MM32 IEC 60730 ClassB For M0 Software Library

User Manual

V0.1

Contents

Introduction

This work instruction guides certification reviewers in identifying the implementation of specific safety mechanisms on the DUT.

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

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

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

Three classes are defined by the 60730-1 standard:

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

1 Overview

The basic structure is detailed, 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.

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 1*.

Table 1. Structure of the common STL packages

STL	Common STL stack source files	
	File	Descrption
Start-up test	IEC60730_B_startup.c	Start-up STL flow control & start-up test
	IEC60730_B_cpustartIAR.s	Cpu start-up self test
	IEC60730_B_cpustartKEIL.s	Cpu start-up self test
Run time test	IEC60730_B_runtimetest.c	Run time self tests structure
	IEC60730_B_flashtest.c	Partial Flash CRC run time self test
	IEC60730_B_transpRam.c	Partial RAM run time self test structure
	IEC60730_B_clocktest.c	Clock run time self test
	IEC60730_B_aux.c	It includes call interfaces for error handling functions, watchdog detection and initialization, CSS interrupt handling, SysTick time base handling, and runtime RAM detection.
Headers	IEC60730_B_clock.h	Clock test header
	IEC60730_B_cpu.h	CPU test header
	IEC60730_B_crc32.h	Flash memory test header
	IEC60730_B_init.h	Support interface header
	IEC60730_B_lib.h	Overall STL includes control
	IEC60730_B_param.h	Support param header
	IEC60730_B_Ram.h	RAM test header
	IEC60730_B_startup.h	Initial process STL header
	IEC60730_B_var.h	Support var header

Ic 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 IEC60730_B 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 an hazard thanks to another applied redundant software procedure or hardware protection function. This is not a case of much more severe Class C level, where fault at a safety critical software results in a hazard due to lack of next protection mechanisms.

2 Generic tests included in STL firmware package

The certified MindMotion 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

An overview of the methods used for the MCU-specific tests (described in deeper detail in the following sections) is given in *Table 2*.

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 2. Methods used in MindMotion specific tests of associated STL package

Component of Table R.1 to be verified	Method used	References to Annex R of IEC 60335-1 and Annex H of IEC 60730-1	
		Component(s)	Definition
CPU registers	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 nondirect methods) link register is tested by PC monitoring. If any error is found, the software jumps directly to the Fail Safe routine.	1.1	H.2.16.5 H.2.16.6 H.2.19.6
Program Counter	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.	1.3	H.2.18.10.2 H.2.18.10.4

Addressing And Data path	Not all the products satisfy the recognized methods for this test. This lack can be compensated by implementing a wider set of indirect methods like RAM functional and Flash memory integrity test, program timing and flow control, class B variables integrity and stack boundaries checks supported by other HW methods like proper handling of CPU exceptions.	4.3, 5.1 and 5.2	H.2.19.18.1 H.2.19.18.2 or Indirect methods
Clock	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.	3	H.2.18.10.1 H.2.18.10.4
Invariable Memory	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 CRC calculation unit is used.	4.1	H.2.19.4.1 H.2.19.8.1
Variable Memory	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.	4.2	H.2.19.6.2 H.2.19.8.2

3 Software package

This section highlights the basic common principles used in MindMotion software solution. The workspace organization is described together with its configuration and debugging capabilities. The differences between the supported development environments.The code base integration can be carried out based on the "Migration of Class B software library(Chapter 7)"

3.1 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 end of IEC60730_B_aux.c file. The goal of this procedure is to provide a unique output and allow the user to react immediately.

3.2 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*.

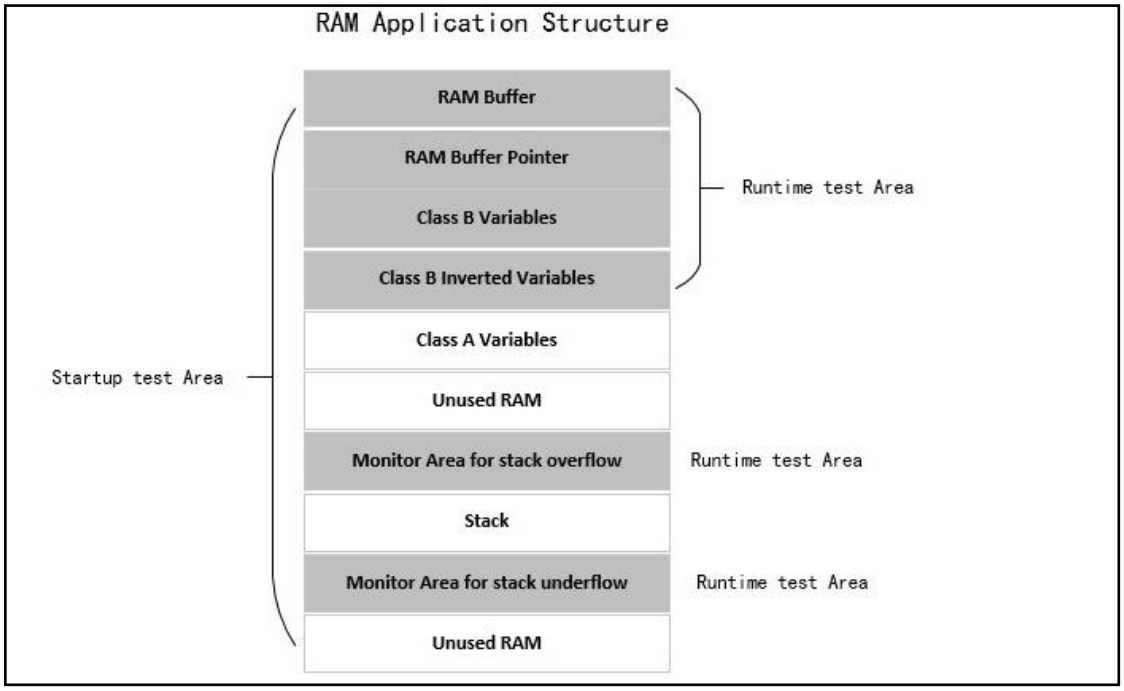


Figure 1. Example of RAM memory configuration

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.

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.

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.

3.3 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. This method is an efficient tool to identify bus matrix issues (e.g. data transfer, addressing). 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).

where a routine performing a component test is called in the controlled flow sequence and the four-step checking service is shown in *Figure 2*. 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.

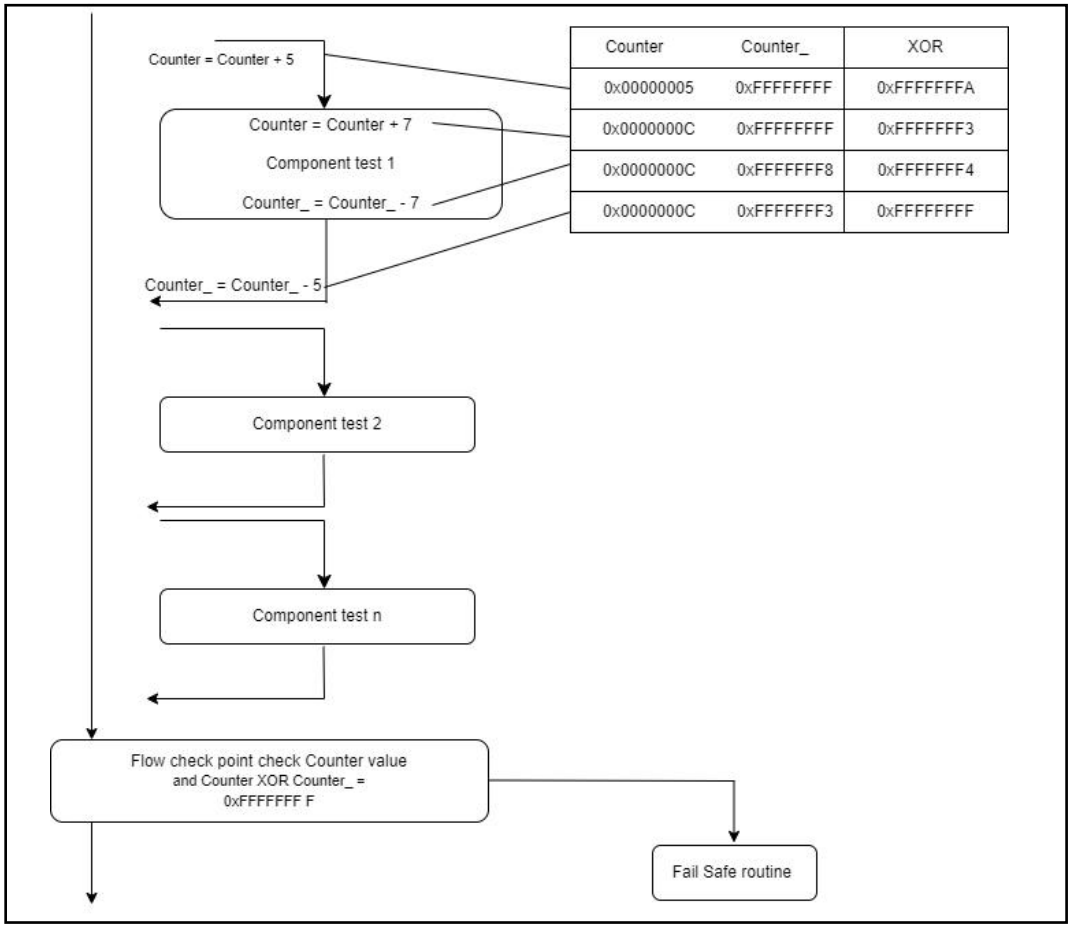


Figure 2. Four-step check principle of control flow

3.4 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.

3.5 Package configuration and debugging

The IEC60730_B 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 solution can be configured, modified and debugged.

Some run-time tests can be skipped depending on the end-application. If the periodicity of the test is connaturated 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.

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.

4 Software library structure (Program sequence)

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.

Redundancy is applied to all class B variables 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 3 shows the basic principle of Class B software package integration into user software solution.

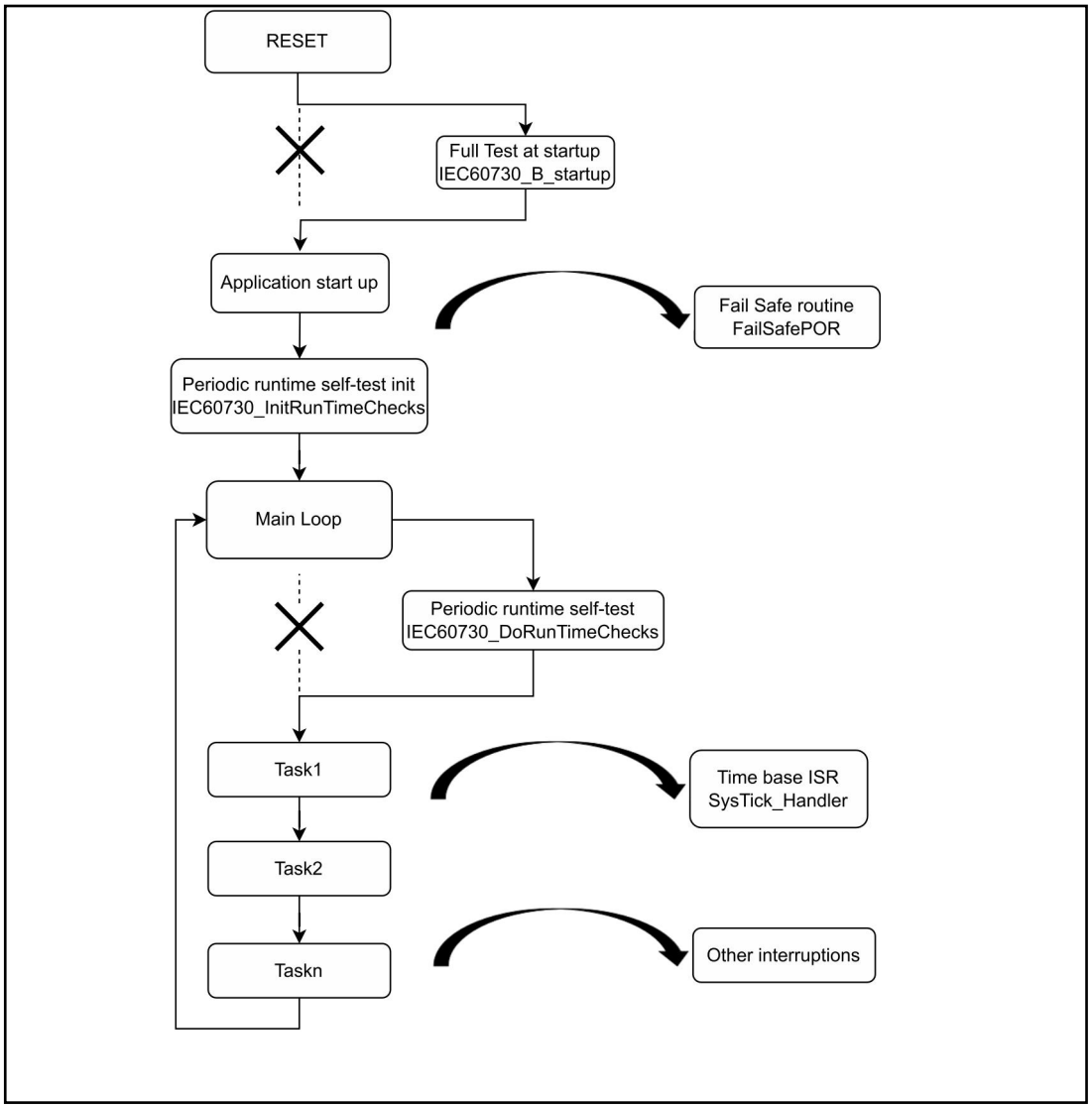


Figure 3. Integration of startup and periodic runtime self-tests into the application

In principle, the following steps must be provided by the user when IEC60730_B 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 IEC60730_B 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.
- 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.

4.1 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 4*.

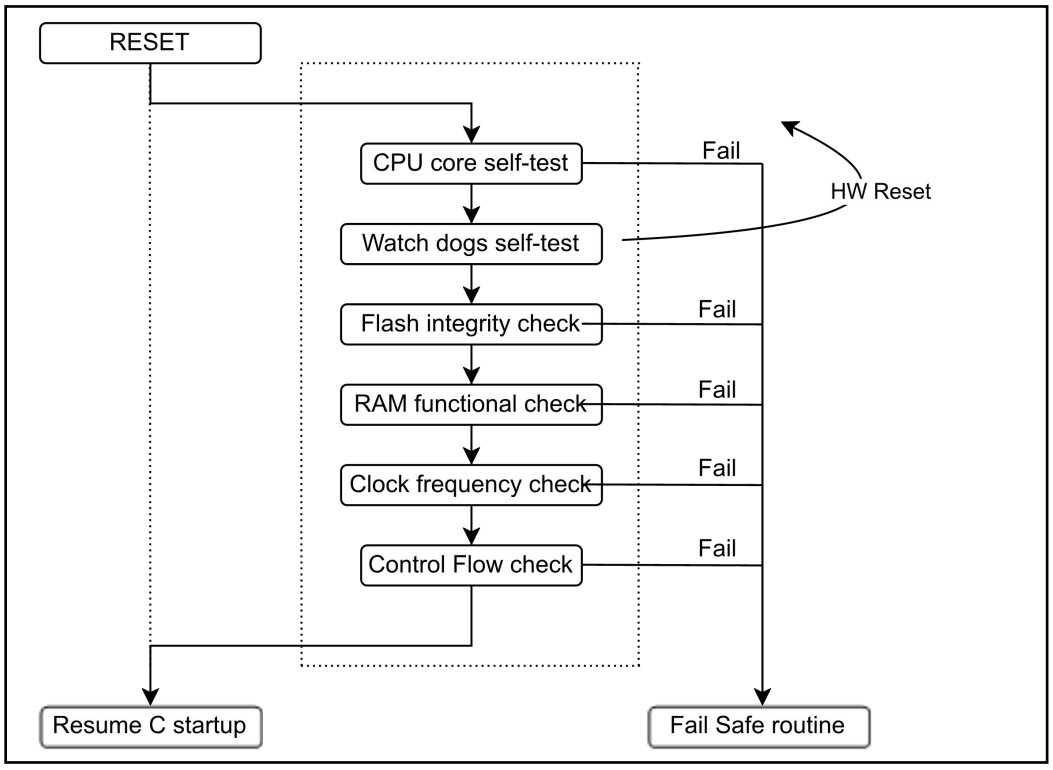


Figure 4. Startup self test structure

4.1.1 CPU start-up self test

CPU self check mainly checks whether the kernel flags, registers, and stack pointers are correct. If an error occurs, the fail safe handling function will be called. This part of the detection source code is written in assembly, and there are differences between KEIL and IAR environments.

The basic structure is shown in *Figure 5*.

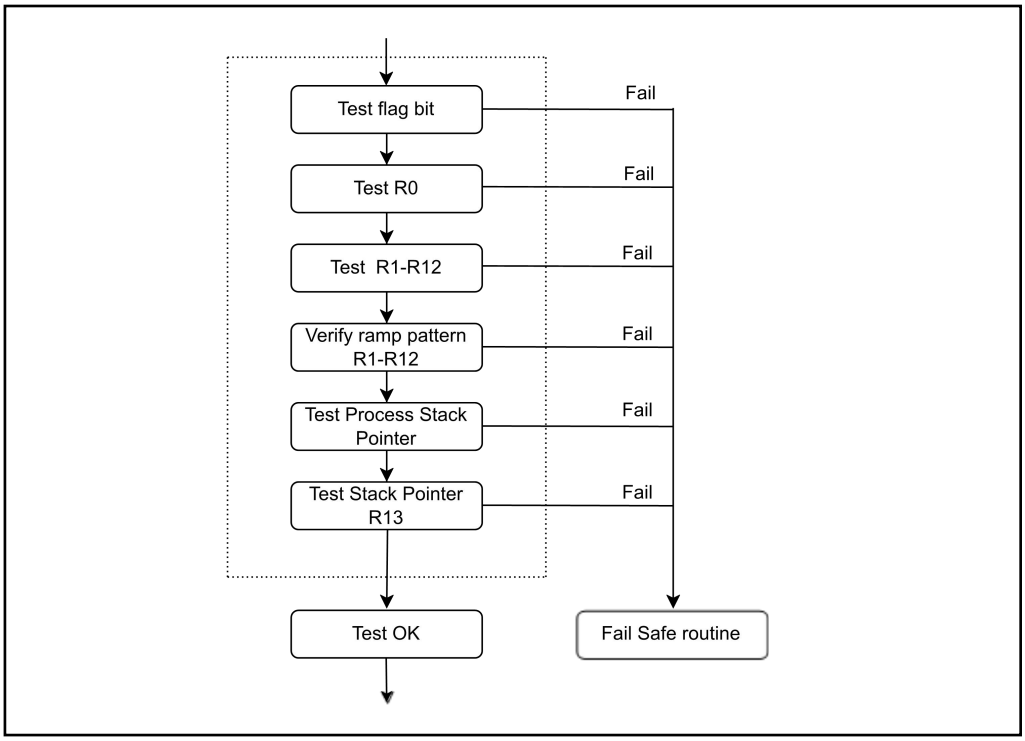


Figure 5. CPU startup self-test structure

4.1.2 Watchdog start-up self test

The test determines whether the execution was successful by determining the identification of the reset status register. After the test is completed, all flag bits are cleared (see *Figure 6*).

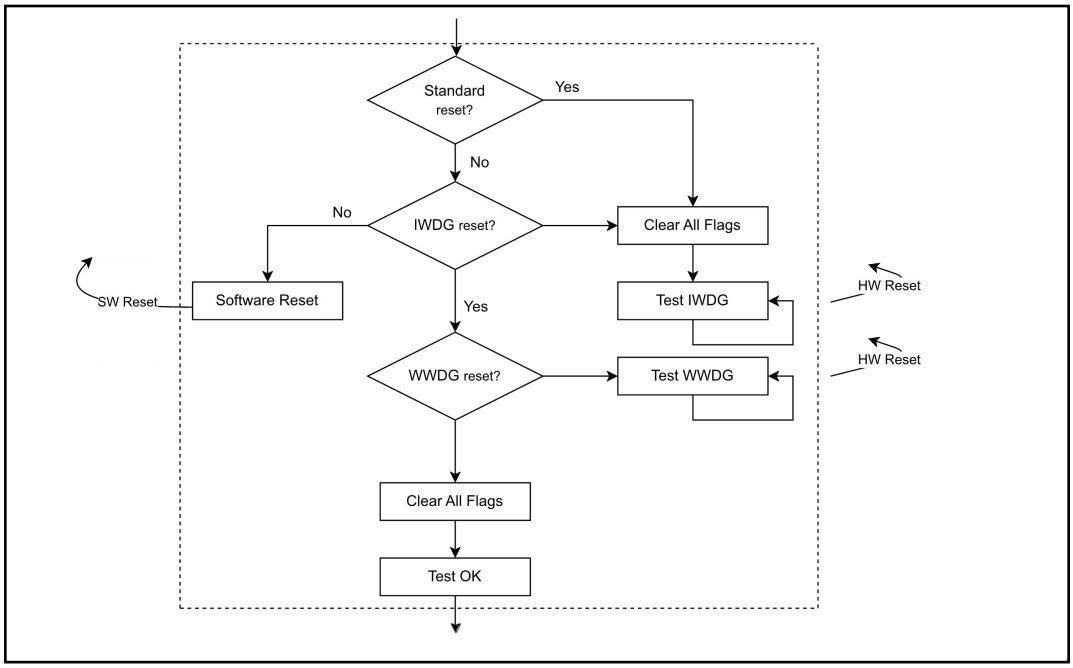


Figure 6. Watchdog startup self-test

4.1.3 RAM start-up self test

The test consists of 6 cycles, with values 0x00 and 0xFF alternately checking and filling the entire RAM word by word. The first 3 cycles are executed as address increment, and the last 3 cycles are executed as address decrement. The entire RAM detection algorithm process is shown in the following *Figure 7*:

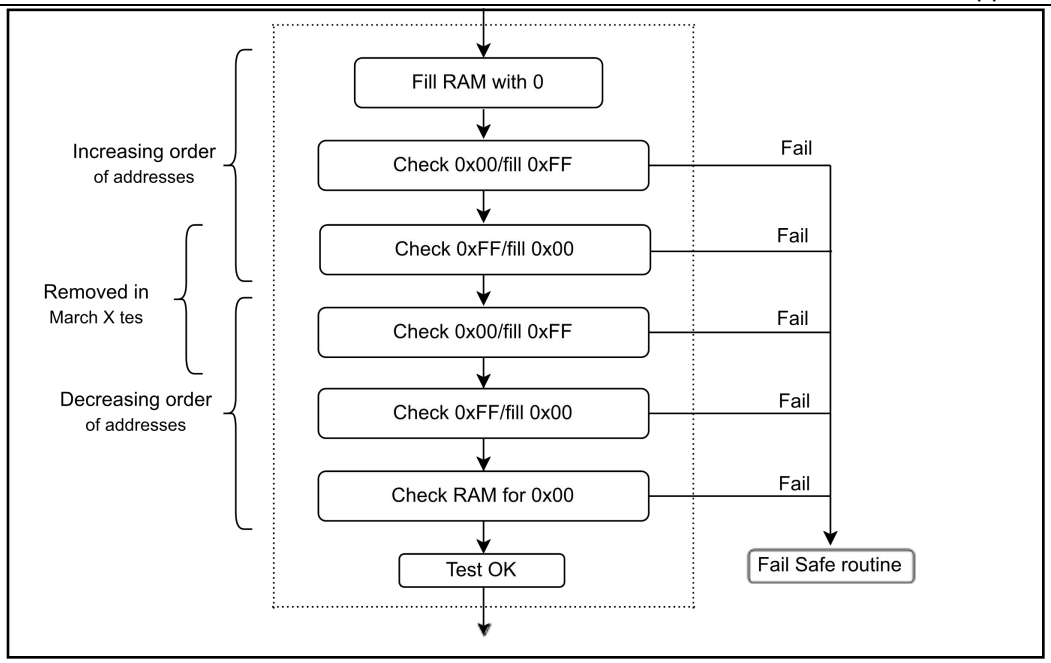


Figure 7. RAM startup self-test structure

4.1.4 Clock start-up self test

The test flow is shown in *Figure 8*.

The user specifies HSE or HSI as the clock source for the system clock, and can choose HSE, HSI, or LSI according to the situation to verify whether the clock source is within the normal range. When setting HSI as the clock source, capture the HSE clock using the HSI clock (-40 ℃~105 ℃,+/-2.5% HSI) as the standard, and compare the captured HSE clock with the maximum and minimum values of the standard HSE to determine whether the current HSI clock is within the normal range; The same applies when HSE is used as a clock source.

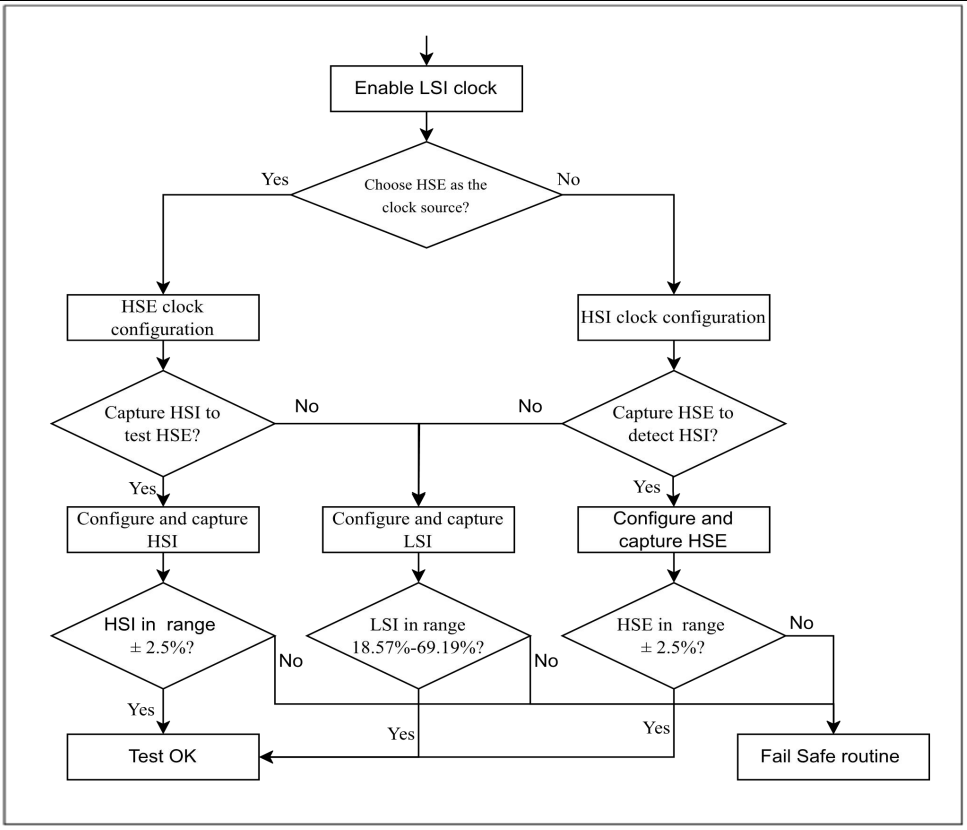


Figure 8. Clock startup self-test subroutine structure

4.1.5 Flash memory complete check sum self test

FLASH self check is the process of using the CRC algorithm to calculate flash data in a program, and comparing the resulting value with the pre calculated CRC value stored in the designated location of FLASH during compilation(see *Figure 9*):

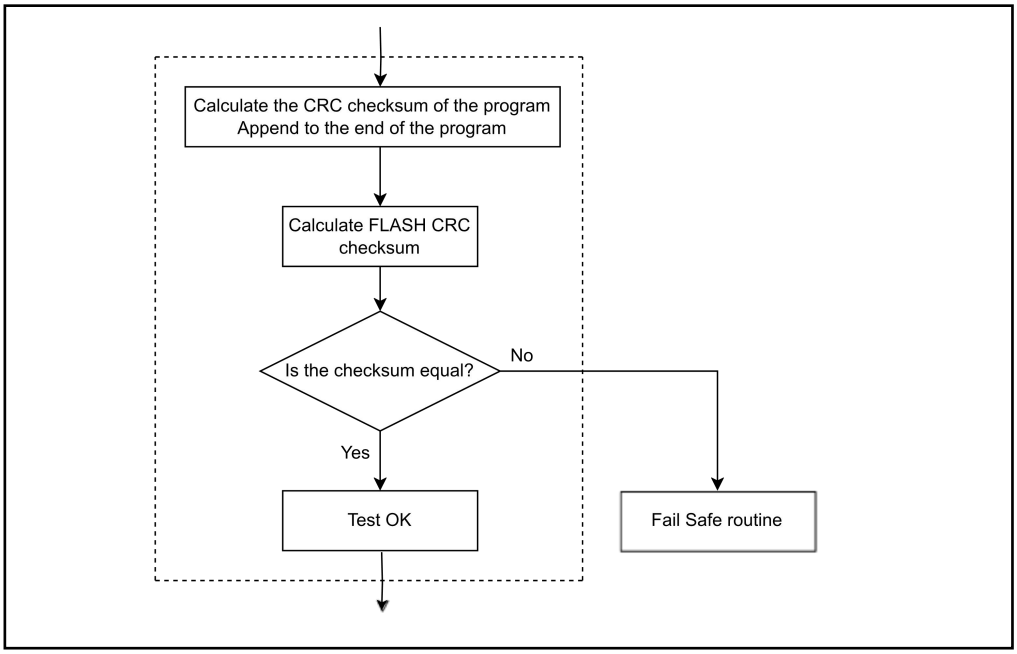


Figure 9. Flash memory startup self-test structure

4.2 Run time self tests

The periodic self check during runtime must be initialized before entering the main loop. At runtime, detection is performed periodically using systick as the time base.

The runtime cycle detection includes:

- Local CPU kernel register detection
- Stack boundary overflow detection
- System clock operation detection
- Flash CRC segmentation detection
- Watchdog detection
- Local RAM self check (conducted in the interrupt service program)

The process is as follows(see *Figure 10*):

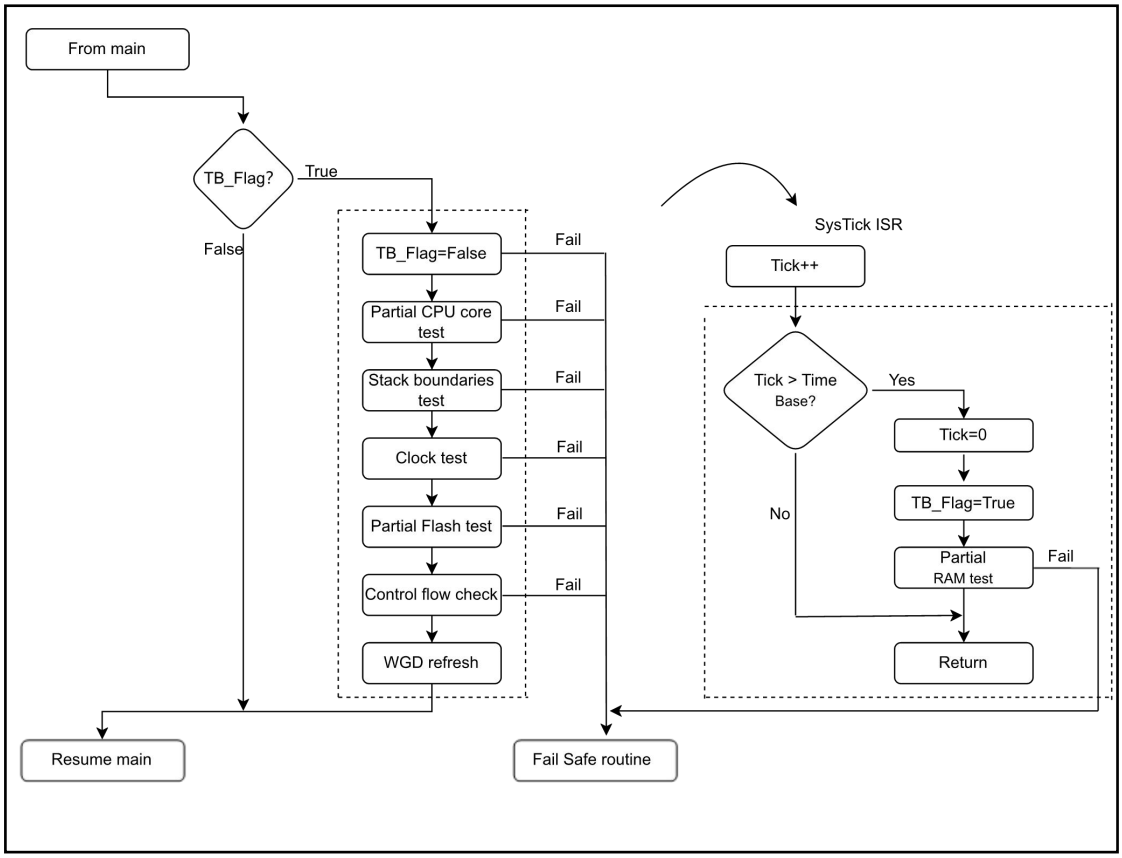


Figure 10. Periodic runtime self-test and timebase interrupt service structure

4.2.1 CPU light run time self test

The CPU runtime cycle self check is similar to the self check at startup, except that it does not detect kernel flags and stack pointers(see *Figure 11*).

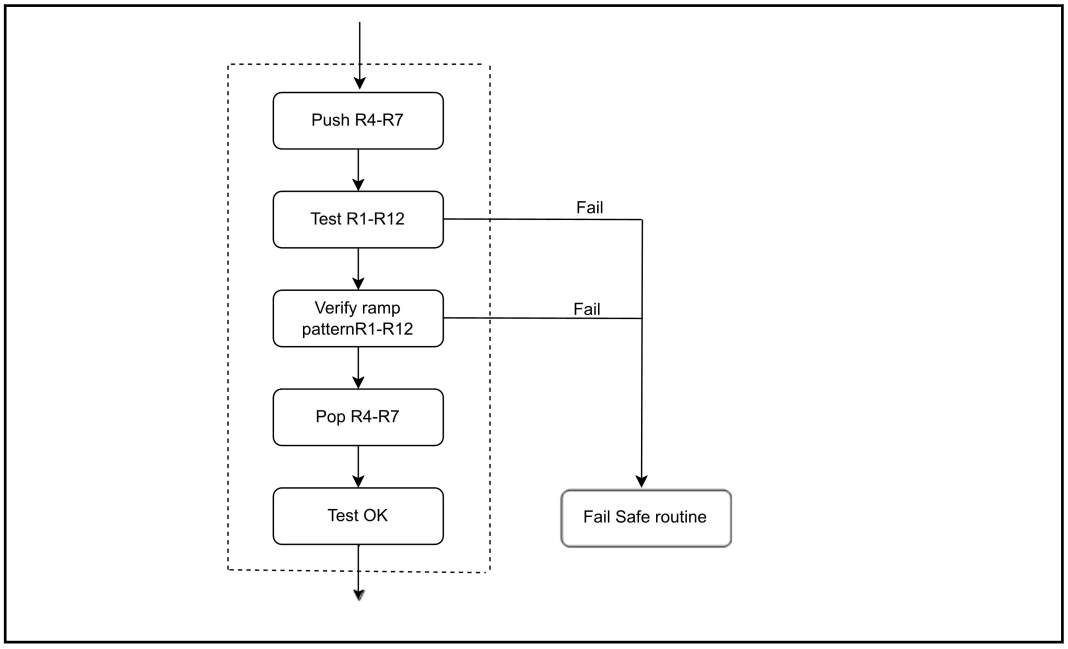


Figure 11. CPU light runtime self-test structure

4.2.2 Stack boundary run time self test

This test detects stack overflow by examining the integrity of the Magic pattern stored at the top of the space reserved for the stack. If the original pattern is destroyed, the test fails and the fail safe program is called.

The Pattern is placed at the lowest address reserved for the stack area. This area can have different configurations depending on the device. Users must define sufficient areas for the stack and ensure that the patterns are placed correctly(see *Figure 12*).

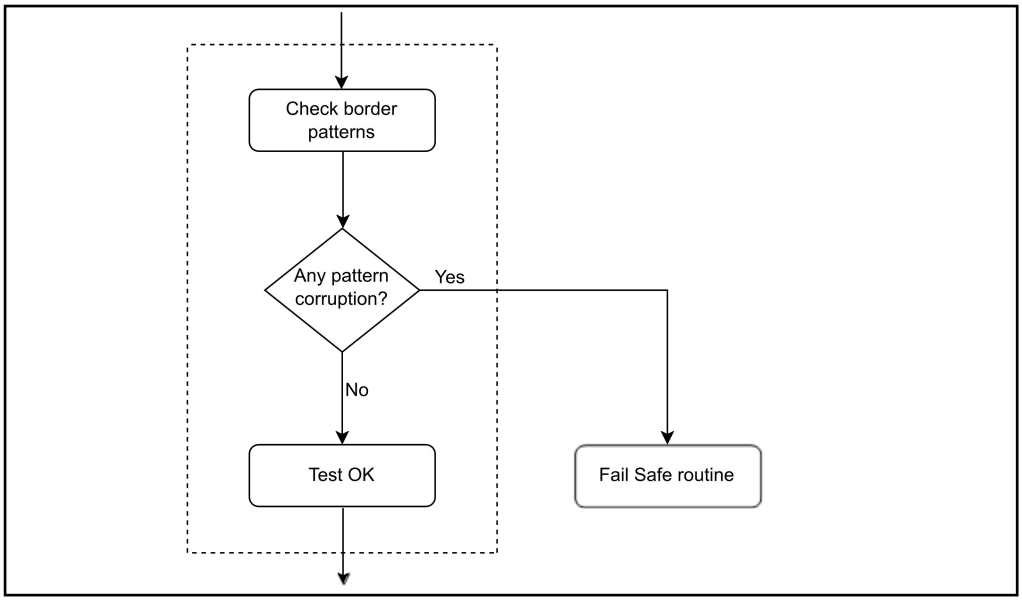


Figure 12. Stack overflow runtime test structure

4.2.3 Clock run time self test

The detection of the system clock during runtime is similar to the detection of the clock during startup, and the process is as follows(see *Figure 13*):

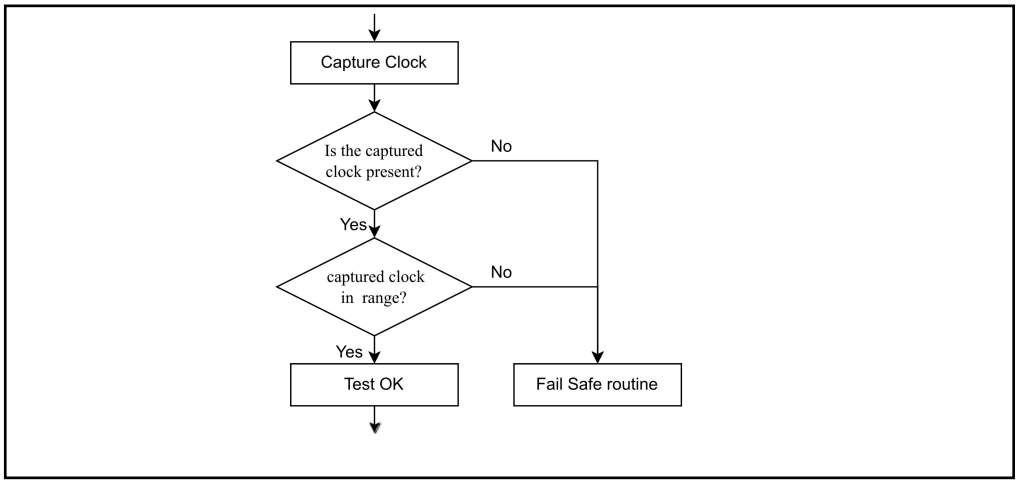


Figure 13. Clock runtime self-test structure

4.2.4 Partial flash run time self test

Perform Flash CRC self check during runtime. Due to different detection ranges and different time consumption, segmented CRC calculation can be configured based on the size of the user application. When the calculation reaches the last range, the CRC value is compared. If it is inconsistent, the test fails(see *Figure 14*).

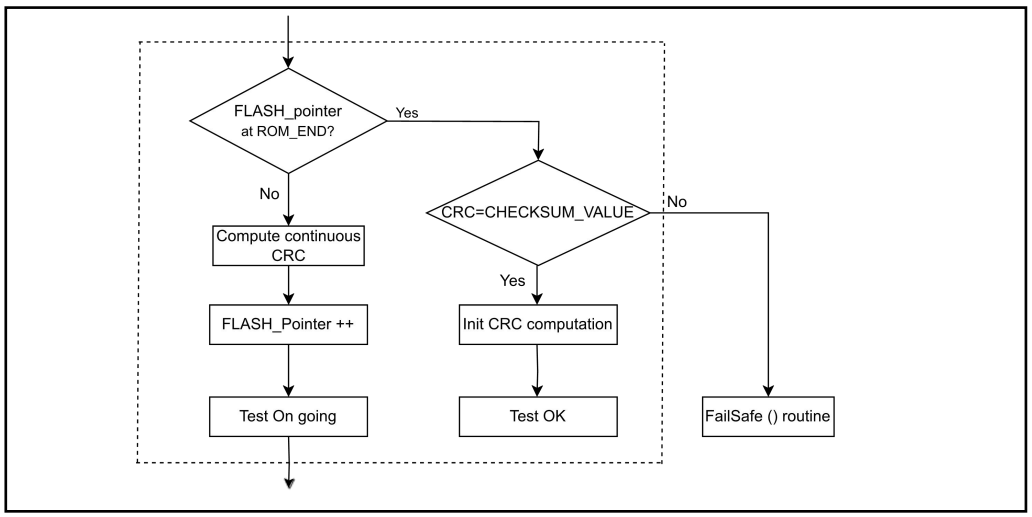


Figure 14. Partial Flash CRC runtime self-test structure

4.2.5 Partial RAM run time self test

The runtime RAM self check is performed in the systick interrupt function. The test only covers the portion of memory allocated to the class B variable(see *Figure 15*).

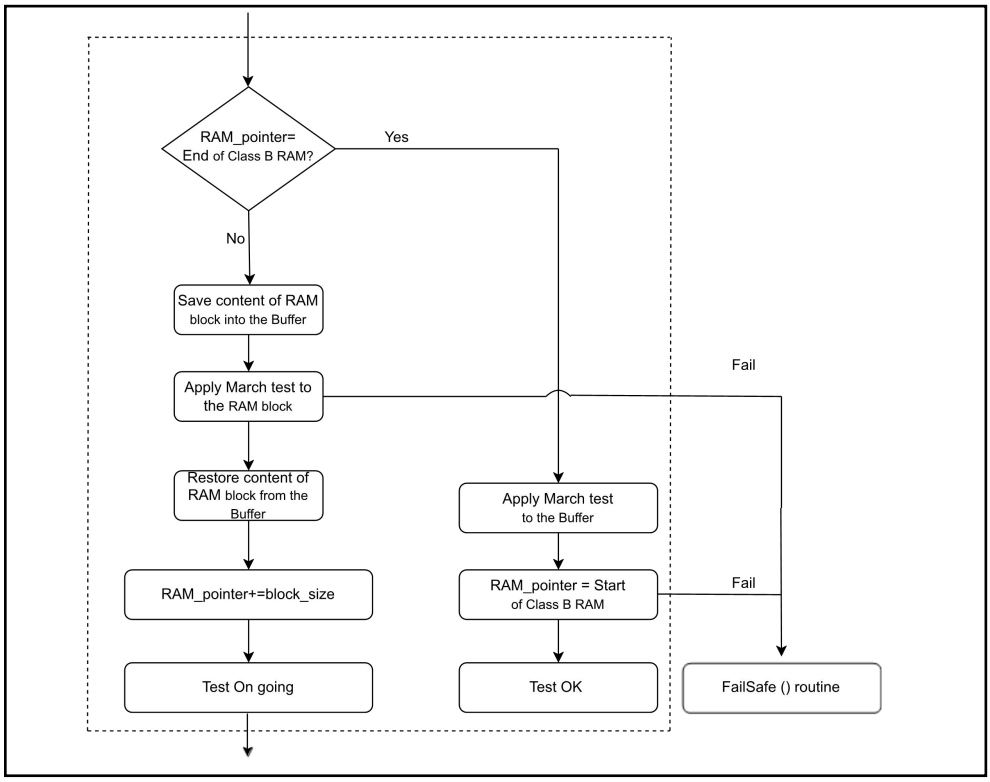


Figure 15. Partial RAM runtime self-test structure

5 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 MindMotion 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.

A software version management system at the module level is needed.

5.1 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.

Software modifications are based on a modification request which details the proposed change & reasons for change.

5.2 Architecture planning

The STL packet structure is the result of a long experience with repeatedly certified FW, where modules were integrated into MindMotion 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 MindMotion 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.

The technical specifications for the Modification need to be confirmed by the development team meeting.

5.3 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.

5.4 Coding

Coding is based on principles defined by internal MindMotion 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.

5.5 Testing modules

Modules have been tested for functionality on different products, with different development tools.

Verification after modifying requirements requires appropriate and determined test cases.

5.6 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.

The assessment of the modification is carried out based on the specified verification and validation activities, which may include:

- a reverification of changed software modules

- a reverification of affected software modules

- a revalidation of the complete system

5.7 Maintenance

For the FW maintenance MindMotion 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.

6 Migration of software library

6.1 Software Library Directory Structure

(see *Figure 16*)

KEIL IAR

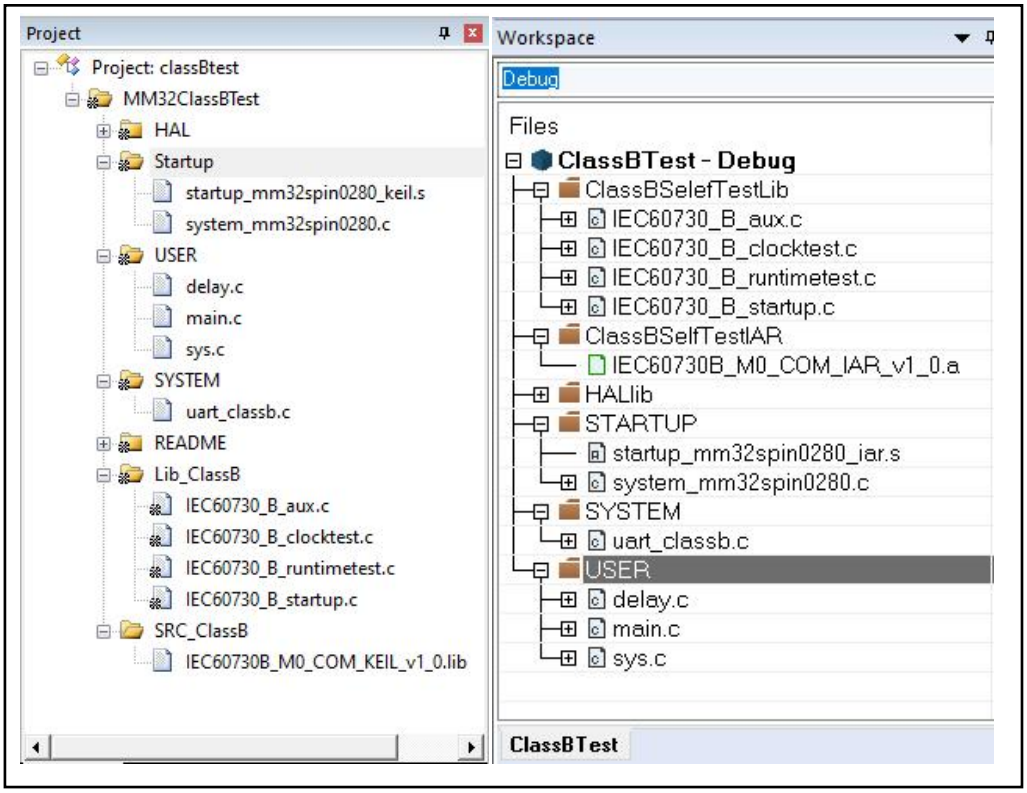


Figure 16. CLASS B Directory Structure

6.2 Migration of Class B functional security software library

Demonstration based on **MM32SPIN0280** platform

1. Copy the ClassBTest folder to the actual application project. The ClassBTest directory contains the Library directory, which includes the library files, C source files, and header files required for migration(see *Figure 17*)；

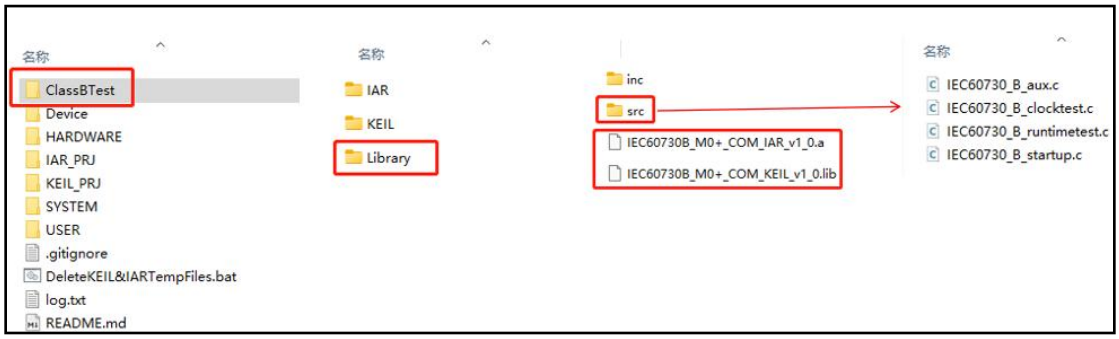


Figure 17. Library Directory

2. Add the library files and C source files required for migration included in the Library directory to the MDK or IAR workgroup, and add the header file path in the actual project(see *Figure 18*)；

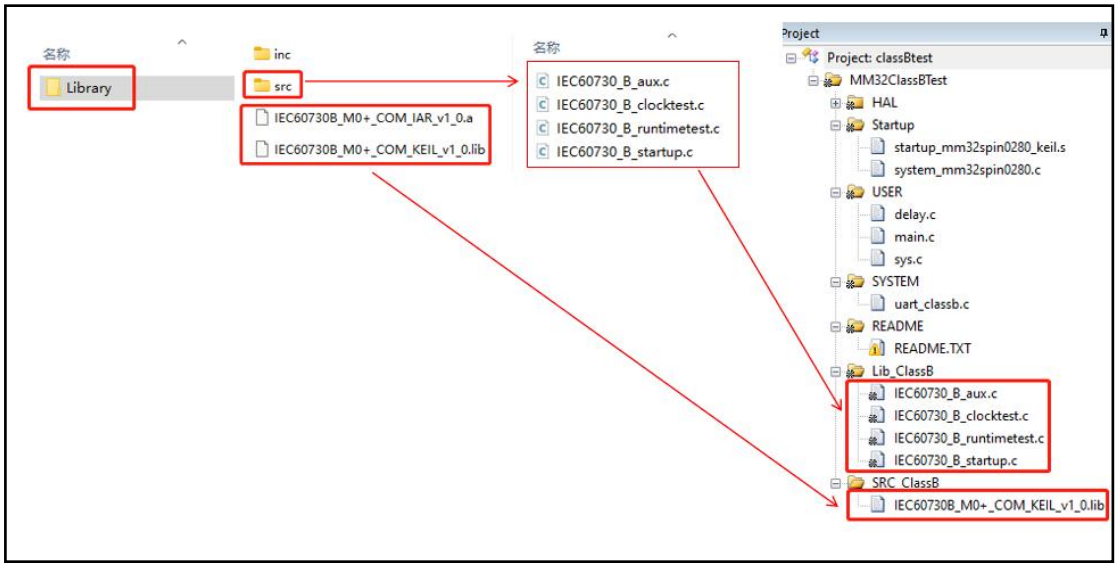


Figure 18. Required Add Files

3. The detection code at startup is located In the $Sub $$main function of the

IEC60730_ B_startup. c file in MDK, modify the project entry address in the IAR file to IEC60730_ B_Startup function(see *Figure 19*)；

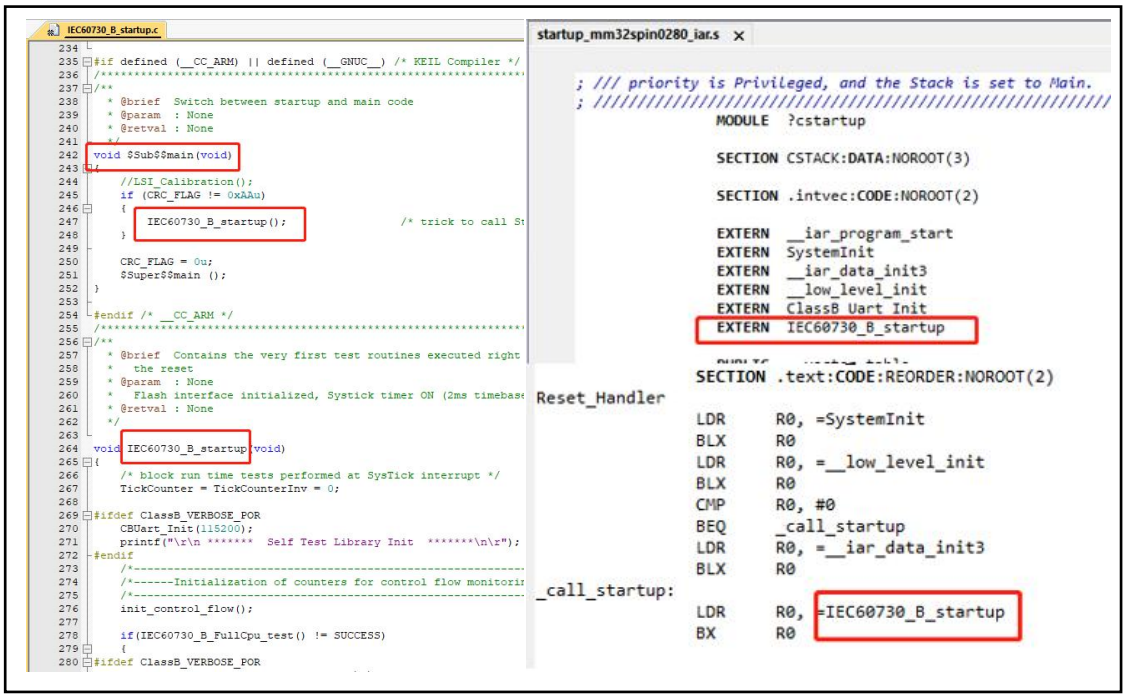


Figure 19. Entry point for initiating detection

4. Runtime detection calls IEC60730_InitRunTimeChecks()function before the while(1) loop in main. c and call IEC60730_DoRunTimeChecks() function during the while(1)loop(see *Figure 20*);

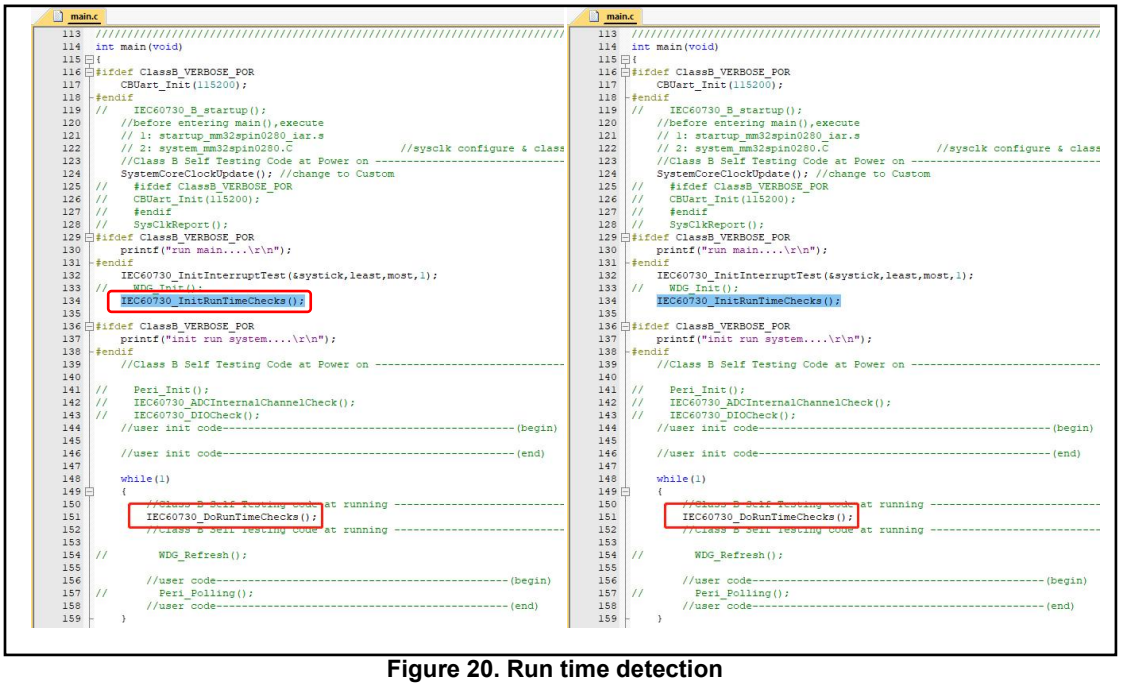


Figure 20. Run time detection

5.Copy the ClassBtest.sct, crc_gen_keil.bat, crc_load.ini, MM32_CRC.exe files in the KEIL_PRJ folder to the directory where the actual application.uvprojx resides; Set the crc_gen_keil.bat running mode in the user configuration(see *Figure 21*)：

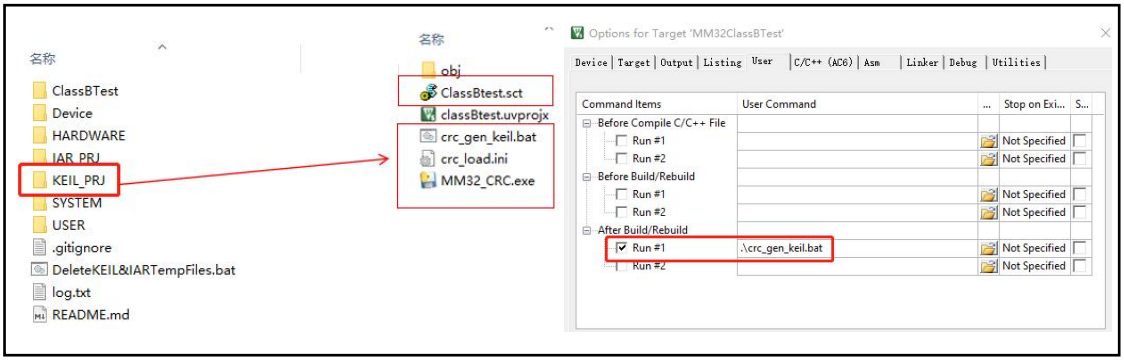


Figure 21. keil platform FLASH detection required files

6.Set the crc_load.ini path in the Utilities option to load the hex file containing CRC during debugging, uncheck the default scattered load file in the linker option, and set the SCT file path in the routine(see *Figure 22* and *Figure 23*)：

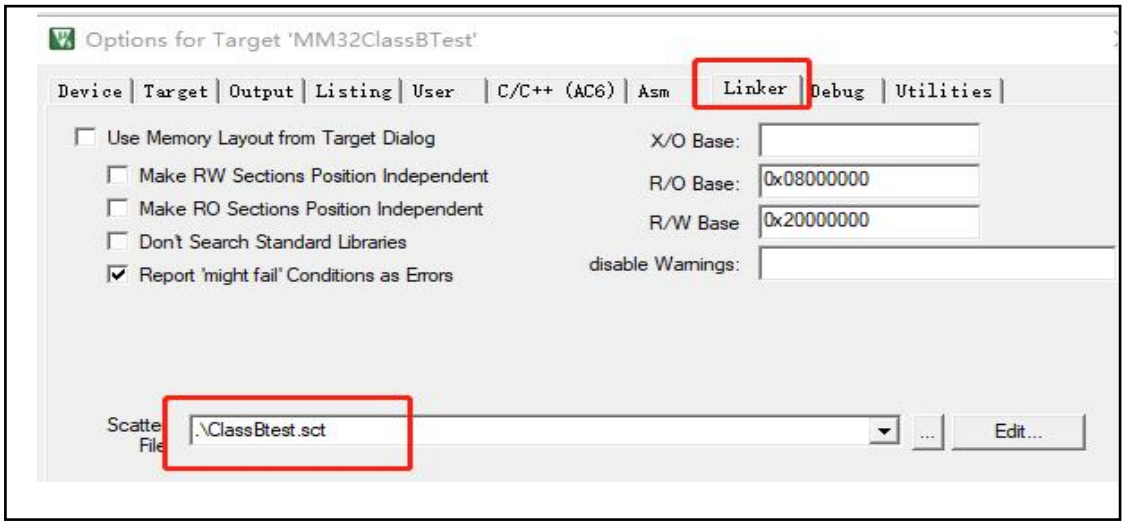


Figure 22. keil SCT file path configuration

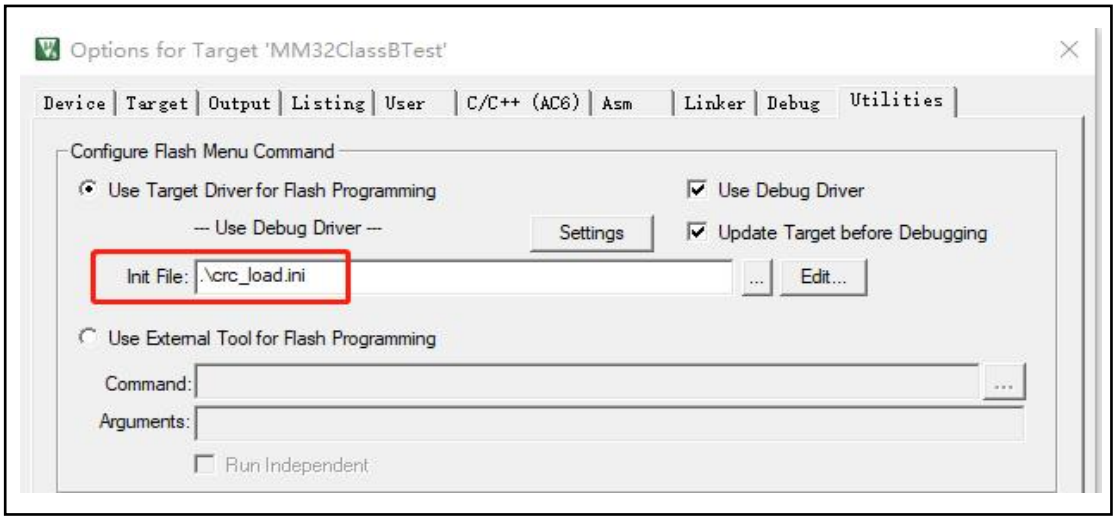


Figure 23. keil platform debugging script configuration

7. In IAR, copy the MM32SPIN0280_CLASSB.icf, crc_gen_keil.bat, and MM32_CRC.exe files in IAR_PRJ folder to the directory where the.eww resides(see *Figure 24*);

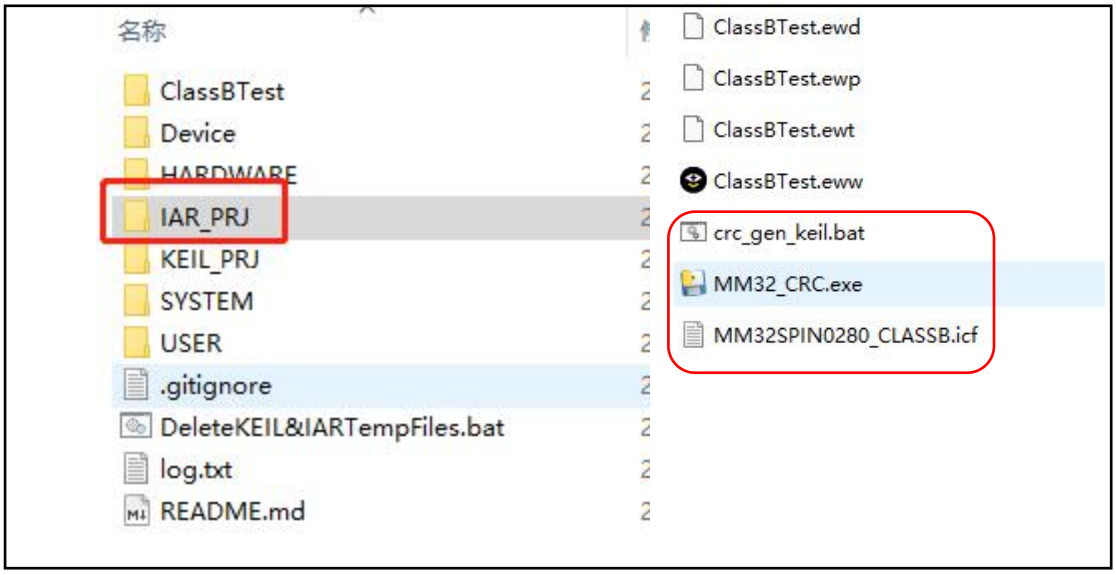


Figure 24. Files required for IAR FLASH detection

8. In Build Action configuration, set the crc_gen_keil.bat running mode, and in linker, set the icf file path in the routine(see *Figure 25* and *Figure 26*);

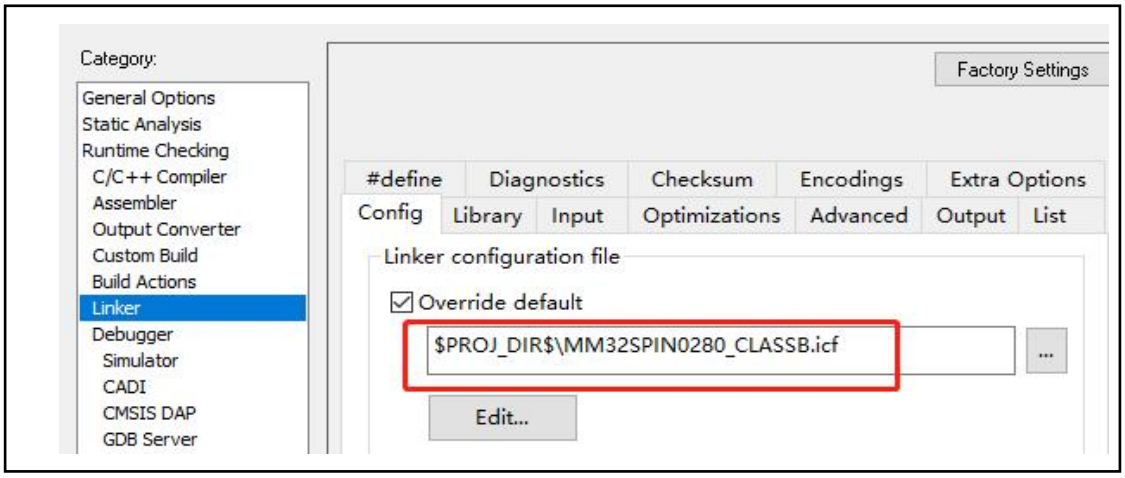


Figure 25. IAR platform ICF file path configuration

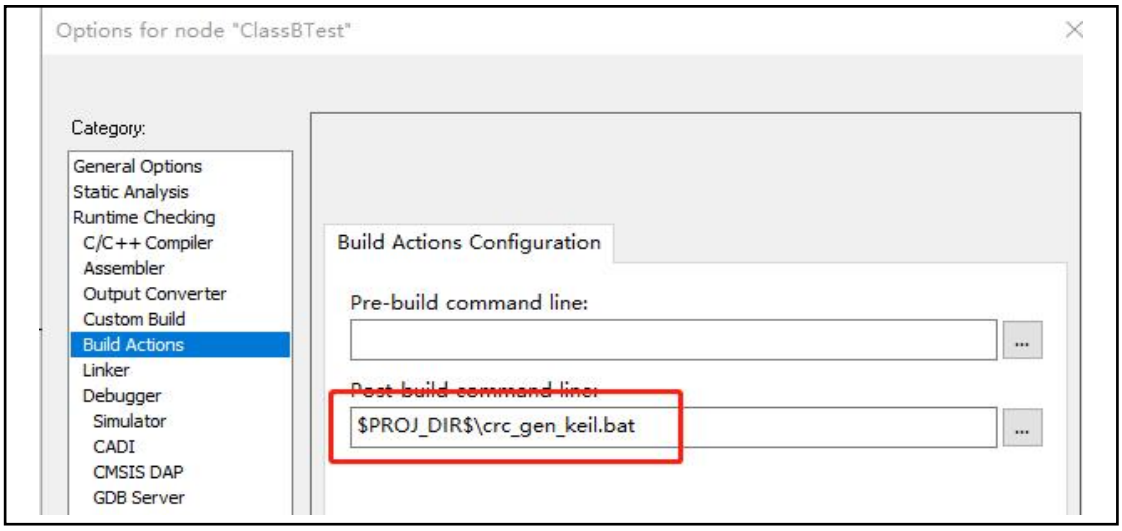


Figure 26. IAR platform CRC script file path configuration

9. In the crc_gen_keil.bat file, you can change the actual project name and the path of the project output file. If the project name is changed, the names of executable files in the crc_load.ini file must also be changed(see *Figure27*);

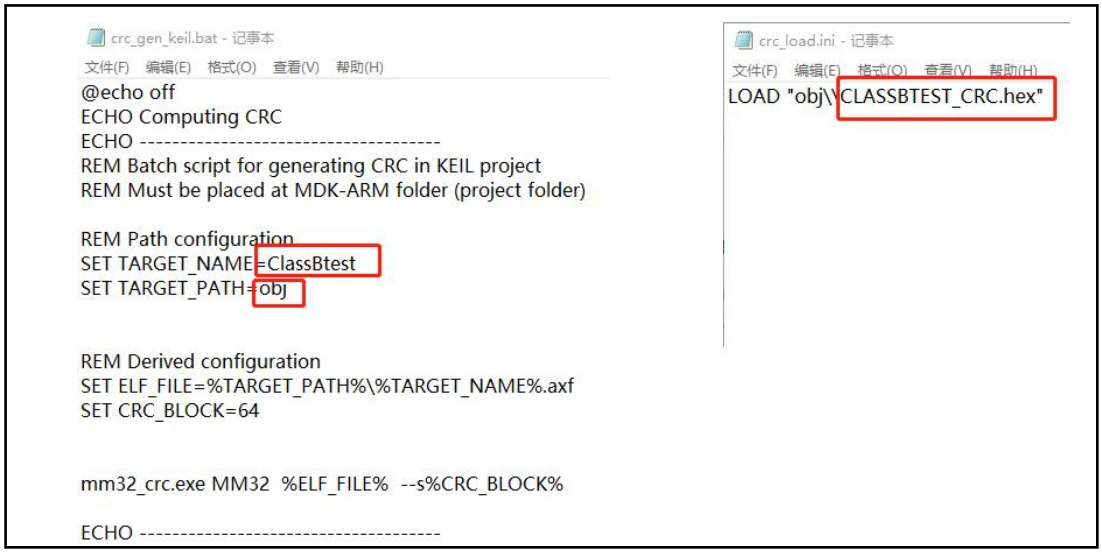


Figure 27. Project name configuration in the keil&IAR platform CRC script file

10. In the IEC60730_B_startup.c file set the on-chip RAM size of the corresponding type of chip, the detection area of RAM during running, the storage address of the class b variable, and make corresponding modifications in the corresponding SCT or icf file(see *Figure 28*,*Figure 29*,*Figure 30*);

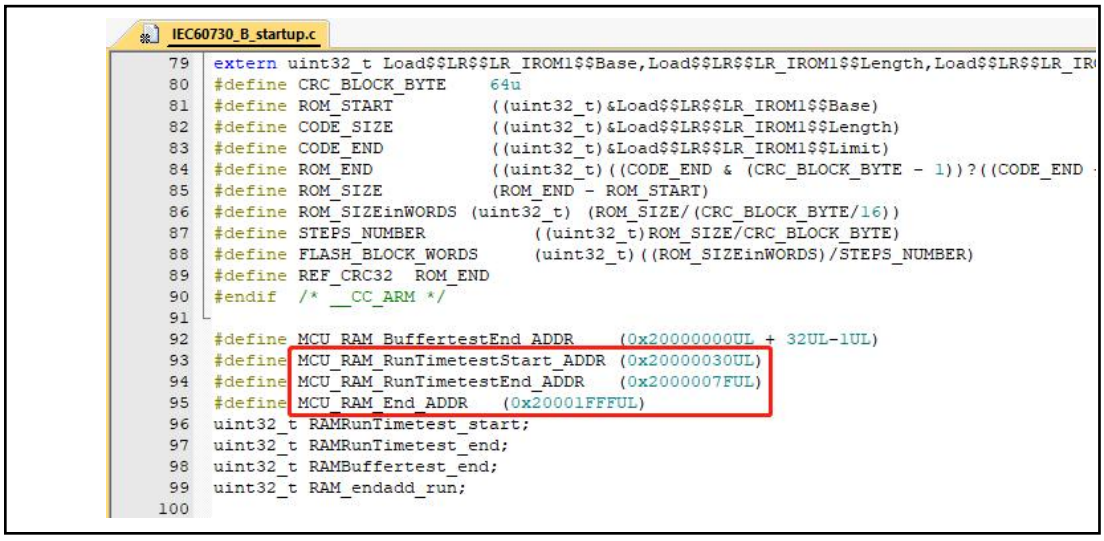


Figure 28. RAM detection area configuration

```
18
19      RAM_BUF 0x20000000 UNINIT 0x20 {; Run-time transparent RAM test
20        IEC60730_B_startup.o (RUN_TIME_RAM_BUF)
21      }
22
23      RAM_PNT 0x20000020 UNINIT 0x10 {; Run-time transparent RAM test
24        IEC60730_B_startup.o (RUN_TIME_RAM_PNT)
25      }
26
27      CLASSB 0x20000030 UNINIT 0x28 {          ; Class B variables
28      IEC60730_B_startup.o (CLASS_B_RAM)
29      }
30
31      CLASSB_INV 0x20000058 UNINIT 0x28 {              ; Class B inve
32      IEC60730_B_startup.o (CLASS_B_RAM_REV)
33      }
```

Figure 29. Modification of RAM detection area in SCT file

```
MM32SPIN0280_CLASSB.icf - 记事本
文件(F)  编辑(E)  格式(O)  查看(V)  帮助(H)
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x0801FFFF;      /* max 0x0801FFFF */
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20001FFF;      /* max 0x20001FFF */

/* leave gaps at begin and and of class B region due to run time test overlap */
define symbol __ICFEDIT_region_CLASSB_start__   = 0x20000030;
define symbol __ICFEDIT_region_CLASSB_midd__    = 0x20000058;
define symbol __ICFEDIT_region_CLASSB_end__     = 0x20000087;
define symbol __ICFEDIT_region_user_RAM_start__ = 0x20000088;
```

Figure 30. Changes to the RAM detection area in the ICF file

11. The clock detection part of startup and running time can modify the hardware timer resources used in the routine according to the needs of actual applications. Timer2 in the routine captures the 128-frequency division input of internal LSI or external HSE; Timer3 captures the MCO output of the A8 pin, which is configured as an internal HSI output(see *Figure 31*,*Figure 32*,*Figure 33*,*Figure 34*);

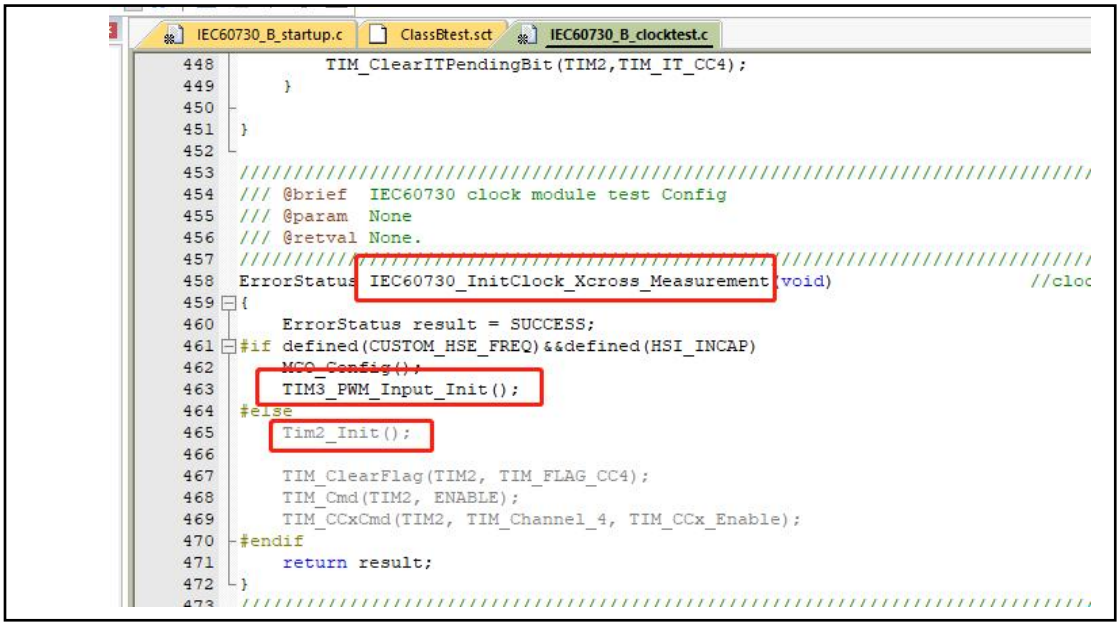


Figure 31. Clock detection part of the code

16.5.19 TIM2_OR Input Option Register

Address offset:0x50

Reset value:0x0000

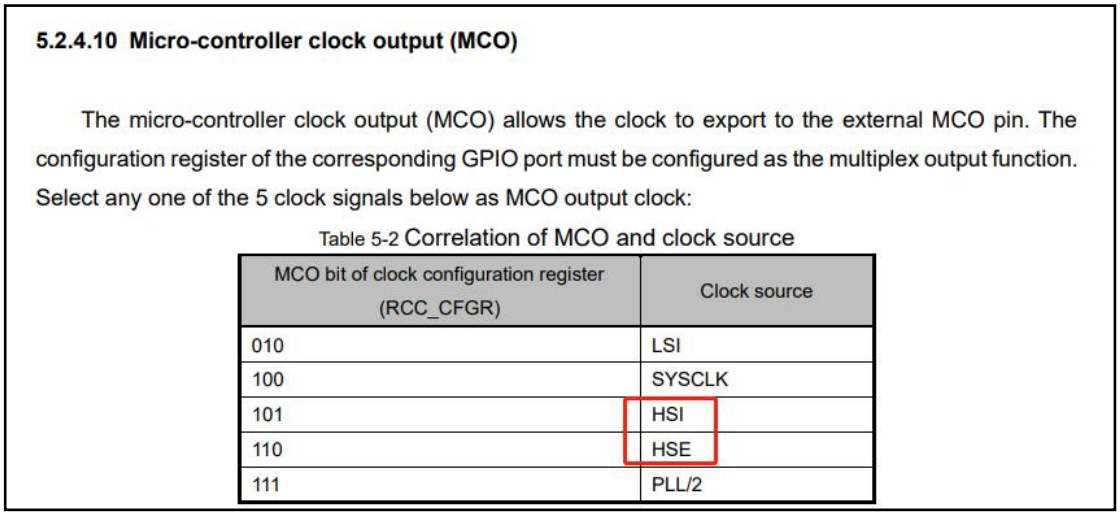
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								TI4_RMP		Reserved				ETR_RMP	
								rw	rw					rw	rw

Bit	Field	Description
15:8	Reserved	Reserved, must be kept at reset value.

Figure 32. Internal Timer input options register

Bit	Field	Description
15:8	Reserved	Reserved, must be kept at reset value.
7:6	TI4_RMP	TI4 multiplex 00：CH4 GPIO or CPT input 01：LSI clock input 10：Reserved 11：HSE_CLK_DIV_128 clock input

Figure 33. Internal Timer input option selected



5.2.4.10 Micro-controller clock output (MCO)

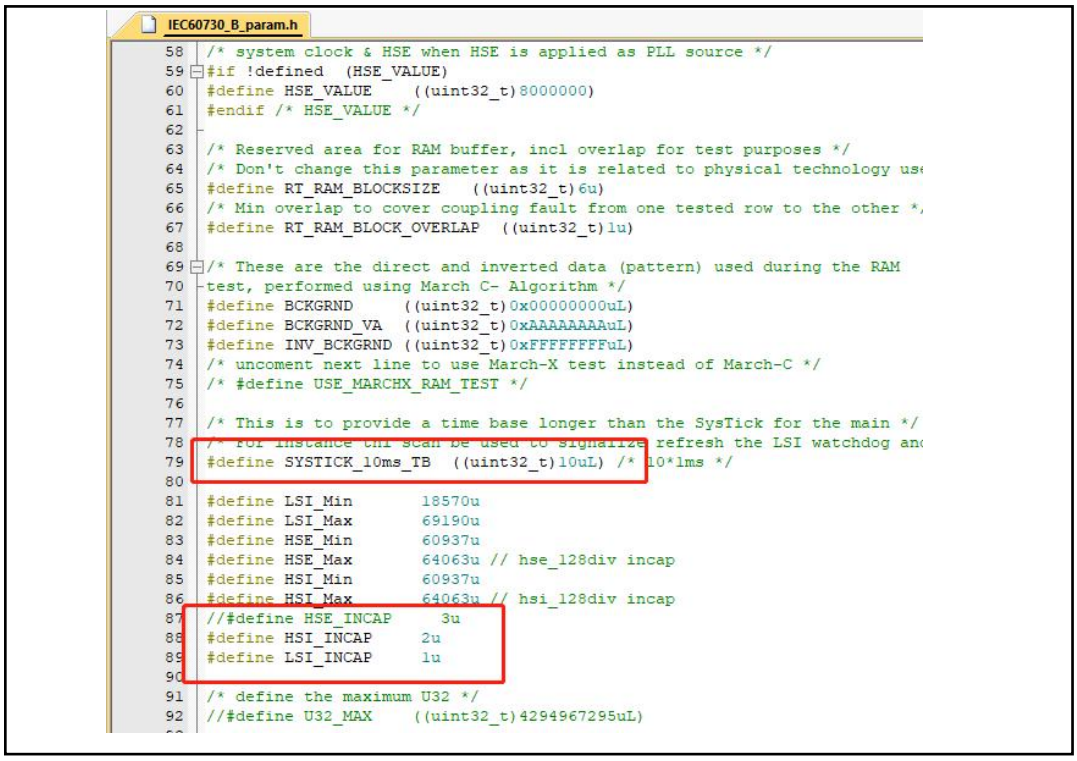
The micro-controller clock output (MCO) allows the clock to export to the external MCO pin. The configuration register of the corresponding GPIO port must be configured as the multiplex output function. Select any one of the 5 clock signals below as MCO output clock:

Table 5-2 Correlation of MCO and clock source

MCO bit of clock configuration register (RCC_CFGR)	Clock source
010	LSI
100	SYSCLK
101	HSI
110	HSE
111	PLL/2

Figure 34. Internal MCO output options are selected

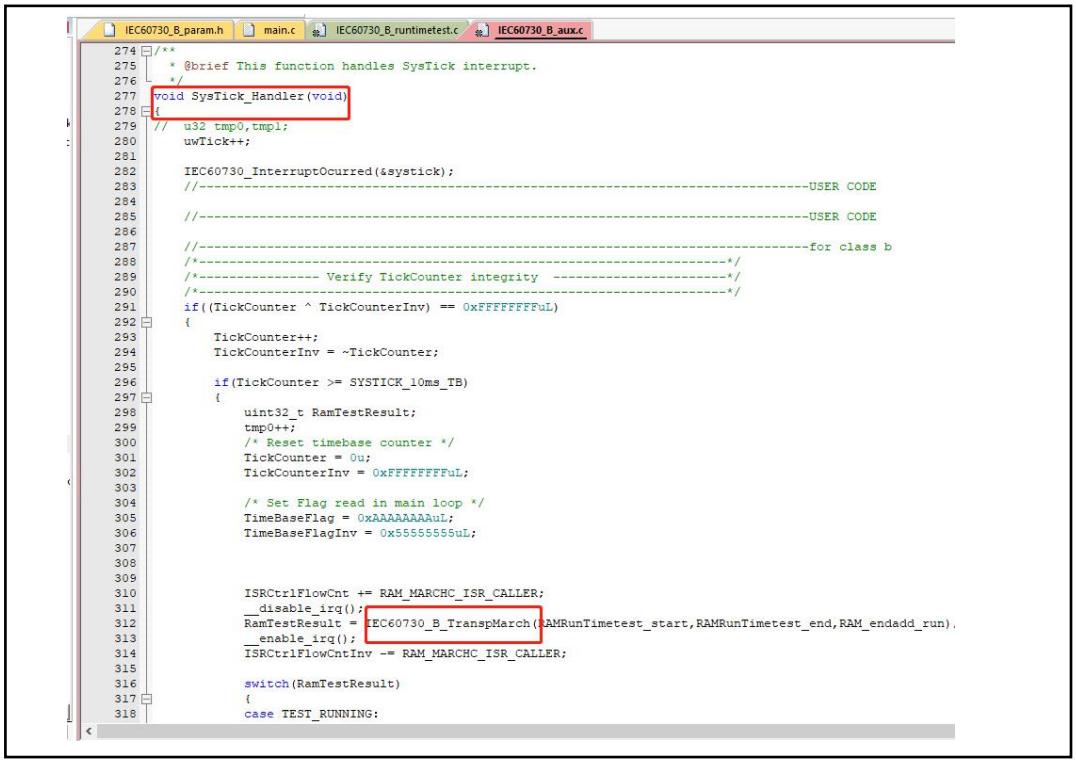
12. The runtime CPU, FLASH, clock, stack detection cycle can be modified in the IEC60730_B_param.h file, the clock capture selection can also be set in IEC60730_B_param.h; Run-time RAM detection is handled in the 1ms timebase interrupt in the IEC60730_B_aux.c file. Since the interrupt is turned off by the RAM run-time detection, the RAM run-time detection function can be placed in the interrupt of higher priority if necessary(see *Figure 35*,*Figure 36*);



IEC60730_B_param.h

```
58  /* system clock & HSE when HSE is applied as PLL source */
59  #if !defined  (HSE_VALUE)
60  #define HSE_VALUE    ((uint32_t)8000000)
61  #endif /* HSE_VALUE */
62
63  /* Reserved area for RAM buffer, incl overlap for test purposes */
64  /* Don't change this parameter as it is related to physical technology use
65  #define RT_RAM_BLOCKSIZE    ((uint32_t)6u)
66  /* Min overlap to cover coupling fault from one tested row to the other *,
67  #define RT_RAM_BLOCK_OVERLAP  ((uint32_t)1u)
68
69  /* These are the direct and inverted data (pattern) used during the RAM
70  test, performed using March C- Algorithm */
71  #define BCKGRND     ((uint32_t)0x00000000uL)
72  #define BCKGRND_VA  ((uint32_t)0xAAAAAAAAuL)
73  #define INV_BCKGRND ((uint32_t)0xFFFFFFFFuL)
74  /* uncoment next line to use March-X test instead of March-C */
75  /* #define USE_MARCHX_RAM_TEST */
76
77  /* This is to provide a time base longer than the SysTick for the main */
78  /* for instance this can be used to signalize refresh the LSI watchdog and
79  #define SYSTICK_10ms_TB  ((uint32_t)10uL) /* 10*1ms */
80
81  #define LSI_Min       18570u
82  #define LSI_Max       69190u
83  #define HSE_Min       60937u
84  #define HSE_Max       64063u // hse_128div incap
85  #define HSI_Min       60937u
86  #define HSI_Max       64063u // hsi_128div incap
87  //#define HSE_INCAP     3u
88  #define HSI_INCAP     2u
89  #define LSI_INCAP     1u
90
91  /* define the maximum U32 */
92  //#define U32_MAX    ((uint32_t)4294967295uL)
```

Figure 35. class B Parameter Settings



IEC60730_B_param.h | main.c | IEC60730_B_runtimetest.c | IEC60730_B_aux.c

```
274 /**
275   * @brief This function handles SysTick interrupt.
276   */
277 void SysTick_Handler(void)
278 {
279 //  u32 tmp0,tmp1;
280     uwTick++;
281
282     IEC60730_InterruptOcurred(&systick);
283     //---------------------------------------------------------------------------USER CODE
284
285     //---------------------------------------------------------------------------USER CODE
286
287     //---------------------------------------------------------------------------for class b
288     /*-------------------------------------------------------------------------*/
289     /*---------------- Verify TickCounter integrity  ----------------------*/
290     /*-------------------------------------------------------------------------*/
291     if((TickCounter ^ TickCounterInv) == 0xFFFFFFFFuL)
292     {
293         TickCounter++;
294         TickCounterInv = ~TickCounter;
295
296         if(TickCounter >= SYSTICK_10ms_TB)
297         {
298             uint32_t RamTestResult;
299             tmp0++;
300             /* Reset timebase counter */
301             TickCounter = 0u;
302             TickCounterInv = 0xFFFFFFFFuL;
303
304             /* Set Flag read in main loop */
305             TimeBaseFlag = 0xAAAAAAAAuL;
306             TimeBaseFlagInv = 0x55555555uL;
307
308
309
310             ISRCtrlFlowCnt += RAM_MARCHC_ISR_CALLER;
311             __disable_irq();
312             RamTestResult = IEC60730_B_TranspMarch(RAMRunTimetest_start,RAMRunTimetest_end,RAM_endadd_run),
313             __enable_irq();
314             ISRCtrlFlowCntInv -= RAM_MARCHC_ISR_CALLER;
315
316             switch(RamTestResult)
317             {
318             case TEST_RUNNING:
```

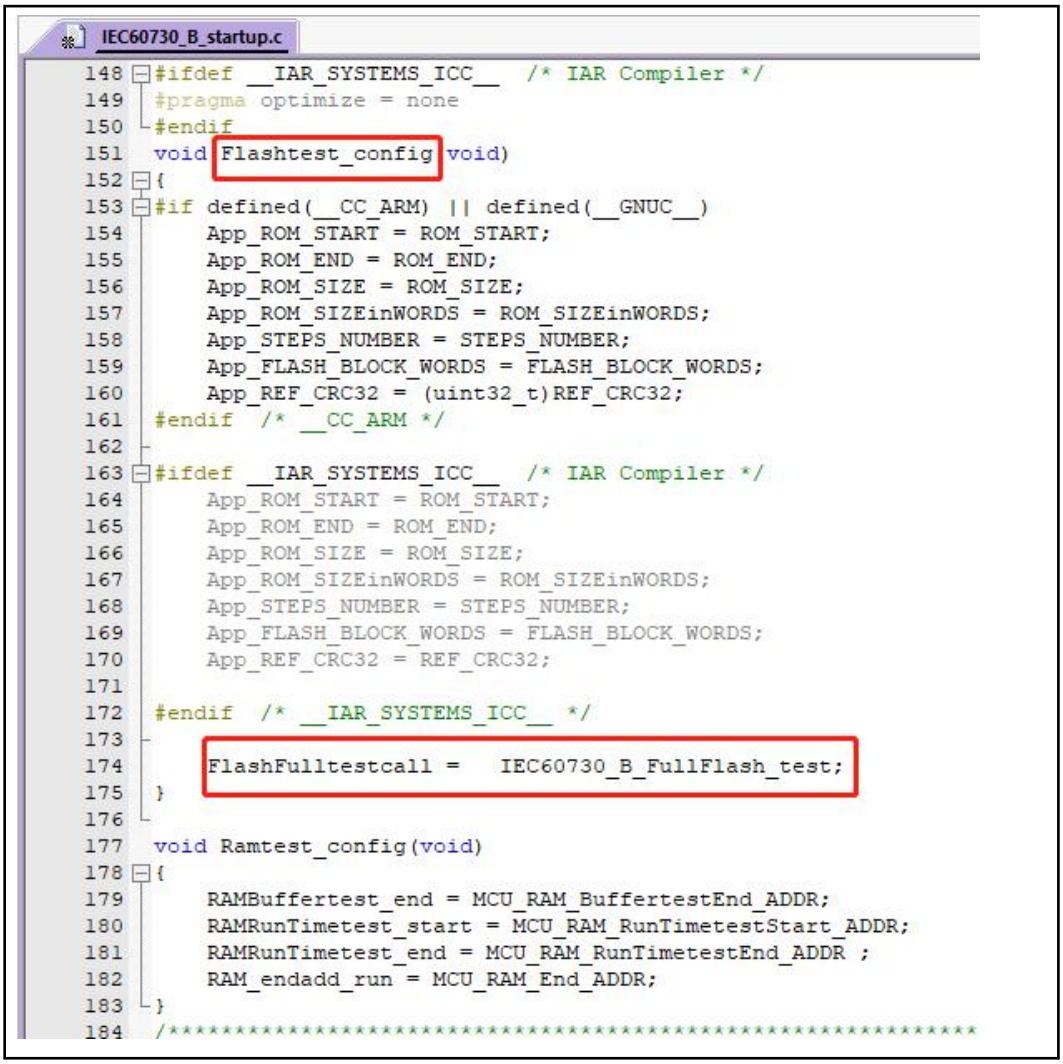
Figure 36. Runtime RAM detection

13. The FLASH detection of the routine adopts CRC verification, and CRC peripherals are used by default. For chips without CRC peripherals, you can change the FLASH detection function interface in the IEC60730_B_startup.c and IEC60730_B_runtimetest.c to CRC verification in software mode(see *Figure 37*,*Figure 38*);

IEC60730_B_startup.c | IEC60730_B_runtimetest.c

```
84      /* Initialize SysTick to generate 1ms time base */
85      if (SysTick_Config(SystemCoreClock/1000uL) != 0x00)
86      {
87  #ifdef ClassB_VERBOSE_POR
88          printf("Run time base init failure\n\r");
89  #endif /* ClassB_VERBOSE_POR */
90          FailSafePOR(SYSTICK_INIT_CALLEE);
91      }
92
93      CRC_ResetDR();
94      RefCrc32 = CRC_CalcBlockCRC(0u,0u);
95      RefCrc32Inv = ~RefCrc32;
96      /* Initialize variables for invariable memory check */
97      Flashtest_config();
98      IEC60730_FlashCrc32Init();
99      printf("&pRunCrc32Chk:%x\n\r",(uint32_t)pRunCrc32Chk);
100     FULL_FLASH_CHECKED = ((uint32_t)DELTA_MAIN * App_STEPS_NUMBER + LAST_DELTA_MAIN);
101     FlashRuntestcall = IEC60730_crc32Run;
102     /* wait for  incapture value is valid */
103     LSIPeriodFlag = 0u;
104     while (LSIPeriodFlag == 0u)
105     { }
106
107     /* Initialize variables for main routine control flow monitoring */
108     CtrlFlowCnt = 0u;
109     CtrlFlowCntInv = 0xFFFFFFFFuL;
110
111 }
```

Figure 37. Function interface for FLASH detection at startup



IEC60730_B_startup.c

```
148 #ifdef __IAR_SYSTEMS_ICC__  /* IAR Compiler */
149 #pragma optimize = none
150 #endif
151 void Flashtest_config(void)
152 {
153 #if defined(__CC_ARM) || defined(__GNUC__)
154     App_ROM_START = ROM_START;
155     App_ROM_END = ROM_END;
156     App_ROM_SIZE = ROM_SIZE;
157     App_ROM_SIZEinWORDS = ROM_SIZEinWORDS;
158     App_STEPS_NUMBER = STEPS_NUMBER;
159     App_FLASH_BLOCK_WORDS = FLASH_BLOCK_WORDS;
160     App_REF_CRC32 = (uint32_t)REF_CRC32;
161 #endif  /* __CC_ARM */
162
163 #ifdef __IAR_SYSTEMS_ICC__  /* IAR Compiler */
164     App_ROM_START = ROM_START;
165     App_ROM_END = ROM_END;
166     App_ROM_SIZE = ROM_SIZE;
167     App_ROM_SIZEinWORDS = ROM_SIZEinWORDS;
168     App_STEPS_NUMBER = STEPS_NUMBER;
169     App_FLASH_BLOCK_WORDS = FLASH_BLOCK_WORDS;
170     App_REF_CRC32 = REF_CRC32;
171
172 #endif  /* __IAR_SYSTEMS_ICC__ */
173
174     FlashFulltestcall =   IEC60730_B_FullFlash_test;
175 }
176
177 void Ramtest_config(void)
178 {
179     RAMBuffertest_end = MCU_RAM_BuffertestEnd_ADDR;
180     RAMRunTimetest_start = MCU_RAM_RunTimetestStart_ADDR;
181     RAMRunTimetest_end = MCU_RAM_RunTimetestEnd_ADDR ;
182     RAM_endadd_run = MCU_RAM_End_ADDR;
183 }
184 /*****************************************************************
```

Figure 38. Function interface for run time FLASH detection

14. The following information is printed when the routine runs normally(see *Figure 39*,*Figure 40*);

```
 *******  Self Test Library Init  *******
 Start-up CPU Test OK
Pin reset
SW reset
... Power-on or software reset, testing IWDG ...

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

 *******  Self Test Library Init  *******
 Start-up CPU Test OK
Pin reset
IWDG reset
WWDG reset
... WWDG reset, WDG test completed ...
ROM size = 10a0    0x08000000   0x08004280   0x00004280
FLASH 32-bit CRC Address and Value  :8004280   bcc85794
 Start-up FLASH 32-bit CRC OK
 Control Flow Checkpoint 1 OK
 Full RAM Test OK
STL_ClockStartUpTest

Read Incapture Frequency OK freq = 40851Hz !
 Clock frequency OK
 Control Flow Checkpoint 2 OK
run main....
init run system....
```

Figure 39. Debug information about the debugging version

```
[18:01:17.721]收←◆run main....
&pRunCrc32Chk:8000000
init run system....
```

Figure 40. Debug information for the release version

7 Revision history

Date	Revision	Changes
2023.10.31	V0.1	Initial private beta