data EEPROM</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EEPROM ECC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IWDG window option</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Clock system(3)</td>
<td>HSI14, HSI48 (LSI~40 kHz)</td>
<td>(LSI~32 kHz)</td>
<td>(LSI~32 kHz)</td>
<td>HSI48 (LSI~40 kHz)</td>
</tr>
<tr>
<td>Clock cross ref measurement(4)</td>
<td>TIM14/Ch1</td>
<td>TIM5/Ch4(1)</td>
<td>TIM5/Ch4(1)</td>
<td>TIM16 Ch1</td>
</tr>
<tr>
<td>Clock reference next option</td>
<td>GPIO, RTC, HSE/32, MCO</td>
<td>-</td>
<td>GPIO, RTC, HSE/32, MCO</td>
<td>LSE, HSE/32, MCO</td>
</tr>
<tr>
<td>Voltage scaling management</td>
<td>Yes(1)</td>
<td>No</td>
<td>Yes(1)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1. Available on some products only.
2. When product features SRAM parity bit, address is included, except the STM32F0 Series where parity is computed only over data.
3. Parity bit does not cover the whole RAM, but only a portion of it. Check the product datasheet for detailed info.
4. Embedded RAM features ECC.
5. All the family members feature HSI16, HSE, PLL, LSI and LSE clock sources. Additional sources are listed in the table.
6. Timers dedicated to clock cross reference measurements are 16-bit wide, except STM32F2, STM32F4, STM32F7 and TIM5 of STM32G4, where 32-bit ones are used.
7. TIM16/Ch1 is used for STM32F30xx.
3.1 CPU tests

Some specific operations are inaccessible by high level compilers. That is why code for both start-up and run time tests are written in assembly, and differs slightly in mnemonics among used compilers.

These tests are product dependent, as sets of available instructions differ between Cortex® cores used by STM32 microcontrollers. As an example, due to restricted instruction set of Arm® Cortex®-M0+ core, instructions loading immediate 32-bit constant operands are replaced by instructions loading those constants placed at code memory.

When user applies a contemporary version of compiler to an older version of the STL, the assembly testing routine applied at startup can become incompatible and require adaptations. For more information refer to Section 3.9: Incompatibility with previous versions of the STL.

3.2 Clock tests and time base interval measurement

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, or to detect any significant discrepancy in the application timing. Different product dependent timers are dedicated to perform such cross check measurement.

Initial configuration of the specific timers is slightly different while dedicated interrupt vectors are used for the measurement in dependency on concrete timer at given device.

Some older products do not support cross-reference measurement feature.

If the system clock doesn't use the HSE quartz clock, user can set up the clock measurement HSI vs. LSI commenting out the parameter HSE_CLOCK_APPLIED in the stm32xx_STLparam.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), user has to consider the accuracy of such a clock source over the whole temperature range. This is necessary to prevent any false clock failure detection, especially when the unit under self test has to operate over a wider temperature range. 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.

Clock security system hardware feature (CSS) is activated by default by the library during startup test as a supplementary testing method if HSE is used as system clock (see stm32xx_STLclockstart.c file).

3.3 SRAM tests

Hardware techniques that ensure single bit redundancy protection of both data words and their addresses are the minimum requirement to fulfill the associated standards on volatile memories (to detect errors not only in areas dedicated to data, but also on their internal address and data path). Some of the older ST products do not feature this (partial or full) hardware redundancy, and then these requirements shall to be met indirectly by applying proper software methods, better if in combination with hardware.
Unfortunately, execution of these tests uses a portion of microcontroller computing power, and makes overall diagnostic tests longer. As a consequence, software methods are applicable on static errors only. Even very sophisticated tests are not able to cover transient errors efficiently, so diagnostic coverage of these tests is always limited in principle.

The SRAM test must follow a topological pattern. Testing by word can be used as logically adjacent bits (belonging to a single word) are physically far away from each other split in the array, while pairs of words with subsequent logical addresses share physically adjacent neighbor cells (bits). In addition, when the sequence of addresses is not regular (as in some older STM32 products), the physical order of words addresses (so called scrambling) has to be respected during this test.

User has to ensure that a proper test corresponding to the RAM design is implemented for the product used in the application. This is done by definition of ARTISAN symbol for the assembly compiler. This symbol has to be defined for STM32F0xx, STM32F1xx and STM32F3xx products exclusively.

Optionally, user can simplify and speed-up the Marching C- algorithm used during run time testing when USE_MARCHX_TEST symbol is applied. If this symbol is defined, two middle marching steps are skipped and not implemented during the transparent test (see Section 6.4.7: Partial RAM run time self test). It is suggested to keep full March C- algorithm at least during the initial test.

Some ST microcontrollers feature a built-in word protection with single bit redundancy (hardware parity check) applied on CCM RAM or at least on a part of the SRAM. This hardware protection is one of the acceptable methods required by the IEC 60335 and IEC 60730. The built-in parity feature includes address and data with the exception of the STM32F0 Series, where the parity is only computed with the data.

Despite the hardware method is recognized by the standard, it is advised to keep the execution of software March test and the data redundancy as supplementary safety methods for the volatile memory testing, and not to rely exclusively on the hardware (the main reason is that there is no way how to test functionality of the parity feature during run time).

Reliability of the information stored in the SRAM can be increased by applying additional indirect testing techniques, such as double storage of safety critical information in physically separated areas in the form of two inverted patterns (this is based on the fact that corruption caused by radiation or EMI usually attacks a limited physical memory section), or by applying dedicated check sum signature to each part of these data.

The hardware RAM parity check is an optional feature. When enabled, it is advised to perform a SW initialization of the whole RAM at the beginning of the code execution, to avoid getting parity errors when reading non-initialized locations (this is the case of local variables when they are allocated and read out while using different data access to memory). The best way to do this is during start-up procedure. A simple loop inserted into start-up assembly code can solve the problem and initialize parity system in the dedicated RAM area:

```assembly
; Program starts here after reset
;------------------------------------------------------------------------
Reset_Handler
; Parity system initialization has to be performed here prior to the
; startup self-test procedure
```
Main differences between STL packages from product point of view

14/64 AN4435 Rev 7

---

; 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 is not relevant because the purpose of this loop is to initialize the parity system. The RAM content is initialized later by the compiler standard start-up procedure. RAM_block_begin, RAM_block_size and end memory address setting must be aligned with the number of data copied by STMIA instruction to prevent any undefined memory access, especially at the end of the loop.

Note: For new products featuring optional HW initialization of the SRAM, there is no need to perform initialization of the memory by the upper SW loop if the user activates this option.

When the initial software March test is performed over a RAM area dedicated to stack, it destroys all the stack content including the return address of the test routine itself stored there when high level compiler is used. The store and restore procedure of the return address depends on the compiler implementation and can differ for different optimization levels. Besides an optimization effort, this is main reason why the routines supporting SRAM testing are written in assembly, to be independent from the higher compiler implementation. On the other side this solution brings a light tool dependency, and different assembly source files have to be kept to pass their compilation correctly.

When user applies a contemporary version of compiler to older version of the STL the assembly testing routine applied at startup can become incompatible and require small adaptations. For more information see Section 3.9: Incompatibility with previous versions of the STL.

3.4 Flash memory integrity tests

Flash memory test is based on built-in HW CRC unit. Some of the STM32 microcontrollers feature configurable units so that the initial configuration can differ slightly, however the polynomial calculation used is the same for all the products.

User must comment definition of parameter CRC_UNIT_CONFIGURABLE in the stm32xx_STLparam.h configuration header file for all products where CRC is not configurable.

The area where the pattern of CRC calculation is stored has to be excluded from the range of the calculation. The boundaries of the checked area must be aligned with multiples of
tested block size used during the test. By default, the block size is set to 16 words (64 bytes) by parameter FLASH_BLOCK_WORDS defined in the stm32xx_STLparam.h file. Unused memory areas included in the check have to be identified with predefined values. An all-1 pattern is used by default.

The range under the nonvolatile memory test is defined by the user. During runtime, if the test of the overall memory range is not acceptable because too long, the user can split it into segments that correspond to local areas where the program is being executed. This requires to dynamically modify the area under test, so that the testing is performed exclusively over those areas.

The STL integration examples perform tests of single contiguous areas described by single checksum descriptor. When user considers additional separated segments, the test procedure has to be adapted, as done in the X-CUBE-STL library targeting SIL.

The result of the CRC calculation has to be compared with the corresponding reference pattern provided either automatically by compiler (IAR™ case) or added by the end user from a computation handled externally (MDK-Arm® and Ac6 cases).

When the tool does not support CRC pattern placement, specific script files (crc_gen_keil.bat or crc_gen_gcc.bat) are provided in the implementation example projects to run post-built procedures calculating the checksum automatically. They are based on installation of Srecord GNU tool, freely available from http://srecord.sourceforge.net. Default HEX file provided by the linker output is modified and the CRC reference pattern is inserted in a new HEX file. User has to ensure that the modified HEX file (output_name_CRC.hex) is used for the application download (e.g. by implementation of crc_load.ini file or by proper modification of launch configuration properties when debug or download session starts).

When testing the Flash memory integrity, CRC computation done via hardware CRC generator decreases significantly the CPU load. The Flash memory can be tested while DMA is used for feeding CRC block, too. In comparison with software test, the CPU load decreases significantly when DMA is applied but the speed of the test itself doesn't change so much, because DMA needs at least a few cycles to read and transfer data. The test can even slow down when DMA services some other transfers or interrupts in parallel. Moreover some additional DMA configuration is required at initialization of the test. User can find detailed information about use of DMA for CRC calculation in Using the CRC peripheral in STM32 family (AN4187).

Some of the oldest STM32 devices do not feature the CRC hardware block, use software routines for CRC computation. Former versions of the STL based on obsolete SPL libraries provide 16-bit wide calculation method based on predefined look-up table constants to speed up the calculation.

### 3.5 Start-up and system initialization

There are differences between initial system configuration and setup of debug and diagnostic utilities (e.g. recognizing reset cause) because of hardware deviations, dedicated debugging tools and used compilers. Standard product start-up file (tool-dependent) is modified to include a set of start-up tests at the very beginning.
3.6 Firmware configuration parameters

All the STL configuration parameters and constants used in the STL code written at C-level are collected into one file, `stm32xx_STLparam.h`. Configuration differences respect mainly different sizes of tested areas, different compilers and slight deviations of control flow.

User must be careful, when modifying the initial or run time test flow, of possible corruption of the implemented control flow. In this case, values summarized at complementary control flow counters can differ from the constants defined for comparison at flow check points (see Section 5.1.3: Flow control procedure). To prevent any control flow error, user must change definition of these constants in an adequate way.

There are a few parameters to be defined for dedicated assembly compiler, for more details see Tool specific integration of the library.

Configuration options are summarized in Table 8.
Table 8. How to manage compatibility aspects and configure STL package

<table>
<thead>
<tr>
<th>Feature</th>
<th>IAR™-EWARM</th>
<th>MDK-Arm®</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arm® Cortex® core</td>
<td>Include proper CPU testing start-up and runtime procedures, proper handling of core hard faults and exceptions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>Handling SYSTCLK_AT_RUN_HSE / SYSTCLK_AT_RUN_HSI / HSE_VALUE / HSE_CLOCK_APPLIED / LSI_Freq parameters in stm32xx_STLparam.h</td>
<td>Handling ROM_START / ROM_END in stm32xx_STLparam.h</td>
<td>Handling ROM_START / ROM_END in stm32xx_STLparam.h</td>
</tr>
<tr>
<td>Flash memory density (KB)</td>
<td>Handling Checksum option in Project linker option menu ROM_region in project.icf.file</td>
<td>Setup LR_IROM1 load region in project.sct file.</td>
<td>Define Flash memory region in project ld file. Implement proper post-built script files for the automatic CRC check sum calculation.</td>
</tr>
<tr>
<td>ECC on Flash</td>
<td>Implement handling ECC event by interrupt or by pulling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRC configurable</td>
<td>Handling CRC_UNIT_CONFIGURABLE parameter in stm32xx_STLparam.h</td>
<td>Setup RAM_BU, RAM_PNT, CLASSB, CLASS_INV RW_IRAM1 in project.sct file</td>
<td>Define CLASSB_RAM and RAM regions in project ld file.</td>
</tr>
<tr>
<td>RAM density (KB)</td>
<td>Setup RUN_TIME_RAM_BUF_region, RUN_TIME_RAM_PNT_region,</td>
<td>Handling RAM_START, RAM_END, CLASS_B_START, CLASS_B_END parameters in stm32xx_STLparam.h</td>
<td>Handling RAM_START, RAM_END, CLASS_B_START, CLASS_B_END parameters in stm32xx_STLparam.h.</td>
</tr>
<tr>
<td>RAM parity</td>
<td>Handling RAM parity event by interrupt or by pulling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM scrambling(1)</td>
<td>Define ARTISAN=1 in Project Assembler / Preprocessor option menu when scrambling is applied</td>
<td>Define ARTISAN=1 in Option for Target / Asm / Conditional assembly control symbols menu when scrambling is applied</td>
<td>Define ARTISAN=1 in Properties for Assembly / Tool Settings / MCU GCC Assembler / General / Assembler Flags when scrambling is applied.</td>
</tr>
<tr>
<td>March-X flow during transparent RAM test</td>
<td>Define USE_MARCHX_TEST=1 in Project Assembler / Preprocessor option menu when the flow is applied</td>
<td>Define USE_MARCHX_TEST=1 in Option for Target / Asm / Conditional assembly control symbols menu when the flow is applied</td>
<td>Define USE_MARCHX_TEST=1 in Properties for Assembly / Tool Settings / MCU GCC Assembler / General / Assembler Flags when the flow is applied.</td>
</tr>
<tr>
<td>ECC on E2PROM</td>
<td>Implement handling ECC event by interrupt or by pulling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IWDG option</td>
<td>Handling IWDG_FEATURES_BY_WINDOW_OPTION parameter in stm32xx_STLparam.h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Main differences between STL packages from product point of view

18/64 AN4435 Rev 7

**Clock cross reference measurement**

Set up proper timer system for cross reference measurement and handling its events.

<table>
<thead>
<tr>
<th>Feature</th>
<th>IAR™-EWARM</th>
<th>MDK-Arm®</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock cross reference measurement</td>
<td>Setup proper timer system for cross reference measurement and handling its events</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debugging option(2)</td>
<td>Handling STL_VERBOSE_POR, STL_VERBOSE, STL_EVAL_MODE, STL_EVAL_LCD parameters in stm32xx_STLparam.h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Tool specific procedures (source code written in assembler).
2. Evaluation board specific procedures (not part of the safety code, but used as an application integrating example).
3.7 Firmware integration

Self test procedures and methods targeting Class B requirements are provided in the project examples showing how to integrate correctly the firmware into a real application. Every integration example uses dedicated products and evaluation HW boards. Apart from common drivers and procedures, it also includes product, evaluation board or compiler specific drivers not directly related to the safety task but rather included for demonstration or debugging purposes (details are given in Section 2: Package variation overview).

User has to take care of dedicated linker file content and project specific settings to integrate the STL stack and all the methods used properly into the target application.

Pay attention to the definition of memory areas under test (RAM and Flash), to the allocation of memory space for Class B variables and stack, and to the definition of the control flow. Additional details are provided in the following sections of this document.

3.8 HAL driver interface

When all the debug and verbose support (UART channel, LCD display, LEDs) is removed from the packages, the interface between HAL layer and STL procedures is reduced to drivers needed to control specific peripherals used during start-up and run time self tests. An overview is given in Table 9.

<table>
<thead>
<tr>
<th>HW component</th>
<th>HAL drivers used</th>
<th>STL files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core SysTick timer</td>
<td>HAL_SYSTICK_Config</td>
<td>stm32xx_STLmain.c</td>
</tr>
<tr>
<td>NVIC</td>
<td>HAL_NVIC_SetPriority, HAL_NVIC_EnableIRQ, HAL_NVIC_SystemReset</td>
<td>stm32xx_STLstartup.c, stm32xx_it.c</td>
</tr>
<tr>
<td>Clock system</td>
<td>HAL_RCC_OscConfig, HAL_RCC_ClockConfig, HAL_RCC_EnableCSS</td>
<td>stm32xx_STLstartup.c, stm32xx_STLclockstart.c, stm32xx_STLclockrun.c</td>
</tr>
<tr>
<td>Timers</td>
<td>HAL_TIM_IC_Init, HAL_TIMEx_RemapConfig, HAL_TIM_IC_ConfigChannel, HAL_TIM_IC_Start_IT, __TIMx_CLK_ENABLE</td>
<td>stm32xx_it.c</td>
</tr>
<tr>
<td>CRC unit</td>
<td>HAL_CRC_Init, HAL_CRC_DelInit, HAL_CRC_Accumulate, HAL_CRC_Calculate, __HAL_CRC_DR_RESET, __CRC_CLK_ENABLE()</td>
<td>stm32xx_STLstartup.c, stm32xx_STLcrc32Run.c</td>
</tr>
</tbody>
</table>
Table 9. Overview of HAL drivers used by STL stack procedures (continued)

<table>
<thead>
<tr>
<th>HW component</th>
<th>HAL drivers used</th>
<th>STL files</th>
</tr>
</thead>
<tbody>
<tr>
<td>IWDG and WWDG</td>
<td>HAL_IWDG_Init, HAL_WWDG_Init, HAL_IWDG_Start, HAL_WWDG_Start, HAL_IWDG_Refresh, HAL_WWDG_Refresh, _HAL_RCC_CLEAR_FLAG, _HAL_RCC_GET_FLAG, _WWDG_CLK_ENABLE()</td>
<td>STM32x_STL_startup.c, STM32x_STLmain.c</td>
</tr>
<tr>
<td>HAL layer</td>
<td>HAL_Init, HAL_IncTick, HAL_GetTick</td>
<td>STM32x_STL_startup.c, STM32x_STLmain.c, stm32xx_it.c</td>
</tr>
</tbody>
</table>

Note: Be careful when using a HAL version newer than that used for the STL certification, check changes summarized in release notes. For more information refer to Section 3.9.

3.9 Incompatibility with previous versions of the STL

User has to be careful when a different version of compiler or HAL is applied to implement the STL testing procedures (see Table 1).

The push towards optimization of code size and speed makes the providers of compilers to apply specific and more sophisticated methods of making code structure, even if users do not need these optimizations, which make the code too compressed and difficult to be analyzed or debugged.

One of the requirements is that each subroutine keeps the content of core registers R4 to R11 (this was not required with lower levels of optimization, and so not kept in previous versions of the STL for product specific startup tests written in assembly). In case of CPU test the modification is easy (it is only needed to push these registers into stack at the beginning of the procedure and restore them back before return, either by push and pop or stmdb and ldmia instructions:

```
STL\_StartUpCPUTest:
    ; Save preserved registers values into stack
    STMDB SP!, {R4, R5, R6, R7, R8, R9, R10, R11}
    ... 
    ; Restore preserved registers values from stack
    LDMIA SP!, {R4, R5, R6, R7, R8, R9, R10, R11}
    BX LR          ; return to the caller
```

Another problem related to the compiler optimization can be the replacement of a control flow based on step by step filling of the specific dedicated flow control registers between the test caller and caller procedures with a single filling of these registers with final values computed by the compiler after the last procedure call. This is why it is strongly suggested to set the lowest possible optimization for the stm32xx\_STL\_startup.c file compilation configuration.
One of the crucial definitions of commonly used error status enumerator is changed in HAL library so that the assembler-written STL code, which relies on the original definition, has to be either adapted or declared independently from the common operator recognized at C-level. To make the FW independent on HAL definition of this enumerator user can define specific enumerator dedicated to the test and modify the declaration of the related test function accordingly. For example, to correctly evaluate result of RAM startup test, the `stm32xx_STLRam.h` file has to be modified in following way:

```c
typedef enum
{
  SRAMTEST_SUCCESS = 1,
  SRAMTEST_ERROR = !SRAMTEST_SUCCESS
} SRAMErrorStatus;
```

and

```c
SRAMErrorStatus STL_FullRamMarchC(uint32_t *beg, uint32_t *end, uint32_t *pat, uint32_t *bckup);
```

while comparison of result when startup test is called in the `stm32xx_STLstartup.c` file has to be modified in following way:

```c
... if (STL_FullRamMarchC(RAM_START, RAM_END, BCKGRND, RAM_BCKUP) !=
SRAMTEST_SUCCESS)
{
...
```

Note that there is an additional back-up pointer included as parameter when calling the startup SRAM test procedure, not used in previous versions of the library. This is an optional solution to solve the problem seen during the CPU startup test, and to save preserved registers. The `RAM_BCKUP` pointer held at R3 during the call has to point at the beginning of some volatile memory area with 16 bytes capacity to save registers R4-R7 used and modified by the self test. The situation here is a bit more complicated as the test can destroy the stack content if the stack area is under test. User then has to save the preserved registers to another SRAM or into a part of memory excluded from the test. The test has to start and finish by storing and restoring the registers from the address provided by R3:

```c
STL_FullRamMarchC:
STMIA R3!,{R4-R7}    ; Save content of preserved registers R4-R7
...
LDMDB R3!,{R4-R7}     ; Restore R4-R7 content
BX    LR          ; return to the caller
```

The content of the R3 cannot be modified by the procedure. Its original implementation has to be replaced by using R7. If there is no area available for such a backup, user can adopt another optional solution and save the R4-R7 registers one by one in some RW peripheral registers not used at startup.
Compliance with IEC, UL and CSA standards

IEC (International Electrotechnical 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.

The standards applied during inspection are always listed in the final report provided by the certification authority. These standards undergo a continuous development and upgrade process. When a new edition appears, not necessarily the FW certified according to a previous version does not meet the new requirements, and not necessarily the applied testing methods become wrong. Often new editions just collect sets of already existing updates, and the relevant part of the standard is not (or slightly) impacted.

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 does not necessarily result into a 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 must 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 10.

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 STM32 microcontrollers, like double independent watchdogs, CRC units or system clock monitoring.

<table>
<thead>
<tr>
<th>Table 10. MCU parts that must be tested under Class B compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Group</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Microcontroller specific</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
4.1 Generic tests included in STL firmware package

The certified STM32 STL firmware package is composed by the following micro specific software modules:

- CPU registers test
- System clock monitoring
- RAM functional check
- Flash CRC 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).

An overview of the methods used for the MCU-specific tests (described in deeper detail in the following sections) is given in Table 11.

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

When tests are removed the user must consider side effects, as not applied tests can play a role in the testing of other components as well.

<table>
<thead>
<tr>
<th>Group</th>
<th>Component to be tested according to the standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application specific</td>
<td>Interrupt handling</td>
</tr>
<tr>
<td></td>
<td>External communication</td>
</tr>
<tr>
<td></td>
<td>Timing</td>
</tr>
<tr>
<td></td>
<td>I/O periphery</td>
</tr>
<tr>
<td></td>
<td>Analog A/D and D/A</td>
</tr>
<tr>
<td></td>
<td>Analog multiplexer</td>
</tr>
</tbody>
</table>
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 has to 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.

<table>
<thead>
<tr>
<th>Components to be verified</th>
<th>Method used</th>
<th>IEC/UL 60730 references</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Table</td>
</tr>
<tr>
<td>CPU registers</td>
<td>Functional test of all registers and flags including R13 (stack pointer), R14 (link register) and PSP (Process stack pointer) is done at start-up. At run test R13, R14, PSP and flags are not tested. Stack pointer is tested for overflow, (underflow is checked by non-direct methods) link register is tested by PC monitoring. If any error is found, the software jumps directly to the Fail Safe routine.</td>
<td>H.1</td>
</tr>
<tr>
<td>Program counter</td>
<td>Two different watchdogs running with two independent clock sources can reset the device when the program counter is lost or hanged-up. The Window watchdog, driven by the main oscillator, performs time slot monitoring and Independent one, driven by low speed internal RC oscillator, is impossible to disable once enabled. Program control flow is monitored using a specific software method additionally.</td>
<td>H.1</td>
</tr>
<tr>
<td>Addressing and data path</td>
<td>This is tested indirectly by RAM functional and Flash memory integrity tests, stack overflow (specific pattern is written at low boundary of stack space and checked for corruption in regular intervals, to check underflow a second pattern can be written at high boundary if it is not at top of RAM). In addition, BUS error exception vector is fetched by the CPU if memory access fault occurs.</td>
<td>H.1</td>
</tr>
<tr>
<td>Clock</td>
<td>A cross check measurement between two independent sources of frequency is used while measured frequency clocks the timer and second one gates the timer clock input. As an example, wrong frequency of external crystal (harmonic/sub harmonic) can be detected using time base of internal low speed RC oscillator for gating the timer.</td>
<td>H.1</td>
</tr>
<tr>
<td>Invariable memory</td>
<td>32-bit CRC check sum test of full memory is done at start-up and partial memory test is repeated at run time (block by block). Fast built-in hardware 32-bit CRC calculation unit is used.</td>
<td>H.1</td>
</tr>
<tr>
<td>Variable memory</td>
<td>March C- full memory test is done at start-up and partial memory test is repeated at run time (block by block over the Class B storage area exclusively). Scrambled order of physical addresses in RAM is respected in the tests for optimal coverage of coupling faults. Faster March X can be optionally used for testing at run time. Word protection with double redundancy (inverse values stored in non adjacent memory space) is used for safety critical Class B variables, Class A variables space, stack and not used space are not tested during run time.</td>
<td>H.1</td>
</tr>
</tbody>
</table>
4.2 Application specific tests not included in ST firmware self test library

User must focus on all the remaining required tests covering application specific MCU parts not included in the ST firmware library:
- test of analog parts (ADC / DAC, multiplexer)
- test of digital I/O
- external Addressing
- external communication
- timing and interrupts.

A valid solution for these components is strongly dependent on application and device peripheral capability. The recommendation is to respect the suggested testing principles from the very early stages of application design.

Very often this method leads to redundancy at both HW and SW levels.

HW methods must be based on:
- multiplication of inputs and/or outputs
- reference point measurement
- loop-back read control at analog or digital outputs like DAC, PWM, GPIO
- configuration protection.

SW methods must be based on:
- repetition in time, multiple acquisitions, multiple checks, decisions or calculations made at different times or performed by different methods
- data redundancy (data copies, parity check, error correction/detection codes, checksum, protocoling)
- plausibility check (valid range, valid combination, expected change or trend)
- periodicity and occurrence checks (flow and occurrence in time controls)
- periodic checks of correct configuration (e.g. read back the configuration registers).

4.2.1 Analog signals

Measured values must be checked for plausibility and verified by measurements performed by other redundant channels, while free channels can be used for reading some reference voltages in conjunction with testing of analog multiplexers used in the application. The internal reference voltage must also be checked.

Some STM32 devices feature two (or even three) independent ADC blocks. It makes sense to perform conversions on the same channel using two different ADC blocks for security reasons. Multiple acquisition at one channel or compare redundant channels followed by average operation can be applied.

Here are some tips for testing the functionality of analog parts at STM32 microcontrollers.
ADC input pin disconnection
Can be tested by applying additional signal source injection on the tested pin
- Some STM32 devices feature internal pull-down or pull-up resistor activation on the
  analog input or free pin with DAC functionality (or a digital GPIO output) can be used
  for the injection.
- Some STM32 devices feature routing interface. It can be used for internal connection
  between pins to make testing loop-back, additional signal injection or duplicate
  measurement at some other independent channel.

Note: User has to prevent a critical injection into the analog pin. This can happen when digital and
analog signals are combined while different power levels are applied to analog and digital
parts ($V_{DD} > V_{DDA}$).

Internal reference voltage and temperature sensor (VBAT for some devices):
- Ratio between these signals can be verified within the allowed ranges
- Additional testing can be performed where the $V_{DD}$ voltage is known.

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

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

Comparator functionality
- Comparison between known voltage and DAC output or internal reference voltage can
  be used for testing comparator output on another comparator input.
- Analog signal disconnection can be tested by pull-down or pull-up activation on tested
  pin and comparing this signal with DAC voltage as reference on another comparator
  input.

Operational amplifier
- Functionality can be tested forcing (or measuring) a known analog signal to the
  operational amplifier (OPAMP) input pin, and internally measuring the output voltage
  with the ADC. The input signal to OPAMP can be also measured by ADC (on another
  channel).

4.2.2 Digital I/Os
Class B tests must detect any malfunction on digital I/Os, too. It could be covered by
plausibility checks together with some other application parts (e.g. change of an analog
signal from temperature sensor must be checked when 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. 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.

### 4.2.3 Interrupts

Occurrence in time and periodicity of events must be checked. Different methods can be used; one of them uses set of incremental counters where every interrupt event increments a specific counter. The values in the counters are then cross-checked periodically with other independent time bases. The number of events occurred within the last period depends upon 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. Unused interrupt vectors must be diverted into common error handler. Polling is preferable for non safety relevant tasks if possible to simplify application interrupt scheme.

### 4.2.4 Communication

Data exchange during communication sessions must 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. Periodicity and occurrence in time of the communication events together with protocol error signals has to be checked permanently.

User can find more information and methods in product dedicated safety manuals.

### 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 the 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 modules were integrated into ST standard peripheral libraries dedicated to different products in the past. Main goal of the new FW has been to remove any HW dependence on different products and integration of safety dependent parts into a single common stack of
self tests based on new unique hardware abstraction interface (HAL) developed for the whole STM32 family.

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 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 control mechanism and synchronized with system tick interrupts 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, Keil® MDK-Arm® and Ac6 Eclipse™) are addressed.

5.1 Common 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.

5.1.1 Fail safe mode

A dedicated procedure, `FailSafePOR()`, is called when a fail is detected by the self test procedures. The routine is predefined at the beginning of `stm32xx_STLstartup.c` file. The goal of this procedure is to provide a unique output and allow the user to react immediately.

By default, there is no specific handling inside the procedure except for debug support and an empty loop waiting for a watchdog reset (the reset can be prevented in debug mode). It is fully upon user responsibility to build up a handler inside this routine and perform all the necessary steps to bring the application in a safe state, while taking a decision on the next cycle in dependency of the severity of the problem found.

Optionally, the user can redefine the procedure and pass a specific input parameter (a simple constant) when calling it from different places of the program to identify the severity of the problem and simplify the decision flow inside the procedure.

The debug or verbose mode described in Section 5.4: Package configuration and debugging can be used to identify error occurred.

5.1.2 Safety related variables and stack boundary control

It is highly recommended to handle critical values related to system safety in a specific way.

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 into non adjacent memory locations. A partial transparent RAM March C- or March X run time test is continuously performed, step by step, on these RAM areas by a specific interrupt subroutine. The buffer used for temporarily storage and back recovery of the tested area is within the range tested permanently, too.

User has to ensure that every pair is always compared for integrity before the value is used. Fail Safe mode has to be invoked if any pair integrity is corrupted. If the value of a variable is changed on purpose, both storage locations need to be updated to keep the correct integrity of the pair.

An example of RAM configuration is shown in Figure 1. User can adapt the RAM space allocation according to the application needs and hardware capability. For better consistency of the run time test, all the class B regions are merged together within a single compact memory location.
The user has to align the size of the tested area to multiply single transparent steps while respecting overlay used for the first and last step of the test, including address scrambling.

That is why the user has to allocate dummy gaps at the beginning and at the end of the area dedicated to Class B variables. Size of these gaps has to correspond to applied overlay of the tested single block.

Backup buffer and pointer to Class B area has to be allocated out of the area dedicated to Class B variables, at a specific location tested separately each time the overall Class B area test cycle is completed.
Specific pattern is written at the low boundary of stack space and checked for corruption at regular intervals to detect the stack overflow. To check its underflow, a second pattern can be written at high boundary if the stack is not placed at top of RAM. If the stack is placed at top of RAM an underflow event raises hardware exception event.

When more than a single stack area is used by the application, it is advisable to separate the adjacent areas by boundary patterns to check that all the pointers operate exclusively within their dedicated areas.

Stack and non safety RAM area are not checked by the Transparent RAM test at run time.

5.1.3 Flow control procedure

Program flow control is a method highly recommended by the standards, because is 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 only one member of the complementary counter pair. 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 principle

Note: The unique number for calling point of Component test 1 at 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 0x0000 0000 and 0xFFFF FFFF for simplicity. The table in upper right corner of Figure 2 shows how the counters are changed in four steps and their consistent complementary state after the last step of checking policy (return from procedure) is done.

5.2 Tool specific integration of the library

This section describes how the ST solution is organized in relation to different tools used.

5.2.1 Projects included in the package

The FW includes implementation examples supporting different evaluation boards dedicated to STM32 products. Three projects are added for every evaluation board supporting IAR™-EWARM, Keil® MDK-Arm®, Ac6 Eclipse™ or GCC STM32CubeIDE workbenches.

It is recommended to check and apply correctly the following tool-specific actions:

- Corresponding Project.eww, Project.uvproj or .cproject project files must be configured for specific STM32 family member and evaluation board used. Proper configuration symbols has to be declared in the preprocessor setting sections.
- *.icf (for IAR™), *.sct (for Keil®) and *.id (for Ac6) templates of linker script files has to be checked where all the memory regions including Class B specific ones are defined. For safety critical variables, the RAM region consistency procedure is described in...
Section 5.1.2, for CRC Flash integrity check see Section 3.4: Flash memory integrity tests.

- Standard startup_stm32xxxx.s file (for IAR™ and Ac6 compilers) or $Sub$$main() procedure in code (case of Keil® compiler - see stm32xx_STLstartup.c) has to be modified to insert a call of STL_StartUp() procedure at the beginning of the program flow, before entering the main. If all the start-up tests are passed, macro GotoCompilerStartUp() is called (defined at stm32xx_STLparam.h file) to continue at the standard C start-up procedure. Procedure ___iar_program_start() is called for IAR™ or __main() for Keil®, and Startup_Copy_Handler() for Ac6. For the Keil® compiler a specific trick is applied to decide if the start-up testing procedures or main flow has to be called. A specific pattern signaling completion of start-up set of tests is stored into independent data register of the CRC unit (see stm32xx_STLstartup.c file).

- CRC generation must be enabled and region under test properly set at project options (IAR™) or CRC check sum result has to be specified and implemented by specific methods described in detail in Section 3.4: Flash memory integrity tests for the Keil® and Ac6 compilers. Proper constants must be defined in the stm32xxx_STLparam.h file.

A summary is given in Table 8.

5.2.2 Start-up file

Specific start-up files are prepared for each project targeting to run initial set of self test procedures as the very first task after device reset. Self test start-up routines are not altering neither disabling the compiler standard C start-up files. Variables and stack/heap are initialized in the usual way just after start-up testing is finished.

5.2.3 Defining new safety variables and memory areas under check

Duplicate allocation of the safety critical variable image in CLASS_B_RAM and CLASS_B_RAM_REV is needed to ensure redundancy of the safety critical data (Class B) stored in variable memories. All other variables defined without any particular attributes are considered as Class A variables and are not checked during transparent RAM test.

Sizes of Class A and Class B variable regions can be modified in the linker configuration file. New Class B variables must be declared in stm32xx_STLclassBvar.h header file, with following syntaxes:

IAR™

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

Keil®

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

Ac6

extern uint32_t MyClassBvar __attribute__((section (".class_b_ram")));
extern uint32_t MyClassBvarInv __attribute__((section (".class_b_ram_rev")));
Consistency has to be always kept between definition of the variables in the stm32xx_STLclassBvar.h header file, linker configuration file and the self test library configuration file stm32xx_STLparam.h to align safety critical variables placement with the definition of memory range areas to be tested both at start-up and during run time SRAM tests. The start and end addresses of RAM/ROM regions are not exported when using Keil® and Ac6 environments. These addresses modifications must be handled by user in the stm32xx_STLparam.h file, which contains specific addresses and constants definitions required to perform correct transparent RAM and ROM (Flash) tests.

Procedures for SRAM testing are written in assembly and are collected in the compiler specific stm32xxxxx_STLRamMcMxyyy.s assembly file. Product specificities and configuration parameters for assembly compilation (ARTISAN and USE_MARCHX_TEST) have to be considered when integrating the procedures into the project. The procedures are called with parameters determining begin and end of the tested area and the checking pattern as well. User has to respect available physical address space by the range applied for the test.

When the SRAM design features a scrambling of physical addresses, ARTISAN parameter has to be defined and user memory allocation has to respect the scrambled addresses pattern repetition. The start of the memory block under marching test has to be aligned with the physical pattern start, while the tested block granularity is limited by the pattern size as a minimum.

Implementation of calculated CRC pattern into code and definition of the area under test used for nonvolatile memory check is different at compiler level. IAR™ compiler can be set to include 32-bit CRC results compatible with STM32 hardware directly into code (see IAR™ documentation), while CRC calculation is not fully supported neither by Keil® nor by Ac6 compilers for STM32 microcontrollers. This is why the result provided by STM32's internal CRC generator cannot be used directly for the memory check in these projects.

User has to apply some other post-built method to implement correct checksum pattern into the code (see more details in Section 3.4: Flash memory integrity tests), or ignore the negative comparison of the pattern with output of the CRC computation process.

Note: Some former revisions of the IAR™ compiler include an incompatibility issue in the configuration of checksum computation results. That is why the dedicated ST HAL driver cannot be used to store data into the CRC unit. Direct access to CRC_DR has to be applied instead while additionally implemented __REV intrinsic function reverses order of the input data bytes to comply with the issue. User can expect correction in future updates of the compiler. Then user can switch back to standard HAL driver calls when feeding the CRC input data correctly (HAL_CRC_Calculate() or HAL_CRC_Accumulate() to stm32xx_STLstartup.c and stm32xx_STLmain.c files).

5.2.4 Application implementation examples

A short demonstration example of a user application is provided at main.c file included in the dedicated project (see Section 6: Class B solution structure). It provides an example of how Class B routines can be integrated into an application-specific solution.

While using conditional compilation, user can include some additional software controlling hardware of dedicated evaluation board for demonstration or debugging purpose. This part
of code is not safety related and it is supposed to be removed from final application code. It uses following hardware:

- LCD display for demonstration purpose if available on evaluation board
- dedicated UART Tx port (sending text info messages for hyper terminal window)
- dedicated outputs to drive LEDs indicating that software routines are executed properly.

5.3 Execution timing measurement and control

Dedicated I/Os can be used for timing measurement of procedures executed both at start-up and during run time.

Start-up tests are performed within a single run, their duration depends from the MCU performance and from the size of the tested area. Tests during run time are performed block by block, so their duration depends also upon the size of the block under test and upon the frequency of repetition of testing.

The user has to find a balance between the performance needed to run the application and that for testing the hardware running it. The main challenge is to achieve a short overall diagnostic test interval while keeping application process safety time within acceptable length. In critical cases run time testing can be limited to areas collecting critical code or data. The partial test interval is derived from SysTick 1 ms interval and it is set to 10 ms by default by parameter SYSTICK_10ms_TB.

When shortening it, user must consider that the interval is used to calculate clock cross reference ratio between system clock and LSI during run time too, so its length shall never drop below an interval corresponding to the number of LSI periods applied for the clock cross-measurement (set to eight by default).

A specific principle is used when monitoring these I/O signals connected to LEDs available on the board, as shown in Figure 3:

![Figure 3. Diagnostic LED timing signal principle](image)

A dedicated LED toggles when specific tests (RAM, Flash memory, clock) are active and every time the testing procedure is completed (both at start-up and during run time). These LED signals can be used to measure among others; length of the tests; frequency of partial tests and time needed to finish an overall single testing cycle during both start-up and run time.
When the dedicated area under test has been completely inspected, a set of partial tests starts again from the beginning.

Typical waveforms related to monitoring of dedicated I/Os signals at start-up or during run time are shown in the oscilloscope screenshots shown, respectively, in Figure 4 and in Figure 5.

**Figure 4. Typical test timing during start-up**
Table 12 summarizes the I/O signals, while Table 13 lists the typical values measured during the test of the FW packages.
Table 12. Signals used for timing measurements

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange LED2</td>
<td>PD9 (CN8-6)</td>
<td>PD13 (CN9-4)</td>
<td>PG8 (CN3-22)</td>
<td>PG8 (CN3-22)</td>
<td>PC8 (CN6-51)</td>
<td>PA5 (CN5-D13)</td>
<td>PD7 (CN10-15)</td>
<td>PA8 (CN11-3)</td>
<td>PF10 (CN7-1)</td>
<td>PB0 (CN10-22)</td>
<td>PD5 (CN9-9)</td>
<td>PA5 (CN10-11)</td>
</tr>
<tr>
<td>(yellow)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Red LED3</td>
<td>PD10 (CN8-4)</td>
<td>PD3 (CN8-25)</td>
<td>PI9 (CN1-12)</td>
<td>PI9 (CN1-12)</td>
<td>PC9 (CN6-46)</td>
<td>PA6 (CN5-D12)</td>
<td>PG14 (CN10-6)</td>
<td>PA9 (CN11-1)</td>
<td>PA4 (CN6-19)</td>
<td>PB1 (CN10-24)</td>
<td>PD8 (CN9-1)</td>
<td>PA6 (CN10-13)</td>
</tr>
<tr>
<td>(green)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blue LED4</td>
<td>PD11 (CN8-2)</td>
<td>PD4 (CN8-26)</td>
<td>PC7 (CN3-20)</td>
<td>PC7 (CN3-20)</td>
<td>PC10 (CN6-55)</td>
<td>PA7 (CN5-D11)</td>
<td>PG15 (CN10-5)</td>
<td>PA10 (CN11-5)</td>
<td>PA11 (MFX-I011)</td>
<td>PA4 (CN10-17A)</td>
<td>PD9 (CN9-2)</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Notes:
- Orange LED2: PD9 (CN8-6) for STM32052B, PD13 (CN9-4) for STM3210C, PG8 (CN3-22) for STM32G2-45-46G, PG8 (CN3-22) for STM32G2173C, PC8 (CN6-51) for STM32G2141G, PA5 (CN5-D13) for STM32G2756G, PA5 (CN5-D13) for STM32G2NUCLEO, PD7 (CN10-15) for STM32G2L152B, PA8 (CN11-3) for STM32G2L78G, PA8 (CN11-3) for STM32G2H74S1, PD7 (CN10-15) for STM32G2M5P, PA8 (CN11-3) for STM32G2G91B, PA8 (CN11-3) for STM32G2G44RE.
- Red LED3: PD10 (CN8-4) for STM32052B, PD3 (CN8-25) for STM3210C, PI9 (CN1-12) for STM32G2-45-46G, PI9 (CN1-12) for STM32G2173C, PC9 (CN6-46) for STM32G2141G, PA6 (CN5-D12) for STM32G2756G, PA6 (CN5-D12) for STM32G2NUCLEO, PG14 (CN10-6) for STM32G2L152B, PA9 (CN11-1) for STM32G2L78G, PA9 (CN11-1) for STM32G2H74S1, PA9 (CN11-1) for STM32G2M5P, PA9 (CN11-1) for STM32G2G91B, PA9 (CN11-1) for STM32G2G44RE.
- Blue LED4: PD11 (CN8-2) for STM32052B, PD4 (CN8-26) for STM3210C, PC7 (CN3-20) for STM32G2-45-46G, PC7 (CN3-20) for STM32G2173C, PC10 (CN6-55) for STM32G2141G, PA7 (CN5-D11) for STM32G2756G, PA7 (CN5-D11) for STM32G2NUCLEO, PG15 (CN10-5) for STM32G2L152B, PA10 (CN11-5) for STM32G2L78G, PA10 (CN11-5) for STM32G2H74S1, PA10 (CN11-5) for STM32G2M5P, PA10 (CN11-5) for STM32G2G91B, PA11 (MFX-I011) for STM32G2G44RE.
<table>
<thead>
<tr>
<th>Feature</th>
<th>STM32 Series</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F0</td>
</tr>
<tr>
<td>XTAL (MHz)</td>
<td>8</td>
</tr>
<tr>
<td>PLL frequency (MHz)</td>
<td>48</td>
</tr>
<tr>
<td>Flash memory test, start-up (ms)</td>
<td>IAR™</td>
</tr>
<tr>
<td></td>
<td>Keil®</td>
</tr>
<tr>
<td></td>
<td>Ac6</td>
</tr>
<tr>
<td>Tested Flash memory (KB)</td>
<td>IAR™</td>
</tr>
<tr>
<td></td>
<td>Keil®</td>
</tr>
<tr>
<td></td>
<td>Ac6</td>
</tr>
<tr>
<td>Full RAM test, start-up (ms)</td>
<td>4.5</td>
</tr>
<tr>
<td>Tested RAM (KB)</td>
<td>8</td>
</tr>
<tr>
<td>Clock test, start-up (ms)</td>
<td>1.0</td>
</tr>
<tr>
<td>Single RAM test, run time (μs)</td>
<td>19</td>
</tr>
<tr>
<td>Single RAM test, Ac6, run time (μs)</td>
<td>-</td>
</tr>
<tr>
<td>Single Flash memory test, run time (μs)</td>
<td>30</td>
</tr>
<tr>
<td>Single Flash memory test, Ac6, run time (μs)</td>
<td>24</td>
</tr>
</tbody>
</table>

1. The values in this table are indicative, measurements have been performed without debugging support (except LED control). These data can change with different optimization settings of compilers, with other package configurations, and depend upon areas to be checked.
5.4 Package configuration and debugging

The STL package has to be configured to respect correct setting of the actual product.

Sometimes a simple application structure involves suspending or excluding a functional part of the STL package (e.g. system is fed from internal clock). On the contrary, some features can be temporary added during package debug, as they help developer in this phase.

This section describes how the ST solution can be configured, modified and debugged.

5.4.1 Configuration control

Configuration of the software is done at two basic levels:

- **Project settings**: the setting has to respect the differences between the different STM32 products. This part is mainly done automatically by proper configuration of the project and its associated configuration files.
- **User settings**: these are centralized in the Class B configuration file `stm32xx_STLparam.h`. A set of constants defined in this file controls the conditional compilation of some functions. Adjustable constants must have specific settings to run all the tests properly.

Some run-time tests can be skipped depending on the end-application. If the periodicity of the test is connotated with the frequency of use, then power-on tests are sufficient and transparent/run-time tests can be avoided (this is the case, for instance, of a washing machine: the user switches on/off the application every time he uses it). This point must be discussed with the chosen test institute on a case by case basis.

For 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 when the application development is in the closing stage. This is not done by default in the STM32 self test library demonstration.

It is recommended to implement window feature at the closing stage of the testing, and to apply freezing watchdog option at break in the debug module control during debugging.

User has to respect duration of initial RAM and Flash tests especially when tested area is large and overall watchdog period is not sufficient. In this case, the test has to be divided into few parts and extra watchdog refresh services separated from the test flow have to be done between them. These services must not to be part of any loop in the code.

Stack overflow detection and watchdog self-check are not mandatory according to the 60370-1 standard. They are added for indirect testing of micro functionality and can be disabled or skipped if user prefers to apply other methods.

It can help decreasing the CPU load during runtime if 32-bit CRC checks are made using the STM32 internal CRC generator (32-bit wide CRC computation uses the standard 0x04C11DB7 polynomial). The validated 32-bit CRC value can be then saved as a reference for comparing with all the subsequent run time checks.

CRC generation is not supported on the Keil® and Ac6 environments; calling CRC checking routines to be compared with an improper CRC pattern causes the application to reset continuously (as if failures were detected).

If users wants to disable CRC check temporarily at start-up, the easiest way is negotiate the output logic control during the evaluation of the test result (assuming the computed CRC differs from the reference value).
In `stm32xx_STLstartup.c` file modify:
if(crc_result != *(uint32_t *)&REF_CRC32)
into:
if(crc_result == *(uint32_t *)&REF_CRC32)

5.4.2 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). Verbose mode is enabled by default and can be disabled during start-up and/or runtime by commenting lines with the `STL_VERBOSE_POR` and `STL_VERBOSE` defined in the Class B configuration file `stm32xx_STLparam.h`. Each successful completion of SRAM and Flash test at run time is signaled by printing a specific character '#' or '*' at terminal window. The verbose messages can differ slightly between packages. 

Figure 6 shows an example of verbose mode.

Figure 6. STM32 demo hyper terminal output window in verbose mode

```
******* Self Test Library Init *******
Start-up CPU Test OK
Pin reset
IWDOG reset
... IWDOG reset from test or application, testing WWDOG

******* Self Test Library Init *******
Start-up CPU Test OK
Pin reset
IWDOG reset
WWDOG reset
... WWDOG reset, WDG test completed ...
Start-up FLASH 32-bit CRC OK
Control Flow Checkpoint 1 OK
Full RAM Test OK
Clock frequency OK
Control Flow Checkpoint 2 OK

STM32F2xx Cortex-M3
IEC60935 test 01ARc
... main routine starts ...
```

5.4.3 Debugging the package

If any of the self test routines fails, an MCU reset is triggered in the FailSafePOR function defined in the `stm32xx_STLstartup.c` file. This makes the debugging of the application difficult, and can cause the debugger to loose the execution context.
While debugging the package it is useful to disable:

- the call macro `HAL_NVIC_SystemReset()` in `FailSafePOR()` routine to prevent loosing execution context when resetting the micro. It can be done uncommenting definition of flag `NO_RESET_AT_FAIL_MODE` in `stm32xx_STLparam.h`
- the control flow monitoring when adding or removing self test routines, in particular runtime self-diagnostics. This can be done by redefinition of function `control_flow_check_point()` defined in the `stm32xx_STLstartup.c` file,
- all program memory CRC check sum tests when using software break points in the code to prevent program memory check sum error occurrence
- the window watchdog to prevent improper service out of the time slot window dedicated to its refresh (or keep its refresh window sufficiently wider).

During the debugging phase it may be useful to enable:

- verbose diagnostic mode to watch Class B status text messages at UART terminal, by uncommenting flag `STL_VERBOSE_POR` or `STL_VERBOSE` in `stm32xx_STLparam.h` file,
- BSP LED output signals indicating the testing phases on memories, by uncommenting flag `STL_EVAL_MODE` in `stm32xx_STLparam.h` file
- LCD control by including definition of `STL_EVAL_LCD` flag in `stm32xx_STLparam.h` file
- user defined auxiliary GPIO signals indicating the testing phases on volatile and nonvolatile memories by uncommenting flag `STL_USER_AUX_MODE` in `stm32xx_STLparam.h` file. User has to declare and write initialization procedure `User_AUX_Init()` to configure these outputs to push pull mode during startup and put proper definition of constants and macros to control them during execution of the STL stack. For example, to implement this functionality at PA5 and PA6 pins, user must include next lines into the main.h file:

```c
#define AUX_GPIO_PORT   GPIOA
#define AUX_VLM     GPIO_PIN_5
#define AUX_NVM     GPIO_PIN_6

#define User_AUX_On(msk)     { HAL_GPIO_WritePin(AUX_GPIO_PORT, msk, GPIO_PIN_SET); }  
#define User_AUX_Off(msk)    { HAL_GPIO_WritePin(AUX_GPIO_PORT, msk, GPIO_PIN_RESET); }  
#define User_AUX_Toggle(msk) { HAL_GPIO_TogglePin(AUX_GPIO_PORT, msk); }  

void User_AUX_Init(uint32_t msk);
```
6 Class B solution structure

6.1 Integration of the software into the user application

Class B routines are divided into two main processes, namely start-up and periodic run time self tests. The periodic run time test must be initialized by set-up block before it is applied. All the processes are covered by sufficient flow of caller-called controls.

All class B variables are redunded, in doubled control registers stored in a Class B variable space defined by the user. This space is split into two separate RAM regions which are under permanent control of transparent test as a part of run time tests.

Figure 7 shows the basic principle of Class B software package integration into user software solution.

Figure 7. Integration of start-up and periodic run time self tests into application
In principle, the following steps must be provided by the user when STL modules are integrated into an application:

- Execution of initial start-up tests before user program starts
- Periodic execution of run time self tests set within dedicated time period related to safety time
- Setup independent and window watchdogs and prevent their expiration when application is running (ideal case is to tune their refresh with the STL testing period)
- Setup correct testing zones for both start-up and run time tests of RAM and Flash
- Respect error results of the tests and handle proper safety procedures
- Handle Safe state procedure and its content
- Handle HardFault exception procedure and its content
- Prevent possible conflicts with application SW (especially interrupts, DMA, CRC - see Table 14)
- Run tests of application specific microcontroller parts related to application safety tasks
- Exclude all debug features not related to any safety relevant task and use them for debugging or testing purposes only.

When any debug support is enabled during start-up tests user has to ensure passing proper C compiler initialization as some peripheral HAL drivers rely on content copied into RAM (e.g. fields of configuration constants supporting HW of evaluation boards). This could be a problem after complete RAM test is done, as this test destroys all the RAM content. To prevent any problem with those HAL drivers user has to ensure recovery of the RAM content if they are used at this program phase between RAM test and entry to main level.

While application is running, process periodic tests are performed at certain intervals defined by value of Class B variable `TimeBaseFlag`. Frequency of these repetitions provides a basic safety interval. Its setting is defined by the user and depends on application limitations and needs. When user calculates overall runtime test cycle completion the tested areas at RAM and Flash have to be considered together with size of single blocks under these partial tests additionally.

To initialize periodic testing, user must call the `STL_InitRunTimeChecks()` routine at beginning of main and then ensure periodical calls of `STL_DoRunTimeChecks()` at main level - inside of main loop in best case. Periodicity is derived from SysTick interrupts defined and initialized in the HAL layer. `SysTick_Handler()` interrupt service counts 1ms ticks and performs short partial transparent RAM March C or March X checks at defined intervals (set

### Table 14. Possible conflicts of the STL processes with user SW

<table>
<thead>
<tr>
<th>Test</th>
<th>Possible risk of conflict</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Content of CPU registers under test is changed or applied by user SW</td>
<td>User interrupt</td>
</tr>
<tr>
<td>RAM</td>
<td>RAM content under test is changed or applied by user SW</td>
<td>User interrupt or DMA activity</td>
</tr>
<tr>
<td></td>
<td>Data change done by user within range of just performed block of transparent test can be ignored or corrupted</td>
<td></td>
</tr>
<tr>
<td>Flash memory</td>
<td>Corruption of CRC calculation</td>
<td>CRC peripheral is used by some other user component(1)</td>
</tr>
<tr>
<td>Clock</td>
<td>Interrupt capturing service is delayed (over captured)</td>
<td>User interrupt</td>
</tr>
</tbody>
</table>

1. User has to handle content of the CRC registers to keep continuity of the test in this case.

When any debug support is enabled during start-up tests user has to ensure passing proper C compiler initialization as some peripheral HAL drivers rely on content copied into RAM (e.g. fields of configuration constants supporting HW of evaluation boards). This could be a problem after complete RAM test is done, as this test destroys all the RAM content. To prevent any problem with those HAL drivers user has to ensure recovery of the RAM content if they are used at this program phase between RAM test and entry to main level.
to 10 ms by default) when \texttt{TimeBaseFlag} rises, too, to synchronize the rest run checks called from main level. \texttt{FailSafePOR()} routine is discussed in \textit{Section 5.1.1: Fail safe mode.}

User must pay special care to the possibility of corrupting the checking processes by accidental interrupts (including NMI, if it is used for purposes different from the result of an internal fault), and consider all the possible consequences of such events. He must also ensure that no interrupts can corrupt execution of the STL tests or significantly change their timing.

When other time critical libraries (e.g. real time operating system, motor control, touch sensing control) are implemented, the user has to ensure adequate system performance for the repeated execution of all the self tests integrated into the target application, and consider the impact of all possible interferences between the applied libraries.

If there is no room to handle all the processes in parallel, the user can consider to postpone run time testing procedures when application performs some critical operation, or give up the run time testing and rely either on the results of the startup test or on HW safety features (like ECC or parity). Such a solution is acceptable when the application duration period is short, or when the application restarts frequently. Anyway, such a specific implementation of the STL library must always be matter of consultation with the certification authority, and well documented by the applicant.
6.2 Description of start-up self tests

The start-up self test must be run during initialization phase as the first check performed after resetting the microcontroller, as indicated in Figure 8.

Figure 8. start-up self tests structure

The start-up test structure is shown in Figure 9, and includes the following self tests:

- CPU start-up self test
- Watchdog start-up self test
- Flash memory complete check sum self test
- Full RAM March-C self test
- Clock start-up self test
- Control flow check
6.2.1 CPU start-up self test

The CPU start-up self test checks the core flags, registers and stack pointers for correct functionality. If any error is found, Fail Safe routine call is performed.

The source files are written in assembly and they differ slightly in dependency on core due to limited support of instructions set for some lighter cores. There are different versions for the IAR™, Keil® and Ac6 solutions.

The basic structure is shown in Figure 9.

Figure 9. CPU start-up self test structure
6.2.2 Watchdog start-up self test

The test structure is based on reset status register content registering all previous reset causes by corresponding flags until the register is cleared (see Figure 10).

**Figure 10. Watchdogs start-up self test structure**

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

User must take care about proper setting of both IWDG and WWDG. Their periods and parameters of refresh window must be set in accordance with time base interval because a normal refresh is performed at the successful end of the periodical run time test at main loop.

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 must 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, as indicated in Figure 15.
6.2.3 Flash memory complete check sum self test

The CRC checksum computation is performed on the entire Flash memory space defined in the linker checksum structure. The result is compared with that of the linker: if they differ, the test fails (see Figure 11).

Figure 11. Flash start-up self test structure

Refer to Section 5.4.3: Debugging the package for additional comments on CRC procedures.

6.2.4 Full RAM March-C 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 12. The first three loops are performed in incremental order of addresses, the last three in reverse decremental order.

The order of tested addresses can be scrambled for some products, 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 15.

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 represents the physical layout. Bold frames highlight the places where the logical order is scrambled.

Table 15. Physical order of RAM addresses organized into blocks of 16 words

<table>
<thead>
<tr>
<th>Physical order of addresses</th>
<th>Rows</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address scrambling is not present on new ST products. User has to apply ARTISAN assembly compilation parameter for those products with the implemented scrambling (more details are given in Section 3.3: SRAM tests).
Some new products have implemented hardware word protection with single bit redundancy (hardware parity check) applied on CCM RAM or on part of SRAM at least. This aspect is discussed in Section 3: Main differences between STL packages from product point of view.

The algorithm of the entire test loop is shown in Figure 12. If an error is detected, the test is interrupted and a fail result is returned.

**Figure 12. RAM start-up self test structure**

![RAM start-up self test structure diagram](image)

**Note:** The RAM test is word oriented but the more precise bit March testing algorithm can be used. However this method is considerably more time and code consuming.

### 6.2.5 Clock start-up self test

The test flow is shown in Figure 13.

Initially the internal low speed clock (LSI) source is started. The external high speed source (HSE) is started in the next step and PLL fed by HSE is set as system clock. A dedicated timer is initialized for cross-reference measurement of HSE frequency gated by predefined number of LSI periods. Both timer and channel dedicated to such measurement are product specific\(^a\). Difference between two consequently captured values at this timer counter gives the ratio between LSI and HSE frequency. The captured values and their difference are handled during regular interrupt service of the timer channel. This ratio measurement is compared with expected range: if it differs more than ±25% from nominal value an error is signaled. The range is defined by macros `HSE_LimitHigh()` and `HSE_LimitLow()`, defined in the `stm32xx_STLparam.h` file based on a few constant definition.

User is responsible for correct inputs of these macros, and has to decide if the values of these limits can be kept constant, or if they have to be adapted dynamically by the

---

\(^a\) The product specific function `STL_InitClock_Xcross_Measurement()` handling the clock cross-check initial configuration can’t be a part of files keeping generic STL code, but it is implemented in the file `stm32yyxx_it.c`, collecting all the associated interrupt services applied to the device.
application (e.g. because of the varying accuracy of the reference clock signal over the temperature range). The CPU clock is switched back to default HSI source after the test is finished.

**Figure 13. Clock start-up self test subroutine structure**

6.2.6 Control flow check

The start-up test is completed by the control flow check point procedure. Before completion a magic pattern is stored at the top of the space separated for stack.

6.3 Periodic run time self tests initialization

Assuming that all start-up self tests are passed successfully and standard initialization has been completed, the runtime self test package must be initialized just before the program enters in the main loop performing regular calling of the runtime self tests (see Figure 14). The timing must be set properly to ensure that the procedures of the run time tests are called at intervals, to keep sufficient application process safety time (see Section 5.3: Execution timing measurement and control for more details).

All class B variables are initialized. Zero and its complement value are stored into every class B variable complementary pair. Dedicated timer is configured for the system clock and reference frequencies cross-check measurement. The same method of start-up test is used.
6.4 Description of periodic run time self tests

6.4.1 Run time self tests structure

Run time self test is a block of tests performed periodically at main loop level. The execution period 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 7).

Most of the tests here are performed at regular intervals signaled by TimeBaseFlag when user calls their execution from main loop. Only the partial transparent RAM test and the clock measurement backup are performed within the SysTick and dedicated timer interrupt services.

Tests listed below must be included at run time:
- CPU core partial run time test
- Stack boundaries overflow test
- Clock run time 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 run time self test and time base interrupt service structure

Note: Test of analog part, communication peripherals and application interrupts occurrence are not included and their implementation depends upon device capability and user application needs.

6.4.2 CPU light run time self test

The runtime CPU core self test is a simplified version of the runtime test described in Section 6.2.1: CPU start-up self test. Flags and stack pointer are not tested here.

Figure 16. CPU light run time self test structure

If any error code is returned, Fail Safe procedure is called.
6.4.3 Stack boundaries runtime test

This test checks the stack overflow by the integrity of 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. This area differs among the devices. User has to define sufficient area for stack and ensure proper placement of the pattern.

Figure 17. Stack overflow run time test structure

![Diagram showing the stack overflow runtime test]

1. The high end pattern has to be checked for the stack underflow case when the stack area is not placed at the physical end of the RAM space.

6.4.4 Clock run time self test

The clock runtime self test uses a procedure similar to the one used in the start-up self test (see Section 6.2.5: Clock start-up self test). Plausibility check of the clock cross reference ratio is based on the difference between last two consequent results of timer capture events. These results are stored during regular interrupt service of the dedicated timer providing the cross reference measurement between system (HSE) and reference (LSI) frequencies.

Test checks if the HSE ratio falls within the expected range only (±25% of its nominal value). If a larger difference is found, or HSE clock signal is missing, or measurement interrupt disappears, then the CPU clock source is immediately switched back to HSI, and HSE fail status is returned. Otherwise the test returns OK status.

The test checks integrity of all associated variables reporting clock measurement results prior the HSE range is compared.

If HSE is not used to feed the system, the other applied clock source (e.g. HSI) must be cross checked instead. User has to modify the setup of the dedicated timer to ensure the proper pair of clock cross measurement (see the STL_InitClock_Xcross_Measurement() function defined in the stm32xxxx_it.c file) and check that such measurement is supported for real products.
6.4.5 Partial Flash CRC run time self test

The partial 32-bit CRC checksum of the block in Flash is performed at each step while using built-in CRC HW block. The overall tested area and the size of the block involved in each single step of the test has to be properly defined by the user. When the last block is reached the CRC checksum is compared with the value stored by linker. In case of difference the Fail Safe routine is called else new computation cycle is initialized.

The test checks integrity of all associated variables before a block is calculated.

Refer to Section 5.4.3: Debugging the package and Section 3.4: Flash memory integrity tests for additional comments on CRC procedures.
6.4.6 Watchdog service in run time test

If the runtime service block is successfully completed, the window and independent watchdogs must both be refreshed as a last step, just before returning to the main loop. For the watchdogs to be refreshed correctly, proper timing of the call to the runtime block is essential. The period when the block must be called is signaled internally by a time base flag tested at the beginning of the `STL_DoRunTimeChecks()` routine (see Figure 15). User must ensure not to pass calling this procedure at main level to be able to react for the time base flag change properly and consequently refresh the watchdogs at 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 must 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 must be outside any software infinite loop (it must only be put in a straightforward part of the code).

6.4.7 Partial RAM run time self test

The partial transparent RAM test is performed step by step inside the timebase interrupt service routine. The test covers just the part of the RAM allocated to class B variables. One block of six words is tested in each step of the test by default. To guarantee coupling fault coverage, every tested block of memory is always overlapped by two additional neighbor words from previous and next step of the test.

This test can be skipped on devices having SRAM parity HW checks, as described in Section 3.3: SRAM tests.

The order of testing operations for this block is summarized in Table 16. Note that it must always respect the physical order of addresses in the memory, no matter if the test performs ascending or descending filling or checks. If the physical order of the memory cells is not continuous, user has to keep the block size within a multiple of word address corresponding to repetition of the applied scrambling pattern. If no scrambling is applied, any number of subsequent words can be tested in the block. For more details about scrambling see Section 6.2.4: Full RAM March-C self test.

The test checks integrity of all associated variables before a block is checked.
A single step of the test is performed in three phases, as shown in Figure 21 and in Figure 22: the first one shows a continuous address model, while the second refers to a model with physical scrambling (the different and not continuous testing order is highlighted by gray boxes).

In the first phase, the current content of all the cells involved in the block testing (cells D1 to D4) and the overlapped words (cells D0 and D5) is stored into the storage buffer. In the next phase destructive March tests are performed over all the words of the tested RAM block. In the final phase, the original content is restored from the storage buffer back to the tested location.

The storage buffer itself is tested by a March test as the last step of the overall sequence. The buffer area is tested together with the two next additional neighbor words to cover coupling faults on the buffer itself. Size of storage buffer has to be adapted to the size of tested block. After the storage buffer is successfully tested, the whole test is reinitialized and restarts from the beginning. If any fault is detected, the Fail Safe routine is called. The test checks integrity of all associated variables before a block is checked.

The size of the block used for partial testing is fixed to four words by default (the next two are involved into the test as overlap). This model corresponds to the repetition of the scrambling pattern. When user changes the size of this block, the algorithm has to respect the period of the scrambling (if present). When SRAM is designed without any scrambling the size of the block is free, but the user has anyway to modify routines written in assembly (they are written for a fixed default block size).
Figure 21. Partial RAM run time self test - Fault coupling principle (no scrambling)

Figure 22. Partial RAM run time self tests - Fault coupling principle (with scrambling)

Note: The scrambled order of addresses is respected, see Table 15.
The order of the operations is given in Table 16. The March X algorithm can be used instead of March C when symbol USE_MARCHX_TEST is applied for assembly compilation. The test is faster because two middle marching steps 2 and 3 are skipped and not executed.

Table 16. March C phases at RAM partial test

<table>
<thead>
<tr>
<th>March phase</th>
<th>Partial words test over the block</th>
<th>Address order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>Write 0x00000000 pattern</td>
<td>Increasing</td>
</tr>
<tr>
<td>1</td>
<td>Check 0x00000000 pattern, write 0xFFFFFFFF pattern</td>
<td>Increasing</td>
</tr>
<tr>
<td>2</td>
<td>Check 0xFFFFFFFF pattern, write 0x00000000 pattern</td>
<td>Increasing</td>
</tr>
<tr>
<td>3</td>
<td>Check 0x00000000 pattern, write 0xFFFFFFFF pattern</td>
<td>Decreasing</td>
</tr>
<tr>
<td>4</td>
<td>Check 0xFFFFFFFF pattern, write 0x00000000 pattern</td>
<td>Decreasing</td>
</tr>
<tr>
<td>5</td>
<td>Check 0x000000000 pattern</td>
<td>Decreasing</td>
</tr>
</tbody>
</table>
7 Revision history

Table 17. Document revision history

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Description of changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>20-Jun-2014</td>
<td>1</td>
<td>Initial release</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Changed document classification.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Introduction, Section 1: Reference documents, Section 2: Package variation overview, Section 3: Main differences between STL packages from product point of view, Section 3.3: SRAM tests, Section 3.4: Flash memory integrity tests, Section 3.6: Firmware configuration parameters, Section 3.7: Firmware integration, Section 3.8: HAL driver interface, Section 4: Compliance with IEC, UL and CSA standards, Section 4.2: Application specific tests not included in ST firmware self test library, Section 4.3: Safety life cycle, Section 5.1.1: Fail safe mode, Section 5.1.2: Safety related variables and stack boundary control, Section 5.1.3: Flow control procedure, Section 5.2.1: Projects included in the package, Section 5.2.3: Defining new safety variables and memory areas under check, Section 5.2.4: Application implementation examples, Section 5.4.1: Configuration control, Section 6.4.4: Full RAM March-C self test. Section 6.4.4: Clock run time self test and Section 6.4.7: Partial RAM run time self test.</td>
</tr>
<tr>
<td>12-Jan-2016</td>
<td>2</td>
<td>Updated Table 1: Overview of STL packages, Table 2: Organization of the FW structure, Table 4: Structure of the common STL packages, Table 5: Structure of the product specific STL packages, Table 9: Overview of HAL drivers used by STL stack procedures and Table 16: March C phases at RAM partial test.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Figure 1: Example of RAM memory configuration, Figure 2: Control flow four steps check principle, Figure 12: RAM start-up self test structure, Figure 15: Periodic run time self test and time base interrupt service structure, Figure 17: Stack overflow run time test structure and Figure 22: Partial RAM run time self tests - Fault coupling principle (with scrambling).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Section 5.3: Execution timing measurement and control.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added 2, Table 8: How to manage compatibility aspects and configure STL package, Table 12: Signals used for timing measurements and Table 13: Comparison of results.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Figure 4: Typical test timing during start-up and Figure 5: Typical test timing during run time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added footnote 1 to Figure 17.</td>
</tr>
<tr>
<td>07-Mar-2016</td>
<td>3</td>
<td>Changed document classification.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Introduction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Table 7: Compatibility between different STM32 microcontrollers.</td>
</tr>
</tbody>
</table>
Updated document title and Introduction.

Updated Section 2: Package variation overview, Section 3.2: Clock tests and time base interval measurement, Section 3.3: SRAM tests, Section 3.4: Flash memory integrity tests, Section 3.5: Start-up and system initialization, Section 4: Compliance with IEC, UL and CSA standards, Section 4.2: Application specific tests not included in ST firmware self test library, Coding, Section 5.1.2: Safety related variables and stack boundary control, Section 5.1.3: Flow control procedure, Section 5.2.1: Projects included in the package, Section 5.2.3: Defining new safety variables and memory areas under check, Section 5.3: Execution timing measurement and control, Section 5.4.1: Configuration control, Section 5.4.3: Debugging the package, Section 6.2.5: Clock start-up self test, Section 6.3: Periodic run time self tests initialization and Section 6.4.1: Run time self tests structure.

Updated Table 1: Overview of STL packages, Table 5: Structure of the product specific STL packages, Table 6: Integration support files, Table 7: Compatibility between different STM32 microcontrollers and Table 13: Comparison of results.

Updated Figure 5: Typical test timing during run time.

Updated Section 1: Reference documents, Section 2: Package variation overview, Section 3.2: Clock tests and time base interval measurement, Section 4: Compliance with IEC, UL and CSA standards, Section 4.2: Application specific tests not included in ST firmware self test library, Coding, Section 4.2.1: Analog signals, Section 4.2.3: Interrupts, Section 4.2.4: Communication, Maintenance, Section 5.1.2: Safety related variables and stack boundary control, Section 5.2.3: Defining new safety variables and memory areas under check and Section 6.1: Integration of the software into the user application.

Updated Table 1: Overview of STL packages, Table 7: Compatibility between different STM32 microcontrollers, Table 8: How to manage compatibility aspects and configure STL package, Table 11: Methods used in micro specific tests of associated ST package, Table 12: Signals used for timing measurements, Table 13: Comparison of results and Table 14: Possible conflicts of the STL processes with user SW.

Updated Figure 1: Example of RAM memory configuration.

Updated Introduction and Section 3: Main differences between STL packages from product point of view.

Updated title of Table 2: Organization of the FW structure.
Table 17. Document revision history (continued)

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Description of changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>24-Sep-2019</td>
<td>7</td>
<td>Updated Introduction, Section 1: Reference documents, Section 2: Package variation overview, Section 3.1: CPU tests, Section 3.2: Clock tests and time base interval measurement, Section 3.3: SRAM tests, Section 3.4: Flash memory integrity tests, Section 3.6: Firmware configuration parameters, Section 4: Compliance with IEC, UL and CSA standards, Section 5.2.1: Projects included in the package, Section 5.2.3: Defining new safety variables and memory areas under check and Section 5.4.3: Debugging the package.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Table 1: Overview of STL packages, Table 3: Used IDEs and toolchains, Table 4: Structure of the common STL packages, Table 6: Integration support files, Table 7: Compatibility between different STM32 microcontrollers, Table 8: How to manage compatibility aspects and configure STL package, Table 12: Signals used for timing measurements and Table 13: Comparison of results.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Figure 1: Example of RAM memory configuration.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Section 3.9: Incompatibility with previous versions of the STL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Table 3: Used IDEs and toolchains.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minor text edits across the whole document.</td>
</tr>
</tbody>
</table>
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. For additional information about ST trademarks, please refer to www.st.com/trademarks. 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.

© 2019 STMicroelectronics – All rights reserved