Application Note

Safety Startup Application Note

Introduction

Safety plays an increasingly important role in the field of electronic applications. In electronic design, the level of component safety requirements is constantly increasing, and electronic device manufacturers are incorporating many new technology solutions into new component designs. Software technologies are constantly emerging to improve safety. Standards for hardware and software safety requirements are also under continuous development.

This document describes how the project in N32G003 MCU to perform the requirements of IEC60730 software safety related operations, as well as related application code content.

This document applies to the N32G003 series products of Nsing Technologies.

Content

1. IEC60730 Class B Software Standard

To ensure the safety of electrical appliances, risk control measures during software operation need to be evaluated.

IEC60730, issued by the International Electrotechnical Commission, introduces the requirements for the evaluation of software for household appliances. In Appendix H (H.2.21), software is classified as follows:

Class A software: the software only implements the functions of the product and does not involve the safety control of the product. For example, software for room thermostats, lighting controls, etc.

Class B software: the software is designed to prevent unsafe operation of electronic devices. For example, the washing machine software with automatic door lock control, the induction cooker software with overheating control, etc.

Class C software: the software is designed to avoid certain specific hazards. For example, automatic burner control and hot break of closed water heater (mainly for some explosive equipment), etc.

The specific evaluation requirements of class B software include components to be tested and related faults and test schemes, which are sorted out in the following table (refer to IEC60730 Table H.11.12.7):

Table 1-1 The Specific Evaluation Requirements of Class B Software

Components to be Detected		Fault/Error	Fault Classification	Nsing with Library	Test Solution Overview
1.CPU	1.1 Register	Hysteresis (Stuck at)	MCU related	Y	Write relevant registers and check
	1.3 Program counter	Hysteresis (Stuck at)	MCU related	Y	When the PC runs fly, start the watchdog reset
2.Interruption		No interrupts or interrupts too frequently	Application of the relevant	N	Count the number of interrupts
3. The clock		Incorrect frequency	MCU related	Y	Use HSI to measure LSI clock frequency
4. Memory	4.1 Non-volatile memory	All single bit errors	MCU related	Y	Flash CRC integrity check
	4.2 Volatile memory	DC fault	MCU related	Y	1. SRAM March C test 2. Stack overflow detection
	4.3 Addressing (related to non-volatile and volatile memory)	Hysteresis (Stuck at)	MCU related	Y	Flash/SRAM tests are included

5. Internal data path	5.1 Data	Hysteresis (Stuck at)	MCU related	N	Only for MCU using external memory, monolithic MCU is not required
	5.2 Addressing	Incorrect address	MCU related	N	
External communication	6.1 Data	The Hamming distance is 3	Application of the relevant	N	Add verification in data transfer
	6.2 Addressing	Incorrect address	Application of the relevant	N	
	6.3 Timing	Incorrect timing	Application of the relevant	N	Count the number of communication events
7. Input and output	7.1 Digital I/O	Error defined in H27	Application of the relevant	N	None
	7.2 Analog input and output	Error defined in H27	Application of the relevant	N	None

2. Test Point Process Description

Class B software package program detection content is divided into two main parts: self-check at startup and periodic self-check at runtime. Self-check at startup includes:

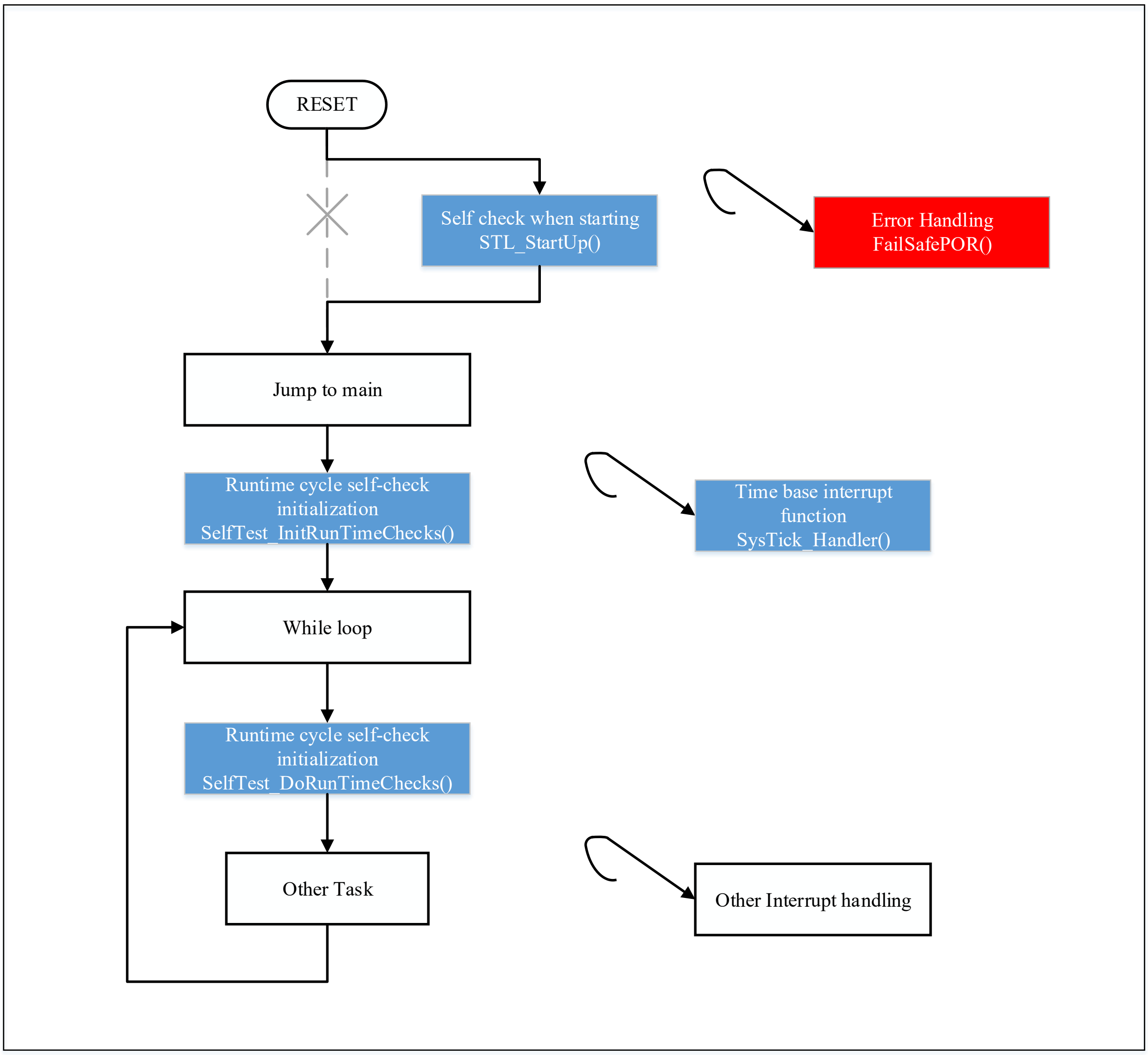
- CPU self-check
- Watchdog self-check
- Flash integrity self-check
- RAM function self-check
- System clock self-check
- Control flow self-check

Periodic self-check at runtime:

- Local CPU core register self-check
- Stack boundary overflow self-check
- System clock runtime self-check
- Flash CRC segment self-check
- Watchdog self-check
- Local RAM self-check (performed in interrupt service routines)

The overall flow diagram is as follows:

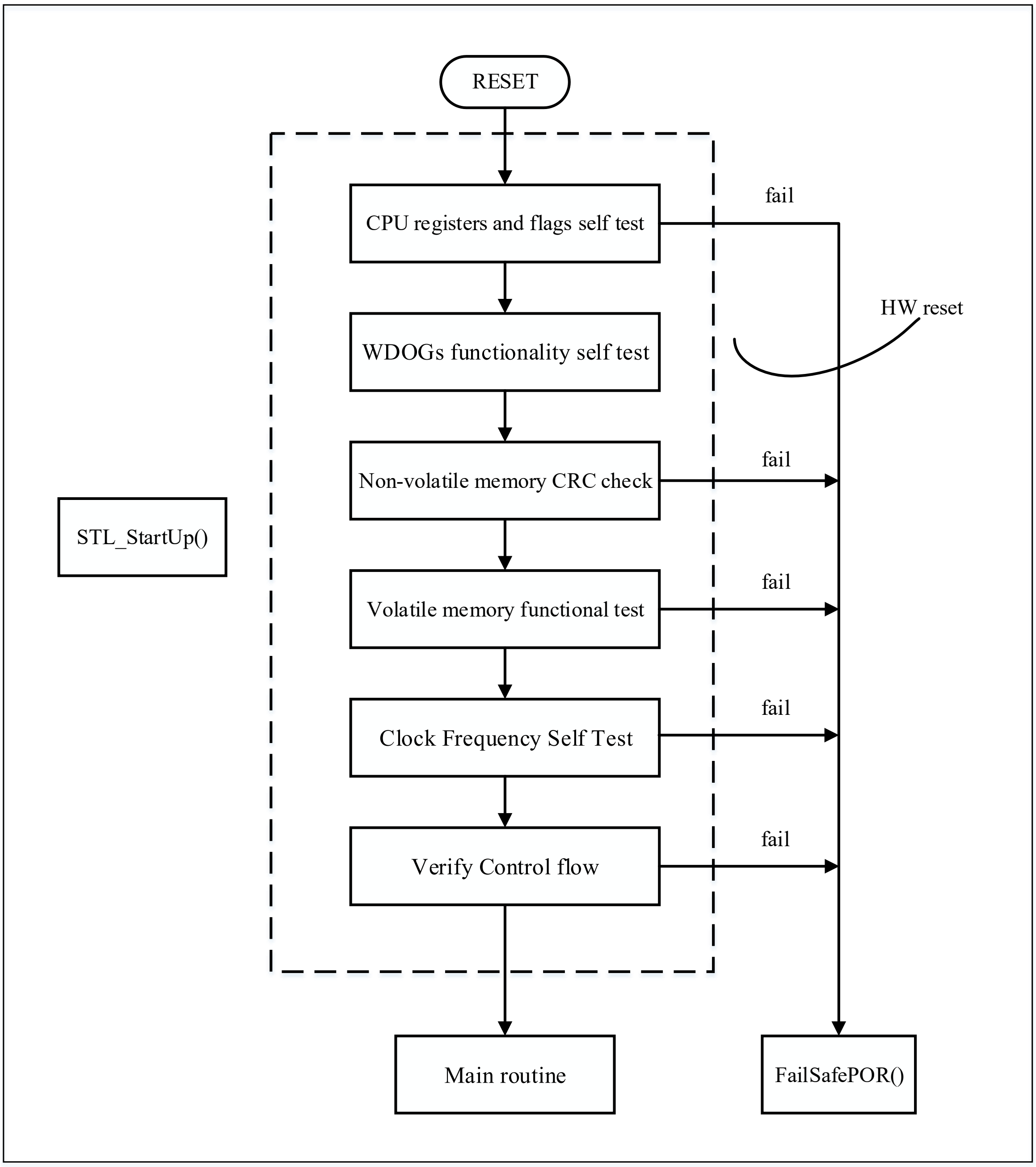
Figure 2-1 Class B Software Self-Check Flow Diagram



2.1 Startup Self-check Process

Before the chip enters **main** function from startup, the startup detection is performed first, and the startup file is modified to execute this part of the code. After the detection process is completed, the **__iar_program_start** function is called to jump back to **main** function.
The following is a flow diagram for performing a startup self-check:

Figure 2-2 Class B Software Self-Check Flow Diagram



2.1.1 CPU Startup Self-check

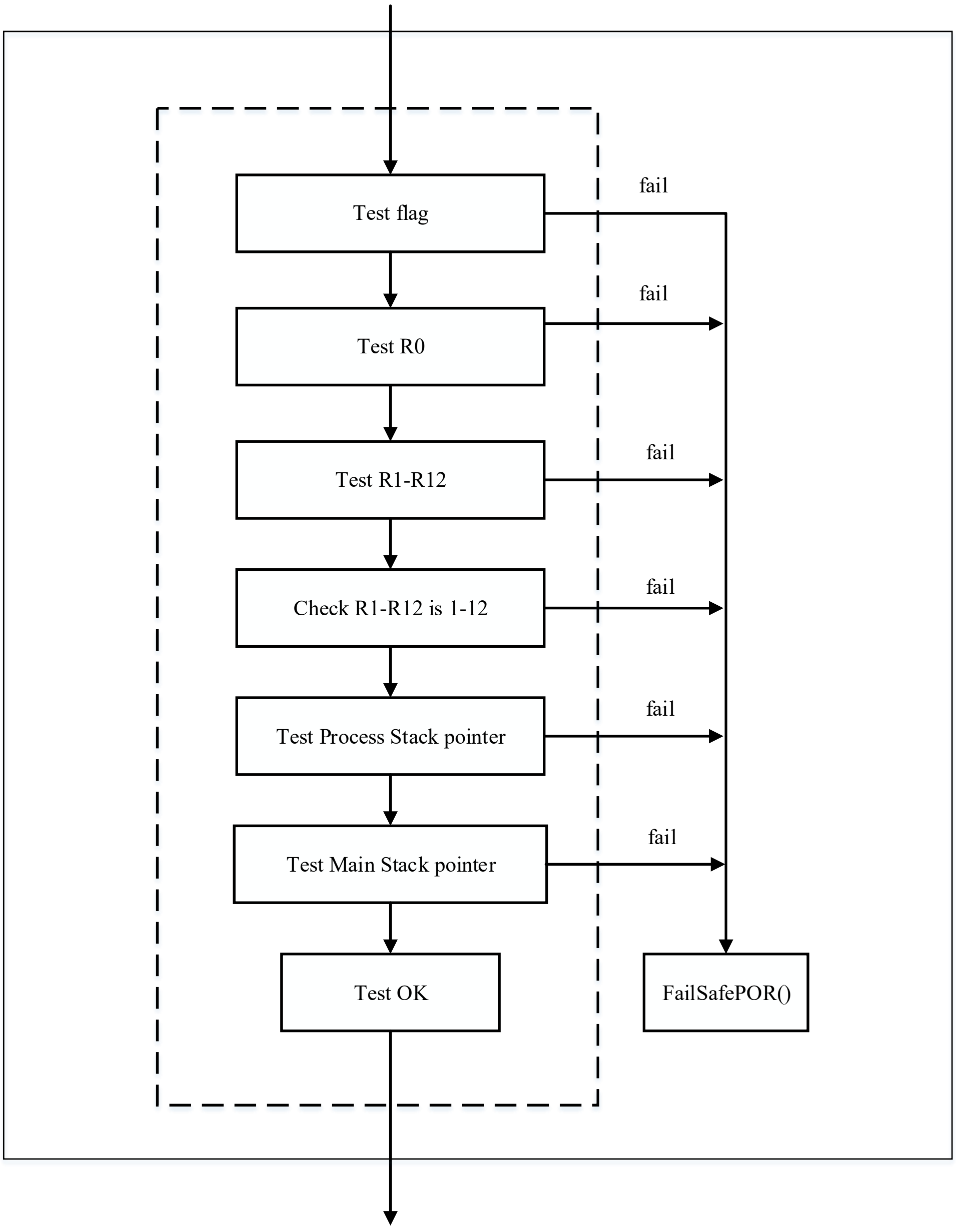
CPU self-check mainly checks whether the core flags, registers and so on are correct. If an error occurs, the fault-safe handing function **FailSafePOR()** is called.

CPU self-check is performed both at startup and runtime. At startup, a functional test of the flags in R0~R12, PSP, MSP registers and Z(zero), N(negative), C(carry), V(overflow) is conducted once as part of self-check process; At runtime, periodic self-check only detect registers R1~R12.

The specific implementation method for register detection is as follows: write 0xAAAAAAAA and 0x55555555 to the registers respectively, and then compare the read values to check if they match the written value. Write 1 after R1 is tested, write 2 after R2 is tested, and so on.

The specific implementation method of flag detection is as follows: set the flag respectively. If any flag is checked incorrectly, proceed the fault handling function. The detection flow diagram is as follows:

Figure 2-3 CPU Startup Self-check Flow Diagram



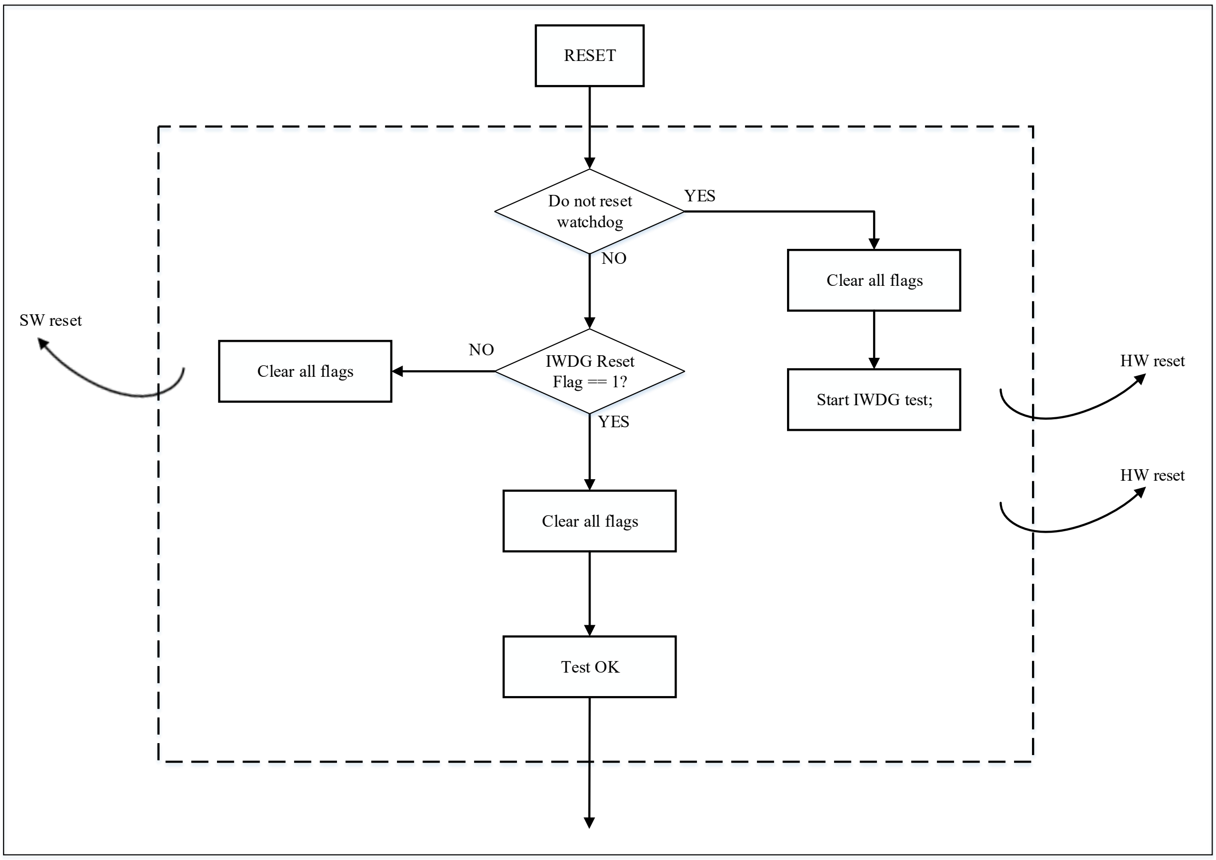
2.1.2 Watchdog Startup Self-check

Test to verify that independent watchdog can be reset correctly, ensuring that it can be reset in time to prevent jam during program execution.

After the initial reset, clear all reset status register flags, start the IWDG test to reset the chip. If the IWDG reset flag is set, the watchdog test passes, and then clear all flags.

The flow diagram is as follows:

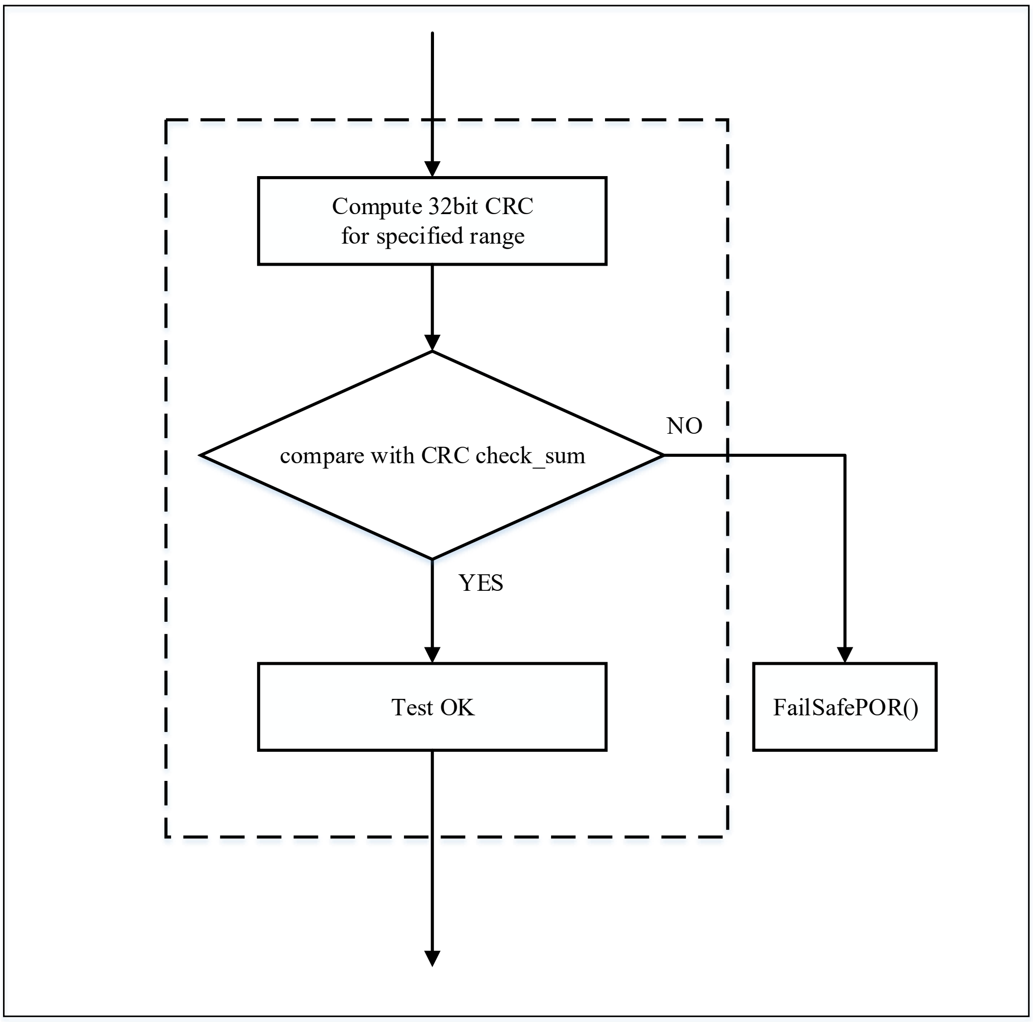
Figure 2-4 Watchdog Startup Self-check Flow Diagram



2.1.3 FLASH Startup Self-check

Flash self-check is a program that calculates Flash data with CRC algorithm. Then compares the result value with the CRC value calculated during compilation and stored in the specified location of Flash to confirm the integrity of Flash.
The flow diagram is as follows:

Figure 2-5 Flash Startup Self-check Flow Diagram

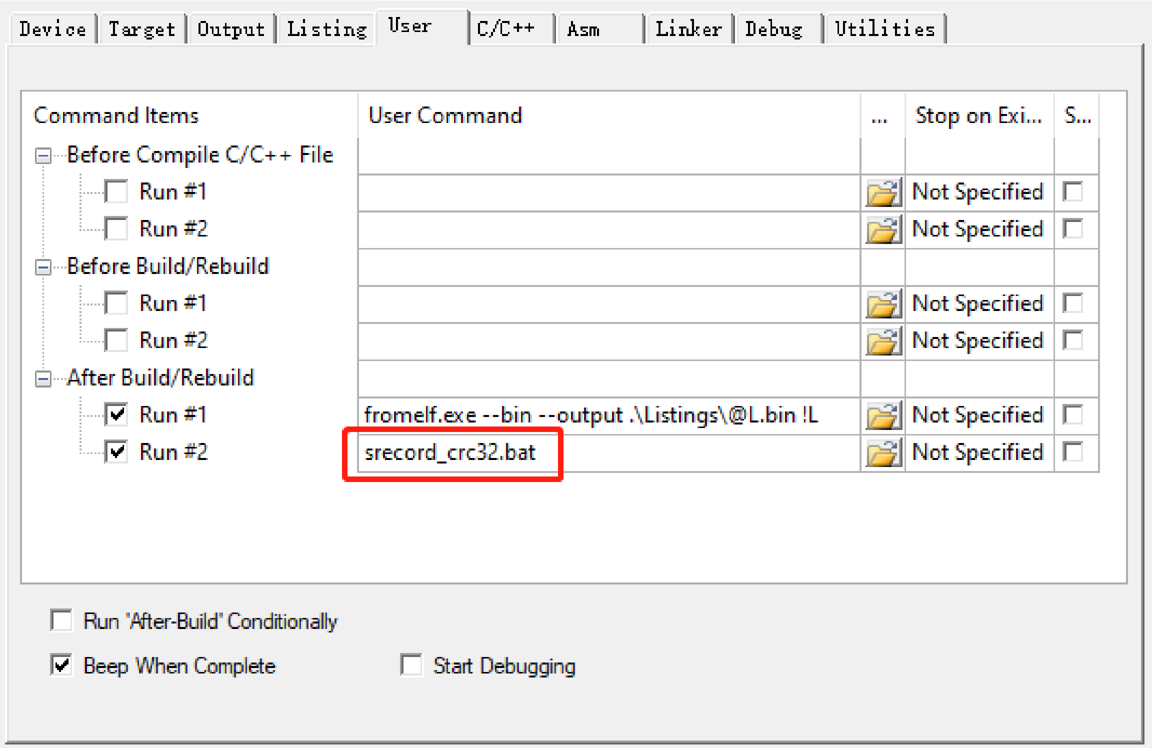


Flash range for CRC calculation is configured according to the actual situation of the whole program:

The configuration of Keil is more complicated. ARM officially recommends using the third-party software SRecord for ROM Self-Test in MDK-ARM.

According to the project configuration, after the compilation is completed, the script file srecord_crc32.bat will be called. Through the srec_cat.exe software, the data in the N32G003_SelfTest.hex file generated by Keil compilation will be calculated to generate CRC verification result. This result is then added to the specified location to obtain a new N32G003_SelfTest_CRC.hex file:

Figure 2-6 Flash Calculation Range Configuration at Linker



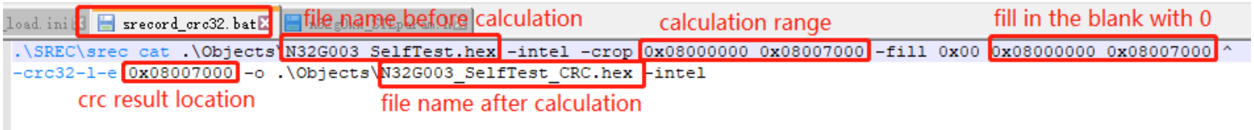
Command Items	User Command	...	Stop on Exi...	S...
⊟ Before Compile C/C++ File				
☐ Run #1			Not Specified	☐
☐ Run #2			Not Specified	☐
⊟ Before Build/Rebuild				
☐ Run #1			Not Specified	☐
☐ Run #2			Not Specified	☐
⊟ After Build/Rebuild				
☑ Run #1	fromelf.exe --bin --output .\Listings\@L.bin !L		Not Specified	☐
☑ Run #2	srecord_crc32.bat		Not Specified	☐

☐ Run 'After-Build' Conditionally

☑ Beep When Complete ☐ Start Debugging

Open the .bat file with Notepad or other tools, and modify the following according to the actual application:

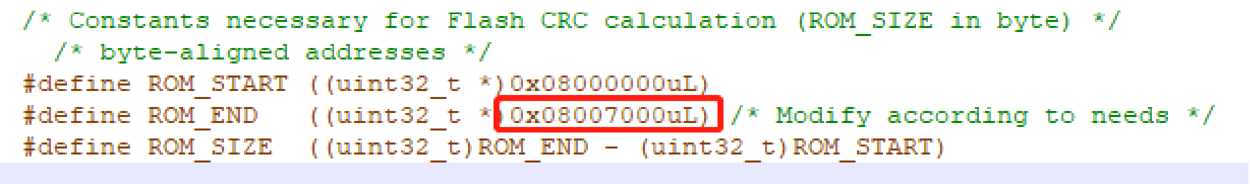
Figure 2-7 Configuration of srecord_crc32.bat



```
.\SREC\srec_cat .\Objects\N32G003_SelfTest.hex -intel -crop 0x08000000 0x08007000 -fill 0x00 0x08000000 0x08007000 ^
-crc32-l-e 0x08007000 -o .\Objects\N32G003_SelfTest_CRC.hex -intel
```

Range of calculating CRC in the program is configured in the n32g003_STLparam.h file, which can be modified according to the requirements to align with the configuration mentioned above:

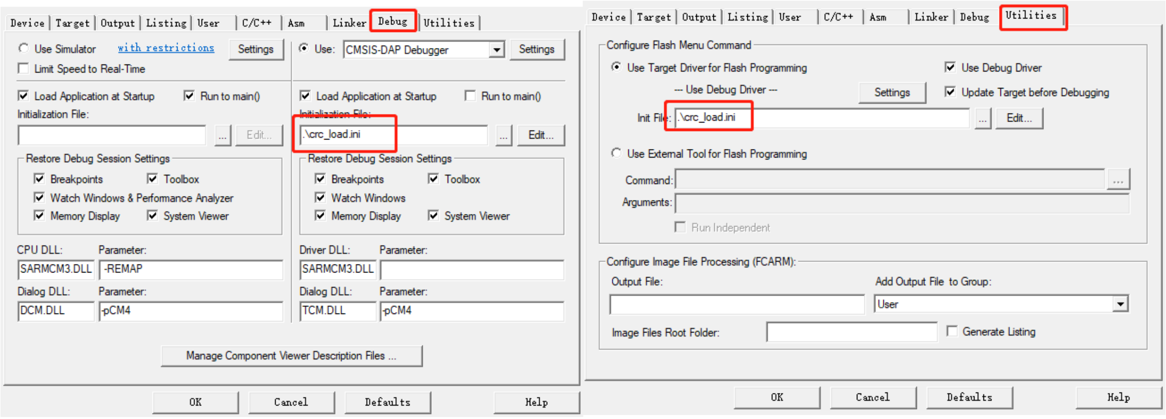
Figure 2-8 Configuration of Range of CRC Calculation



```
/* Constants necessary for Flash CRC calculation (ROM_SIZE in byte) */
  /* byte-aligned addresses */
#define ROM_START ((uint32_t *)0x08000000uL)
#define ROM_END   ((uint32_t *)0x08007000uL) /* Modify according to needs */
#define ROM_SIZE  ((uint32_t)ROM_END - (uint32_t)ROM_START)
```

Therefore, whether it is for downloading or debugging, the final generated N32G003_SelfTest_CRC.hex file needs to be used, so the .ini file needs to be added to the Keil configuration option to download the new .hex file. The configuration is as follows:

Figure 2-9 .ini File is Added to Keil Configuration



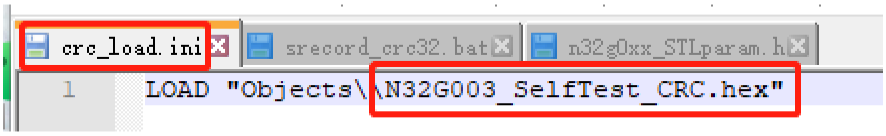
○ Use Simulator
☐ Limit Speed to Real-Time
☑ Load Application at Startup ☑ Run to main()
Initialization File:
☑ Breakpoints ☑ Toolbox
☑ Watch Windows & Performance Analyzer
☑ Memory Display ☑ System Viewer
CPU DLL: SARMCM3.DLL Parameter: -REMAP
Dialog DLL: DCM.DLL Parameter: -pCM4

● Use: CMSIS-DAP Debugger
☑ Load Application at Startup ☐ Run to main()
Initialization File: .\crc_load.ini
☑ Breakpoints ☑ Toolbox
☑ Watch Windows
☑ Memory Display ☑ System Viewer
Driver DLL: SARMCM3.DLL Parameter:
Dialog DLL: TCM.DLL Parameter: -pCM4

Configure Flash Menu Command
● Use Target Driver for Flash Programming ☑ Use Debug Driver
--- Use Debug Driver --- ☑ Update Target before Debugging
Init File: .\crc_load.ini
○ Use External Tool for Flash Programming
Command:
Arguments:
☐ Run Independent
Configure Image File Processing (FCARM):
Output File: Add Output File to Group: User
Image Files Root Folder: ☐ Generate Listing

It should be noted that the .ini file should be configured with the file name of the actual application requirements that need to be modified

Figure 2-10 Configuration of crc_load.ini



```
LOAD "Objects\\N32G003_SelfTest_CRC.hex"
```

2.1.4 RAM Startup Self-check

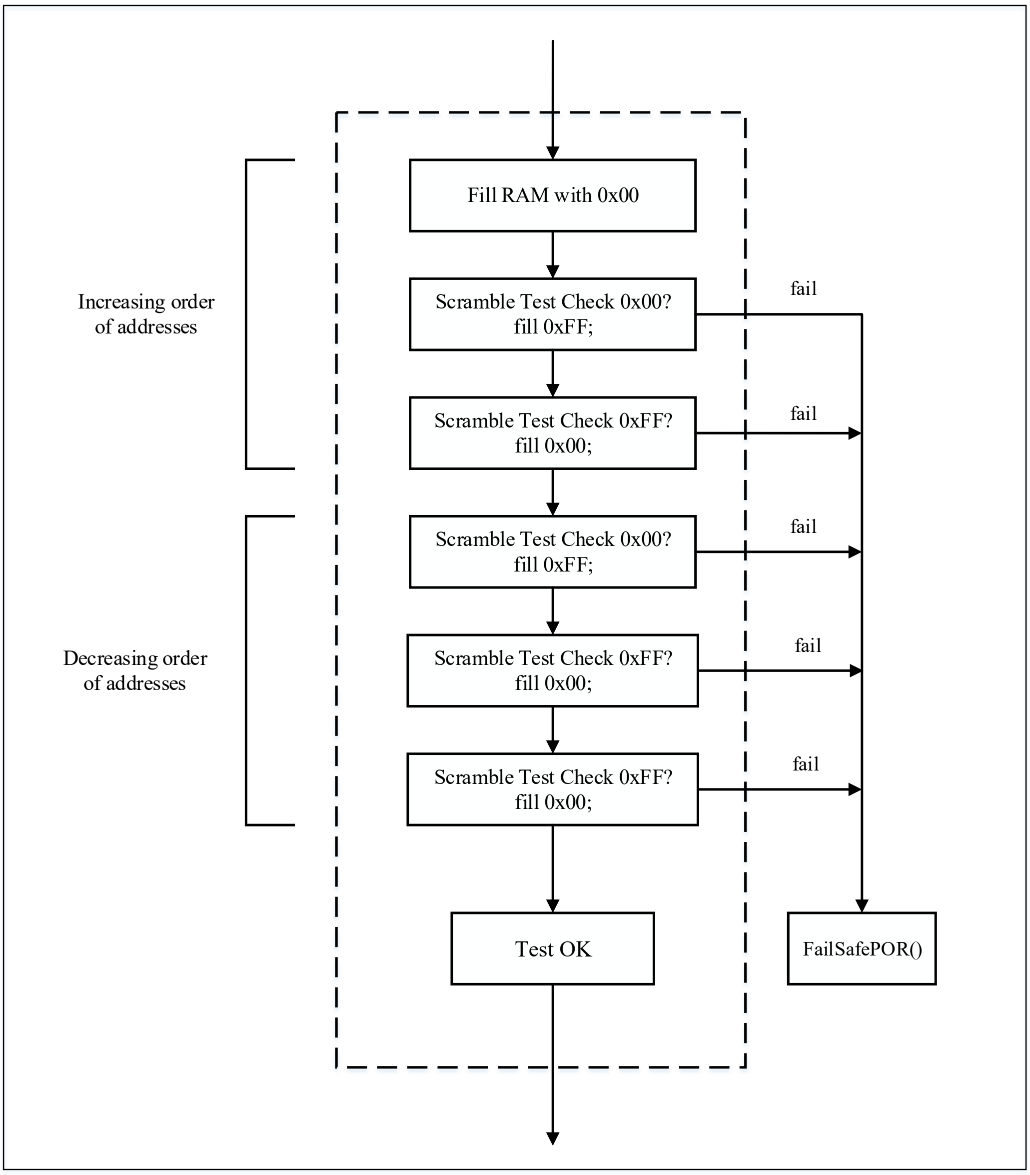
SRAM detection detects errors not only in the data region, but also in its internal address and data path.

SRAM self-check uses the March-C algorithm, which is an algorithm used for SRAM testing of embedded chips and is a part of safety certification. All ranges of SRAM are detected at startup. Firstly, the entire SRAM is cleared, and then each bit is set to 1 sequentially. After setting each bit, it is tested whether the bit is indeed 1. If it is, the process continue; if not, an error is reported. After all bits are set, each bit is cleared to 0 sequentially. After clearing each bit, it is tested whether the bit is cleared to 0. If it is, it is considered correct; otherwise, an error is reported. This process continues until the entire RAM space is test.

The test is divided into 6 loops, and the entire RAM is checked and filled word by word alternately with the values 0x00 and 0xFF. The first 3 loops are executed in increasing address order, while the last 3 cycles are executed in decreasing address order.

The entire RAM detection algorithm process is shown in the figure below:

Figure 2-11 RAM Startup Self-check Flow Diagram



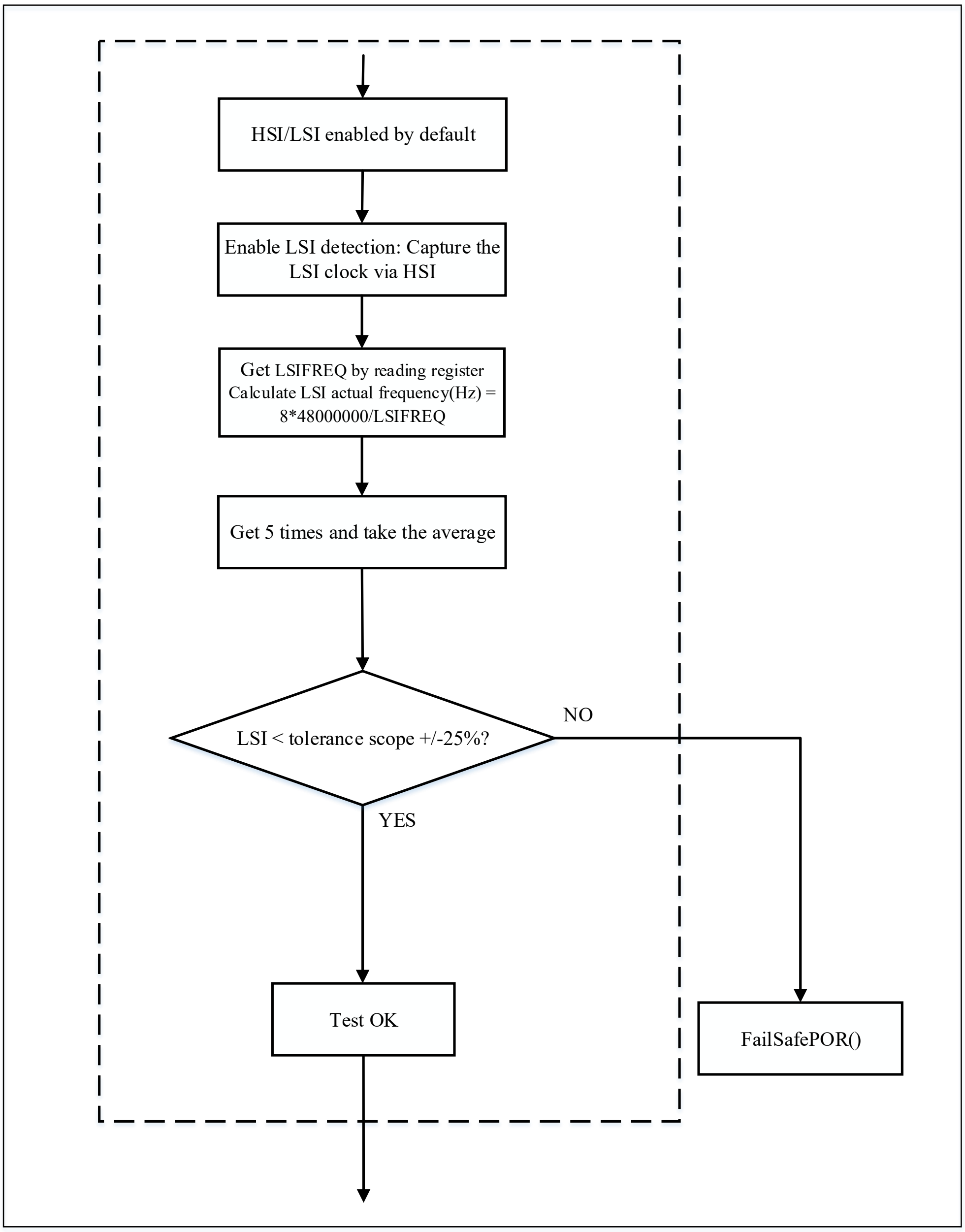
2.1.5 Clock Startup Self-check

N32G003 provides LSI detection function, which involves capturing and calculating the actual frequency of LSI using HSI. The testing steps are as follows:

1. Enable RCC_LSICTRL.LSIDETEN to initiate LSI clock calibration.
2. Enquire RCC_LSICTRL.LSIDETFF flag. When it is 1, the calibration is completed. Read RCC_LSICTRL.LSIFREQ;
3. Obtain the RCC_LSICTRL.LSIFREQ multiple times to average the values. Then clear RCC_LSICTRL.LSIDETFF flag, and disable RCC_LSICTRL. LSIDETEN;
4. Calculate the LSI actual frequency according to the formula LSICLK(Hz) = 8*HSICLK/LSIFREQ (HSICLK frequency is 480000000), then compare this frequency value with the expected range value: if it exceeds +/-25%, the test fails.

The expected range value can be adjusted by the user according to the actual application. The macros are defined as LSI_LimitHigh and LSI_LimitLow.

Table 2-12 Clock Startup Self-check Flow Diagram



2.1.6 Control Flow Startup Self-check

The self-check part of the startup ends with the control flow detection pointer program. Initialize the variables CtrlFlowCnt to 0, CtrlFlowCntInv to 0xFFFFFFFF. In each test step, CtrlFlowCnt adds a fixed value, CtrlFlowCntInv subtracting the same fixed value. At the end of the startup self-check, judge whether the sum of the two values is still 0xFFFFFFFF.

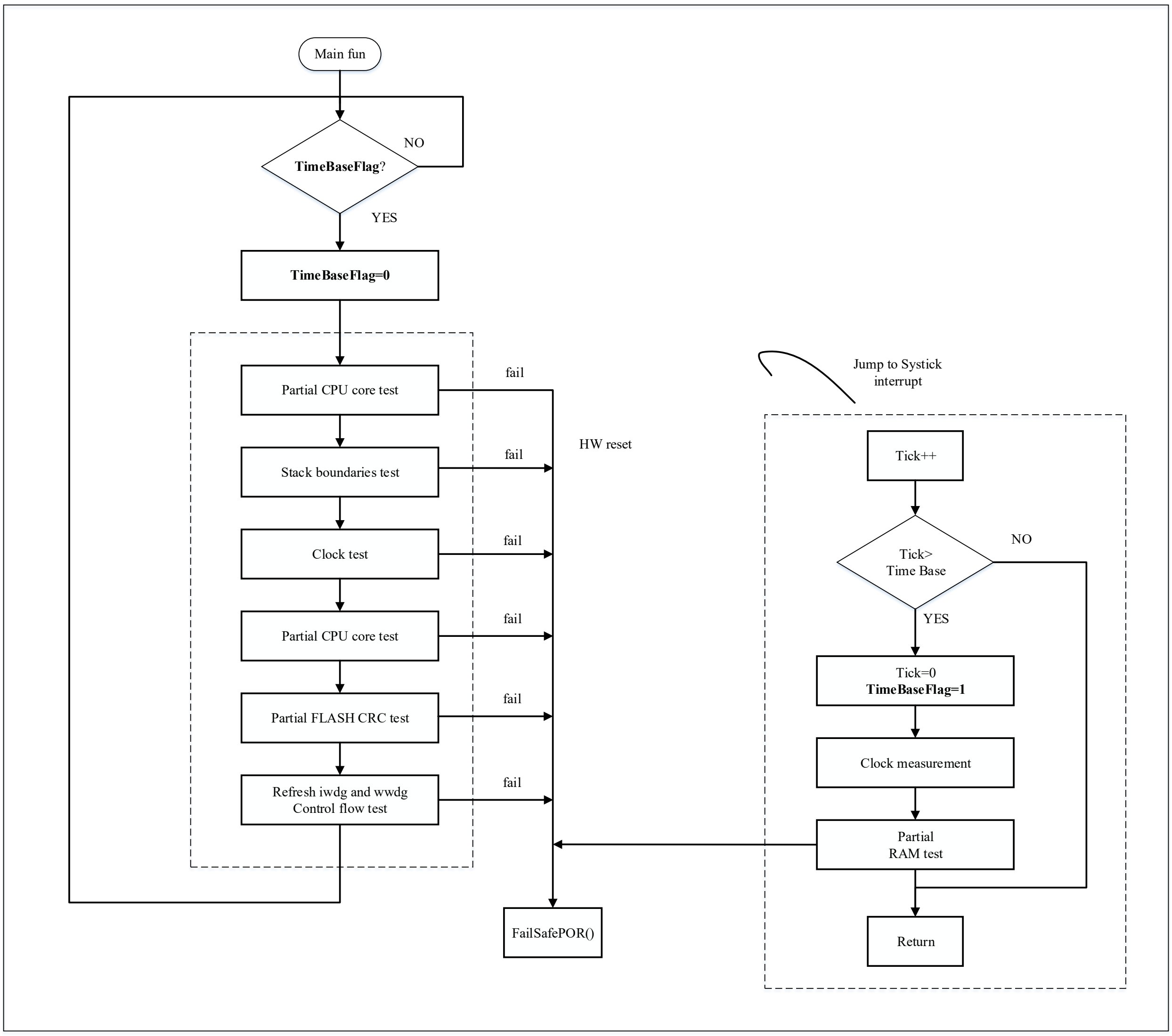
2.2 Runtime Self-check Process

If the startup self-check passes successfully, the runtime periodic self-check must be initialized before entering the main loop.

The runtime self-check is performed periodically based on SysTick.

The runtime periodic detection process is as follows:

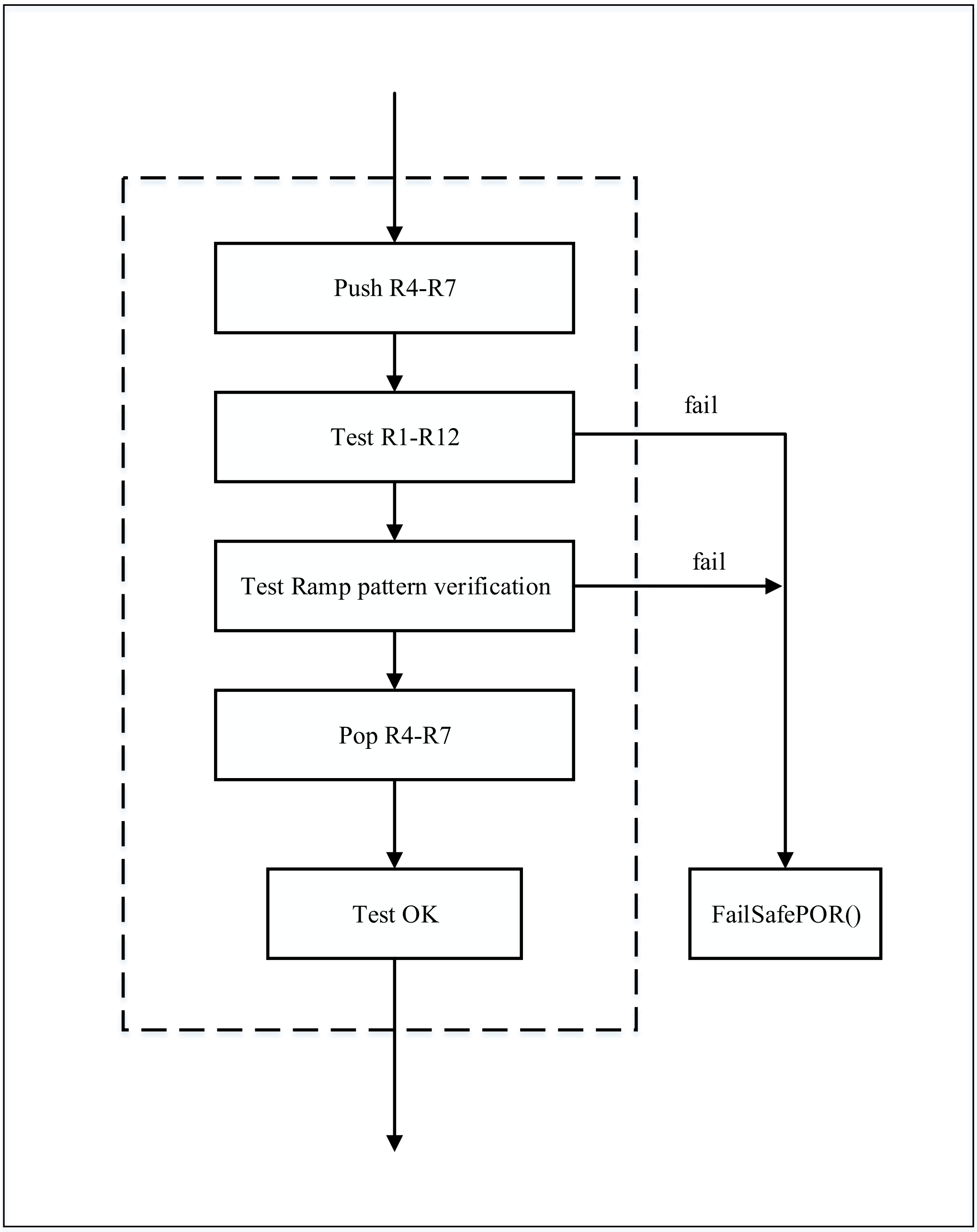
Figure 2-13 Runtime Self-check Flow Diagram



2.2.1 CPU Self-check

The CPU runtime periodic self-check is similar to the self-check at startup, except that the core flags and stack pointers are not detected.

Figure 2-14 CPU Runtime Self-check Flow Diagram



2.2.2 Stack Boundary Runtime Overflow Self-check

This test detects stack overflow by determining the data integrity of pattern array in the boundary detection area. If the original pattern data is corrupted, the test fails and a fault-tolerant program is invoked.

The lower address closely following the stack area is defined as the stack boundary detection area. This area can be configured differently depending on the device. The user must define enough areas for the stack and ensure that pattern is placed correctly.

Figure 2-15 Stack Area Diagram

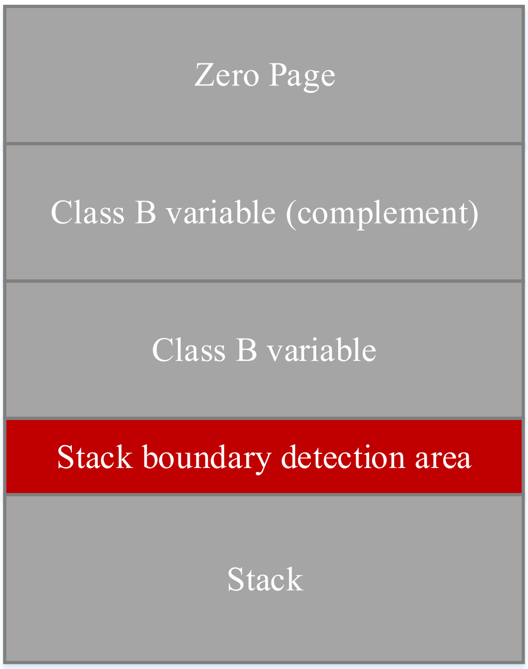
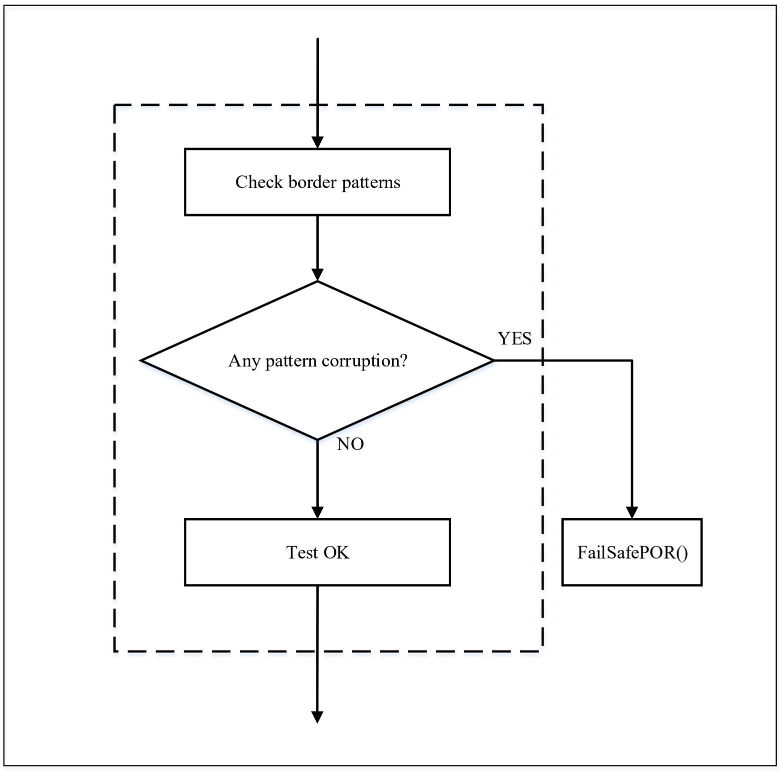


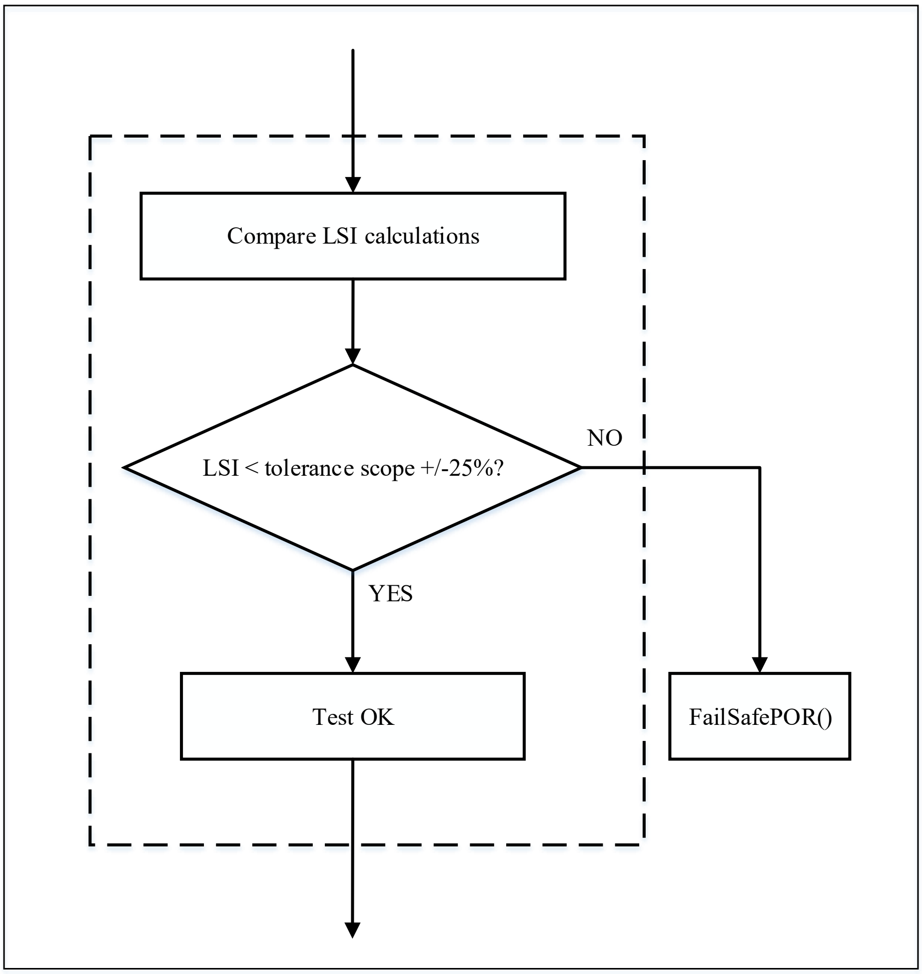
Figure 2-16 Stack Boundary Overflow Runtime Self-check Flow Diagram



2.2.3 System Clock runtime Self-check

The detection of the system clock at runtime is similar to the clock detection at startup, and the LSI calculation value is directly compared, and the flow is as follows:

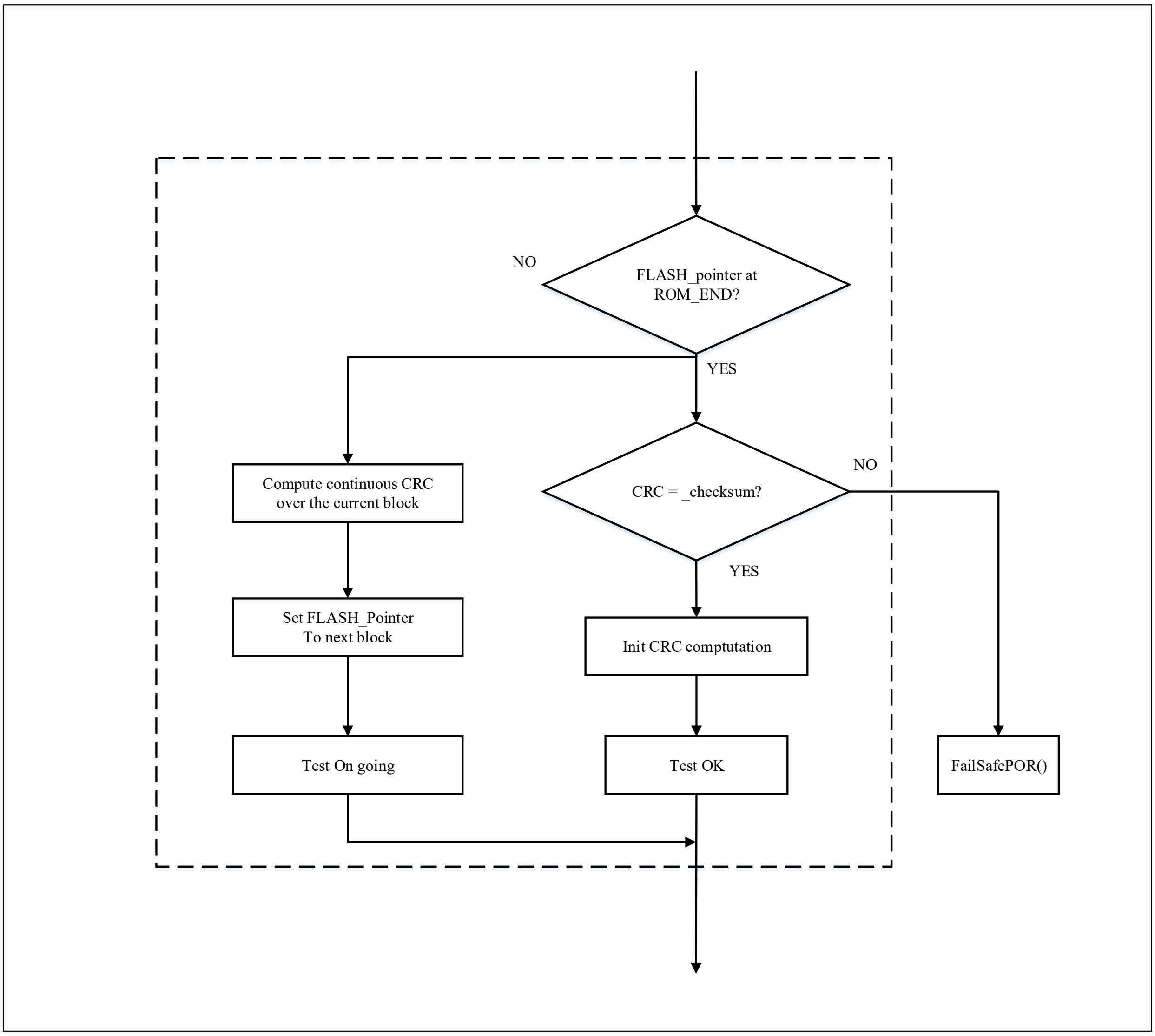
Figure 2-17 System Clock Runtime Self-check Flow Diagram



2.2.4 FLASH Runtime Self-check

The Flash CRC self-check can be performed during the runtime. Because the self-check tie length varies with the check range required, user can configure segmented CRC calculation based on the size of the user application. When the CRC values are calculated for the last segment range, the CRC values are compared. If they do not match, the test fails.

Figure 2-18 Flash Runtime Self-check Flow Diagram



2.2.5 Watchdog Runtime Self-check

During runtime, watchdog needs to be fed periodically to ensure the normal operation of the system. The watchdog feeding part is placed at the end of STL_DoRunTimeChecks().

2.2.6 Local RAM Runtime Self-check

The RAM self-check at runtime is performed in the SysTick interrupt function. The test covers only the portion of memory allocated to the class B variables.
According to the regions allocated for the class B variables, every 6 bytes is a block. Before the March-C test, the block data is saved in the RunTimeRamBuf, and then the RunTimeRamBuf is placed back to the original location of the class B after the test is completed. This process continues until all tests in the class B region are completed.
After the class B region test is completed, March-C test is performed on the RunTimeBuf region. After the test is completed, the pointer is restored to the class B start address for the next test.

Figure 2-19 Local RAM Region Diagram

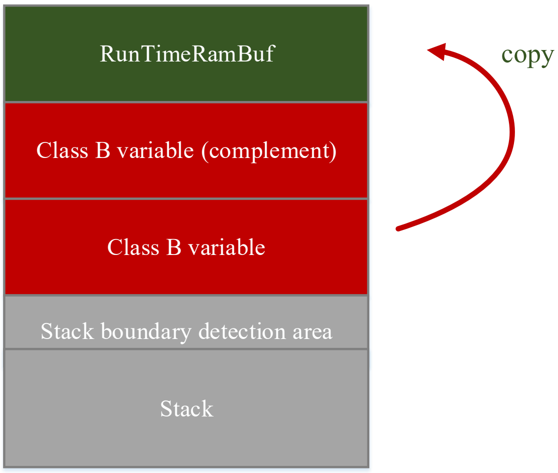
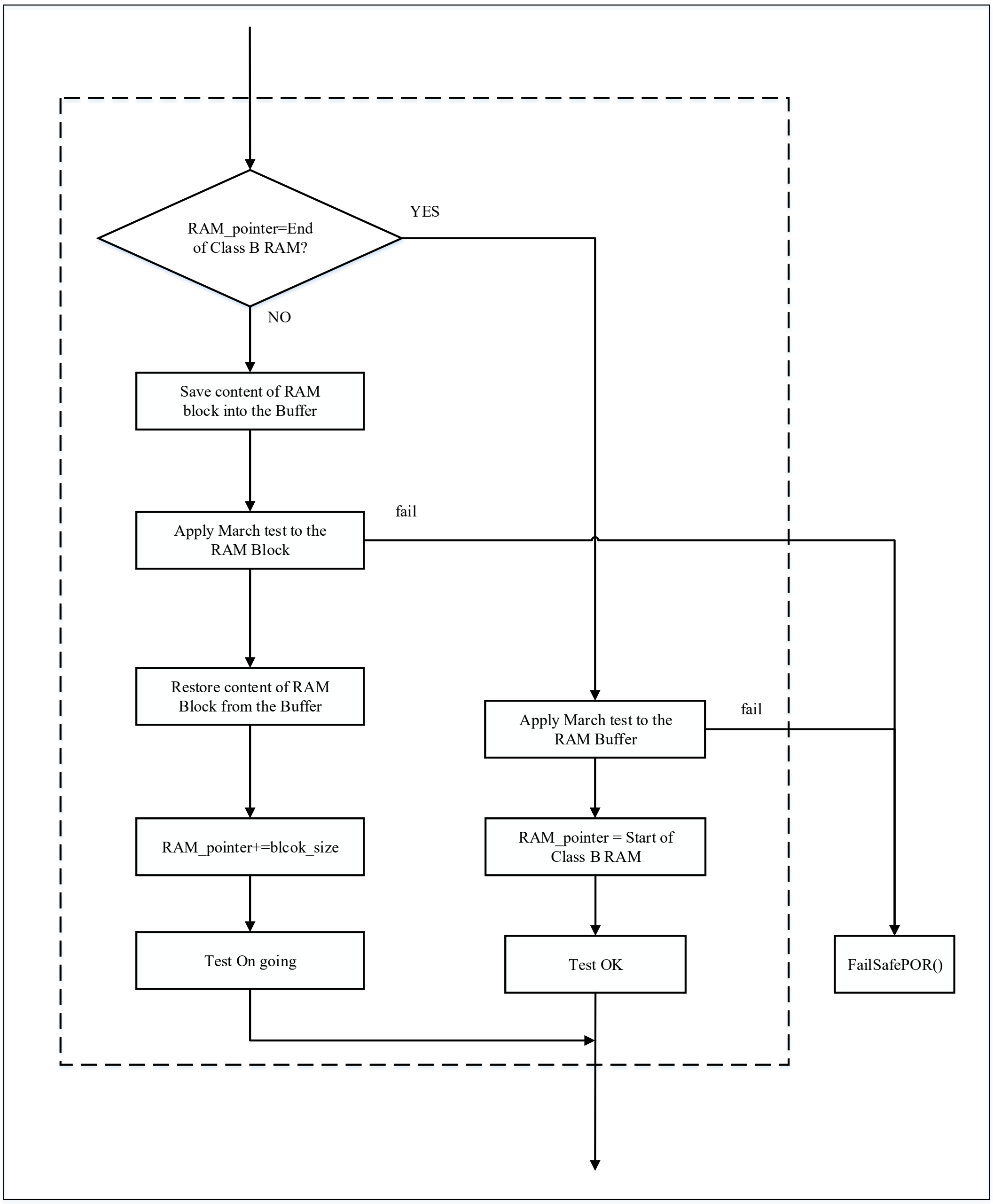


Figure 2-20 Local RAM Runtime Self-check Flow Diagram

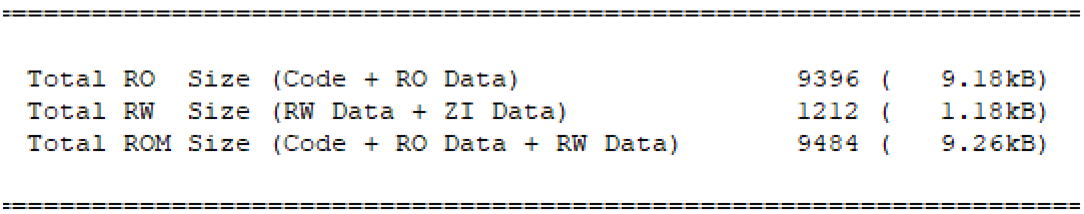


3. Key Points of Software Library Migration

- Before executing the user program, execute the STL_StartUp function (to start the self-check)
- Set IWDG to prevent it from being reset when the program is running properly
- Set up RAM and Flash detection range at startup and runtime
  - The range of CRC checksum, and the location where the checksum is stored in the Flash
  - The range of storage addresses for Class B variables
  - Location of stack boundary detection region
- Troubleshoot detected faults
- Add user-related fault detection content based on specific applications
- Define the frequency of program runtime self-check according to the specific application
- After the chip is reset, the STL_StartUp function must be called for startup self-check before initialization
- Call STL_InitRunTimeChecks() before entering the main loop, and call STL_DoRunTimeChecks() in the main loop
- Users can release Verbose comments to enter diagnostic mode and output text information through the Tx pin(PA2) of UART1

Set the serial port to 115200Bits/s, no parity, 8-bit data, and 1 stop bit

- The memory occupied by the self-test library is as follows:



```
==============================================================================

  Total RO  Size (Code + RO Data)                9396 (   9.18kB)
  Total RW  Size (RW Data + ZI Data)             1212 (   1.18kB)
  Total ROM Size (Code + RO Data + RW Data)      9484 (   9.26kB)

==============================================================================
```

4. Version History

Version	**Date**	**Changes**
V1.0	2023.01.03	Initial release

5. Disclaimer

This document is the exclusive property of NSING TECHNOLOGIES PTE. LTD.(Hereinafter referred to as NSING).

This document, and the product of NSING described herein (Hereinafter referred to as the Product) are owned by NSING under the laws and treaties of Republic of Singapore and other applicable jurisdictions worldwide. The intellectual properties of the product belong to Nations Technologies Inc. and Nations Technologies Inc. does not grant any third party any license under its patents, copyrights, trademarks, or other intellectual property rights. Names and brands of third party may be mentioned or referred thereto (if any) for identification purposes only. NSING reserves the right to make changes, corrections. enhancements, modifications, and improvements to this document at any time without notice. Please contact NSING and obtain the latest version of this document before placing orders. Although NATIONS has attempted to provide accurate and reliable information, NATIONS assumes no responsibility for the accuracy and reliability of this document. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. In no event shall NATIONS be liable for any direct, indirect, incidental, special, exemplary, or consequential damages arising in any way out of the use of this document or the Product.

NATIONS Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, Insecure Usage'. Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, all types of safety devices, and other applications intended to supporter sustain life. All Insecure Usage shall be made at user's risk. User shall indemnify NATIONS and hold NATIONS harmless from and against all claims, costs, damages, and other liabilities, arising from or related to any customer's Insecure Usage Any express or implied warranty with regard to this document or the Product, including, but not limited to. The warranties of merchantability, fitness for a particular purpose and non-infringement are disclaimed to the fullest extent permitted by law. Unless otherwise explicitly permitted by NATIONS, anyone may not use, duplicate, modify, transcribe or otherwise distribute this document for any purposes, in whole or in part.