ROM Self-Test in MDK-ARM

MDK Version 5 – Functional Safety

AN277, May 2015, V 1.0 Matthias Hertel matthias.hertel@arm.com

ARM KEIL Microcontroller Tools

Abstract

Self-Testing abilities are one of the most fundamental strategies to meet safety-critical system requirements. Typically the built-in self-tests for microcontroller systems are separated into 4 categories:

- Bus self-test
- ROM self-test
- RAM self-test
- CPU self-test

This application note discusses **ROM self-test** methods and their application in an MDK-ARM project.

Contents

Prerequisites

To run through the workshop you need to install the following software:

- MDK-ARM Version 5 (any version)
- SRecord 1.64+ utility

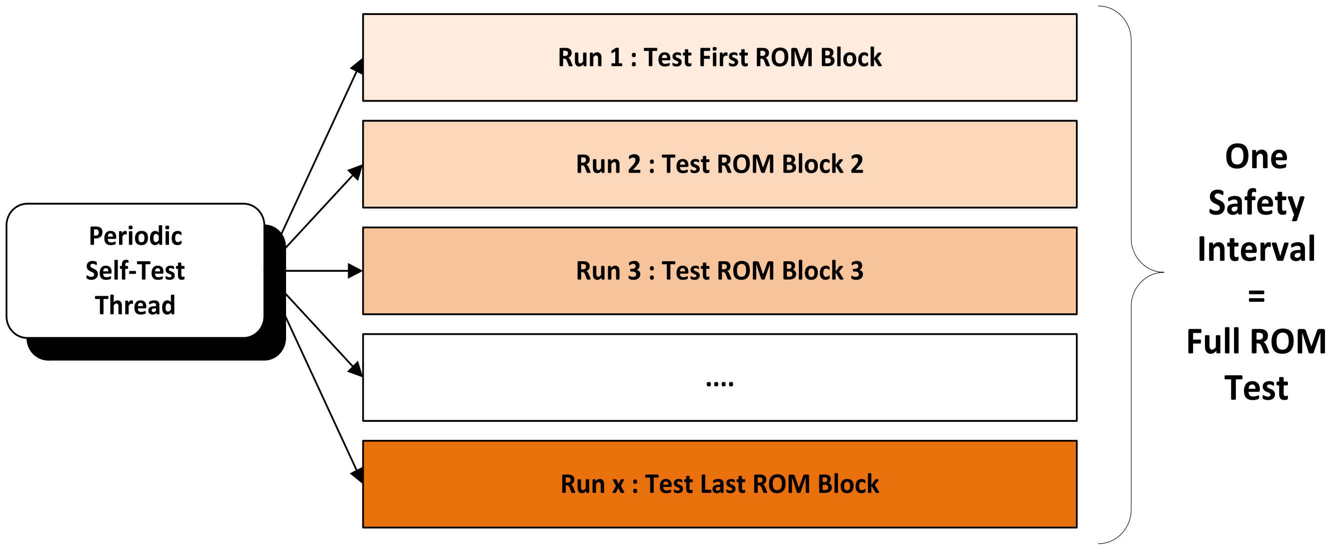
Fundamentals

The self-testing of a systems ROM areas ensures that the tested application code and constant data is stored correctly and can be accessed by the CPU.

A common practice for integrity check is based on a suitable checksum that is integrated into the ROM image. This will be calculated at compile time and the application includes the same checksum calculation to verify this at runtime.

There are three different run-time models that define when to test the checksums. They can be combined according to the requirements:

- Startup Test: This will be executed once at system startup and checks the full ROM area in question. The Example 1 that accompanies this appnote shows a practical approach of this technique.
- Periodic Test: This will be executed periodically. To reduce the run-time effects of the self-test on the application itself the ROM can be tested in smaller fractions of multiple intervals. This will be discussed in the Example 2.
- Test on Exception: A fault of the system was detected (e.g. HardFault). This incident will lead to a self-test and recovery routine which also includes testing of all memory areas.


Periodic Self-Test Thread
Run 1 : Test First ROM Block
Run 2 : Test ROM Block 2
Run 3 : Test ROM Block 3
....
Run x : Test Last ROM Block
One Safety Interval = Full ROM Test

Periodic Self-Test controlled by an RTOS thread

Choosing an algorithm

There are many checksum algorithms to choose from, depending on your application requirements. Some examples are:

CRC	The Cyclic Redundancy Check is available in many variants and will suit the most requirements.
Adler32	Similar to CRC but trades reliability for speed.
MD5	Is a hashing algorithm that was mainly used in encryption. Unless there is certain reasons like a hardware accelerator it will not have any advantage over CRC. Designed for encryption it deliberately does not allow data reconstruction based on the checksum/hash.
Fletcher	Is the base algorithm for Adler. It has weaknesses that make it very unsuitable for flash checksums as it cannot distinguish between blocks of 1's and 0's which are very common in padded ROM images.

CRC32 (32-bit wide cyclic redundancy check) checksums are very proven in use and do cover self-testing of ROM areas on a 32-bit systems very well. The examples of this application make use of a CRC32 with the standard polynomial of 0xedb88320 (X^32+X^26+X^23+X^22+X^16+X^12+X^11+X^10+X^8+X^7+X^5+X^4+X^2+X+1)

The probability that a number of bit failures is still undetected does exist and is related to the data length. To make sure you meet your systems requirements it is recommended to study the specification of CRC: http://en.wikipedia.org/wiki/Cyclic_redundancy_check

CRC32 – Implementation Variants

Calculating the CRC checksum requires a decent amount of time and memory resources. To lower the impact on your system for this additional task there are some implementations to select from.

Naive: The CRC is calculated completely at runtime with a series of modulo divisions. This preserves memory, but also takes typically 10-times as long as a table-based approach. The average throughput of a Cortex-M3 is around 1Mbyte/s at 100 MHz with this approach.

Table based / Lookup-Table runtime generated: A table with every possible remainder for every input byte can be precomputed. While there are still some fundamental operations required (shifts, XOR and lookup) the throughput of this approach can achieve over 10 Mbyte/s at 100 MHz on a Cortex-M3. The downturn is that the size of the lookup table is 256 4-byte integers for a CRC32, occupying 1 Kbyte of RAM space.

Table based / Lookup-Table pre-generated: Similar to the run-time generated table, but with a table generated at compile-time that is stored in the constant memory (ROM) of the MCU. This saves RAM and trades in 1Kbyte of ROM memory instead.

Hardware Acceleration: Some Cortex-M based MCUs offer a CRC peripheral. This has many advantages. It typically does require almost no resources and offers a higher throughput than a software solution. The main advantage though is that DMA transfers totally offload the CPU from CRC calculation down to around 1% of the execution time. This makes a hardware accelerated CRC the ideal choice for systems with periodic ROM testing as it makes the CRC almost non-blocking for the application.

If the ROM access time is high it might be considerable to run the CRC routine from RAM. The naive and dynamic table approach are most suitable.

Example 1 – Startup Testing

SelfTestROM_Ex is an example based on a simulated Cortex-M3 device with 256Kbyte of ROM area starting from address 0x00000000. The application uses three different implementations to calculate the CRC32 checksum.

Before building the example you need to download and copy the SRecord executables into the .*bin* folder.

Preparing an existing project to test the ROM consistency at startup is a process in three steps:

Prepare the ROM image and embed the calculated checksum at link time.
Add a CRC32 routine to the project and call this at system boot time.
Configure the debugger to use the processed file.

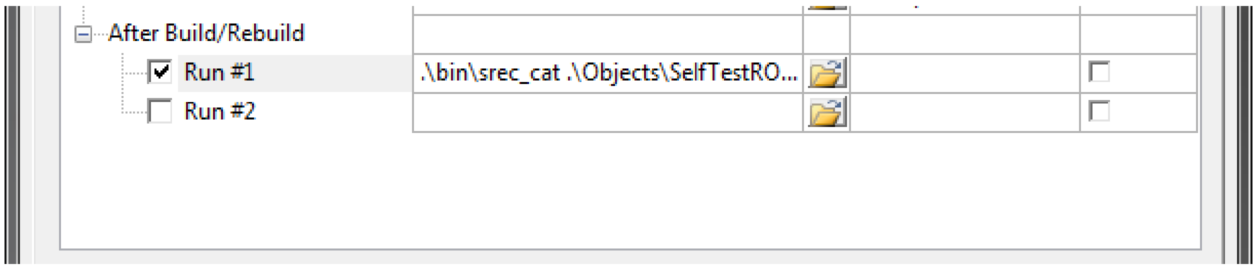
Prepare ROM Image using SRecord

1. Download the SRecord release for Windows from: http://srecord.sourceforge.net/download.html
2. Unzip the executables into a sub-folder of your project called *bin*.
3. Open **Project - Options for Target – Output.**
4. Enable **Create HEX File**.
5. Open **Project - Options for Target – User.**
6. At **After Build – Run #1** add the following command:

```
.\bin\srec_cat .\Objects\SelfTestROM_Ex.hex -intel -crop 0 0x3FFFC -fill 0x00
0x00000 0x3FFFC -crc32-l-e 0x3FFFC -o .\Objects\SelfTestROM_Ex_checked.hex -intel
```

The srec_cat utility command will do the following after linking of the project:

- Read the executable file as intel hex.
- Crop the area that is used to 256Kbyte minus 4 Bytes (at the end, where the Checksum itself will be stored).
- Fill the unused areas with 0x00.
- Calculate a CRC32 checksum of the range 0x0-0x3FFFC and store it at 0x3FFFC in little-endian format.
- Store the result in intel hex format in a file called SelfTestROM_Ex_checked.hex



.\bin\srec_cat .\Objects\SelfTestRO...

Add CRC32 Routines to the Application

Typically the ROM test will be executed in the very beginning of main as follows:I

1. Add a CRC32 implementation module from the example project to your project, e.g.: CRC32_fast_static_lut.c
2. Add defines for your ROM area, the location of the ROM checksum and include the prototype for the CRC routine at the beginning of your main module:

```
#include "CRC32.h"

uint32_t crc_nominal __attribute__((at(0x0003FFFC)));
#define ROM_START 0x00000000
#define ROM_LEN   0x0003FFFC
```

3. Add the test code at the entry of main();

```
crc_actual = crc32_fsl(0,(void*)ROM_START,ROM_LEN);
if (crc_actual != crc_nominal) {
      //handle ROM mismatch exception
}
```

Now the CRC will be calculated at every startup of the system and compared with the pre-calculated checksum stored at the end of the ROM image. Depending on the safety design the system must be brought into a safe state upon a detected ROM mismatch and the error is reported to a master node or user interface.

Use the Processed Hex File for Download and Debugging

1. Create a text file in your project folder called debug.ini. Enter the following content:

```
LOAD .\Objects\SelfTestROM_Ex_checked.hex INCREMENTAL
```

2. Specifiy the debug.ini file at **Initialization File** under **Options for Target – Debug**
3. Specifiy the debug.ini file at **Init File** under **Options for Target – Utilities**

This will make sure that the content of the file that includes the checksum and fill bytes is used for flash download and debugging.

Example 2 – Periodic Testing

The example I is used the same way as Example I. It is required to add the SRecord utility to the .*bin* folder first. The main differences to the startup approach are:

- ROM is partitioned into 4 blocks of 64Kbyte size with individual checksums.
- An RTOS thread is used to periodically test one block per iteration at lowest system priority.

To add periodic testing prepare your example as explained for Example I first. Additional tasks are explained in the following:

Adding block-wise Checksums using the SRecord Utility

As additional multiple calls to the srec_cat tool are required creating a batch file is recommended.

1. Create a batch file called *srecord_crc32.bat*. To create a hex file with 3 64Kbyte partitions that carry a checksum in the last 4 bytes of each block add the following command sequence:

```
.\bin\srec_cat .\Objects\SelfTestROM_Ex2.hex -intel -crop 0x00000 0x0FFFC -fill
0x00 0x00000 0x0FFFC -crc32-l-e 0x0FFFC -o .\Objects\SelfTestROM.0.hex -intel
.\bin\srec_cat .\Objects\SelfTestROM_Ex2.hex -intel -crop 0x10000 0x1FFFC -fill
0x00 0x10000 0x1FFFC -crc32-l-e 0x1FFFC -o .\Objects\SelfTestROM.1.hex -intel
.\bin\srec_cat .\Objects\SelfTestROM_Ex2.hex -intel -crop 0x20000 0x2FFFC -fill
0x00 0x20000 0x2FFFC -crc32-l-e 0x2FFFC -o .\Objects\SelfTestROM.2.hex -intel
.\bin\srec_cat .\Objects\SelfTestROM_Ex2.hex -intel -crop 0x30000 0x3FFFC -fill
0x00 0x30000 0x3FFFC -crc32-l-e 0x3FFFC -o .\Objects\SelfTestROM.3.hex -intel
.\bin\srec_cat .\Objects\SelfTestROM.0.hex -intel ^
               .\Objects\SelfTestROM.1.hex -intel ^
                     .\Objects\SelfTestROM.2.hex -intel ^
                     .\Objects\SelfTestROM.3.hex -intel ^
                     -o SelfTestROM_checked.hex -intel
```

2. Open **Project - Options for Target – User.**
3. At **After Build – Run #1** add the *srecord_crc32.bat*

Creating a Self-Test Thread

1. First declare a list of dummy variables to make sure the linker will reserve the memory locations and the checksum can be added here without conflict:

```
uint32_t crc_nominal_dummy0 __attribute__((at(0x0000FFFC)));
uint32_t crc_nominal_dummy1 __attribute__((at(0x0001FFFC)));
uint32_t crc_nominal_dummy2 __attribute__((at(0x0002FFFC)));
uint32_t crc_nominal_dummy3 __attribute__((at(0x0003FFFC)));
```

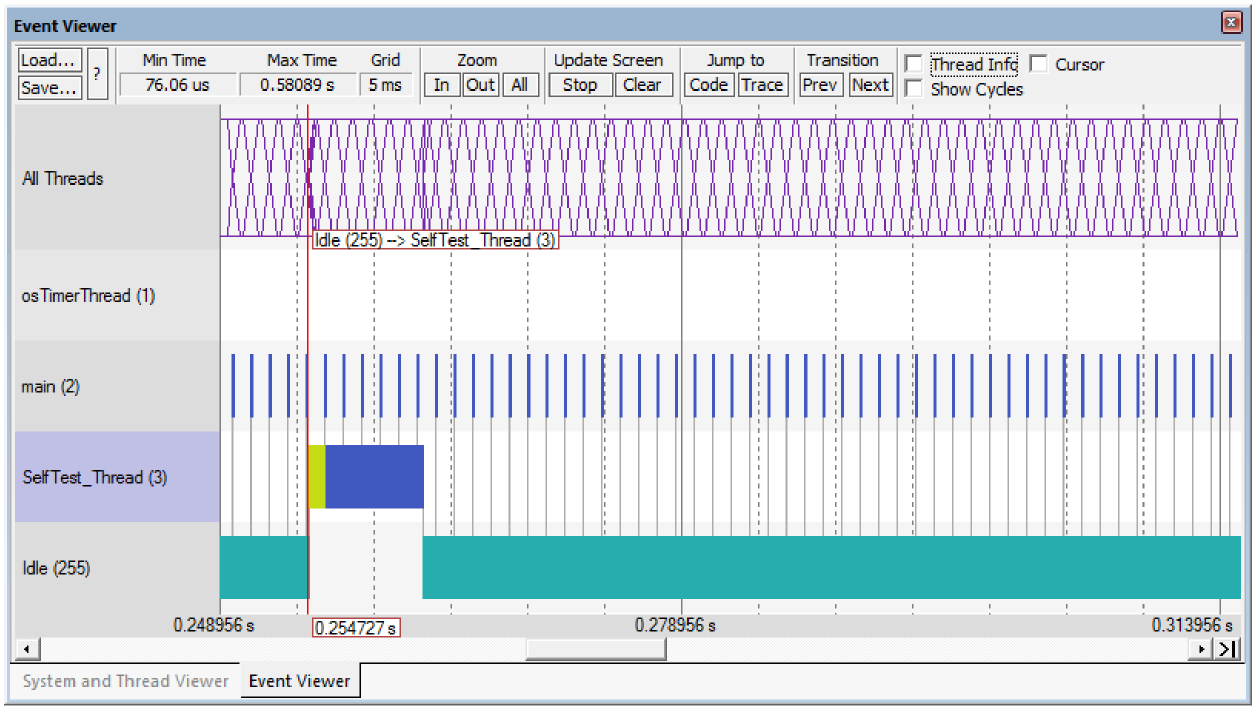
2. Now declare the metrics of the ROM Blocks and create an array that contains the checksum and block start address.

```
#define ROM_BLOCK_LEN     0x0000FFFC
#define ROM_BLOCK_NO      4

typedef struct crc_info
{
    const uint32_t crc_block_nominal;
    uint32_t       blockstart;
} crc_info_t;

crc_info_t g_crc_info[ROM_BLOCK_NO] = {
    0x0000FFFCul, 0x00000000ul,
    0x0001FFFCul, 0x00010000ul,
    0x0002FFFCul, 0x00020000ul,
    0x0003FFFCul, 0x00030000ul
};
```

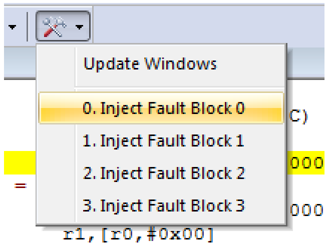
3. The Thread in our example will test one block every second.

```
void SelfTest_Thread(void const *argument)
{
    static uint32_t current_block_index = 0;
    while (1)
    {
        uint32_t crc_actual = 0;

        crc_actual =
crc32_fsl (0, (void*)g_crc_info[current_block_index].blockstart, ROM_BLOCK_LEN);
        if (crc_actual != *(uint32_t *)
g_crc_info[current_block_index].crc_block_nominal)
        {
            //handle ROM mismatch
        }
                          current_block_index++;
        if (current_block_index == ROM_BLOCK_NO)
        {
            current_block_index = 0;
        }

        osDelay(1000);
    }
}
```

The Example 2 is configured to be tested in simulation. Simply build and start the debugger.

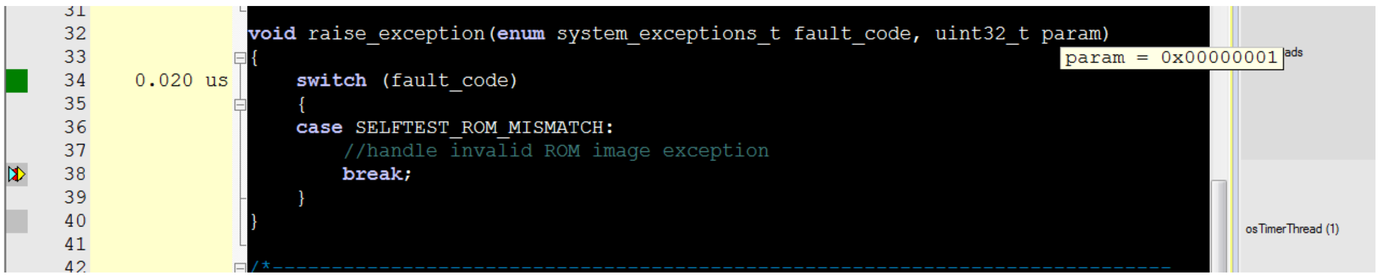


The Event Viewer shows how the SelfTest_Thread behaves.



Testing ROM failures on real hardware can be quite difficult. In simulation you can inject faults using debug commands. The example has this set up in the Toolbox menu.

This will cause the expected abortion of the application execution after a maximum of 4 seconds.



```
void raise_exception(enum system_exceptions_t fault_code, uint32_t param)
{
    switch (fault_code)
    {
    case SELFTEST_ROM_MISMATCH:
        //handle invalid ROM image exception
        break;
    }
}
```

Conclusion

MDK-ARM integrates SRecord for a flexible toolchain that can cover almost every requirement to runtime ROM diagnosis. SRecord also supports CRC generation of STM32 checksums to make use of the STM32 hardware CRC unit and various other special cases.