Aalto University
School of Electrical
Engineering

Special Assignment

Space Science Technology

Bit Flip Detection in Hercules LAUNCHXL2-RM46L852: A Study on Single Event Upsets in Space Conditions

Author:
Arnab Chakraborty
Masters Student, Dept. of Electrical and Automation
Aalto University

Advisor:
Anton Fetzer
Doctoral Researcher, Dept. of Electronics and Nanoengineering
Aalto University

Supervisor:
Jaan Praks
Associate Professor, Dept. of Electronics and Nanoengineering
Aalto University

August 2024

Project Links:

GitHub Repository
GitLab Repository

Contents

1 Introduction

In high-radiation environments like space,Microcontroller Unit(MCU) which is a type of IC(Integrated Circuit) are susceptible to **Single Event Upsets(SEUs)** such as bit flips which might occur in the memory and registers of the MCU.Such SEUs happens when ionizing radiation such as high-energy charged particles from cosmic rays or alpha particles in space strike the onboard electronic components of a spacecraft like MCU and flip the bits within the memory and registers by altering the properties of the electron used to store data[1].Although bit flips are classified as "soft errors",it can cause unexpected behavior in critical systems and lead to data corruption[1].These kind of disruptions is a major concern for spacecraft manufacturers and space mission designers since the success of the mission will depend on how reliably such errors could be mitigated or handled.So,to take into account such errors during risk assessment phase prior to a space mission,SEU cross section of the MCU is performed.During finding SEU cross-section,MCU is irradiated with ionizing particles using a particle accelerator and the number of SEUs that the MCU experiences is noted down based on the number of particles and Linear Energy Transfer (LET) of the particles. The SEU cross section data helps the engineers to predict the likelihood of the number of SEUs that the chip will experience during a mission to a specific orbit.This information is essential for the engineers to design robust fault tolerant recovery mechanisms before a space mission that can mitigate such soft errors and also helps to choose the appropriate MCU that can reliably function in the radiation environment of the desired orbit that the spacecraft is supposed to operate in.Since different orbits have different level of radiation exposure and it had been studied over the years and documented,engineers can now compare these radiation levels to the SEU cross-section of the MCU.Such comparison can provide important details on how long the MCU can withstand the expected radiation dose over the mission's lifetime.This allows the engineers to assess and make informed decisions on design of the spacecraft with appropriate shielding and fault-tolerant systems.The ultimate goal of this project "Bit Flip Detection in Hercules LAUNCHXL2-RM46L852" is to similarly find the SEU cross-section of the MCU **RM46L852CPGET** on the Texas Instruments (LAUNCHXL2-RM46) RM46L852 Hercules Launchpad evaluation board.It is a lockstep ARM® Cortex®-R4F based MCU with the safety features[2] and we plan to irradiate the MCU in a particle accelerator to determine its SEU cross-section.Since SEU cross-section is determined by the number of observed bit flips divided by the number of incident particles per square centimeter of the radiation beam[3], we in this project developed the necessary firmware to observe and count these bit flips when the mentioned MCU will be irradiated in the particle accelerator.Our plan is to use this MCU in future CubeSat missions and knowing its SEU cross-section will help us to predict how many SEUs the MCU will experience during future missions.Through this project,we want to observe all the bit flips that might occur in the RAM(Random Access Memory) of the MCU and also utilized the Lock-step feature of the MCU to detect bit flips that might occur in the registers of the CPU(Central Processing

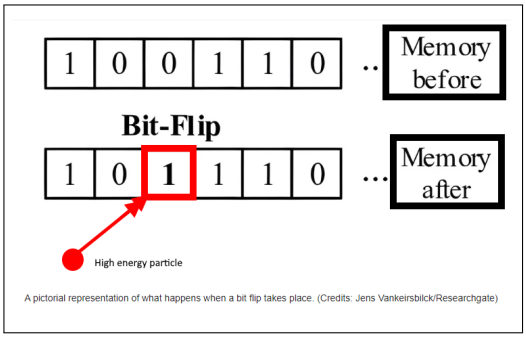


Figure 1: Bitflips in Memory[1]

Unit) causing signal mismatch within the CPU.

2 Methodology

For our project as mentioned in the introduction, we focused on observing the number of bit flips in CPU and RAM of MCU RM46L852CPGET.Below we discuss how we configured CPU and RAM of the MCU in Halcogen(Hardware Abstraction Layer Code Generator),how does the developed firmware(code) work and finally how users can use it or customize it.In addition to that,a python logging script was developed which listens to the status messages(status on bit flips) sent from the Hercules microcontroller via SCI(Serial Communication Interface) and displays the messages in the terminal to the user.

- **CPU**

  We know that CPU has registers which are used by the CPU to store data temporarily during execution of instructions.It also has address registers which are used to hold memory addresses so that CPU can access them when required.Since these registers store these data in the form of bits, they are also susceptible to bit flips when high energy particles such as cosmic rays strike the CPU.To observe and detect such bit flips in the registers of the CPU, we decided to use the Lockstep-mode feature(1oo1D Lock Step Mode) of the Hercules RM46L852 microcontroller.According to the datasheet[4],Hercules RM46L852 MCU has Cortex-R4F CPU developed by ARM and two cores of Cortex-R4F runs in parallel to detect any mismatch in signals.This is essential in safety-critical applications where reliability and run-time detection of faults is critical.The MCU has a CPU Compare Module (CCM-R4F) which is responsible for comparing the bus outputs of both these CPU cores that are running in parallel and signal or raise an error flag in case of signal mismatches.The CPU Compare Module (CCM-R4F) can operate in 4 different operating modes[4]: (i) 1oo1D lock step mode (ii)self-test mode (iii)error forcing mode (iv) self-test error forcing mode.For our purpose of detecting bit flips in the registers of the CPU cores, we will be using the 1oo1D(one-out-of-one, with diagnostics) lock step mode.This is because lock step mode facilitates continuous monitoring and any error due to bit flips is reported almost instantly.On the contrary, the other three operating modes: self-test mode, error forcing mode and self-test error forcing mode are non-continuous and are mostly used or designed for diagnostic purposes. Self-test mode is usually used to check the compare module itself for hardware faults by generating test patterns to verify if the compare module is functioning correctly or not and it does not take in to account the CPU signals.Again, error forcing mode is used to test the error detection module and its error signaling path by intentionally generating a known error in the CPU signals which causes the output to mismatch.Similarly,in self-test error forcing mode,an error is forced to self-test signals verifying that the CCM-R4F module detects and handles the self-test errors correctly.

  In Lock step mode[4],both the Cortex-R4F CPU cores referred to as Master CPU and Checker CPU runs the same instructions(our code) in parallel at almost the same time.The CPU Compare Module (CCM-R4F) then monitors and compares the bus outputs of both the Master and Checker CPU cores continuously.To prevent any sort of

case where both the CPU gets affected simultaneously by an external factor, the signals from both the Master CPU and Checker CPU is delayed intentionally: the input signals to the Checker CPU are delayed by 2 clock cycles while the output signals from the Master CPU are delayed by 2 clock cycles.In addition to that,CCM-R4F module begins its comparison 6 CPU clock cycles after a CPU reset to ensure that the outputs from the CPU cores have reached a stable and expected state.However, once the comparison starts, the compare module keeps monitoring the CPU signals continuously without any interruption.According to the datasheet[4], the compare module compares almost 900 signals from each of the two CPU cores which includes data, addresses and control signals originating from RAM and Flash of the MCU. In case of a difference noticed in

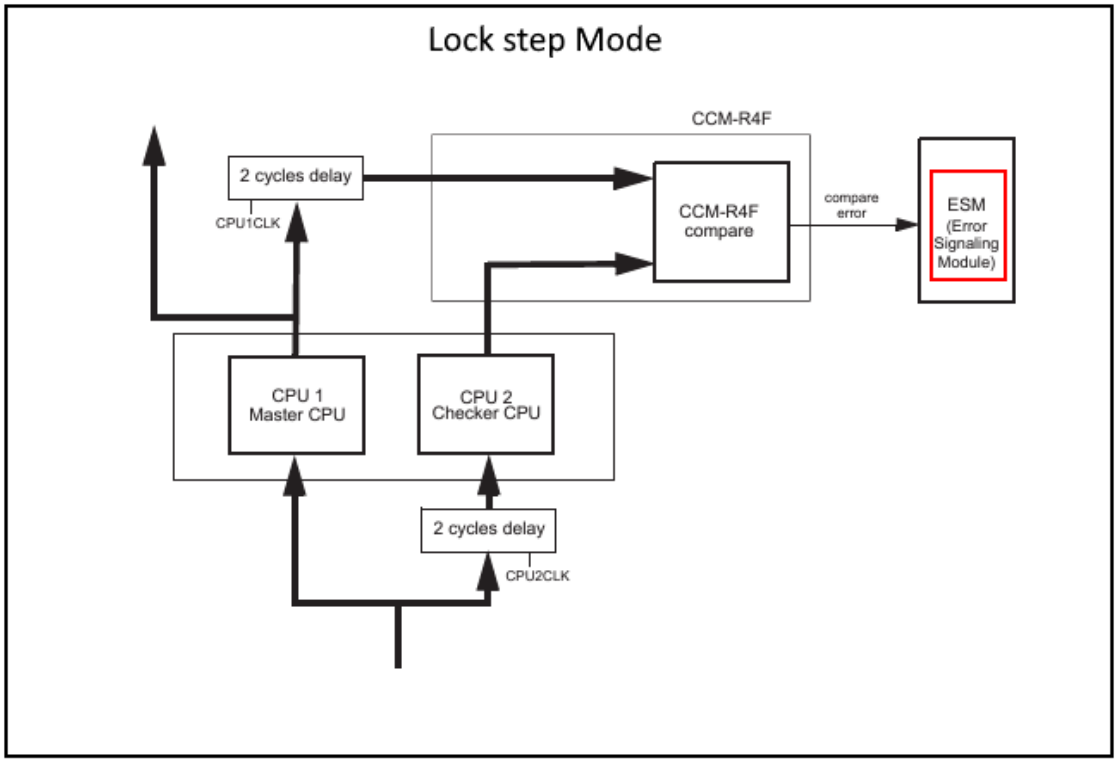


Figure 2: Lock Step Mode in Hercules RM46L852[4]

bus output signals of the Master and Checker CPU, the compare module immediately signals an error to the ESM(Error Signaling Module) which then sets the error flag "CCM-R4F compare" by setting the CMPE(Compare Error) bit to 1 which is the 16th bit of the CCM-R4F Status Register (CCMSR).Such kind of difference is usually noticed when there is a hardware fault or some high energy particles like cosmic rays hits the CPU registers leading to bit flips and eventually causing a signal mismatch.For our project, we will be checking this 16th bit of the CCM-R4F Status Register (CCMSR) continuously to see if there is any mismatch due to bit flips while testing the MCU in the particle accelerator and notify the user reagrding such bit flips as soon as the compare error flag is raised.

– **HALCoGen:**
  HALCoGen is the short form for Hardware Abstraction Layer Code Generator.It

was developed by Texas Instruments for helping the users to configure peripherals,clocks,interrupts,and other Hercules MCU parameters[5] using a GUI(Graphical User Interface).Once the MCU is configured in HALCoGen, user can generate peripheral initialization and low-level driver code which could be imported to TI Code Composer Studio (CCS) and eventually used for developing the main application code[5].This is extremely useful since users can prioritize on application development without needing to manually write low-level initialization code and thereby it helps in reducing the overall application development time.

For configuring the Lock step mode for detecting bit flips in the CPU, we at first need to configure the MCU using HALCoGen in the following way:

1. At first we need to download HALCoGen from the attached **link** and create a new project for the Hercules RM46L852 MCU.

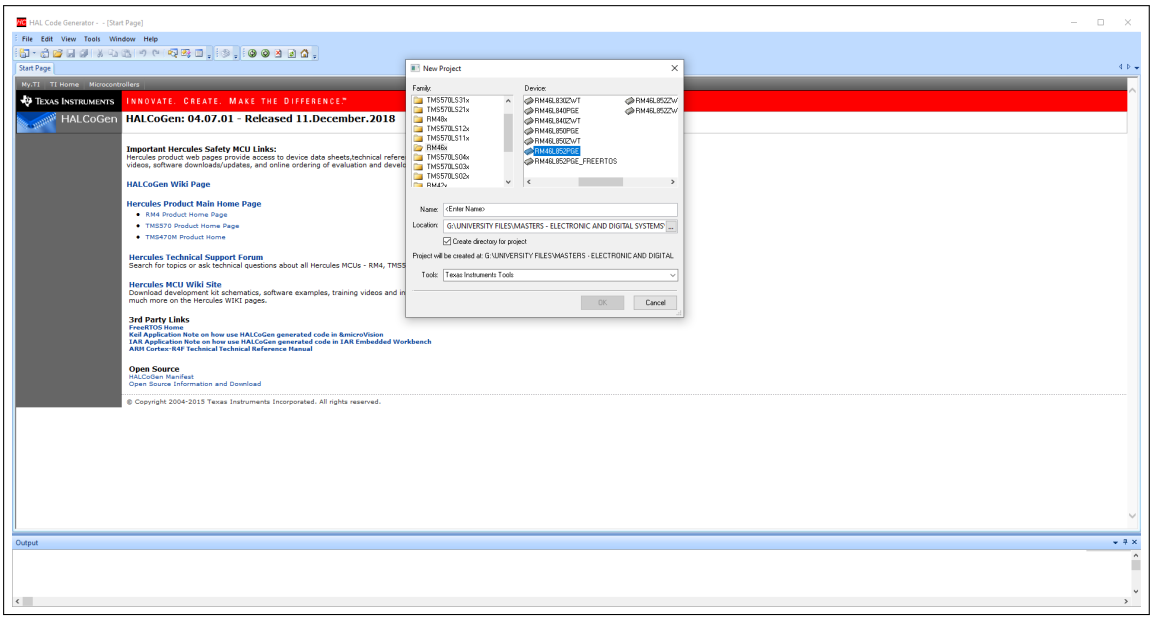


Figure 3: HALCoGen

2. Then, we need to navigate to the "Driver Enable" tab and make sure to enable GIO driver and SCI2 driver(SCILIN:LIN in SCI mode).Alongside that, we need to make sure other drivers are unticked or disabled.GIO(General Input/Output) driver in our firmware is required for controlling the LEDs which provides visual indication of the system's status and informs the user whether a bit flip has been detected or not in the Lock step mode.On the other hand,SCI2(Serial Communication Interface) driver in our firmware is required for sending log messages from the MCU via serial communication to the terminal which helps the user for real-time monitoring of the systems status and log the bit flips as soon as they are detected by the MCU.

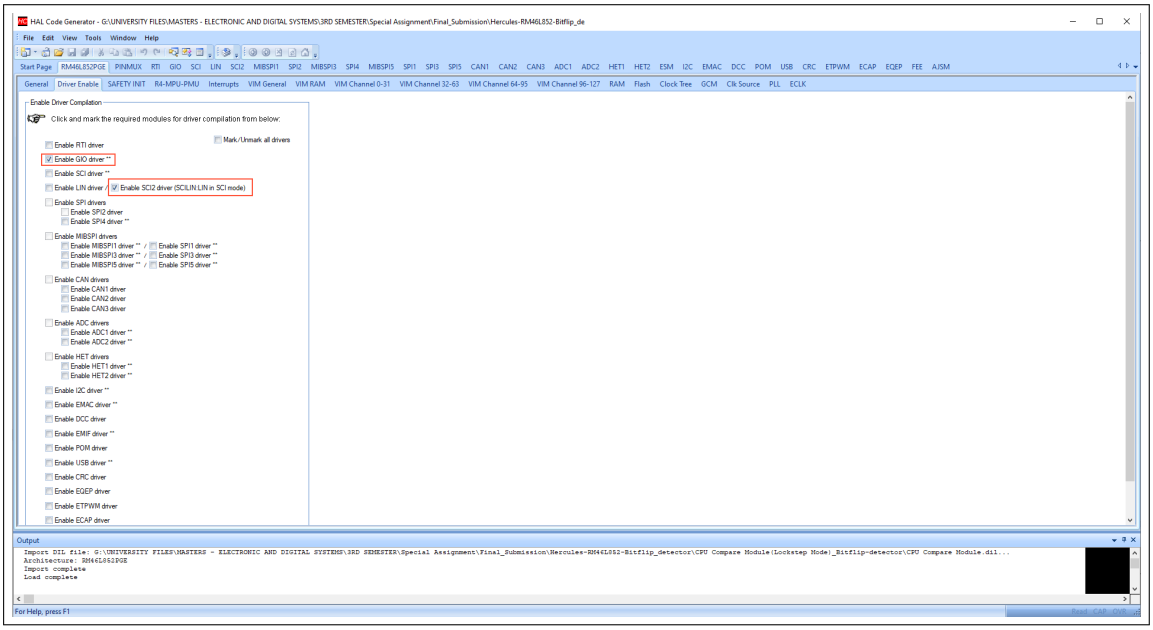


Figure 4: Setting up required drivers

3. For SCI2(Serial Communication Interface) driver to generate transmit interrupt and to send the log messages to the terminal, we need to enable TX INT(Transmit Interrupt) by navigating to SCI2 tab. Additionally,although not required for our code since our code is only transmitting the log messages to the terminal, we should also enable RX INT(Receive Interrupt) for future flexibility if we need to receive commands or data from the terminal to the MCU.In the SCI2 SCI/LIN Global tab, enable both TX INT and RX INT and then navigate to SCI/LIN Data Format tab and set Baudrate to 9600 Hz and stop bits to 2 and length to 8.

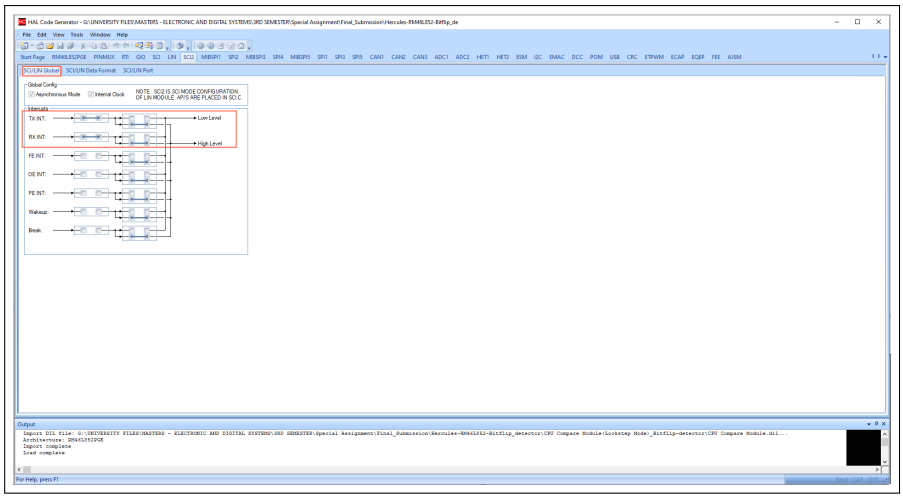


Figure 5: Enabling TX and RX INT

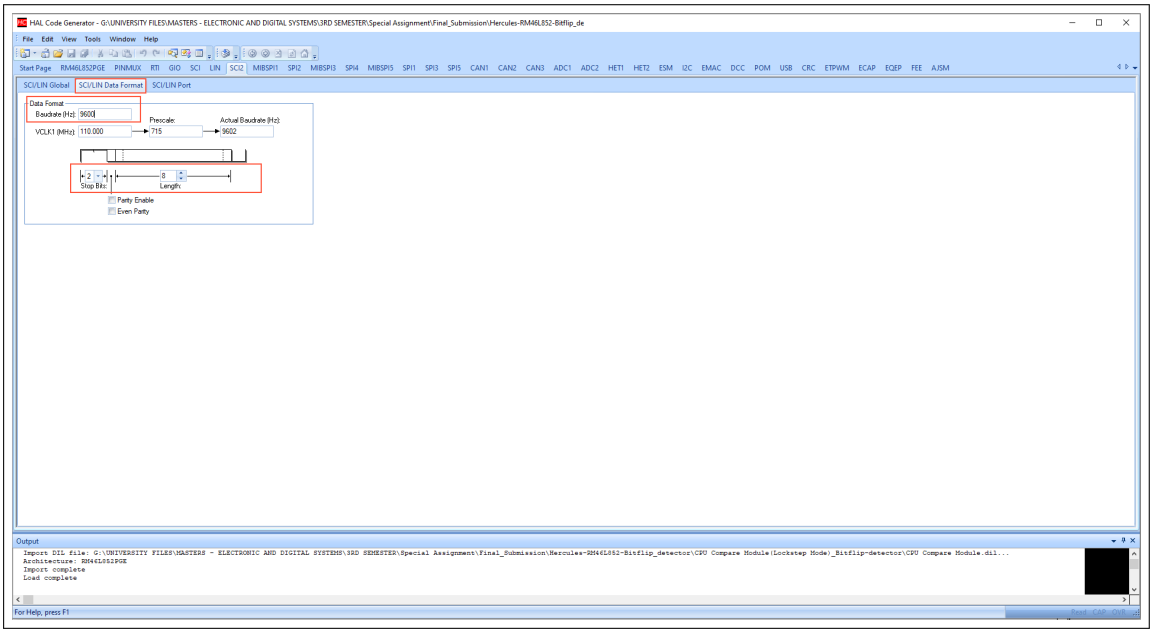


Figure 6: Setting data format for SCI2

4. After that we need to navigate to "SAFETY INIT" tab and enable CPU Self test and CCM Self test.This is required if we want to execute the built-in self-tests to verify that the CPU and the compare module(CCM-R4F) are functioning properly during system startup or before switching to Lockstep mode.This could also be useful after irradiation tests to check if the CPU or Compare module had any hardware faults or not due to the tests performed.Although this is optional and not implemented in our firmware,using the datasheet[4](CPU self-tests pg number: 369 and CCM self-test pg number:385) we can execute these built-in self-tests for diagnostic purposes ensuring the integrity of the CPU and the Compare module.

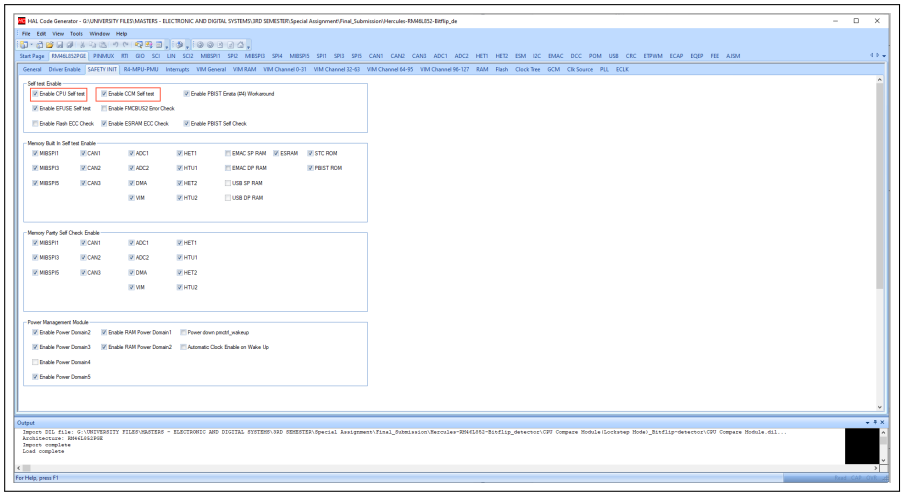


Figure 7: Enabling self-tests

5. We then navigate to General tab > click on VIM > then click on VIM Channel Config > enable the 13th(LIN1 High) VIM channel or you can directly navigate to VIM Channel 0-31 tab and enable the 13th(LIN1 High) VIM channel.This is required because VIM (Vectored Interrupt Manager) in the Hercules MCU is responsible for managing the interrupts in the MCU.The SCI2 peripheral which we used in our code for serial communication and logging the bit flip detected events generates interrupts whenever bit flip is detected and data(message) needs to be transmitted.The VIM then captures these interrupt signals and routes the interrupt request to the CPU.CPU then temporarily halts the current execution of tasks and processes the interrupt helping to transmit the log message to the terminal via the SCI2 module.Here,we need to enable the 13th VIM channel (LIN1 High) since it refers to the SCI2 module's interrupts and enabling this channel will allow the MCU to handle SCI2 interrupts.Without enabling it, the logging mechanism via terminal will fail and the detected bit flip events will not be logged.

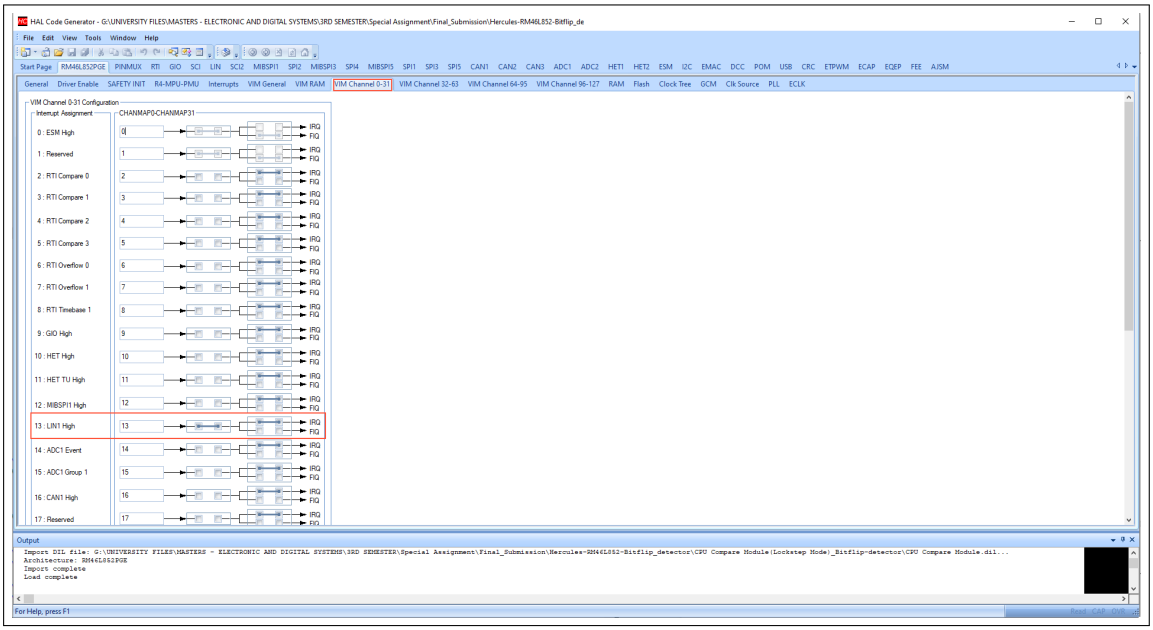


Figure 8: Enabling VIM 13th Channel (LIN1 High)

6. Finally, we can click on File > Generate code which will generate the necessary init or startup files, driver configurations and peripheral setup codes based on the configurations we set.After that, we downloaded Code Composer Studio IDE(**CCS**) and there imported the generated startup files and used those imported HAL files to write our application specific code.We deleted the main.c file generated with the project in CCS and instead used "sys_main.c" file in the source folder to write our firmware code for lock step mode.

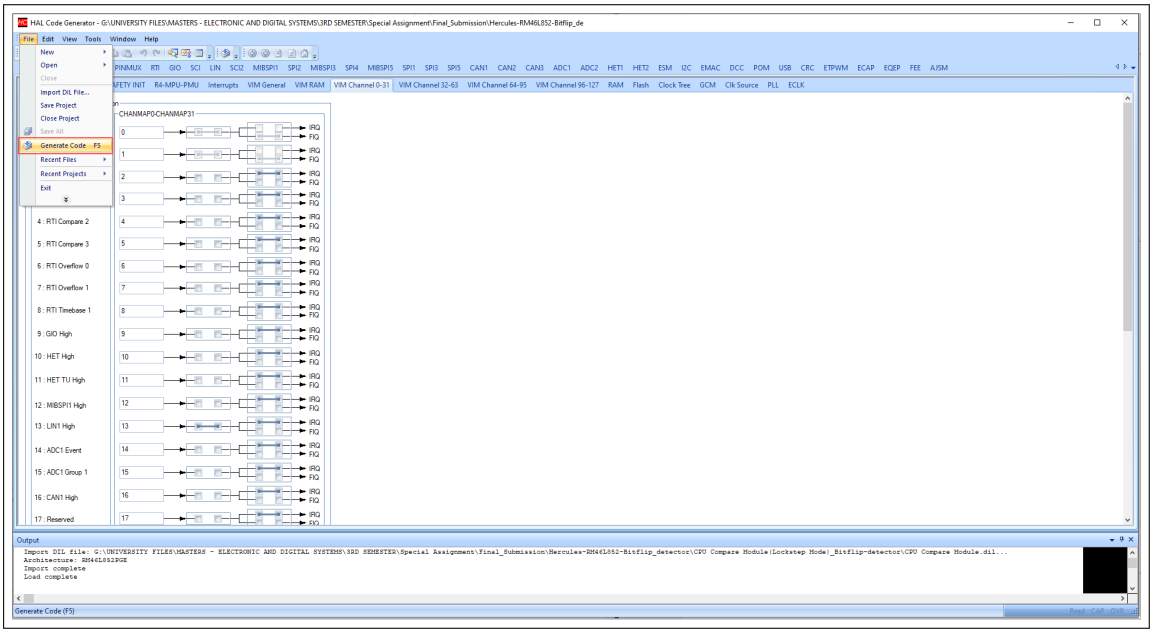


Figure 9: Generate initialization files

– **Code Description:**

Including Header Files:

```
#include "sys_common.h"
#include "system.h"
#include "sci.h"
#include "gio.h"
#include "esm.h"
#include "sys_core.h"
#include "sys_selftest.h"
#include <stdio.h>
#include <string.h>
```

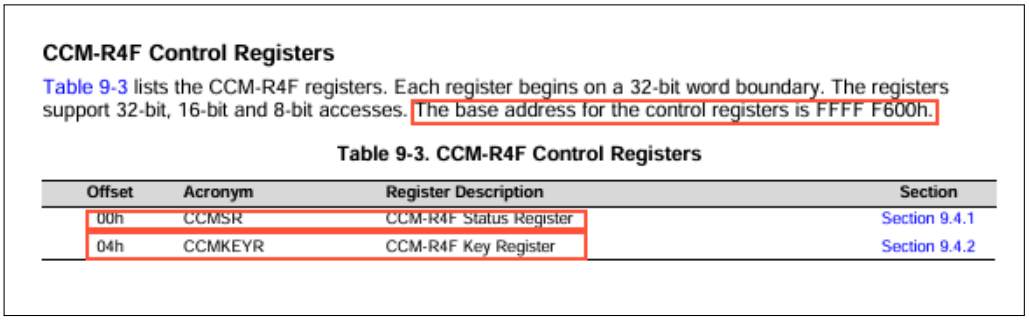
In the beginning of the code, we defined all the header files that were generated by HALCoGen and are needed for building our firmware for detecting bit flips in the CPU of the Hercules in Lock step mode.Here, “sys_common.h” header file provides common system level definitions, “system.h” header file provides system and peripheral control functions and “sys_core.h” header file which provides core system functions and interrupts.These are usually generated by HALCoGen as part of initialization files.We also defined “sci.h” which provides us with all the functions required for SCI(Serial Communication Interface) operations like transmitting log messages from MCU to terminal.Again, we have included “gio.h” which provides all the necessary functions to control General Input/Output (GIO) pins such as LEDs.Finally, we have “stdio.h” and “string.h” which are basic standard C library headers providing printf functions and string related operation functions like strlen() for calculating the length of the string.In addition to that, we have included “esm.h” and “sys_selftest.h” which are usually used for error signaling and

self-test functions.These are not used in our developed code and only have been included for future improvements.So they can be safely commented out if required.

Defining Base Addresses and Offsets:

```
#define CCMR4F_BASE_ADDR  0xFFFFF600  /**< Base
    address for the CCM-R4F module */
#define CCMKEYR_OFFSET    0x04        /**< Offset
    for the MKEY register */
#define CCMSR_OFFSET      0x00        /**< Offset
    for the CCMSR register */
```

Here, we define the base address i.e. the starting address of CCM-R4F module's(Compare module) control registers which is 0xFFFFF600 according to datasheet [4] (pg no: 390).This base address is the part of the memory-mapped IO region of the Hercules MCU and registers related to the compare module(CCM-R4F) start from this base address and can be accessed by adding an offset to this base address.The compare module has two 32-bit registers which are CCM Status Register (CCMSR) and CCM Key Register (CCMKEYR).According to the datasheet,CCM Status Register has an offset of 00h(0x00 in hexadecimal) which we defined as CCMSR_OFFSET and CCM Key Register (CCMKEYR) has an offset of 04h(0x04 in hexadecimal) which we defined as CCMKEYR_OFFSET.So,adding the offset to the base address we get their respective register address: CCM Status Register(0xFFFFF600 + 0x00 = 0xFFFFF600) since offset is zero and for CCM Key Register, we get 0xFFFFF600 + 0x04 = 0xFFFFF604.



CCM-R4F Control Registers

Table 9-3 lists the CCM-R4F registers. Each register begins on a 32-bit word boundary. The registers support 32-bit, 16-bit and 8-bit accesses. The base address for the control registers is FFFF F600h.

Table 9-3. CCM-R4F Control Registers

Offset	Acronym	Register Description	Section
00h	CCMSR	CCM-R4F Status Register	Section 9.4.1
04h	CCMKEYR	CCM-R4F Key Register	Section 9.4.2

Figure 10: CCM-R4F Control Registers[4](pg:390)

The reason we need to access these two registers is because: CCM Status Register helps us to monitor the status of the comparison between the two CPU cores running in lockstep mode in real-time and contains a compare error flag(CMPE) which indicates whether a mismatch between the Master CPU and Checker CPU has been detected or not.If mismatch detected, it sets compare error flag(CMPE) by setting the 16th bit(also known as CMPE bit) of the CCM Status Register to 1.By checking this CMPE bit continuously, we can monitor the bit flips since bit flips will trigger a mismatch between the signals of the Master CPU and Checker CPU.On the other hand, we have CCM Key Register (CCMKEYR) which is used for configuring and controlling the mode of operation for the CCM-R4F module.By writing a specific key to this register, we can set the mode of operation to any of

the following: (i) 1oo1D lock step mode (ii)self-test mode (iii)error forcing mode (iv) self-test error forcing mode.

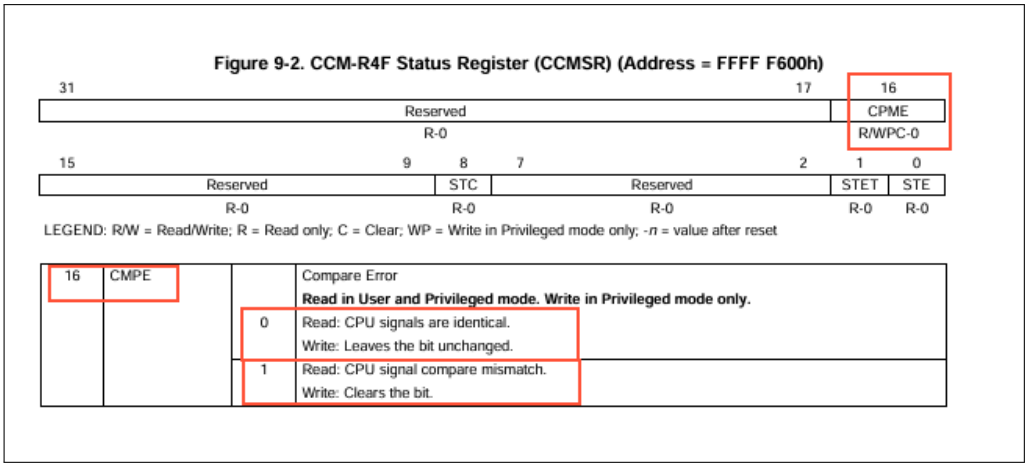


Figure 9-2. CCM-R4F Status Register (CCMSR) (Address = FFFF F600h)

31	17	16
Reserved		CPME
R-0		R/WPC-0

15	9	8	7	2	1	0
Reserved		STC	Reserved		STET	STE
R-0		R-0	R-0		R-0	R-0

LEGEND: R/W = Read/Write; R = Read only; C = Clear; WP = Write in Privileged mode only; -*n* = value after reset

16	CMPE		Compare Error **Read in User and Privileged mode. Write in Privileged mode only.**
		0	Read: CPU signals are identical. Write: Leaves the bit unchanged.
		1	Read: CPU signal compare mismatch. Write: Clears the bit.

Figure 11: CCM-R4F Status Register and CMPE bit[4](pg:391)

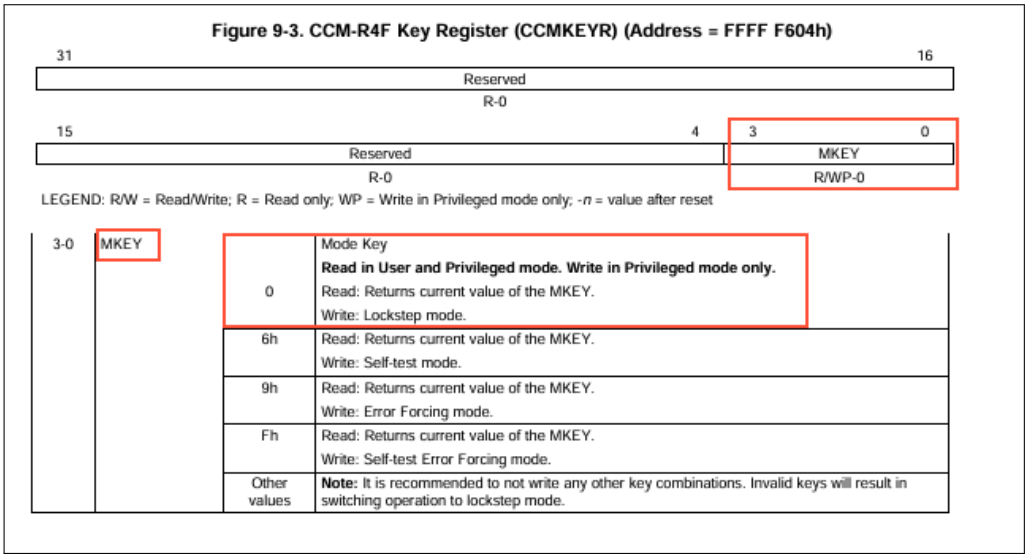


Figure 9-3. CCM-R4F Key Register (CCMKEYR) (Address = FFFF F604h)

31	16
Reserved	
R-0	

15	4	3	0
Reserved		MKEY	
R-0		R/WP-0	

LEGEND: R/W = Read/Write; R = Read only; WP = Write in Privileged mode only; -*n* = value after reset

3-0	MKEY		Mode Key **Read in User and Privileged mode. Write in Privileged mode only.**
		0	Read: Returns current value of the MKEY. Write: Lockstep mode.
		6h	Read: Returns current value of the MKEY. Write: Self-test mode.
		9h	Read: Returns current value of the MKEY. Write: Error Forcing mode.
		Fh	Read: Returns current value of the MKEY. Write: Self-test Error Forcing mode.
		Other values	**Note:** It is recommended to not write any other key combinations. Invalid keys will result in switching operation to lockstep mode.

Figure 12: CCM-R4F Key Register and Lockstep mode[4](pg:392)

Defining Lock step mode key:

```
#define CCMKEYR_LOCKSTEP  0x0          /**< Value
   to set CCM-R4F to lockstep mode */
#define CCMSR_CMPE_MASK   (1 << 16)    /**< Mask
   for the Compare Error flag bit in CCMSR */
```

As can be seen from figure 12, Lockstep mode key is 0x0 and setting this 0x0 value to CCM Key Register(0xFFFFF604) will operate the compare module(CCM-R4F module) in lockstep mode.Therefore,here we defined this 0x0 value as CCMKEYR_LOCKSTEP which will be later used in the code to set the compare module to lock step

mode.Additionally,we have also defined a CCMSR_CMPE_MASK which is used in the later part of the code to isolate the 16th bit of a 32-bit CCM Status Register (CCMSR) that indicates a CPU comparison error.The expression `(1 << 16)` performs a bitwise left shift that shifts the binary 1 sixteen places to the left resulting in a hexadecimal value 0x00010000(In binary:00000000 00000001 00000000 00000000).This value is then used as a bitmask since it can be used to check or isolate the specific bits like the 16th bit in the status register.

Defining LED Port and Pins:

```
#define LED_PORT gioPORTB                    /**< GIO
    port for LEDs */
#define LED_PIN_SUCCESS 1                    /**< GIO pin
    for success LED */
#define LED_PIN_ERROR 2                      /**< GIO pin
    for error LED */
```

In this part, we define the GIO(General Input/Output) port and their pins for controlling the LEDs on the MCU which we will use in the later part of the code to visually indicate the user whether an error has been detected or not.Here, we are using port B to control the LEDs where pin 1 of the GIO Port B will be used to control the success LED(LED2) and pin 2 of GIO Port B will be used to control the error LED(LED3).If there is no error i.e. no bit flips was identified,the MCU will blink the defined LED_PIN_SUCCESS(GIOB_1 or LED2) and if there was a bit flip or CPU comparison error, the MCU will blink the defined LED_PIN_ERROR(GIOB_2 or LED3).

Defining a delay function:

```
void delay(uint32_t count) {
    uint32_t i;
    for (i = 0; i < count; i++);
}
```

Here, we defined a delay function which introduces a time-based delay in our code.It is basically an empty for-loop that iterates from 0 to the value passed as the count argument.For count argument and loop variable i, we used data type uint32_t which is an unsigned 32-bit integer.The reason of using this data type is because it can accommodate larger range of values providing options of wide variety of delay lengths to the user.Moreover, it also makes sure no negative values are passed to the delay function since delay cannot be negative.In the later part of the code, we need this delay function to create blinking effects for the LEDs and control the blinking speed.Moreover, it also helps to introduce a delay between the log messages that are sent by the MCU to the terminal via SCI and prevents overpopulating or overwhelming the terminal output making it easier for the user to monitor the log or status messages in real-time.

Defining logToSerial function:

```
void logToSerial(const char* message) {
    while (!sciIsTxReady(scilinREG));
    sciSend(scilinREG, strlen(message), (unsigned
        char*)message);
}
```

Here, the logToSerial function is defined to send the log messages from the MCU to the connected terminal using the Serial Communication Interface (SCI).The function takes a constant character pointer (message) as an argument which is basically a string that needs to be send over SCI.When a string is passed as an argument to the function, the constant character pointer (message) of the function points to the memory address of the first character of the string.The while loop here continuously checks if the SCI module is ready for transmitting log messages or not via the function sciIsTxReady(scilinREG).The scilinREG argument inside the function sciIsTxReady represents the SCI register for LIN/SCI in SCI mode configured during the configuration of SCI2 in HALCoGen.The sciIsTxReady function returns true(i.e.1) when the transmitter or TX buffer is ready to send the message and returns 0 when it is not ready.The while loop here negates the sciIsTxReady function returned value i.e. if sciIsTxReady(scilinREG) returns 0 (SCI is not yet ready to send message), the negation makes it 1 and as a result, it keeps the loop running(while(1)) whereas when sciIsTxReady(scilinREG) returns 1 (SCI is ready to send message), the negation makes it 0 exiting the while loop and proceeding to the next step which is to send the log message via sciSend() function.This way the while loop makes sure to send the message using sciSend() function only when the buffer is ready.The sciSend() function which is responsible for sending the actual message via SCI interface takes three arguments:(a)scilinREG:It is the base register that is used for SCI interface and refers to SCI2 that was setup in HALCoGen,(b)strlen(message): strlen is a standard C library function which calculates the length of the message received and tells sciSend function about the number of bytes that needs to be sent,(c)(unsigned char*)message: this is the third argument and it casts the message from const char* to an unsigned char*.It is necessary to cast since sciSend function expects the message to be passed as an array of unsigned char as it expects the data to be in unsigned byte format.

Note: sciIsTxReady function and sciSend() function are defined in "sci.c" source file while scilinREG is defined in "reg_sci.h" header file.Both these functions sciIsTxReady() and sciSend() are provided by the SCI driver which is generated by HALCoGen during initialization.

Main function init part:

```
void main(void) {
    sciInit(); /**< Initialize SCI for serial
       communication */
    gioInit(); /**< Initialize GIO for LED control
        */

    /* Set LED pins as output */
    gioSetDirection(LED_PORT, (1 <<
       LED_PIN_SUCCESS) | (1 << LED_PIN_ERROR));
    _enable_IRQ(); /**< Enable interrupts */

    /* Initialize CCM-R4F and set to lockstep mode
        */
    *((volatile uint32_t *)(CCMR4F_BASE_ADDR +
       CCMKEYR_OFFSET)) = CCMKEYR_LOCKSTEP;
```

This is the main function and here in the beginning, we do all the initialization of the drivers required for our code to function as expected.At first, we use sciInit() which is initializing the SCI driver for serial communication and the function is defined in the source file "sci.c".It prepares the SCI module which will be used later for sending the detected bit flip events in the CPU as log messages to the terminal which can be used for real-time monitoring by the user.Then, we use gioInit() function which is defined in "gio.c" source file and used for initialization of the General Input/Output (GIO) module that is required in our code for controlling the LED pins on the MCU.LEDs will be used in our code as visual indicators to inform the user about whether a bit flip was detected or not.After that, we configure the pins of GIO Port B for controlling the defined success and error LEDs.Here, success led is pin 1 of GIO Port B which is referred to as LED2 on the MCU and error led is pin 2 of GIO Port B which is referred to as LED3 on the MCU.The gioSetDirection() function sets the direction of these mentioned pins and marks them as output pins.This is required since it will help us to drive the defined LEDs with respect to the system's status in the later part of the code.After that, we enable the interrupt by implementing the function call _enable_IRQ() which is required by the SCI module for sending log messages to terminal.The function _enable_IRQ() is compiler intrinsic which means it is a function that is provided by the compiler and not defined in any of the source files.The function basically enables IRQ(Interrupt Request) at the CPU level and initializes the processor to start responding to interrupt requests from peripherals like SCI module.Whenever SCI module is ready to send a message(i.e., the TX buffer is empty and ready for the next data to be sent), it generates a transmit (TX) interrupt and this interrupt lets the CPU know that the SCI module is ready to send message. Without enabling IRQ interrupts, the CPU will not be notified and messages will not be sent.Finally, we are initializing the compare module(CCM-R4F) to monitor the CPU cores in Lock step mode and it is done by setting the defined lock step mode key(0x0) in the CCM-R4F Key Register.Here, (CCMR4F_BASE_ADDR + CCMKEYR_OFFSET),we are basically adding the compare module base address which is 0xFFFFF600 to the CCM-R4F Key Reg-

ister offset which is 0x04 according to the datasheet.Adding both, we get the memory address for the CCM-R4F Key Register which is (0xFFFFF600 + 0x04 = 0xFFFFF604) as per datasheet.Since the key register controls the modes in which the CPU cores will operate, assigning a value of 0x0 as per figure 12 which we defined as CCMKEYR_LOCKSTEP will instruct the CPU cores to switch to Lock step mode and will keep operating until the mode is switched to some other modes by assigning a different key.

Main function while loop part:

```
                    while (1) {
                        /* Checking the CCM Status Register's 16th
                            bit using a bitwise AND operation
                         * to detect a CPU signal compare mismatch
                            ; an error is detected if the result
                         * of the operation is non-zero (meaning
                            that the 16th bit is set to 1)
                         */
                        if (*((volatile uint32_t *)(
                            CCMR4F_BASE_ADDR + CCMSR_OFFSET)) &
                            CCMSR_CMPE_MASK) {
                            logToSerial("\rCCM-R4F Lockstep Mode:
                                Error Detected!\r\n");
                            gioSetBit(LED_PORT, LED_PIN_SUCCESS,
                                0);     /**< Turn off success LED
                                */
                            gioToggleBit(LED_PORT, LED_PIN_ERROR);
                                      /**< Toggle error LED */

                            /* Clear error flags */
                            *((volatile uint32_t *)(
                                CCMR4F_BASE_ADDR + CCMSR_OFFSET))
                                = CCMSR_CMPE_MASK; /**< Clearing
                                compare error status bit */
                             //systemREG1->SYSECR = 0x8000;  /**<
                                Writing to SYSECR to trigger a
                                CPU reset when error is detected
                                */
                        } else {
                            logToSerial("\rCCM-R4F Lockstep Mode:
                                No Error Detected\r\n");
                            gioSetBit(LED_PORT, LED_PIN_ERROR, 0);
                                      /**< Turn off error LED */
                            gioToggleBit(LED_PORT, LED_PIN_SUCCESS
                                );     /**< Toggle success LED */

                        }

                        /* Delay before the next iteration */
                        delay(10000000); /**< Add a delay between
                            tests */
                    }
                }
```

This is the main application infinite while loop where we continuously check for bit flips in Lockstep mode.In the loop,we define an if else condition.Here,if block executes when a bit flip is detected i.e. a mismatch between the Master CPU and Checker CPU operating in Lock step mode is detected and else block is executed when there is no signal mismatch between the CPU cores indicating no bit flips were detected.For if block condition,we are checking the 16th bit of the CCM Status Register (CCMSR) since the 16th bit is set to 1 by the compare module when an error or mismatch is detected and it remains 0 when no error has been detected as discussed earlier and can be seen from figure 11.To check the 16th bit of the CCM Status Register continuously, we are performing a bitwise AND operation between the address of CCM-R4F Status Register that we get by adding the CCMSR_OFFSET value with the base address i.e. the starting address of CCM-R4F module and previously defined CCMSR_CMPE_MASK.If the 16th bit in the CCM Status Register is set(1) by the compare module, the bitwise AND operation will result to true(1) and execute the statements inside if condition while in other cases where the 16th bit is not set(i.e. 0), the bitwise AND operation will result in false(0) and the else block will be executed instead.

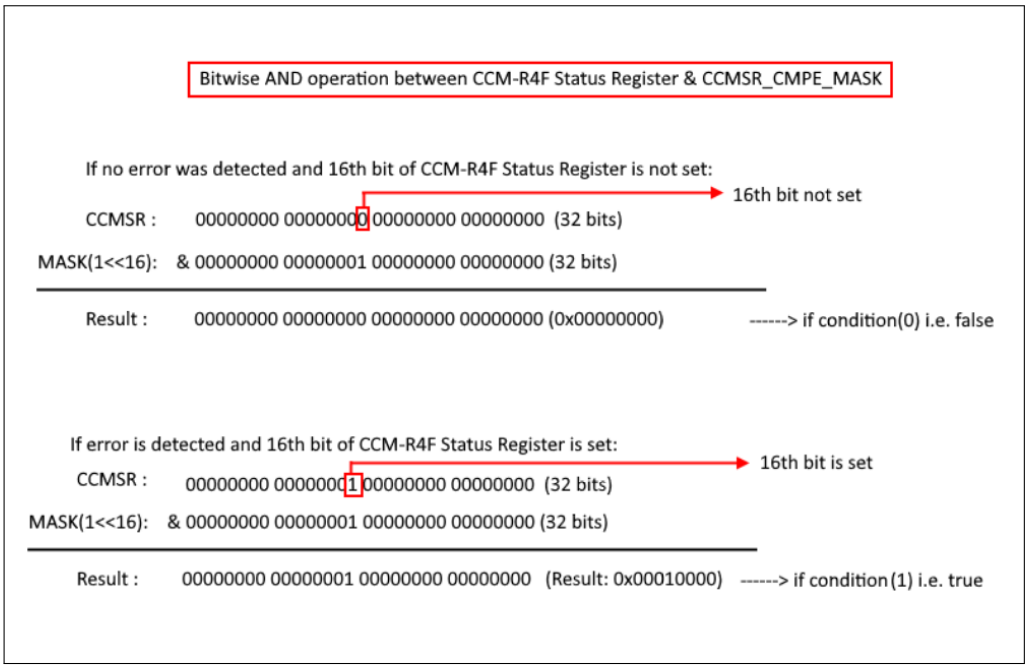


Figure 13: Bitwise AND operation between CCMSR and CCMSR_CMPE_MASK

Now if during the irradition tests, bit flip is detected then the if condition will evaluate to true and enter the if block.After entering the if block, it will at first call the defined logToSerial function to send a log message to the terminal.The log message will notify the user about the error(i.e. a signal mismatch) that has been detected in the lockstep mode indicating a potential bit flip.Next, we use the gioSetBit function from "gio.c" to set the Success LED(LED2) to 0 which turns the LED2 off.This is done to ensure that the success LED is not on when we turn on the Error LED(LED3).After that, we call the function gioToggleBit from "gio.c"

to toggle the state of the Error LED (LED3) which creates the blinking effect with each cycle in the while loop indicating the user about a potential bit flip.Finally, once the bit flip is notified to the user, we clear the error flag in the CCM Status Register (CCMSR) so that the system is ready to detect any subsequent errors or bitflips.To clear the error flag, as per the datasheet and figure 13,we need to write value 1 to the 16th bit(CMPE bit) of the CCM-R4F Status Register.So,for that,we are using again the CCMSR_CMPE_MASK(0x00010000) which has value 1 at 16th bit.By assigning this mask which has a value(0x00010000) to the CCM-R4F Status Register,we are specifically targeting the 16th bit and setting it to bit 1, which clears the error flag.After that,we trigger a CPU reset by writing value 0x8000 to SYSECR(System Exception Control Register) and this helps in resetting the MCU which clears the signal mismatches between the CPU cores that might have been caused due to bit flips[6].Resetting the MCU will remove such soft-errors and allow the MCU to detect any subsequent errors or bitflips in the next loop.However, we have commented this out since testing this reset functionality is not possible without the irradiation environment, like a particle accelerator where a bit flip will occur and the CPU will reset and then in next loop it will enter else condition showing no error.Currently, if I place this reset command in else condition,it keeps resetting continuously blocking the normal program flow.So,it can be uncommented when testing in a particle-accelerator,where actual bit flips are expected to occur.On the other hand,else block is executed when no error or bit flip is detected.Here,we use the same functions as if block and the only key difference is that instead of sending an error detected message and signaling an error, we send no error message,turn off the Error LED (LED3) and toggle the Success LED (LED2) to notify the user that the system is operating correctly and no bit flips were detected.Finally, outside the else block, we introduce a small delay between each iteration of the while loop which helps to provide the blinking effect perceivable by human user and also prevents overpopulating log messages in the terminal with each iteration.

Note:When a signal mismatch will occur due to bit flip in a real particle-accelerator, alongside our defined Error LED(LED3), the compare module will also trigger the ESM module(Error Signaling Module) which will turn on the red ERR LED signaling a mismatch.Since we did not have the option to test the ERR LED(cannot be accessed normally), we defined LED3 as our error led.So if an error occurs during irradiation tests, it is expected that LED3 will blink and at the same time,the ERR LED will turn on.

– **Usage:**

  * If the users want to see the HAlCoGen project and want to modify something, users can open HalCoGen software and click `File>Open>Project` and select the file named "CPU Compare Module.hcg" with the extension .hcg representing the HALCoGen file which one can find in the project folder.After opening it, users need to import the configured settings by importing the DIL file.For importing the DIL file, the user need to select `File>Import DIL File` and

finally choose the file named "CPU Compare Module.dil" which is also provided in the project folder or repository.After that, the user can either have a look at it or modify accordingly.Once modified, the user needs to regenerate the code by selecting `File>Generate code` or simply click F5.

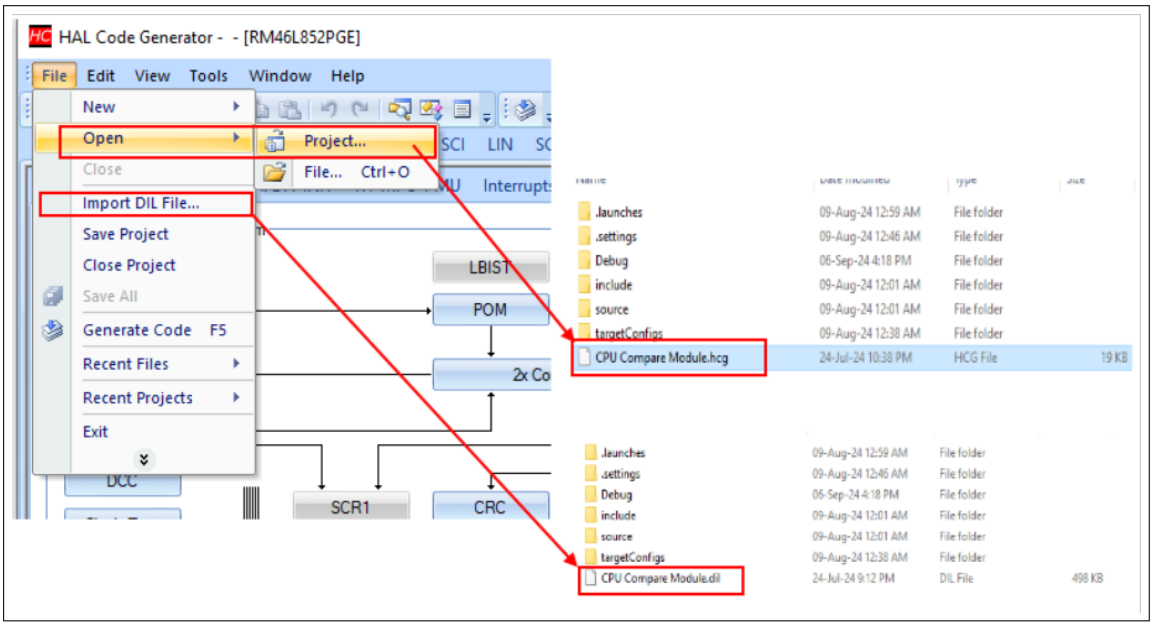


Figure 14: Opening the pre-configured HALCoGen file

∗ In the next step, to run and flash our developed firmware to the MCU, the user needs to install **Code Composer Studio** (recommended version: 12.5.0 (04 Oct 2023) or later).After that, the user needs to create their own workspace and navigate to `File > Open Projects from File System > Click on Directory` and select our project folder named "CPU Compare Module (Lockstep Mode)_Bitflip-detector".

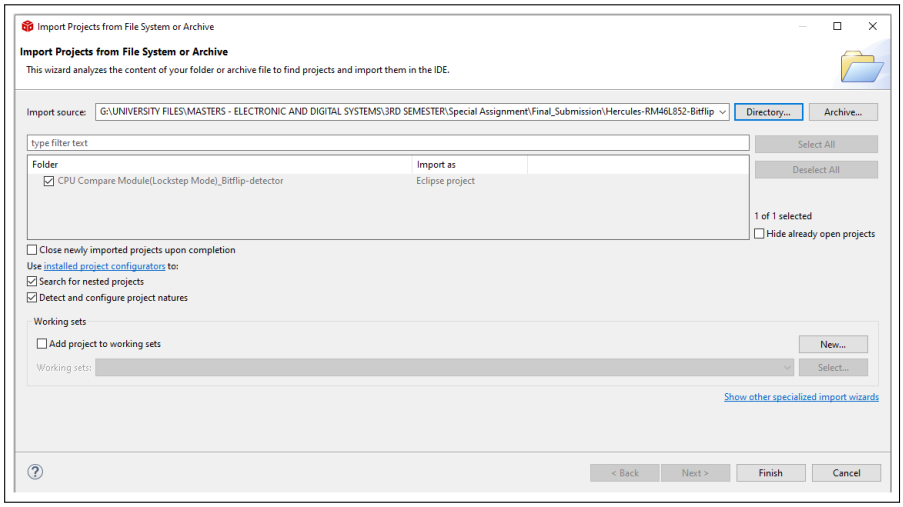


Figure 15: Importing our Lockstep project

* Once the project folder is imported, the users should see the project folder in the project explorer of the CCS(Code Composer Studio).

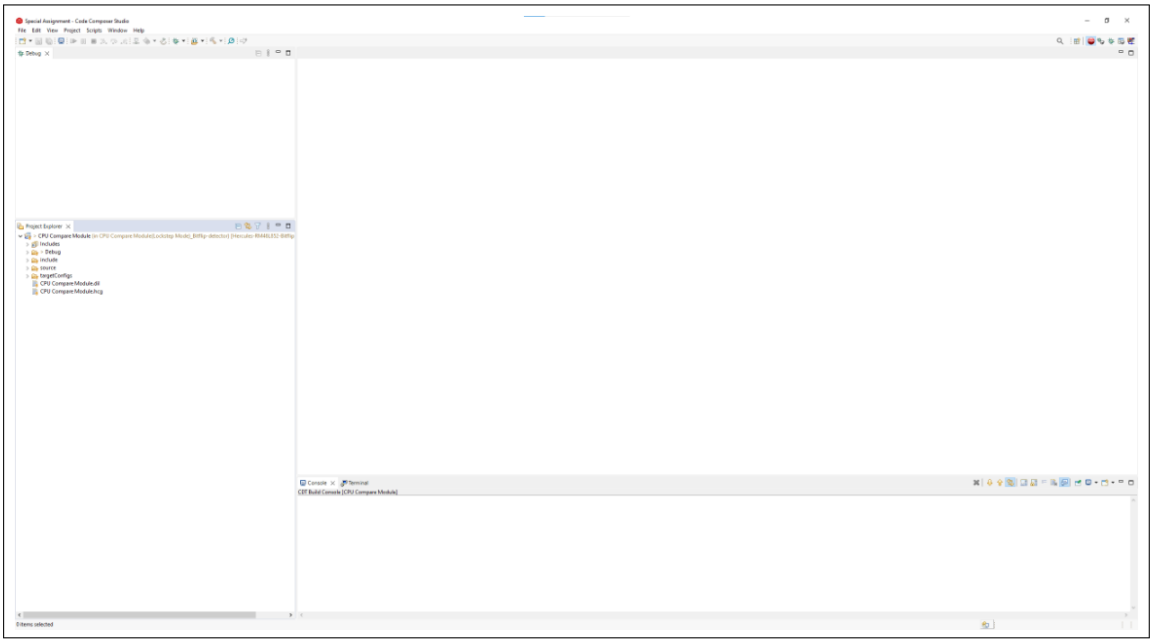


Figure 16: Project explorer

* After that,users should set the properties by right clicking on the project folder in the explorer and selecting Properties.

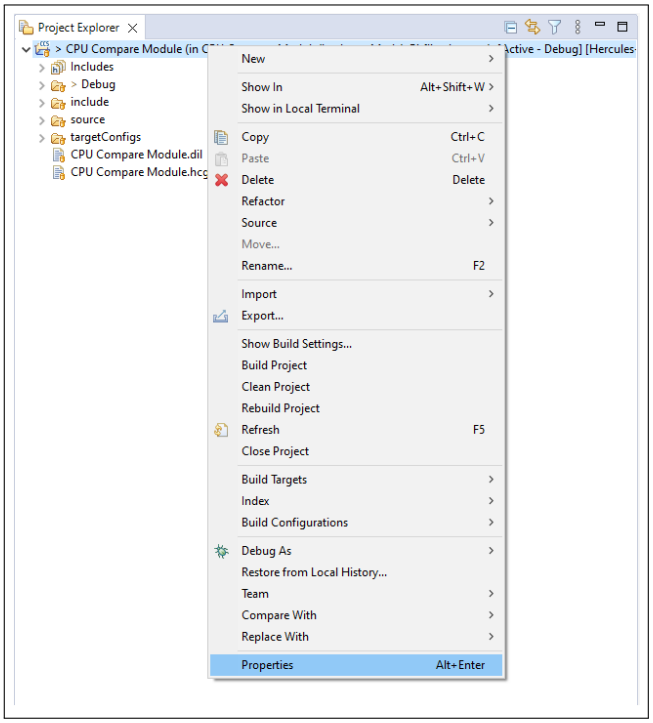


Figure 17: Properties Tab

* In the General properties tab, users should set the Family, Variant, Core, and Connection for the MCU like below picture:

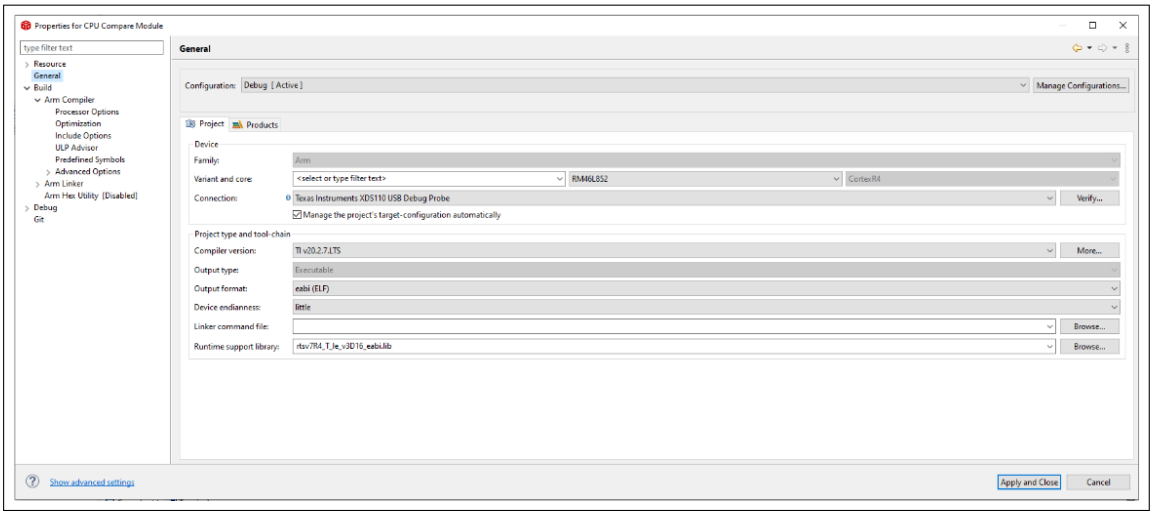


Figure 18: General Properties Tab

* After that, in the same window, user should select `Build > Arm Compiler > Include options` and `click on the plus sign > Workspace > Select include folder` and press OK.This tells the compiler where to look for the header files.

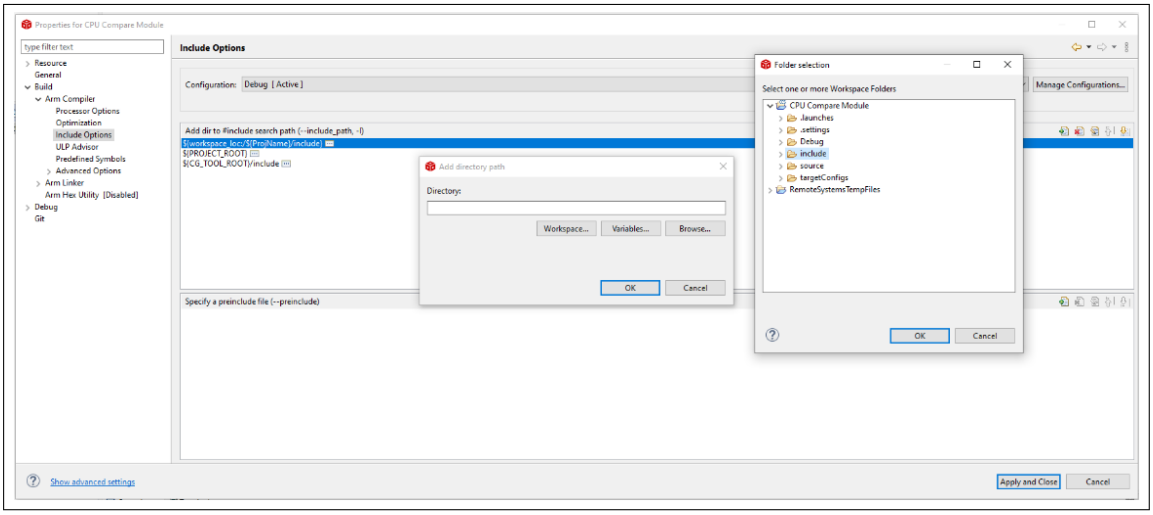


Figure 19: Include header files

* Next,if the user wants to check out the source code, they need to navigate to project explorer where under `source > sys_main.c` , they can find the source code.

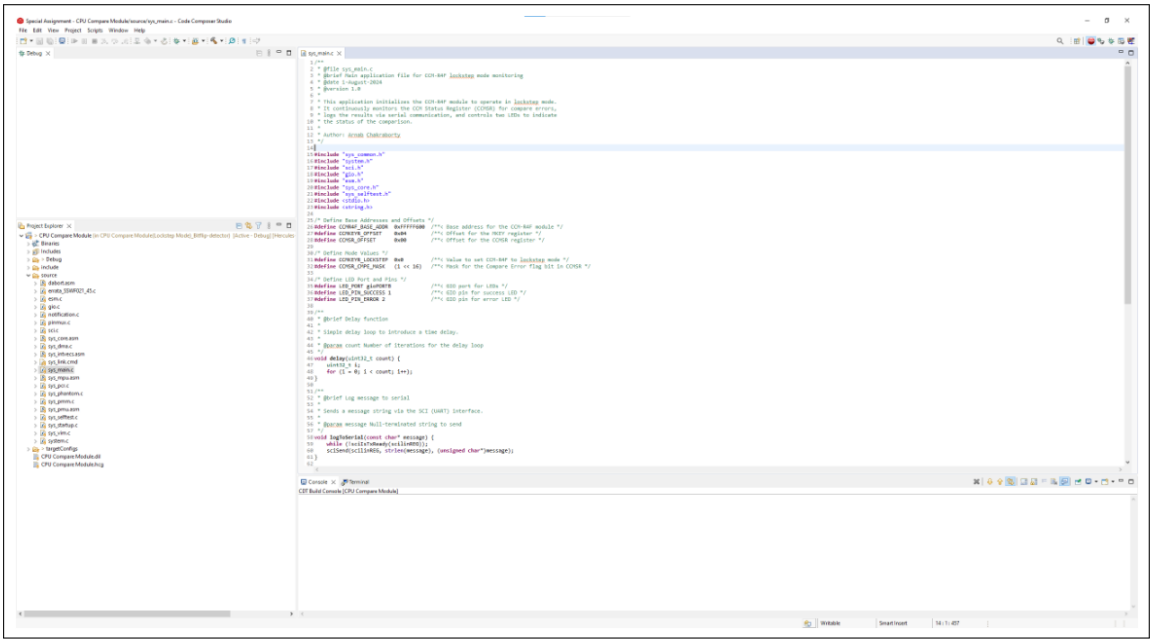


Figure 20: Source code

* If everything seems fine, the user can flash the code to the MCU by pressing the flash button located on the top left side of the CCS like below:

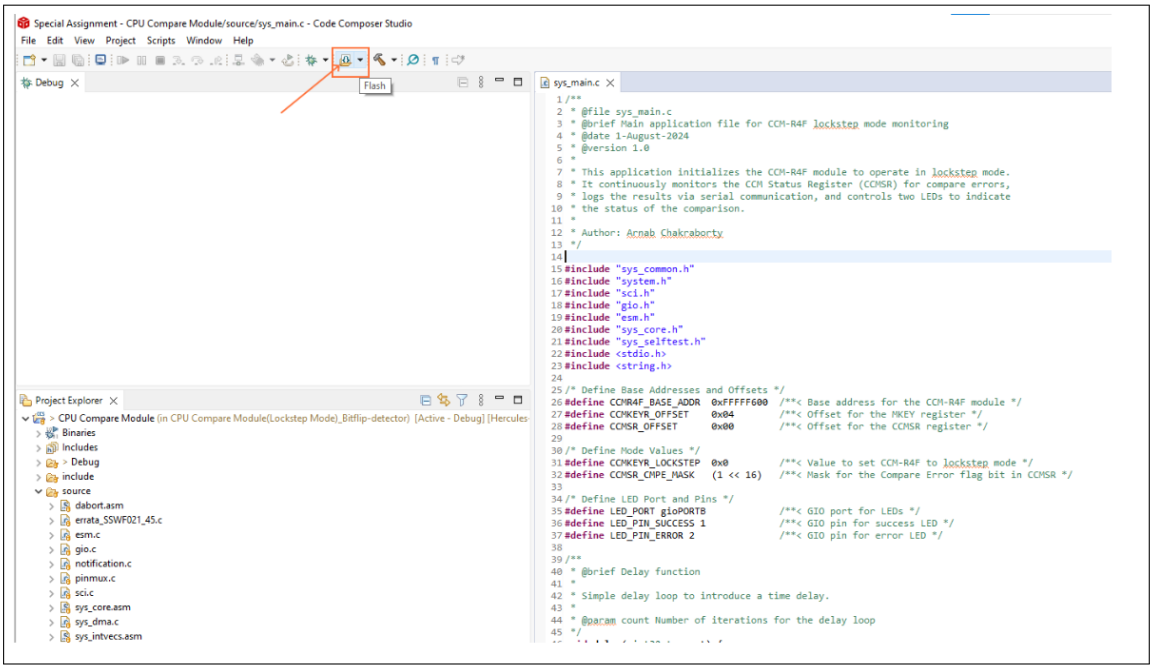


Figure 21: Flash code

* Finally,once the flash is successful, the user needs to press the physical button (labeled as PORRST) on the Hercules microcontroller to start the program

execution.Once the program execution begins,user will see LED2 (GIOB_1) blinking.

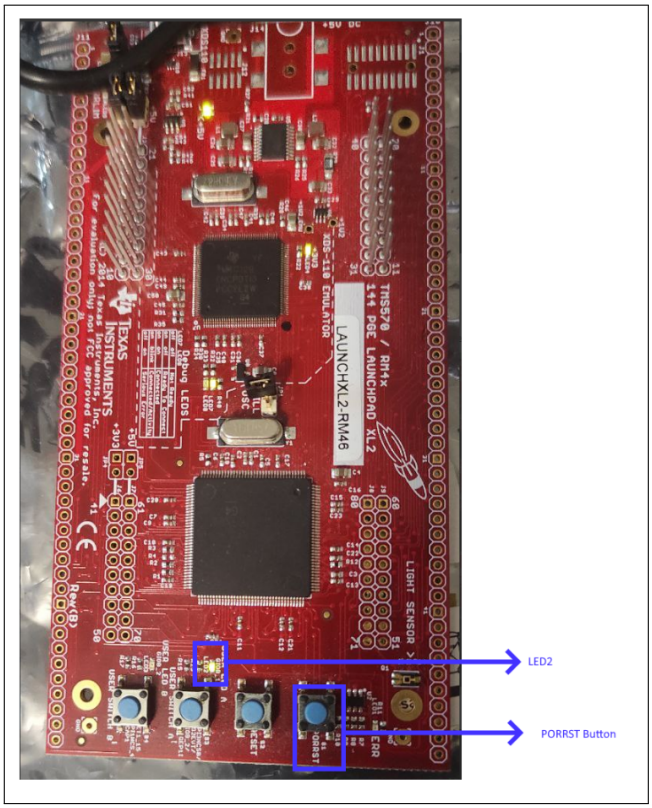


Figure 22: Starting program execution in the MCU

- **RAM**

  - **HALCoGen:**
    For RAM specific application code,the HALCoGen can be used to generate the startup files by following the similar steps that we followed in the above CPU part.However,the only key difference is that we do not need to navigate to the "SAFETY INIT" tab and enable CPU Self test and CCM Self test for RAM configuration.

  - **Code Description:**

```
/* Include Files */
#include "sys_common.h"
#include "sci.h"
#include "gio.h"
#include <stdio.h>
#include <string.h>
```

    Here, we have included the same header files that we did for above CPU section.Details of reasons behind adding these header files have been discussed in the above CPU section.

```
/* Defines the start and size of the memory
   regions */
#define DATA_STORE_START 0x08001600  // Start
   address of data store
#define DATA_STORE_SIZE  0x000174FC  // Each
   region size: 93.25 KB

#define CUSTOM_RAM_START 0x08018AFC  // Start
   address of custom RAM
#define CUSTOM_RAM_SIZE  0x000174FC  // Each
   region size: 93.25 KB and end address: 0
   x0802FFF0
```

In this section of the code, we defined the RAM of the Hercules MCU into two equal sections.We defined the first section as DATA_STORE_START section which in our case will act as storage space for the original data that we want to compare against later.This section serves as a reference to detect and correct bit flips that might occur in the second section of the defined RAM region.We named the second section as CUSTOM_RAM_START where we plan to store the data for testing.Basically, we will fill CUSTOM_RAM_START section of the RAM with data and monitor the bit flips during irradiation tests.By comparing the data in the CUSTOM_RAM_START section with the original data stored in DATA_STORE_START section,we plan to identify and correct the bit flips during irradiation tests.As per the memory map (figure-23) of the Hercules MCU in the datasheet[4], Hercules RM46L852 has a RAM size of 192KB.The start address for RAM is mentioned in the memory map as 0x08000000 and end address as 0x0802FFFF which makes it 192KB.To avoid overwriting data that is already being used by the MCU at the start of the RAM region(0x08000000),we defined our data store section from the memory address 0x08001600.The region starting from 0x08000000 to before 0x08001600 is used by the RAM for system-level data or initialization which we noticed in the memory browser.So, we used the RAM space starting from 0x08001600 to 0x0802FFF0 for our use case which is approximately around 186.5 KB out of 192KB. We divided this region into two equal parts(Each region size: 93.25 KB) and defined DATA_STORE_START section from the memory address 0x08001600 and defined CUSTOM_RAM_START section from the memory address 0x08018AFC.

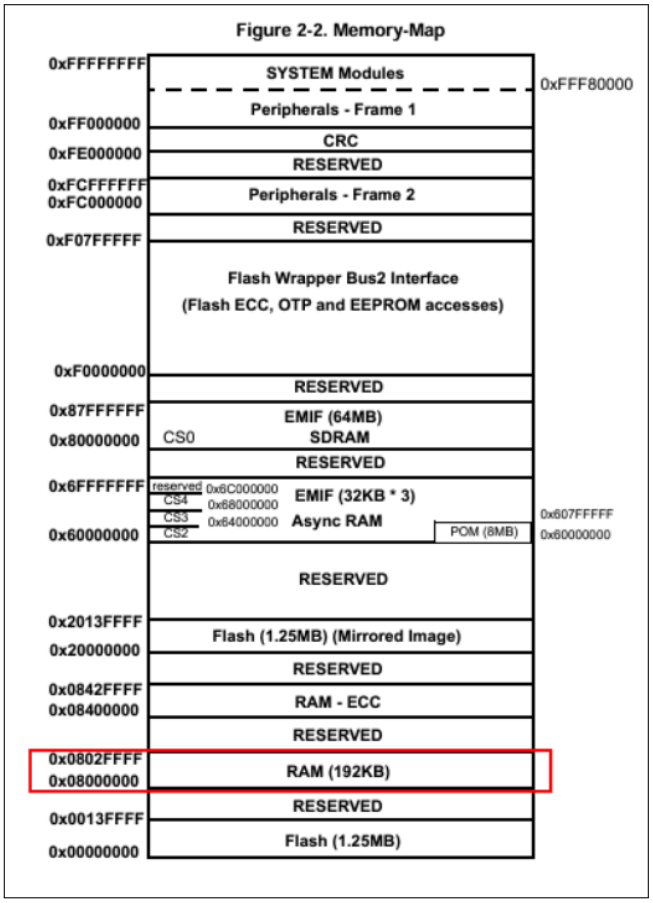


Figure 23: Memory Map(Hercules RM46L852)[4]

```
/* LED port and pin definitions */
#define LED_PORT gioPORTB
#define LED_PIN_SUCCESS 1
#define LED_PIN_ERROR 2
```

Here, I declared again the same GIO port B and their pins(Pin 1 for LED2 and Pin 2 for LED3) as above CPU section for indicating whether a bit flip was detected or not.Details can be read from the above CPU section(Defining LED Port and Pins).

```
/* Memory region declarations */
__attribute__((section(".custom_data_section")))
    uint64_t custom_ram[CUSTOM_RAM_SIZE / sizeof(
    uint64_t)];
__attribute__((section(".data_store_section")))
    uint64_t original_ram[DATA_STORE_SIZE / sizeof
    (uint64_t)];
```

Here, in this section of the code, we are telling the compiler to place the declared array “custom_ram” in the specified memory section named “.custom_data_section” and array “original_ram” in the specified memory section named “.data_store_section” as defined in the Linker script24.Although we defined the starting memory ad-

dresses for both these RAM sections in the beginning of the code,it is essential to define these two sections in the Linker script which can be found under `source>sys_link.cmd`.Linker script is used to map the data into the memory regions of the microcontroller each time we flash the MCU.It helps in ensuring that arrays like "original_ram" which we will use to store the original data that we want to compare against and "custom_ram" which we will use to store data for testing are correctly mapped or placed to their respective specified memory sections during the linking stage.Additionally,we also defined the size of both these arrays by dividing the total RAM available for these sections by the size of each 64-bit data element (i.e., sizeof(uint64_t)).By doing this we get the number of uint64_t elements that can fit into the allocated memory sections.

```
/*---------------------------------------------------------------------------*/
/* Memory Map                                                                */

MEMORY
{
    VECTORS (X)  : origin=0x00000000 length=0x00000020
    FLASH0  (RX) : origin=0x00000020 length=0x0013FFE0
    STACKS  (RW) : origin=0x08000000 length=0x00001500
    RAM     (RW) : origin=0x08001500 length=0x00000100  /*RAM size */
    DATA_STORE (RW) : origin=0x08001600 length=0x000174FC   /* Data Store size */
    CUSTOM_RAM (RW) : origin=0x08018AFC length=0x000174FC  /* CUSTOM RAM size */


/*---------------------------------------------------------------------------*/
/* Section Configuration                                                     */

SECTIONS
{
    .intvecs : {} > VECTORS
    .text    : {} > FLASH0
    .const   : {} > FLASH0
    .cinit   : {} > FLASH0
    .pinit   : {} > FLASH0
    .bss     : {} > RAM
    .data    : {} > RAM
    .sysmem  : {} > RAM

    .data_store : {
        . = ALIGN(8);
        *(.data_store_section)
        . = ALIGN(8);
    } > DATA_STORE

    .custom_data : {
        . = ALIGN(8);
        *(.custom_data_section)
        . = ALIGN(8);
    } > CUSTOM_RAM
```

Figure 24: Linker Script

```
/**
 * @brief Generate PRBS-7 sequence
 *
 * This function generates a pseudo-random binary
    sequence (PRBS) of length 7.
 * It is used to fill the custom RAM with random
    data.
 *
 * @return 7-bit PRBS value
 */
uint8_t prbs7() {
    static uint8_t state = 0x7F; // Initial state
    int newbit = (((state >> 6) ^ (state >> 5)) &
       1); // New bit from the 7th and 6th bits
    state = ((state << 1) | newbit) & 0x7F; //
       Shift left and insert the new bit at LSB,
       mask to keep 7 bits
    return state;
}
```

In this section,we defined a function which can generate pseudo-random binary sequence (PRBS) of length 7.It will be used to generate and fill the defined custom data section of the RAM in the linker script where we want to observe the bit flips. Instead of using a constant pattern (like all 0s or 1s), we are using **PRBS-7** to create a more random bit pattern that is deterministic and predictable while also appearing random over short segments.This randomness is essential for our use case(irradiation tests) because it can be that bits are more likely to flip from 0 to 1 or from 1 to 0 depending on the technology and often multiple neighboring bits are affected.PRBS-7 helps to simulate this realistic scenario and makes it easier to detect and correct errors that might occur due to radiation.Coming to the function definition, in the beginning of the function,we defined a static state which will hold the current state of the PRBS generator and initialized it with 0x7F (binary: 0111 1111).Static type ensures that the state retains its current state during the code execution.In the second statement, we are generating a new bit in the PRBS sequence by first shifting the current state(6 positions to the right to extract the 7th bit and 5 positions to the right to extract the 6th bit).This shift moves the 7th bit and 6th bit to the LSB(Least Significant Bit) position and after shifting,we perform an XOR operation between these two shifted results comparing the isolated 7th and 6th bit.This generates a new bit for the PRBS sequence and we mask this newly generated bit with `&1` to ensure that the newly generated bit is a single-bit value (either 1 or 0).In the third statement,we shift the current state(0x7F) one bit to the left and insert the newly generated bit to the position of the least significant bit (LSB) using Bitwise OR operation.After that we mask it using `&0x7F` to ensure that the result remains within 7 bits.Finally, the function returns the updated state(a new number) which is a 7-bit pseudo-random number and the next value in the PRBS sequence. To visualize we can refer to the below figure[25]:

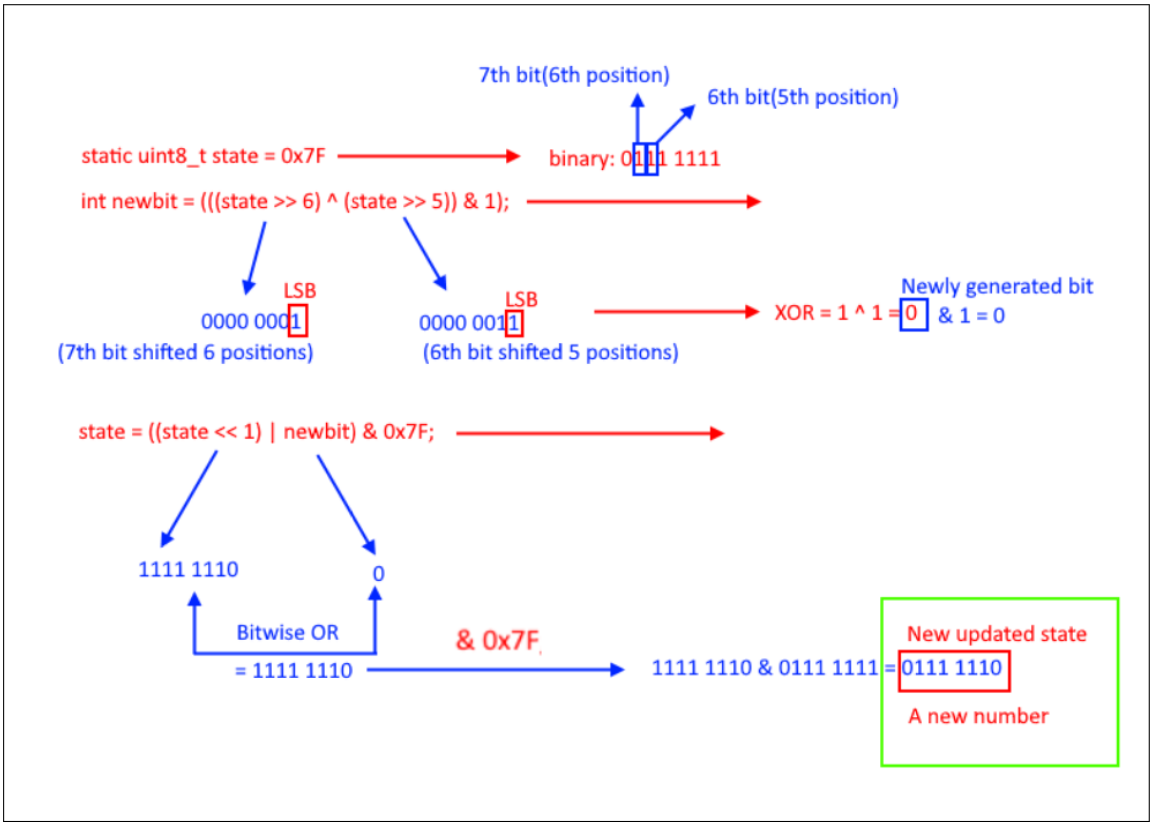


Figure 25: PRBS7

```c
/**
 * @brief Delay function
 *
 * This function provides a simple delay loop.
 * @param count Number of iterations for delay
 */
void delay(uint32_t count) {
    uint32_t i;
    for (i = 0; i < count; i++) {
    }
}

/**
 * @brief Log messages to serial (UART)
 *
 * This function sends a message over UART.
 *
 * @param message The message to be logged
 */
void logToSerial(const char* message) {
    while (!sciIsTxReady(scilinREG));  //
       Waiting until the TX buffer is ready
    sciSend(scilinREG, strlen(message), (
       unsigned char*)message);
}
```

Here,in this section, similarly like the CPU section, we define two functions:Delay function and logToSerial function.Delay function is used to create an empty for loop to create a delay between each while loop iteration.It helps the LEDs to produce a blinking effect and also helps in avoiding flooding the terminal with log messages.On the other hand, logToSerial function helps the MCU to send the event log messages over SCI(Serial Communication Interface) to the connected computer terminal.Details of these functions had already been discussed in the above CPU section.

```
/**
 * @brief Main function
 *
 * This is the entry point of the application. It
    initializes the custom RAM
 * with PRBS-7 data, calculates the expected
    checksum, and continuously
 * verifies the checksum to detect any bit flips.
 *
 * @return (never returns-continously checks for
    bit flips in a while loop)
 */
int main(void) {
    sciInit(); // Initialize the SCI (UART)
    gioInit(); // Initialize GIO
    // Set the direction for LED pins
    gioSetDirection(LED_PORT, (1 <<
       LED_PIN_SUCCESS) | (1 << LED_PIN_ERROR));

    _enable_IRQ(); // Enable interrupts
```

This is the main function and in the beginning of the code,similarly like the CPU section of the report,we initialize the serial communication interface,gio driver required for controlling the LEDs and set the direction of the defined LED pins and mark them as output pins.Finally, we enable the interrupt which is required by the SCI module for sending log messages to terminal.

```
 // Initializing custom RAM with generated PRBS-7
    data and calculating expected checksum
uint64_t i;
for (i = 0; i < CUSTOM_RAM_SIZE / sizeof(uint64_t)
   ; i++) {
    data = 0;
    int j;
    for (j = 0; j < 64; j += 8) {
        prbs_output = prbs7();
        data |= ((uint64_t)prbs_output << j);
    }
    custom_ram[i] = data;
    original_ram[i] = data; // Store original data
    expected_checksum += data;
}
```

In this section of the code, we are filling the defined custom RAM section with the generated PRBS-7 data and calculating the expected checksum because if there is

a bit flip the checksum value will change and differ from the expected checksum indicating that bit flip has occurred in the custom ram section.To fill the custom ram section, we are using a for-loop which iterates through each 64-bit block in the allocated custom ram region.The number of iterations is determined by the number of total 64-bit blocks that can fit into the total size of the custom RAM section.For each 64-bit block, through another inner for-loop, we call the PRBS-7 generator 8 times to generate 8 bits at a time.Then,each 8-bit PRBS value is shifted into the correct byte position of the 64-bit block and after this construction is done, we store this 64 bit data in both custom_ram array and original_ram array(this saves a reference for later comparison and detect and correct bit flips).Finally, while filling it, we also add the value of each 64-bit block continuously to the expected_checksum variable for validation of the data later on.

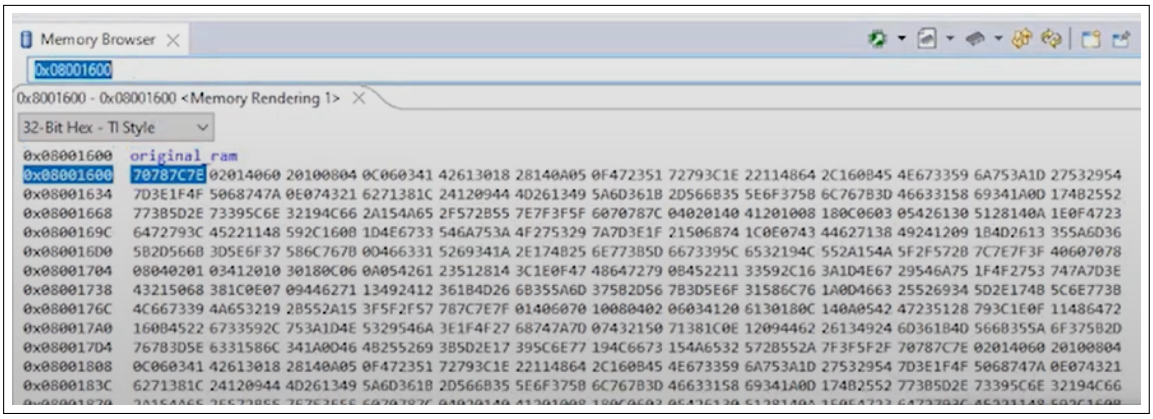


Figure 26: Memory browser verifying successful allocation of PRBS7

```
                    // Infinite loop to verify the checksum and
                       correct errors
                    while (1) {
                        uint64_t calculated_checksum = 0;
                        uint64_t bit_flip_count = 0;

                        // Calculate the checksum and count the
                           number of bit flips
                        for (i = 0; i < CUSTOM_RAM_SIZE / sizeof(
                           uint64_t); i++) {
                            calculated_checksum += custom_ram[i];
                            uint64_t xor_val = custom_ram[i] ^
                               original_ram[i];
                            if (xor_val != 0) {
                                // Correct the bit flip by
                                   restoring the stored original
                                   value from original_ram
                                custom_ram[i] = original_ram[i];
                                // Count the number of bit flips
                                while (xor_val) {
                                                        //Loop
                                   continues as long as xor_val
```

```
                                    is non-zero
                                    bit_flip_count += xor_val & 1;
                                          // Incrementing
                                        bit_flip_count for each
                                        bit flip
                                    xor_val >>= 1; // Right
                                        shifting xor_val to check
                                        the next bit
                                }
                            }
                        }
```

This is the main section of the code where we verify the checksum and detect and correct bit flips in custom_ram section.Inside an infinite while loop, we use a for-loop to continuously iterate over each element in the custom_ram array and compare it to the corresponding value in original_ram section that we stored as reference.For each iteration of the while loop, we do the following:

* We continuously recalculate the checksum using the inner for-loop by summing all the values in the custom_ram array which we use in the later if-block as a condition to check whether this calculated checksum is matching to the expected checksum or not.If it matches, no bit flips had occurred and if it does not match,possible bit flips had occurred.
* Now, to detect the bit flips, inside the inner for-loop, we are continuously performing an XOR operation between the current value in custom_ram[i] and the original value from original_ram[i].The XOR result that we store in the variable xor_val helps us to detect if any bit has flipped or not.If the xor_val value is non-zero, it means there was a bit flip.So, we added another inner if-block which checks whether xor_val value is non-zero or not and if non-zero(bit flipped), we correct the bit flip by restoring the original value by assigning original_ram[i] back to custom_ram[i].
* Now to let the user know, how many bits have flipped, we again use an inner while loop which checks each bit in xor_val to count how many bits have flipped.This is done by performing a bitwise AND operation between xor_val & 1, which checks the least significant bit (LSB) of the xor_val.If the result of xor_val & 1 is 1, it indicates that the LSB was flipped, and we increment the bit flip count(since result of this operation is always either 1 or 0, it increases by 1 when the bit was flipped).After checking the LSB, we right shift xor_val by 1 which moves the next bit into the LSB position and we again do the bitwise AND operation with 1.This process continues until all bits in xor_val have been checked and we get the number of bit flips(if occurred) for that particular memory block.The final count of bit flips is then accumulated and reported to the user if bit flips were detected.

```
                    // Log and indicate the result
                    if (expected_checksum == calculated_checksum
                        && bit_flip_count == 0) {
                        char log_entry[200];
                        snprintf(log_entry, sizeof(log_entry), "\
                            rChecksum matches. No bit flip was
                            detected!\r\n");
                        logToSerial(log_entry);
                        gioSetBit(LED_PORT, LED_PIN_ERROR, 0);
                            // Turning off Error LED
                        gioToggleBit(LED_PORT, LED_PIN_SUCCESS);
                            // Toggling Success LED for creating a
                             blinking effect
                    } else {
                        char log_entry[200];
                        snprintf(log_entry, sizeof(log_entry), "\
                            rChecksum mismatch! %llu bit flips
                            were detected and corrected.\r\n",
                            bit_flip_count);
                        logToSerial(log_entry);
                        gioSetBit(LED_PORT, LED_PIN_SUCCESS, 0);
                              // Turning off Success LED
                        gioToggleBit(LED_PORT, LED_PIN_ERROR);
                            // Toggling Error LED for creating a
                            blinking effect

                    }

                    delay(10000000);  // Delay to avoid flooding
                        the terminal
                }
```

In this last section of the code, we defined the if-else block similarly like the CPU code.In the if block we are continuously checking whether the expected checksum is equal to the calculated checksum and if the bit flip count is zero or not. If there is no bit flip, the expected checksum will be equal to the calculated checksum and bit flip count will be 0 and the if condition will evaluate to true and execute the statements inside the if block.Inside the if block, similarly like CPU code, we are sending the log message to the terminal that no bit flip has occurred and checksum matches along with blinking of the Success LED(LED2).If bit flips are detected, then the code will execute the else block where we are sending checksum mismatches along with the details of the number of bit flips that were detected and corrected as log messages.Alongside,we are also notifying the user about the bit flips by blinking the Error LED(LED3).

- **Usage:** Users can follow the same steps as mentioned in the CPU part above.

- **Logging Script** A logging script was developed using python programming language to continuously listen for the incoming status messages(success or error messages) sent from the Hercules microcontroller via the Serial Communication Interface (SCI).These messages report on bit flips that were detected in the CPU during lockstep mode as well as bit flips detected in RAM when performing irradiation tests.The logging script

captures these event messages from the MCU and displays them to the user in real-time via terminal.Additionally, it also logs the messages with timestamp in a text file which is essential for later analysis.By reviewing the log record file after the irradiation tests have been performed,we can determine how often bit flips occur in both the CPU and RAM which will eventually help us to determine the SEE cross-section of the MCU.

– **Code Description:**

Logging script:

```
import serial
import time

def log_uart_messages(port, baudrate):
    ser = serial.Serial(port, baudrate, timeout=1)
    log_file = open('bitflip_log.txt', 'a')

    try:
        while True:
            if ser.in_waiting:
                message = ser.readline().decode('utf-8
                    ').strip()
                timestamp = time.strftime('%d-%m-%Y %H
                    :%M:%S')
                formatted_message = f"{timestamp} - {
                    message}"
                print(formatted_message)
                log_file.write(f"{formatted_message}\n
                    ")
                log_file.flush()
    except KeyboardInterrupt:
        print("Logging stopped.")
    finally:
        ser.close()
        log_file.close()


if __name__ == "__main__":
    port = 'COM5'
    baudrate = 9600
    log_uart_messages(port, baudrate)
```

In the beginning of the logging script, we imported two python modules named serial and time.Serial module which is imported from pySerial library provides all the functions required for enabling communication with serial devices(in our case reading messages from Hercules MCU) and time module is a built-in module in python which we need for generating timestamps for the log messages in the later part of the code.After importing both the modules, we define a function named log_uart_messages which takes two arguments port(where the MCU is connected) and baudrate(speed of serial communication).The function is responsible for listening to the specified serial port where the MCU is connected for status messages and logging them with a timestamp.Inside the log_uart_messages

function, at first, we create a serial object named ser using the serial.Serial constructor and it is used to connect to the specified port and set the specified baudrate with a timeout of 1s(i.e.the amount of time the function will wait if there is no data for reading).Then, we create or open(if already exists) a log file named bitflip_log.txt in append mode ('a') which we will use to store the incoming log messages from the MCU without replacing the previous log messages.Next, we declare try-except-finally block.Inside the try block, we start an infinite while loop which will be continuously listening to the incoming messages from the MCU.The loop can only be stopped using KeyboardInterrupt(CTRL+C in Linux) which is handled in the except block.Inside the while loop, we placed an if condition(if ser.in_waiting) which basically checks if there is any messages or data that is waiting to be read from the specified serial port.If there is no data or messages, the loop continues waiting and when a message or data arrives, it proceeds to the next step.In the next step, through ser.readline().decode('utf-8').strip(), we are reading a full line of incoming data or message from the serial port and we are decoding the messages from bytes into a string using UTF-8 encoding and finally using the strip function to remove any leading/trailing whitespace from the message.After decoding the message, we store the message in the message variable.Next, we used time.strftime() function which generates the current date and time in the format day-month-year hour:minute:second and we store that in the variable named timestamp.After that,we format the message in a way where the timestamp appears first and then alongside it displays the received message together in one line.Then, we print the formatted message using the print function which prints the messages with timestamp in the terminal from where the user can monitor the status in real-time.Finally,we write the formatted message in the log file bitflip_log.txt, followed by a newline character so that each message starts on a new line.Additionally, we also use the flush() function so that the log message is written to the log file immediately rather than being buffered.This helps in ensuring that the log message does not get lost if the script is interrupted by the user at any point of time.In the except block, if a keyboard interrupt is observed, it prints a message in the terminal notifying the user that the logging has stopped.In the finally block,regardless of how the try block terminates (whether by normal execution or by an exception),we ensure that the serial connection and the log file are properly closed to prevent any sort of data loss.Coming to the main block, we specify the port(in our case it was COM5) and baudrate(it needs to be 9600 since we configured it in the MCU) and finally pass these two arguments to the log_uart_messages function to start the listening and logging process.

– **Usage:**

For using the logging python script for observing the messages sent from the Hercules MCU, we need to follow the below steps:

* In the beginning, user needs to make sure python is installed on the system where they plan to run the script.Secondly, the user needs to install the serial module package since our script uses serial module package.

* Before executing the script, user should check the COM port on which the Hercules microcontroller is connected via the USB cable and change it accordingly.For us, it was COM5 and it can be different for other users.
* Then the user should download an IDE or run the script directly using the command line.For our case, we used Pycharm(IDE) to run the script.For command line, one can run the script using:

```
python main.py
```

* Once the script is running, user can see the messages sent from the MCU in the terminal in real-time.
* To stop the script, in an IDE like Pycharm one can directly press the stop button or in linux command line, one can press `CTRL + C` in the terminal.This will terminate the serial connection and stop the logging process.
* Once the script is terminated, users can find all the logged messages along with timestamp in the text file named "bitflip_log.txt" in the same folder where the script is located.

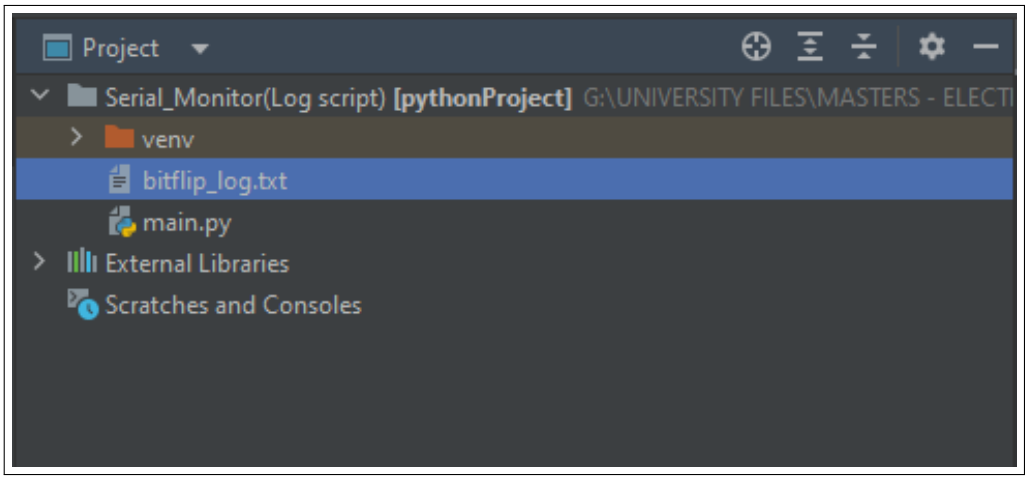


Figure 27: Log file with all the log messages

3 Testing & Results

For testing our developed firmware on the MCU, we followed the below process: **Youtube Link**

- **Testing firmware developed for detecting bit flips in CPU Lock step mode:**
  - No bit flips: We flashed our firmware to the Hercules MCU and in normal conditions we observed that the assigned LED2(Sucess LED) is blinking correctly indicating that there were no signal mismatches between the Master CPU and Checker CPU i.e. no possible bit flips.

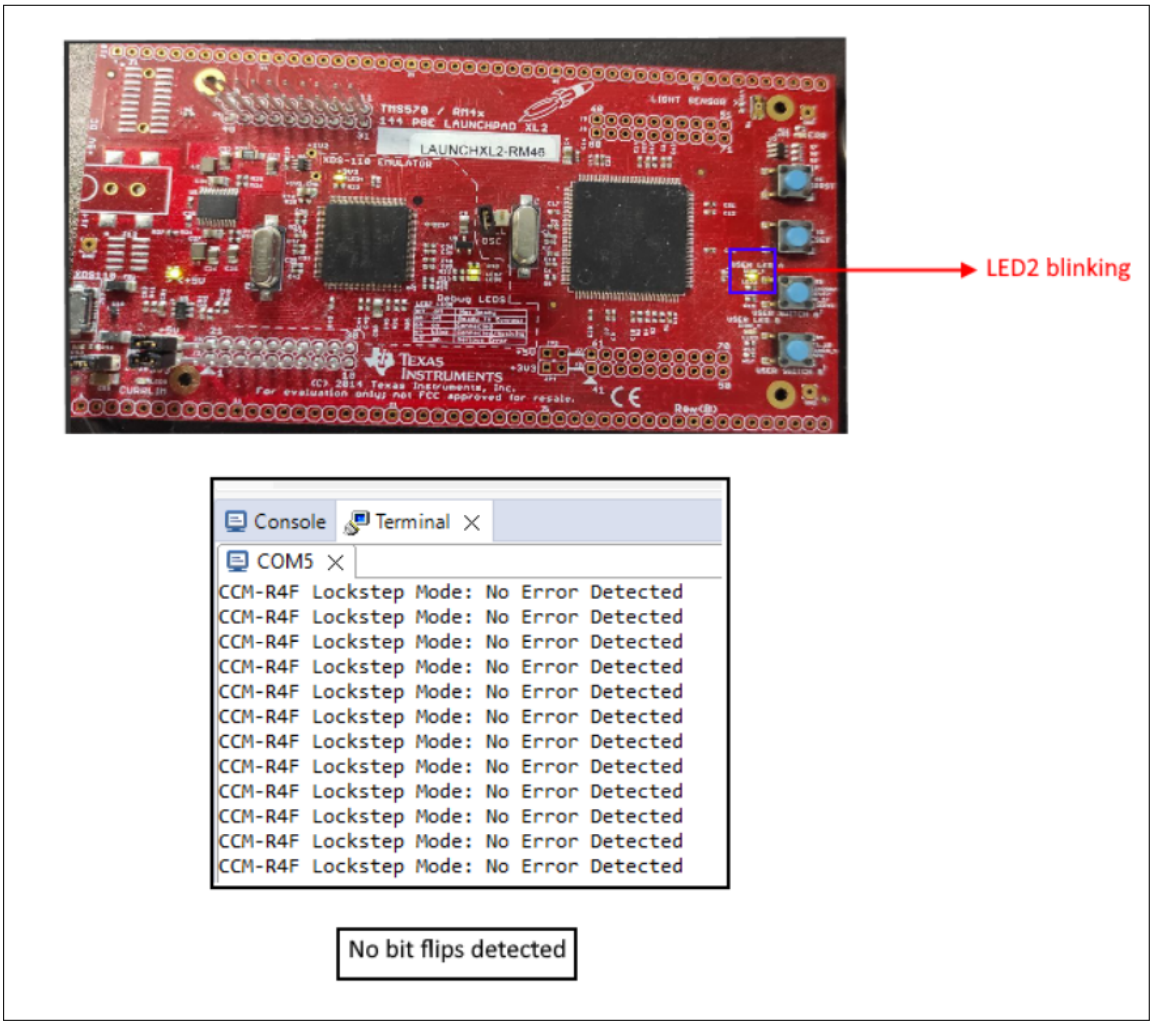


Figure 28: No bit flips

  - Simulated Bit flips detected: In this phase of the development,since there was no opportunity to test our firmware in a particle accelerator and there is no way to cause an intentional signal mismatch between both the CPU cores or simulate a bit flip, we decided to test the bit flip detection part of the code by modifying the if-block condition.Instead of performing the usual bitwise AND operation between

the CCM Status Register (CCMSR) and CCMSR_CMPE_MASK to detect signal mismatches,we changed it temporarily to bitwise OR operation (as shown in figure 29) which will make the if-block always result to true(1) since the CMPE bit in the CCMSR is always zero in normal conditions when no bit flips occur.This was the only way to test if the assigned LED3(Error LED) is blinking correctly or not and if the MCU is publishing the error messages in the terminal correctly or not.Upon testing,we noticed that the error LED3 was blinking as intended and the MCU was correctly transmitting the error messages to the terminal.This proves that the firmware we developed will function as expected during actual irradiation tests in the particle accelerator and respond appropriately when actual bit flips occur.

```
if (*((volatile uint32_t *)(CCMR4F_BASE_ADDR + CCMSR_OFFSET)) | CCMSR_CMPE_MASK) {
    logToSerial("\rCCM-R4F Lockstep Mode: Error Detected!\r\n");
    gioSetBit(LED_PORT, LED_PIN_SUCCESS, 0);     /**< Turn off success LED */
    gioToggleBit(LED_PORT, LED_PIN_ERROR);       /**< Toggle error LED */
```

Figure 29: Bitwise OR for testing purpose

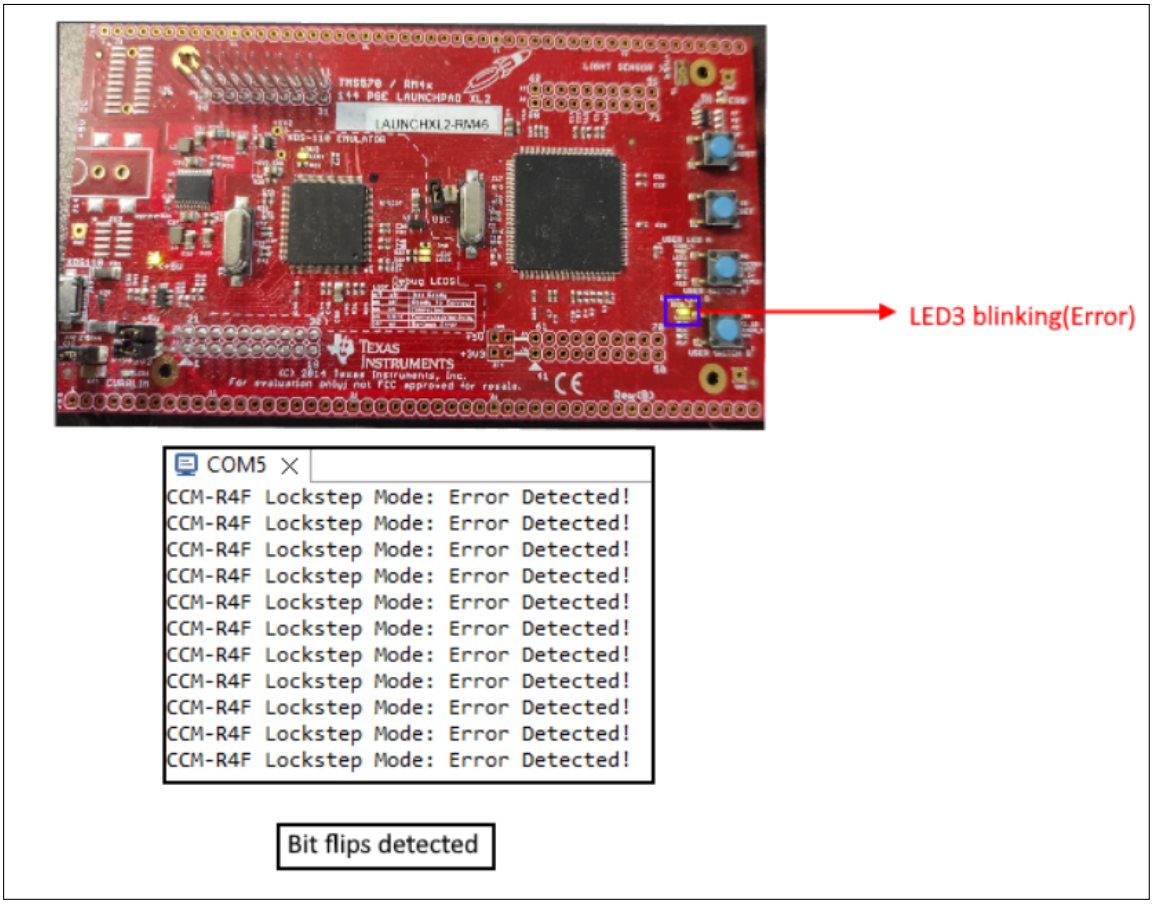


Figure 30: Simulating bit flips condition in CPU

- **Testing firmware developed for detecting bit flips in RAM:**
  - No bit flips: We flashed our firmware to the MCU developed for detecting bit flips in the custom RAM region.In normal conditions, we observe that the assigned LED2 is blinking as intended indicating that there were no bit flips detected in the specified RAM section of the memory.

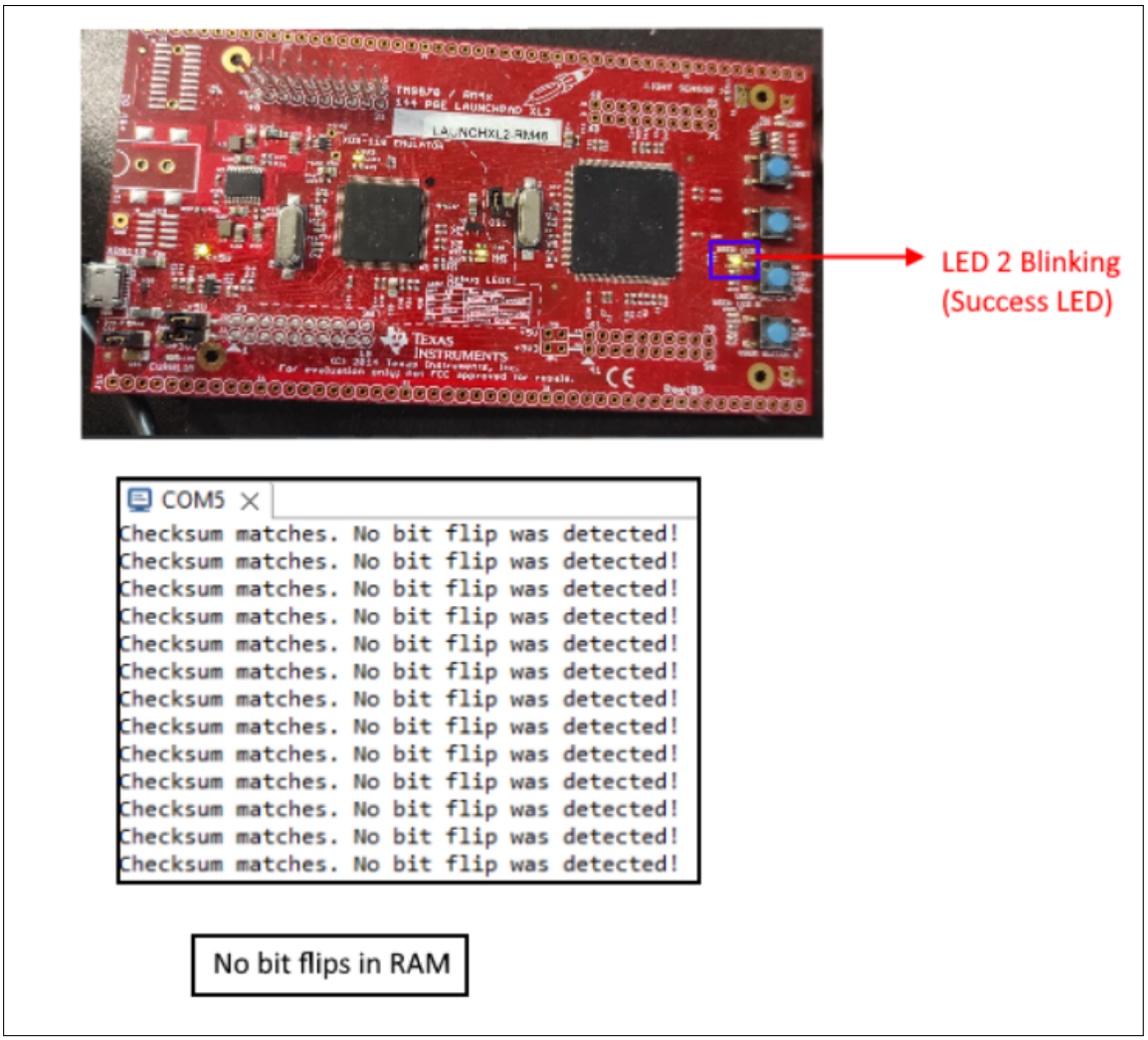


Figure 31: No bit flips in RAM

  - Simulated Bit flips detected: Since we could not test our firmware in a particle accelerator, we decided to simulate bit flips manually in the custom RAM region.We did this by modifying specific bits in the custom RAM array using bitwise XOR operations.For our case, we manually flipped the least significant bit and second least significant bit of the 11th element in the custom ram array intentionally which had been filled with pseudo-random data using the prbs7() function.This allowed us to test whether our firmware is able to detect,correct and log the bit flips correctly or not,toggle the error LED when such bit flips are detected and finally send the appropriate error messages to the terminal with details about

the number of bit flips detected and corrected.This testing confirmed that our firmware is working as expected and is able to detect bit flips when in a particle accelerator.

```
// Simulate bit flips for testing
custom_ram[10] ^= 0x1;     // Flip the least significant bit of the 11th element
custom_ram[10] ^= 0x2;     // Flip the second least significant bit of the 11th element
```

Figure 32: Simulating bit flips using bitwise XOR operation

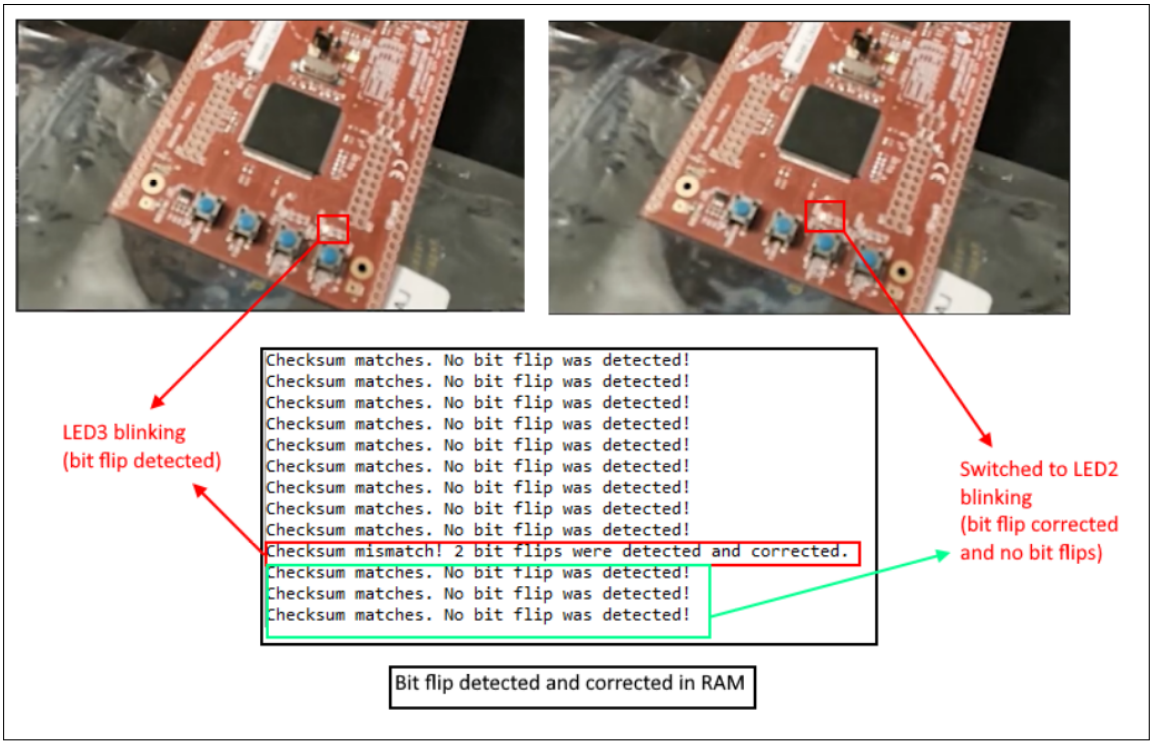


Figure 33: Bit flips in RAM

- **Testing the developed logging python script:**

  The logging script seemed to be working as expected and it was logging all the bit flips and no error messages with correct timestamp in the specified text file.

  – For CPU Lockstep mode:

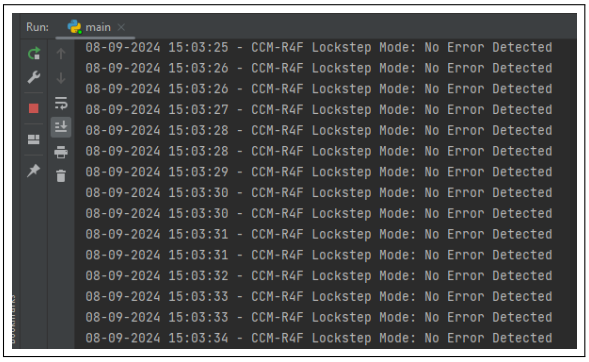


Figure 34: Logging no bit flips for CPU

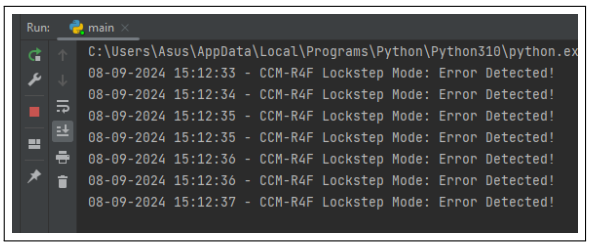


Figure 35: Logging bit flips for CPU

– For RAM:

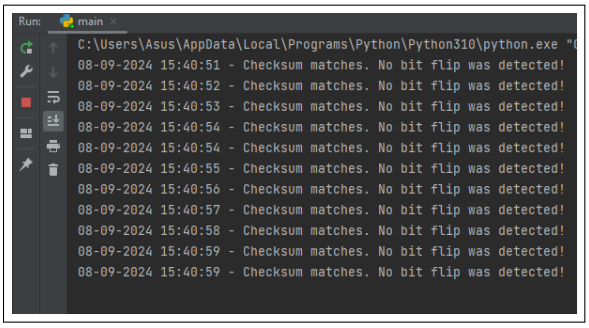


Figure 36: Logging no bit flips for RAM

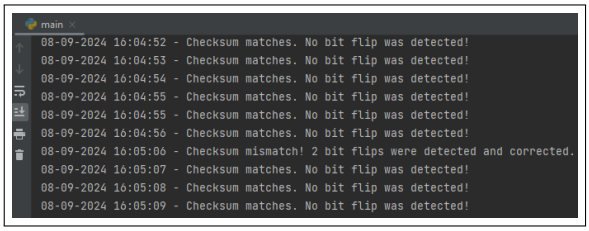


Figure 37: Logging number of bit flips for RAM

Log file:

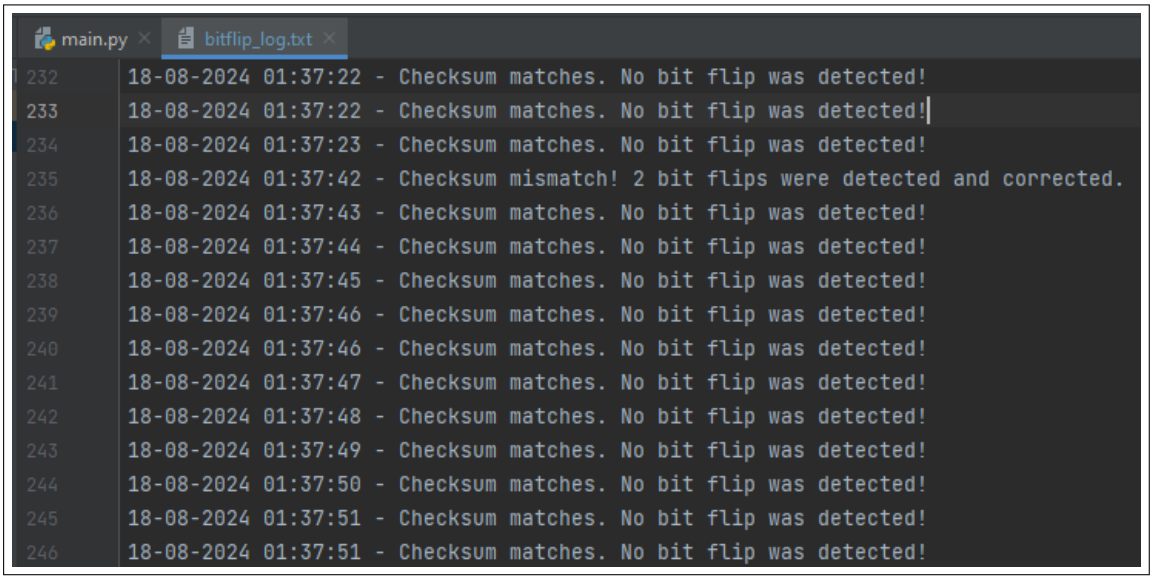


Figure 38: Log records

4 Future improvements & Conclusion

For future improvements, following enhancements can be taken into account:

- Currently, we are dividing the RAM region in two sections and using one section for testing and another section for storing the data which is being used for comparison purpose and we are using the store section to correct the bit flips that might occur in the testing section.It also helps us to count the number of bits that flipped i.e. it can detect and count multi-bit errors.However,in future,we can modify our firmware to use ECC(Error Correcting Code) which is a built-in feature for Hercules MCU to detect and correct single-bit errors automatically.Using this will help us to fill the whole RAM with PRBS7 data instead of dividing the RAM region into two sections.This approach would allow us to detect bit flips over a larger memory region which will increase the chances of identifying more bit flips and thereby providing a more accurate SEU cross-section for the Hercules MCU.

  How ECC works in Hercules MCU[7]:
  ECC works differently for RAM and Flash in the Hercules MCU[7].Although the reference is about TMSx70-Based microcontroller, it should be similar for Hercules MCU.For RAM, ECC is automatically calculated by hardware whenever a write operation takes place.Here, for every 64-bit data word,8 ECC bits are generated and stored in a separate ECC region.During the read operations,the stored ECC bits are read along with the data and if there are any errors detected, ECC bits are used to correct it.On the other hand, for Flash memory, ECC must be generated manually and programmed into the ECC space.Here also,each 64-bit data word consists of 8 ECC bits and these ECC bits are used during read operations to detect and correct any errors by comparing the stored ECC bits with those generated during the read operation.

  How we can enable ECC in HALCoGen,CCS and through Linker script(Not tested): To enable ECC in RAM and Flash:

  - For HALCoGen, we enable ECC for RAM in RAM tab and for Flash in Flash tab.

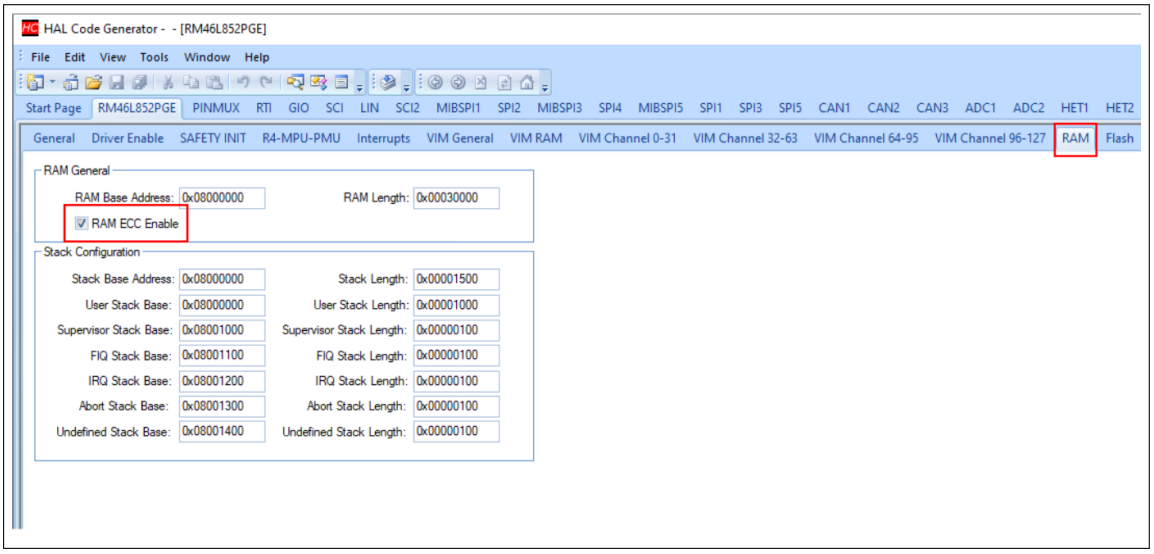


Figure 39: RAM ECC

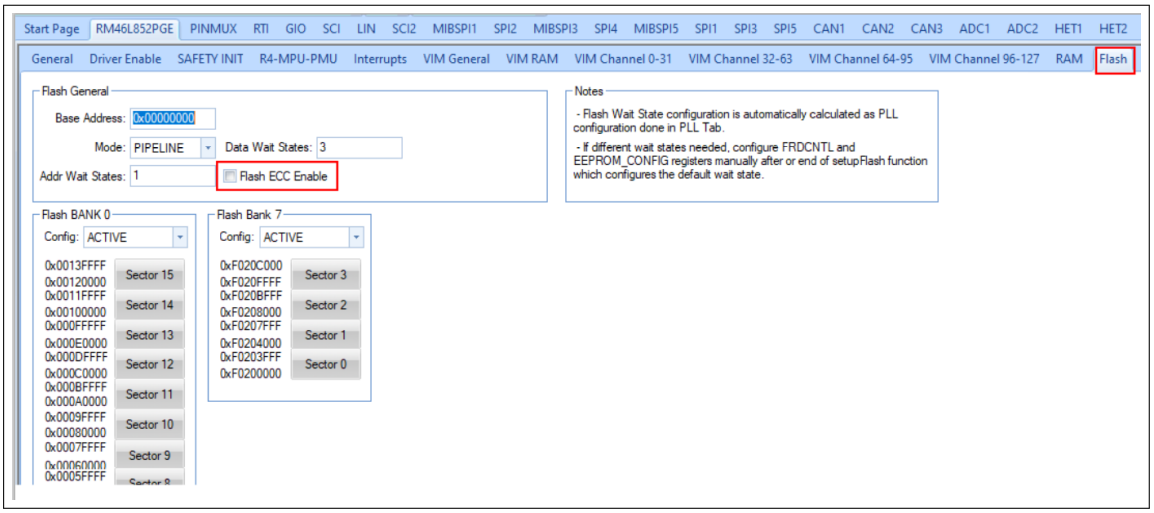


Figure 40: FLASH ECC

– Now for ECC generation(FLASH), either we can use CCS(Code Composer Studio) or Linker Script to generate the ECC.But we cannot use both at the same time.However, most likely you still need to define the ECC section in the Linker script even for CCS method.To read more about it, we can follow the following reference[8].

  * For CCS: Right click on the project folder in the project explorer and navigate to properties.Then turn on ECC generation as shown below[9]:

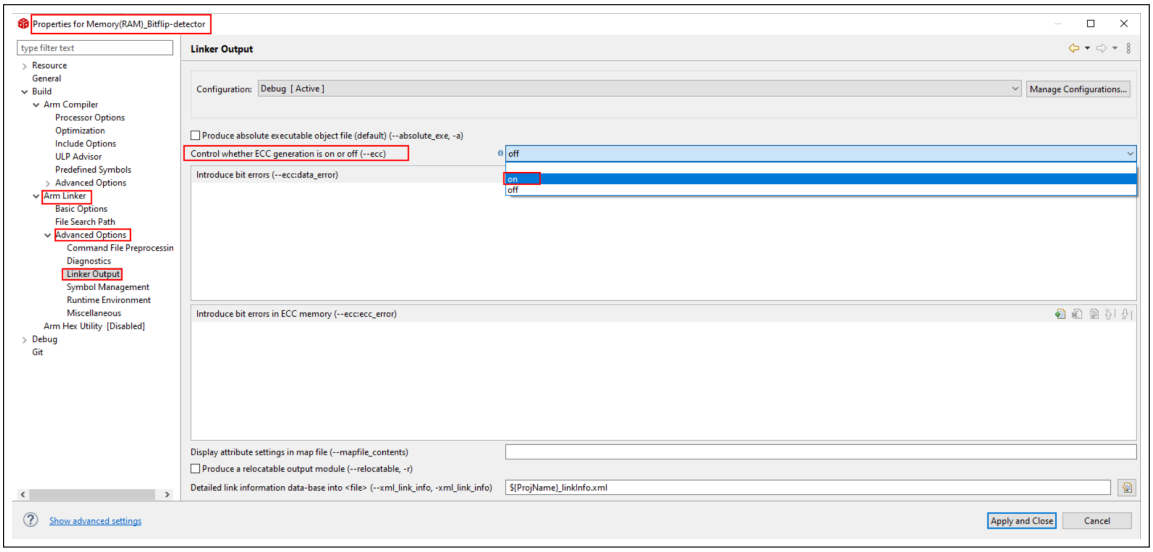


Figure 41: Enabling ECC in CCS

∗ For Linker script(`source>sys_link.cmd`):

```
/* USER CODE END */
    VECTORS (X)  : origin=0x00000000 length=0x00000020
    FLASH0  (RX) : origin=0x00000020 length=0x001FFFE0
    FLASH1  (RX) : origin=0x00200000 length=0x00200000
    STACKS  (RW) : origin=0x08000000 length=0x00001500
    RAM     (RW) : origin=0x08001500 length=0x0007eb00

/* USER CODE BEGIN (3) */
#endif
    VECTORS (X)  : origin=0x00000000 length=0x00000020 vfill = 0xffffffff
    FLASH0  (RX) : origin=0x00000020 length=0x001FFFE0 vfill = 0xffffffff
    FLASH1  (RX) : origin=0x00200000 length=0x00200000 vfill = 0xffffffff
    STACKS  (RW) : origin=0x08000000 length=0x00001500
    RAM     (RW) : origin=0x08001500 length=0x0007EB00

    ECC_VEC (R)  : origin=0xf0400000 length=0x4 ECC={ input_range=VECTORS }
    ECC_FLA0 (R) : origin=0xf0400000 + 0x4 length=0x3FFFC ECC={ input_range=FLASH0 }
    ECC_FLA1 (R) : origin=0xf0440000 length=0x40000 ECC={ input_range=FLASH1 }
/* USER CODE END */
}


/* USER CODE BEGIN (4) */
ECC
{
   algo_name : address_mask = 0xfffffff8
   hamming_mask = R4
   parity_mask = 0x0c
   mirroring = F021
}/* USER CODE END */

/*-----------------------------------------
```

Figure 42: Linkerscript ECC[9]

However, while ECC is beneficial, it has its limitations.ECC can detect and correct only

single-bit errors in Hercules MCU but if there are multi-bit errors,it can only detect them and is not able to correct them.This means that for our use case, where we plan to irradiate the chip in a particle accelerator, if multi-bit errors occur, the firmware using ECC will detect the multi-bit error but will not be able to correct it.This will lead the MCU to send error detected messages continuously as ECC will keep detecting the same bit flip and we will not be able to calculate or get an accurate result for SEU cross-section which is our ultimate goal.On the other hand, our current approach can easily handle such multi-bit errors and correct them without any issue since it is not relying on ECC.Even if bit flips occur in the store section of the RAM, we will still be able to know there was a bit flip since it will cause a mismatch between both the section of the RAM.

- Through this project,we are only checking bit flips in CPU and RAM.However,in future improvements, we can also modify our developed firmware to check for bit flips in other memory regions and flash memory.We tried to fill the flash memory with PRBS7 data but as it was write-protected, the MCU did not allow us to write to it.In future, maybe we can use the Linker script like discussed in above section to disable this write protection and enable ECC(either through Linker Script, CCS or nowECC tool[10] along with nowFlash(not supported anymore and replaced with UniFlash) from Texas Instruments) to detect and correct bit flips in the Flash memory.

5 References

[1] A. Menon, "What are bit flips and how are spacecraft protected from them?" *Science ABC*, Mar. 2024, Accessed: August 24, 2024. [Online]. Available: https://www.scienceabc.com/innovation/what-are-bit-flips-and-how-are-spacecraft-protected-from-them.html.

[2] T. Instruments, *Launchxl2-rm46 hercules rm46x launchpad development kit*, Accessed: August 25, 2024. [Online]. Available: https://www.ti.com/tool/LAUNCHXL2-RM46.

[3] Guatelli, *What is cross section? (correlation between incident neutron and proton for single event upset (seu))*, Accessed: August 25, 2024. [Online]. Available: https://geant4-forum.web.cern.ch/t/what-is-cross-section-correlation-between-incident-neutron-and-proton-for-single-event-upset-seu/7617.

[4] T. I. Incorporated, *Rm46x 16/32-bit risc flash microcontroller*, Accessed: August 31, 2024. [Online]. Available: https://www.ti.com/lit/ug/spnu514c/spnu514c.pdf.

[5] T. I. Incorporated, *Halcogen*, Accessed: August 31, 2024. [Online]. Available: https://software-dl.ti.com/hercules/hercules_docs/latest/hercules/Halcogen/HalCoGen.html.

[6] j. g. Texas Instruments Incorporated, *If cpu compare error (lockstep) is detected during runtime, what needs to be done*, Accessed: Sep 07, 2024. [Online]. Available: https://e2e.ti.com/support/microcontrollers/arm-based-microcontrollers-group/arm-based-microcontrollers/f/arm-based-microcontrollers-forum/1288289/tms570lc4357-if-cpu-compare-error-lockstep-is-detected-during-runtime-what-needs-to-be-done-is-there-any-recommended-fault-reaction-like-mcu-reset.

[7] T. I. Incorporated, *Ecc handling in tmsx70-based microcontrollers*, Accessed: Sep 08, 2024. [Online]. Available: https://www.ti.com/lit/pdf/spna126.

[8] T. I. Incorporated, *Linker generated ecc*, Accessed: Sep 08, 2024. [Online]. Available: https://software-dl.ti.com/hercules/hercules_docs/latest/hercules/How_to_Guides/HowToGuides.html#id1.

[9] J. Cumps, *Hercules microcontroller: Correctly create and load firmware with error checking and correction (tms570lc43 and rm57 specific)*, Accessed: Sep 08, 2024. [Online]. Available: https://community.element14.com/technologies/automotive/b/blog/posts/hercules-microcontroller-correctly-create-and-load-firmware-with-error-checking-and-correction-tms570lc43-and-rm57-specific.

[10] T. I. Incorporated, *Nowecc generation tool version 2.22*, Accessed: Sep 08, 2024. [Online]. Available: http://www.ti.com/lit/pdf/SPNU491.