目录

[一、IEC-60730 CLASS-B 2](#_Toc153182900)

[二、参考文档及例程 2](#_Toc153182901)

[三、代码移植 2](#_Toc153182902)

[四、测试过程 2](#_Toc153182903)

[1、测试前准备工作: 2](#_Toc153182904)

[2、CPU ARM内核测试 3](#_Toc153182905)

[1、CPU的初次自检过程 3](#_Toc153182906)

[2、CPU运行时刻的自检过程 5](#_Toc153182907)

[3、STL FLASH存储器测试 5](#_Toc153182908)

[1、预期CRC的的预计算 5](#_Toc153182909)

[2、STL FLASH自检及运行时刻检查: 5](#_Toc153182910)

[4、STL RAM存储器的测试 7](#_Toc153182911)

[1、RAM存储器的自检及运行时刻检查 7](#_Toc153182912)

[5、系统时钟频率的测试 9](#_Toc153182913)

[1、系统时钟频率运行时的自检代码段 9](#_Toc153182914)

[6、整体测量流程如下: 10](#_Toc153182915)

[7、测试效果： 11](#_Toc153182916)

[8、人为触发错误 11](#_Toc153182917)

[五、编译与链接 12](#_Toc153182918)

[1、变量的存储 12](#_Toc153182919)

[1、编译器默认的存储位置 12](#_Toc153182920)

STM32基于IEC-60730CLASSB标准的实现

# 一、IEC-60730 CLASS-B

IEC-60730安全标准法规由国际电工委员会 ( IEC ) 制定，该安全标准定义了家用电器嵌入式控制软件与硬件安全操作的测试与检测方法，以确保家电中受控设备的安全操作。IEC-60730将家用电器分为3类：

A类 – 不用于确保设备的安全性。 例如：照明控制、湿度控制等。

B类 – 用于防止受控设备的不安全运行。 例如：洗衣机、电冰箱等。

C类 – 用于防止特殊危险，如受控设备爆炸等用途。 例如：自动燃烧器、燃气型烘干机等。

参考:[IEC-60730简介](https://www.wpgdadatong.com/tw/blog/detail/47317)

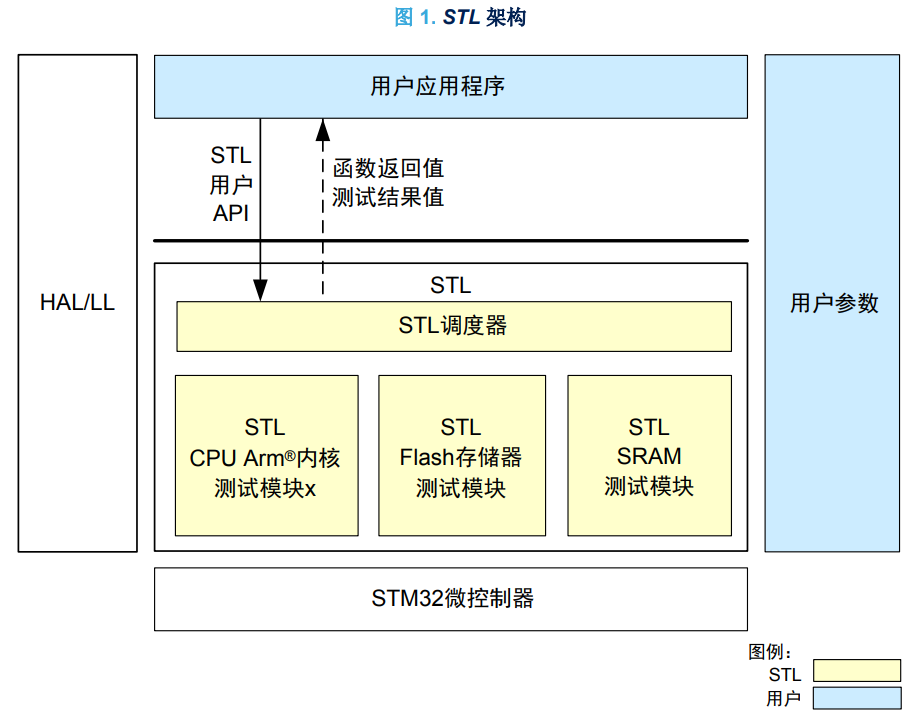
# 二、参考文档及例程

# 三、代码移植

# 四、测试过程

根据ST官方的X-CUBE-STL(版本V4.0.0)软件库，进行如下测试:

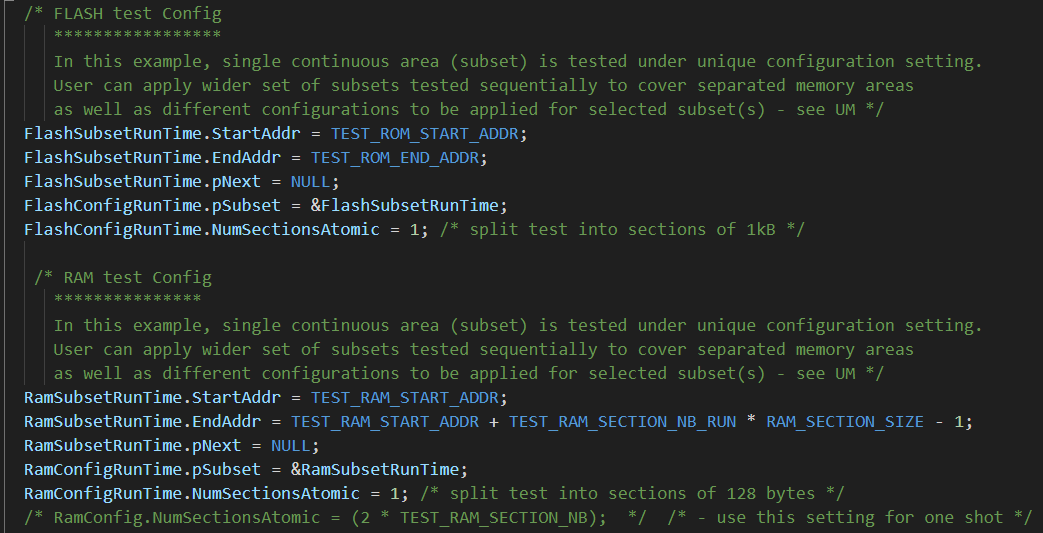
* STL CPU ARM内核测试模块
* STL FLASH存储器测试模块
* STL SRAM测试模块



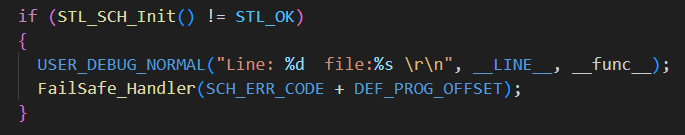
## 1、测试前准备工作:

1. 系统复位之后，执行STL\_Init()函数

* 完成FLASH、RAM相关配置(起始地址/结束地址，section大小等)



* 调用STL软件库函数: STL\_SCH\_Init()



* 执行用户程序STL\_InitClock\_Xcross\_Measurement(),该函数用来初始化TIM16，用于后续的系统时钟频率测试。

## 2、CPU ARM内核测试

STL 包含下列 CPU 测试模块，以及测试功能的概述

TM1L：实现通用寄存器的轻量测试模式

TM7：实现两个栈指针（MSP 和 PSP）的模式和功能测试 PSP

TMCB：实现 APSR 状态寄存器的测试。

### 1、CPU的初次自检过程

调用到的软件库函数如下:

STL\_Status\_t STL\_SCH\_RunCpuTM1L(STL\_TmStatus\_t \*pSingleTmStatus);

STL\_Status\_t STL\_SCH\_RunCpuTM7(STL\_TmStatus\_t \*pSingleTmStatus);

STL\_Status\_t STL\_SCH\_RunCpuTMCB(STL\_TmStatus\_t \*pSingleTmStatus);

代码片段如下:

1. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
2. */\* CPU Test modules \*/*
3. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
4. #ifdef ARTI\_FAILING\_CPU\_TM
5. */\* Artificial failing feature -*
6. when activated, it forces the STL outputs to predefined values \*/
7. ArtifFailing.aCpuTmStatus[0] = STL\_PASSED;
8. ArtifFailing.aCpuTmStatus[1] = STL\_PASSED;
9. ArtifFailing.aCpuTmStatus[2] = STL\_FAILED;
10. STL\_SCH\_StartArtifFailing(&ArtifFailing);
11. #endif */\* ARTI\_FAILING\_CPU\_TM \*/*
12. */\* CPU TM1L \*/*
13. if (STL\_SCH\_RunCpuTM1L(&StlCpuTm1LStatus) != STL\_OK)
14. {
15. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
16. FailSafe\_Handler(TM1L\_ERR\_CODE + DEF\_PROG\_OFFSET);
17. }
18. if (StlCpuTm1LStatus != STL\_PASSED)
19. {
20. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
21. FailSafe\_Handler(TM1L\_ERR\_CODE);
22. }
23. */\* CPU TM7 \*/*
24. if (STL\_SCH\_RunCpuTM7(&StlCpuTm7Status) != STL\_OK)
25. {
26. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
27. FailSafe\_Handler(TM7\_ERR\_CODE + DEF\_PROG\_OFFSET);
28. }
29. if (StlCpuTm7Status != STL\_PASSED)
30. {
31. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
32. FailSafe\_Handler(TM7\_ERR\_CODE);
33. }
34. */\* CPU TMCB \*/*
35. if (STL\_SCH\_RunCpuTMCB(&StlCpuTmCBStatus) != STL\_OK)
36. {
37. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
38. FailSafe\_Handler(TMCB\_ERR\_CODE + DEF\_PROG\_OFFSET);
39. }
40. if (StlCpuTmCBStatus != STL\_PASSED)
41. {
42. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
43. FailSafe\_Handler(TMCB\_ERR\_CODE);
44. }
45. #ifdef ARTI\_FAILING\_CPU\_TM
46. STL\_SCH\_StopArtifFailing();
47. #endif */\* ARTI\_FAILING\_CPU\_TM \*/*

### 2、CPU运行时刻的自检过程

系统的函数调用，同自检过程一致。自检频率为3ms执行一次。

## 3、STL FLASH存储器测试

STL 利用以下原则实现 Flash 存储器的测试（基于用户配置结构的实际内容）：

* 在用户应用程序定义的一个或多个子集的分区上执行测试。
* 针对用户应用程序定义的分区数，连续（一次性）执行或在单个原子步骤部分执行测试。
* 测试结果基于计算所得 CRC 值（在测试执行过程中计算）与预期 CRC 值（在软件二进制文件刷写前计算） 之间的 CRC 比较。

### 1、预期CRC的的预计算

* FLASH存储器测试基于内置的硬件CRC单元。
* 使用ST 提供 CRC 预计算工具。该工具作为 STM32CubeProgrammer 内部的单个功能来提供，会自动用填充位（0x00 模式）填充二进制文件，以便实现 32 位对齐。
* 编译生成可执行的.elf文件，使用命令行:

STM32\_Programmer\_CLI.exe -sl "$(BUILD\_DIR)/$(TARGET).elf" 0x08000000 0x08080000 0x400

重新生成带CRC校验的.elf文件。再生成对应的hex/bin文件。

* 下载至Flash

### 2、STL FLASH自检及运行时刻检查:

调用到的STL库函数为:

1. STL\_Status\_t STL\_SCH\_InitFlash(STL\_TmStatus\_t \*pSingleTmStatus);
2. STL\_Status\_t STL\_SCH\_ConfigureFlash(STL\_TmStatus\_t \*pSingleTmStatus, STL\_MemConfig\_t \*pFlashConfig);
3. STL\_Status\_t STL\_SCH\_RunFlashTM(STL\_TmStatus\_t \*pSingleTmStatus);

代码段如下所示:

1. */\*\*\*\*\*\*\*\*\*\*\*\*/*
2. */\* Flash TM \*/*
3. */\*\*\*\*\*\*\*\*\*\*\*\*/*
4. #ifdef ARTI\_FAILING\_FLASH\_TM
5. */\* forced STL\_FAILED status simulates the TM configuration error and*
6. STL\_NOT\_TESTED status simulates TM run error \*/
7. ArtifFailing.FlashTmStatus = STL\_FAILED;
8. STL\_SCH\_StartArtifFailing(&ArtifFailing);
9. #endif */\* ARTI\_FAILING\_FLASH\_TM \*/*
10. if (StlFlashStatus == STL\_NOT\_TESTED)
11. {
12. */\* Init & configure Flash TM \*/*
13. if (STL\_SCH\_InitFlash(&StlFlashStatus) != STL\_OK)
14. {
15. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
16. FailSafe\_Handler(TMF\_ERR\_CODE + DEF\_PROG\_OFFSET);
17. }
18. if (STL\_SCH\_ConfigureFlash(&StlFlashStatus, &FlashConfigRunTime) != STL\_OK)
19. {
20. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
21. FailSafe\_Handler(TMF\_ERR\_CODE + DEF\_PROG\_OFFSET);
22. }
23. else if (StlFlashStatus != STL\_NOT\_TESTED)
24. {
25. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
26. USER\_DEBUG\_NORMAL("STL Flash Test Fail\r\n");
27. FailSafe\_Handler(TMF\_ERR\_CODE);
28. }
29. }
30. */\* run partial atomic test \*/*
31. if (STL\_SCH\_RunFlashTM(&StlFlashStatus) != STL\_OK)
32. {
33. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
34. FailSafe\_Handler(TMF\_ERR\_CODE + DEF\_PROG\_OFFSET);
35. }
36. if (StlFlashStatus != STL\_PARTIAL\_PASSED)
37. {
38. if (StlFlashStatus == STL\_PASSED)
39. {
40. */\* test completed successfully - to be restarted \*/*
41. StlFlashStatus= STL\_NOT\_TESTED;
42. #if defined(STL\_EVAL\_MODE)
43. *// BSP\_LED\_Toggle(LED2);*
44. HAL\_GPIO\_TogglePin(LED01\_GPIO\_Port,LED01\_Pin);
45. #endif  */\* STL\_EVAL\_MODE \*/*
46. }
47. else
48. {
49. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
50. FailSafe\_Handler(TMF\_ERR\_CODE);
51. }
52. }
53. #ifdef ARTI\_FAILING\_FLASH\_TM
54. STL\_SCH\_StopArtifFailing();
55. #endif */\* ARTI\_FAILING\_FLASH\_TM \*/*

## 4、STL RAM存储器的测试

为了提供正确的 RAM 测试配置，必须具备以下结构。

* 块：16 字节（RAM\_BLOCK\_SIZE）的连续区域，由 STL 进行硬编码（与存储器物理扇区无关）。
* 分区：128 字节（RAM\_SECTION\_SIZE）的连续区域，由 STL 进行硬编码。
* 子集：一个连续区域，大小为两个块的倍数，具有 32 位对齐的起始地址。由于子集的最后一个部分可能小于 一个分区，因此子集大小不一定是分区大小的倍数。
* 子集大小 = N \* RAM\_SECTION\_SIZE + 2 \* M \* RAM\_BLOCK\_SIZE， 其中： – N 为 ≥ 0 的整数 – M 为 0 ≤ M < 4 的整数，当 M > 0 时，子集最后部分的大小与分区大小不一致。 由用户应用程序定义子集。

用户配置参数如下:

1. RamSubsetRunTime.StartAddr = TEST\_RAM\_START\_ADDR;
2. RamSubsetRunTime.EndAddr = TEST\_RAM\_START\_ADDR + TEST\_RAM\_SECTION\_NB\_RUN \* RAM\_SECTION\_SIZE - 1;
3. RamSubsetRunTime.pNext = NULL;
4. RamConfigRunTime.pSubset = &RamSubsetRunTime;
5. RamConfigRunTime.NumSectionsAtomic = 1; */\* split test into sections of 128 bytes \*/*

各个地址如下:

1. #define RAM\_BACKUP\_BUFFER\_SIZE 32
2. #define RAM\_SECTION\_SIZE 128U
3. #define TEST\_RAM\_SECTION\_NB\_RUN 8
4. #define TEST\_RAM\_START\_ADDR (0x20000000U + RAM\_BACKUP\_BUFFER\_SIZE)

### 1、RAM存储器的自检及运行时刻检查

调用到的STL库函数如下：

1. STL\_Status\_t STL\_SCH\_InitRam(STL\_TmStatus\_t \*pSingleTmStatus);
2. STL\_Status\_t STL\_SCH\_ConfigureRam(STL\_TmStatus\_t \*pSingleTmStatus, STL\_MemConfig\_t \*pRamConfig);
3. STL\_Status\_t STL\_SCH\_RunRamTM(STL\_TmStatus\_t \*pSingleTmStatus);

代码的调用如下:

1. */\*\*\*\*\*\*\*\*\*\*/*
2. */\* RAM TM \*/*
3. */\*\*\*\*\*\*\*\*\*\*/*
4. #ifdef ARTI\_FAILING\_RAM\_TM
5. */\* forced STL\_FAILED status simulates the TM configuration error and*
6. STL\_NOT\_TESTED status simulates TM run error \*/
7. ArtifFailing.RamTmStatus = STL\_NOT\_TESTED;
8. STL\_SCH\_StartArtifFailing(&ArtifFailing);
9. #endif */\* ARTI\_FAILING\_RAM\_TM \*/*
10. if (StlRamStatus == STL\_NOT\_TESTED)
11. {
12. */\* Initialize & configure Flash TM \*/*
13. if (STL\_SCH\_InitRam(&StlRamStatus) != STL\_OK)
14. {
15. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
16. FailSafe\_Handler(TMR\_ERR\_CODE + DEF\_PROG\_OFFSET);
17. }
18. if (STL\_SCH\_ConfigureRam(&StlRamStatus, &RamConfigRunTime) != STL\_OK)
19. {
20. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
21. FailSafe\_Handler(TMR\_ERR\_CODE + DEF\_PROG\_OFFSET);
22. }
23. else if (StlRamStatus != STL\_NOT\_TESTED)
24. {
25. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
26. FailSafe\_Handler(TMR\_ERR\_CODE);
27. }
28. }
29. */\* run partial atomic test \*/*
30. if (STL\_SCH\_RunRamTM(&StlRamStatus) != STL\_OK)
31. {
32. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
33. FailSafe\_Handler(TMR\_ERR\_CODE + DEF\_PROG\_OFFSET);
34. }
35. if (StlRamStatus != STL\_PARTIAL\_PASSED)
36. {
37. if (StlRamStatus == STL\_PASSED)
38. {
39. */\* test completed successfully - to be restarted \*/*
40. StlRamStatus= STL\_NOT\_TESTED;
41. #if defined(STL\_EVAL\_MODE)
42. */\* BSP\_LED\_Toggle(LED3); \*/*
43. #endif  */\* STL\_EVAL\_MODE \*/*
44. }
45. else
46. {
47. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
48. FailSafe\_Handler(TMR\_ERR\_CODE);
49. }
50. }
51. #ifdef ARTI\_FAILING\_RAM\_TM
52. STL\_SCH\_StopArtifFailing();
53. #endif */\* ARTI\_FAILING\_RAM\_TM \*/*

## 5、系统时钟频率的测试

时钟频率测量不属于 STL 软件包的集成部分，测量的基本原理如下:

用户使用LSI 触发的定时器检查 TM 时钟的系统时钟频率，以两个独立时钟源的交叉检查测量法为基础，用于频率比较的硬件最初必须被配置（TIM16 的通道 1 被 LSI 触发）

，硬件配置在 softclassb.c 文件中 STL\_Init()程序结束时完成。 API 被编写成使用与 STL 中集成的常规 API 兼容的接口，因此声明中使用相同格式：

STL\_Status\_t STL\_SCH\_RunClockTest(STL\_TmStatus\_t \*pSingleTmStatus)

在该函数调用期间传递的参数充当指向时钟模块测量状态的指针，并且函数本身提供 STL\_KO vs STL\_OK 返回状 态，如果发生防御性编程错误，常规 STL 模块同样如此。如果时钟测量硬件激活且最后一个测量周期（（置为8个连续LSI周期））更新的新周期值处于预期区间（通过宏 CLK\_LimitLow 和 CLK\_LimitHigh 定义）以内，则模 块测量状态值变为 STL\_PASSED。否则按照常规 API 模块置为 STL\_FAILED。

所调用到的函数为:

1. STL\_Status\_t STL\_InitClock\_Xcross\_Measurement(void)
2. void TIM1\_UP\_TIM16\_IRQHandler(void)
3. STL\_Status\_t STL\_SCH\_RunClockTest(STL\_TmStatus\_t \*pSingleTmStatus)

其中：STL\_InitClock\_Xcross\_Measurement()完成对TIM16的初始化。

### 1、系统时钟频率运行时的自检代码段

1. */\*\*\*\*\*\*\*\*\*\*\*\*/*
2. */\* Clock TM \*/*
3. */\*\*\*\*\*\*\*\*\*\*\*\*/*
4. */\* When artificial failing is activated for clock module, the failure status is*
5. forced by the API module itself available as open source code \*/
6. StlClkStatus = STL\_NOT\_TESTED;
7. if (STL\_RunClockTest(&StlClkStatus) != STL\_OK)
8. {
9. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
10. FailSafe\_Handler(CLK\_ERR\_CODE + DEF\_PROG\_OFFSET);
11. }
12. if (StlClkStatus != STL\_PASSED)
13. {
14. USER\_DEBUG\_NORMAL("Line: %d  file:%s \r\n", \_\_LINE\_\_, \_\_func\_\_);
15. FailSafe\_Handler(CLK\_ERR\_CODE);
16. }

## 6、整体测量流程如下:



代码片段如下:

1. scb\_init();
2. scb\_RunFullTest();
3. HAL\_Delay(10);
4. */\* USER CODE END 2 \*/*
5. */\* Infinite loop \*/*
6. */\* USER CODE BEGIN WHILE \*/*
7. while (1)
8. {
9. */\* USER CODE END WHILE \*/*
10. */\* USER CODE BEGIN 3 \*/*
11. iwdg\_refresh();
12. scb\_step();
13. HAL\_Delay(3);
14. }

## 7、测试效果：

系统正常运行，小灯以一定频率连续闪烁。

## 8、人为触发错误

调用到的库函数为：

1. STL\_Status\_t STL\_SCH\_StartArtifFailing(const STL\_ArtifFailingConfig\_t \*pArtifFailingConfig);
2. STL\_Status\_t STL\_SCH\_StopArtifFailing(void);

在用户文件中通过如下宏定义来触发不同的错误

1. *// #define ARTI\_FAILING\_CPU\_TM           (1)*
2. *// #define ARTI\_FAILING\_FLASH\_TM         (1)*
3. *// #define ARTI\_FAILING\_RAM\_TM           (1)*
4. *// #define ARTI\_FAILING\_CLK\_TM           (1)*

系统无法正常运行，会根据STL返回的错误码，进行错误处理。错误处理的函数如下：

1. static void FailSafe\_Handler(uint32\_t fail\_code)
2. {
3. USER\_DEBUG\_NORMAL("Err Code %d\r\n",fail\_code);
4. */\* Here, user has to add an action to setup & keep the application in safe state.*
5. The fail\_code input parameter together with TM dedicated user status value
6. can be used to identify detected issue and adopt specific action \*/
7. while (1)
8. {
9. #if defined(STL\_EVAL\_MODE)
10. */\* LED2 slow flashing loop \*/*
11. HAL\_GPIO\_WritePin(LED01\_GPIO\_Port,LED01\_Pin,GPIO\_PIN\_RESET);
12. HAL\_Delay(100);
13. HAL\_GPIO\_WritePin(LED01\_GPIO\_Port,LED01\_Pin,GPIO\_PIN\_SET);
14. if(fail\_code < DEF\_PROG\_OFFSET)
15. {
16. HAL\_Delay(1900);
17. }
18. else
19. {
20. HAL\_Delay(3900);
21. }
22. #endif  */\* STL\_EVAL\_MODE \*/*
23. }
24. }

# 五、编译与链接

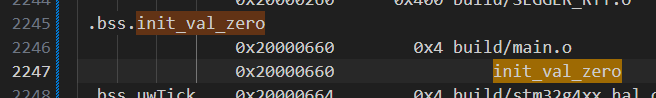
## 1、变量的存储

### 1、编译器默认的存储位置

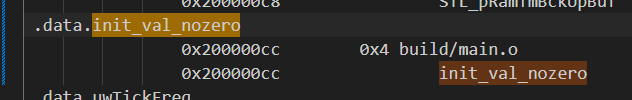
* + 1. 初始值不为0的全局变量
    2. 初始值为0的全局变量 volatile unsigned int init\_val\_nozero = 0xA5A5AAAA;

volatile unsigned int init\_val\_zero = 0;

3、直接查看.map文件



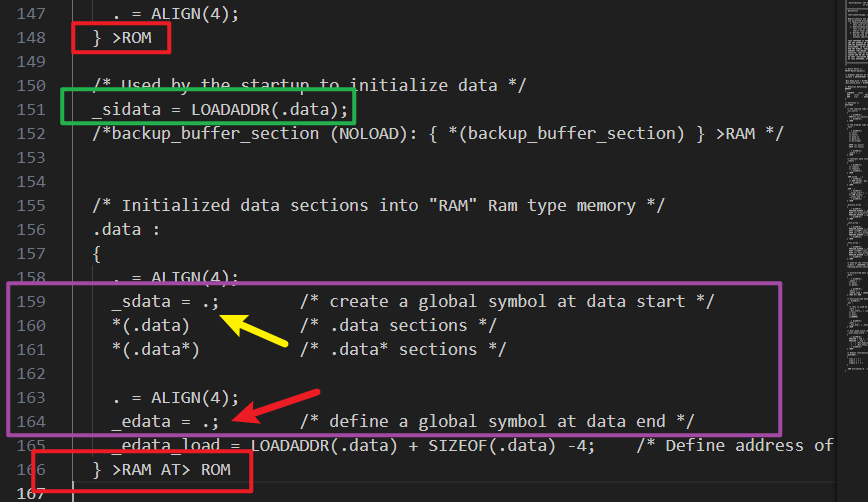
(初始值为0变量在内存上的位置)



（初始值不为0的变量在内存上的位置）

问题1：变量init\_val\_zero为什么出现在0x2000 00cc处？

查看并分析.ld文件。



由上图可知，链接文件里面对RAM区域进行操作时在绿色方框开始，绿色方块的意思是，将.data区域的地址(虚拟地址)赋值给\_sidata。

.data{}紫框里面的\_sdata = .代表将当前地址赋值给\_sdata。当前地址即为FLASH的地址。

合并所有的.data,.data\*区。

\_edata = . 代表将当前地址赋值给\_edata。

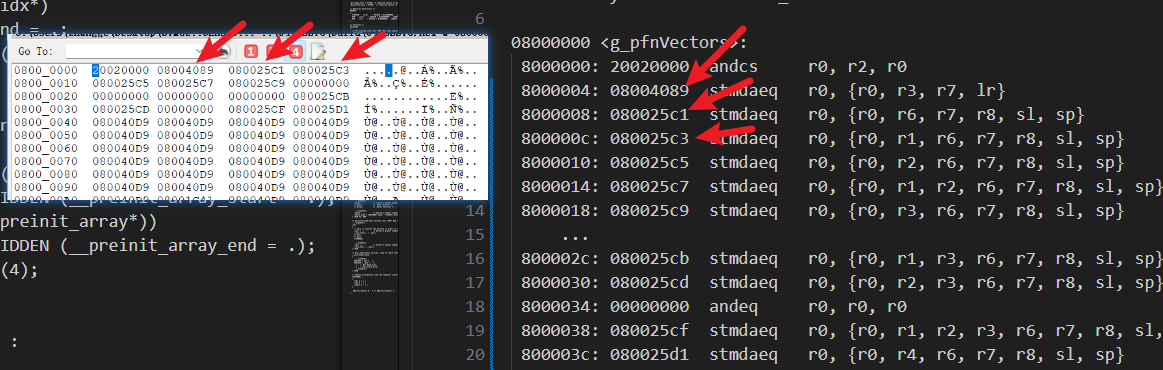
最后一个红框：代表将.data区定位到RAM区域上，并同时将这些数据放入到FLASH上面。

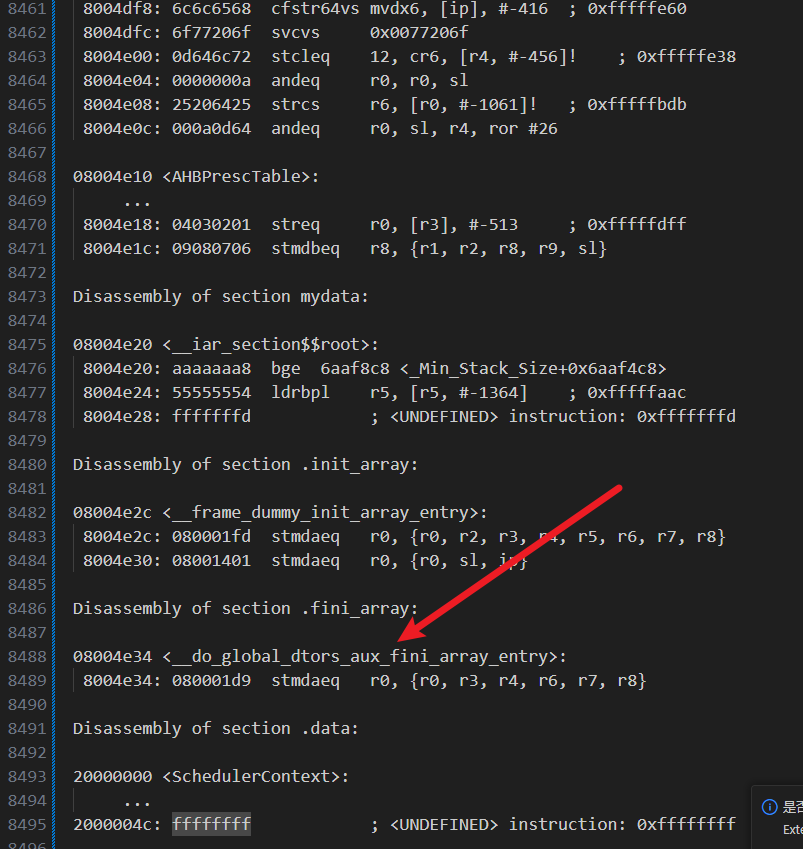
对hex文件的分析

首先hex文件上面存放的都是机器码，这些机器码必然是要存放到FLASH上面，供CPU进行读取调用的。看看.ld文件将什么东西放到了FLASH上面。

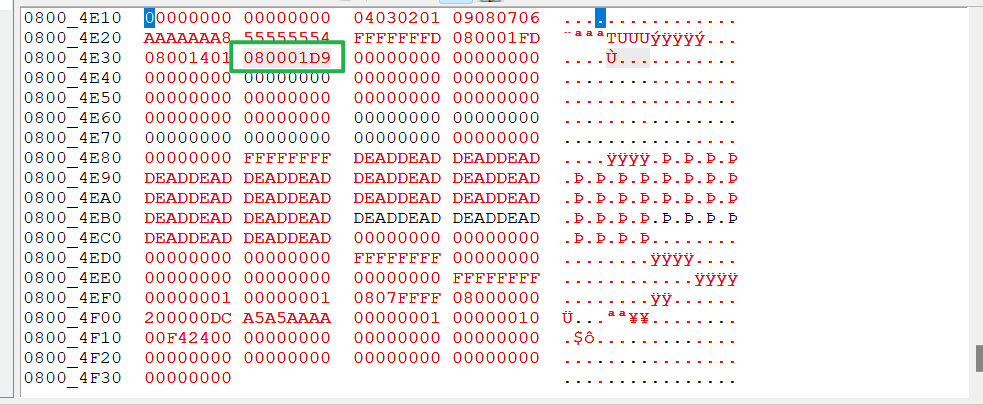
1. */\* Sections \*/*
2. SECTIONS
3. {
4. */\* 将中断向量表放到FLASH开头位置 \*/*
5. .isr\_vector :
6. {
7. . = ALIGN(4);
8. KEEP(\*(.isr\_vector)) */\* Startup code \*/*
9. . = ALIGN(4);
10. } >ROM
11. */\* 紧接着存放所有的代码段放到\*/*
12. .text :
13. {
14. . = ALIGN(4);
15. \*(.text)           */\* .text sections (code) \*/*
16. \*(.text\*)          */\* .text\* sections (code) \*/*
17. \*(.glue\_7)         */\* glue arm to thumb code \*/*
18. \*(.glue\_7t)        */\* glue thumb to arm code \*/*
19. \*(.eh\_frame)
20. KEEP (\*(.init))
21. KEEP (\*(.fini))
22. . = ALIGN(4);
23. \_etext = .;        */\* 所有的代码到此结束 \*/*
24. } >ROM
25. */\* 一些字符串常量及其他标准库或编译器等东西放在FLASH上 \*/*
26. .rodata :
27. {
28. . = ALIGN(4);
29. \*(.rodata)         */\* .rodata sections (constants, strings, etc.) \*/*
30. \*(.rodata\*)        */\* .rodata\* sections (constants, strings, etc.) \*/*
31. . = ALIGN(4);
32. } >ROM
33. .ARM.extab   : {
34. . = ALIGN(4);
35. \*(.ARM.extab\* .gnu.linkonce.armextab.\*)
36. . = ALIGN(4);
37. } >ROM
39. .ARM : {
40. . = ALIGN(4);
41. \_\_exidx\_start = .;
42. \*(.ARM.exidx\*)
43. \_\_exidx\_end = .;
44. . = ALIGN(4);
45. } >ROM
46. .preinit\_array     :
47. {
48. . = ALIGN(4);
49. PROVIDE\_HIDDEN (\_\_preinit\_array\_start = .);
50. KEEP (\*(.preinit\_array\*))
51. PROVIDE\_HIDDEN (\_\_preinit\_array\_end = .);
52. . = ALIGN(4);
53. } >ROM
55. .init\_array :
56. {
57. . = ALIGN(4);
58. PROVIDE\_HIDDEN (\_\_init\_array\_start = .);
59. KEEP (\*(SORT(.init\_array.\*)))
60. KEEP (\*(.init\_array\*))
61. PROVIDE\_HIDDEN (\_\_init\_array\_end = .);
62. . = ALIGN(4);
63. } >ROM
65. .fini\_array :
66. {
67. . = ALIGN(4);
68. PROVIDE\_HIDDEN (\_\_fini\_array\_start = .);
69. KEEP (\*(SORT(.fini\_array.\*)))
70. KEEP (\*(.fini\_array\*))
71. PROVIDE\_HIDDEN (\_\_fini\_array\_end = .);
72. . = ALIGN(4);
73. } >ROM
74. */\* Used by the startup to initialize data \*/*
75. \_sidata = LOADADDR(.data);
76. */\*backup\_buffer\_section (NOLOAD): { \*(backup\_buffer\_section) } >RAM \*/*
77. */\* Initialized data sections into "RAM" Ram type memory \*/*
78. .data :
79. {
80. . = ALIGN(4);
81. \_sdata = .;        */\* create a global symbol at data start \*/*
82. \*(.data)           */\* .data sections \*/*
83. \*(.data\*)          */\* .data\* sections \*/*
84. . = ALIGN(4);
85. \_edata = .;        */\* define a global symbol at data end \*/*
86. \_edata\_load = LOADADDR(.data) + SIZEOF(.data) -4;    */\* Define address of the last word of this section, which is also the last word in the flash. \*/*
87. } >RAM AT> ROM
88. */\* Uninitialized data section into "RAM" Ram type memory \*/*
89. . = ALIGN(4);
90. .bss :
91. {
92. */\* This is used by the startup in order to initialize the .bss section \*/*
93. \_sbss = .;         */\* define a global symbol at bss start \*/*
94. \_\_bss\_start\_\_ = \_sbss;
95. \*(.bss)
96. \*(.bss\*)
97. \*(COMMON)
98. . = ALIGN(4);
99. \_ebss = .;         */\* define a global symbol at bss end \*/*
100. \_\_bss\_end\_\_ = \_ebss;
101. } >RAM
102. */\* User\_heap\_stack section, used to check that there is enough "RAM" Ram  type memory left \*/*
103. .\_user\_heap\_stack :
104. {
105. . = ALIGN(8);
106. PROVIDE ( end = . );
107. PROVIDE ( \_end = . );
108. . = . + \_Min\_Heap\_Size;
109. . = . + \_Min\_Stack\_Size;
110. . = ALIGN(8);
111. } >RAM
112. */\* Remove information from the compiler libraries \*/*
113. /DISCARD/ :
114. {
115. libc.a ( \* )
116. libm.a ( \* )
117. libgcc.a ( \* )
118. }
119. .ARM.attributes 0 : { \*(.ARM.attributes) }
120. }

打开.dis反汇编文件:

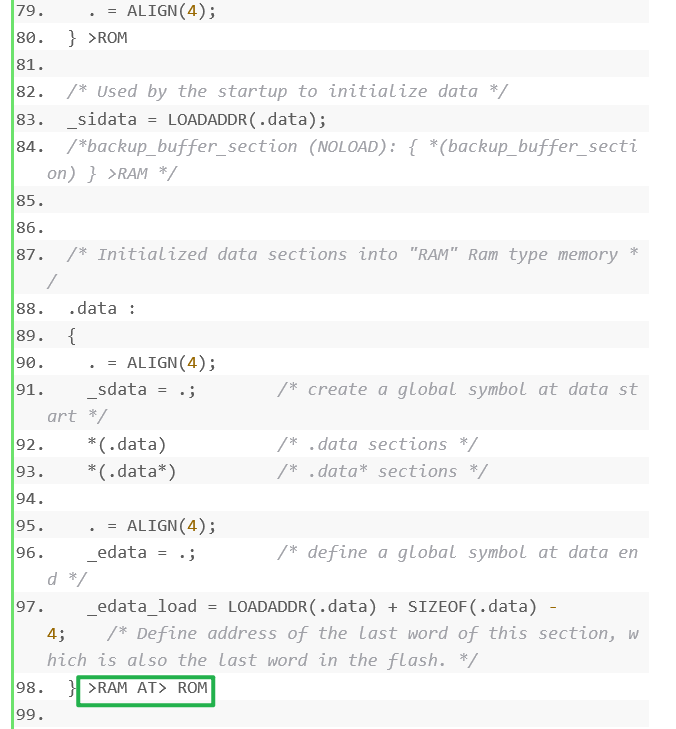
通过反汇编文件和hex文件，可以对比看出片上FLASH的数据可以在反汇编文件上体现出来。



可以看出在红色箭头位置，FLASH已经结束了。通过JFLASH也可以从侧面验证这一个猜想。



FLASH结束的位置在4F34处。也就说4E34-->4F34里面这些数据来自何处？



刚才提到要把.data区的内容放到FLASH里面去，这就是原因。

我们试着把\_edata\_load这个值读出来



0x0800 4F23

通过JLINKFLASH打开对应的hex文件分析：