Getting Started with X-NUCLEO-SPINAND-TOSH

Based on TC58CVG2S0HRAIF
Revision 1.0, 10.04.2017

Quick Start Guide

Revision History

Revision, Date	Editor	Subject (major changes)
Revision 1.0, 10.04.2017	Quang Hai Nguyen	Initial release

Table of Contents

List of Figures

Introduction

This document provides the information about the hardware and software requirements for running the demo. It also guides the user how to run the X-NUCLEO-SPINAND-TOSH demo and how to create one from scratch.

Folder structure

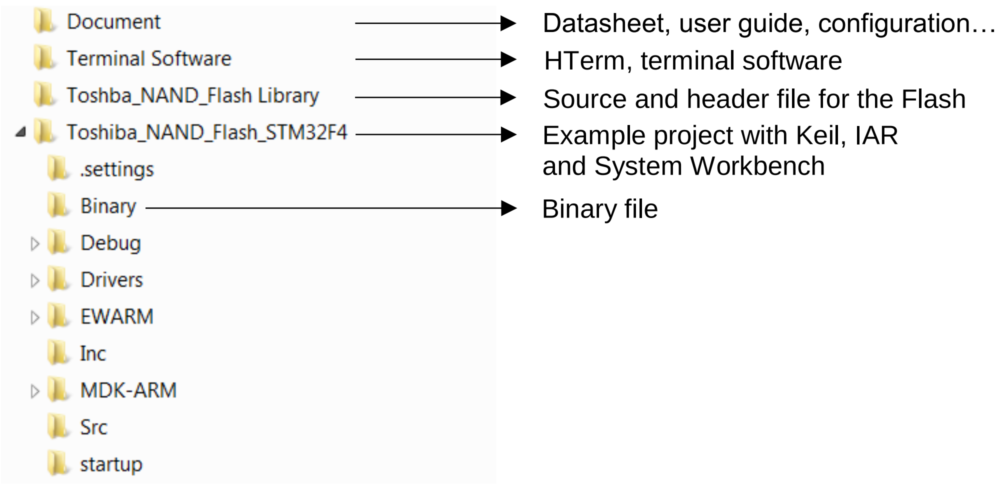


Figure 1: Folder structure

Requirements for the demo

Running the demo

- STM32F401 Nucleo kit
- X-NUCLEO-SPINAND-TOSH
- Mini USB cable
- STM32 – ST LINK Utility
- Terminal Program (HTerm, TeraTerm, Putty…)

Creating the demo from scratch

- STM32CubeMX
- Preferred IDE (Keil, IAR, System Workbench)

Running the demo

Flashing the board with STM32 – ST LINK Utility

Plug the X_NUCLEO onto the STM32F4 Nucleo and power it with the mini USB cable

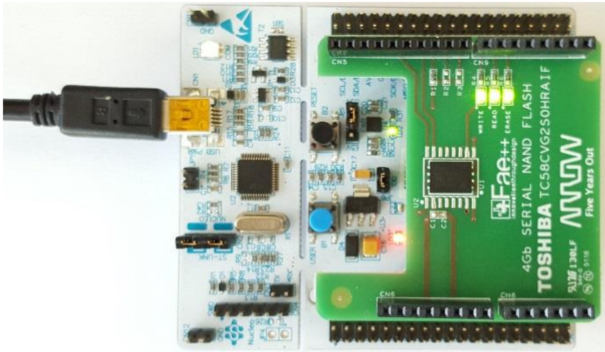


Figure 2: Hardware for running the demo

Open STM32 ST-Link Utility and connect it to the board by pressing the connect button or go to **Connect → Target**.

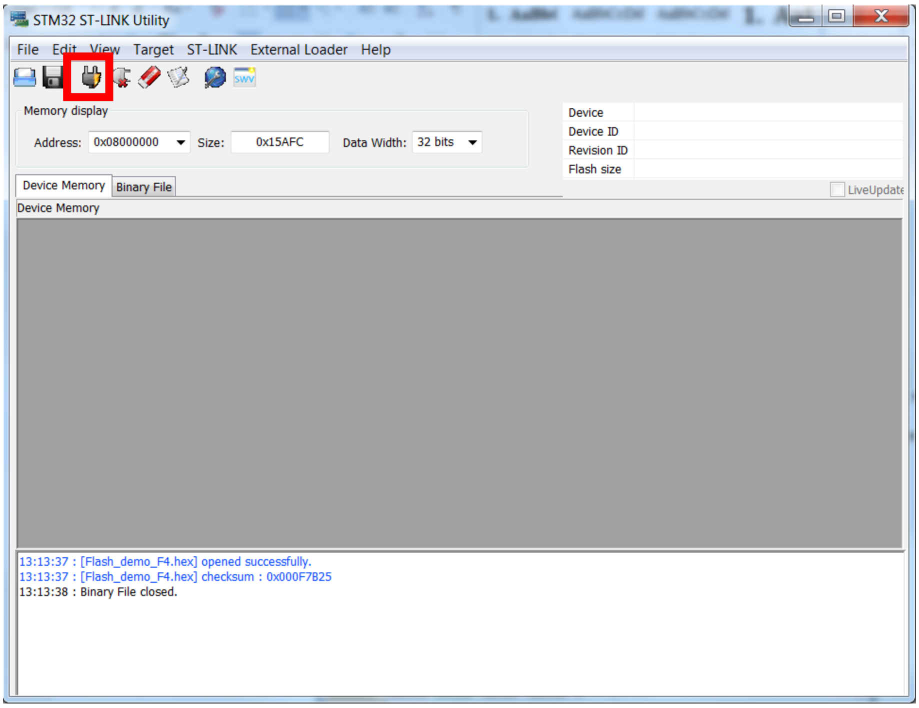


Figure 3: Connect the board to STM32 ST-LINK Utility

Open the binary file by clicking **File → Open File**… then navigate to the folder which the binary file is stored (Figure 1) or simply drag and drop the file into the program.

Binary find can be found in the Binary folder (Figure 1)

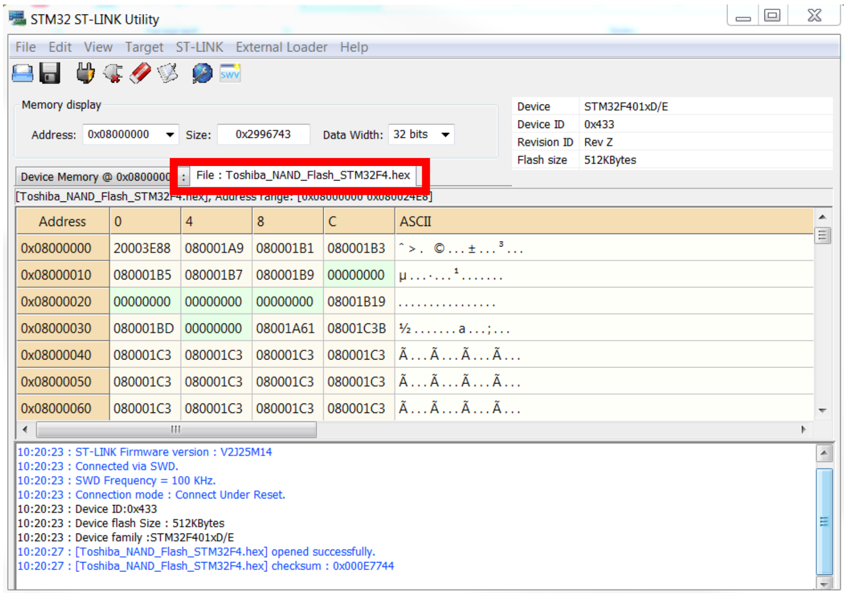


Figure 4: Loading the binary into the STM32 ST-LINK Utility

Finally the board can be programmed by clicking **Target → Program & Verify → Start** or clicking on the Program Verify button, then press start.

When the program is finished, the console will show the complete message as below and the board is ready.

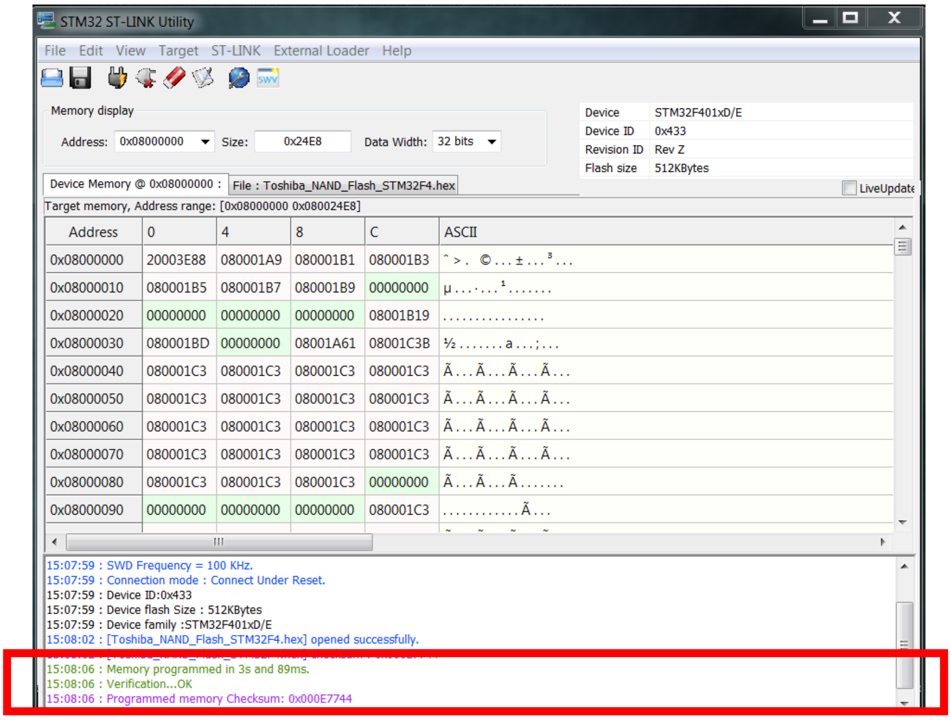


Figure 5: Programming the board with STM32 ST-LINK Utility

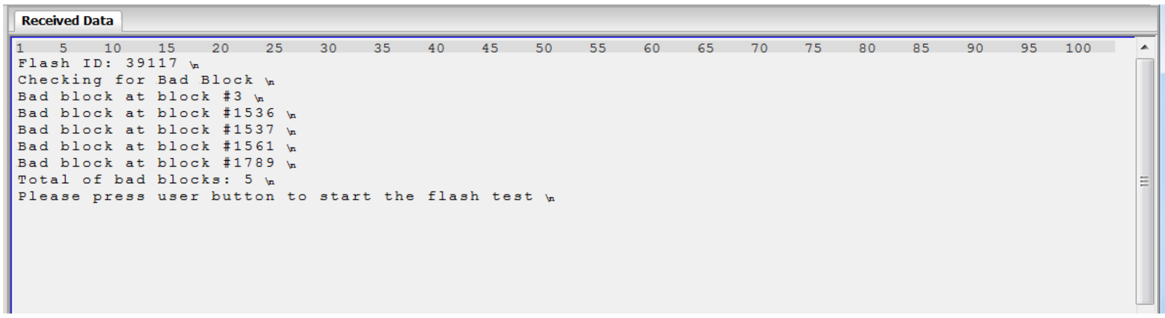
Running the Demo

Open the preferred Terminal program. HTerm is used in this document and can be found in the Terminal Software folder. Configure it as following (*Note that the port is different on your computer*):

Figure 6: Terminal Program configuration

By pressing **Connect** button on HTerm and the **reset button** on the controller (black one), there will be a message displayed to show the address of the flash, also running the process of checking bad blocks. Please note that, every memory is shipped with some bad blocks, and the bad blocks are different from one memory to the others.

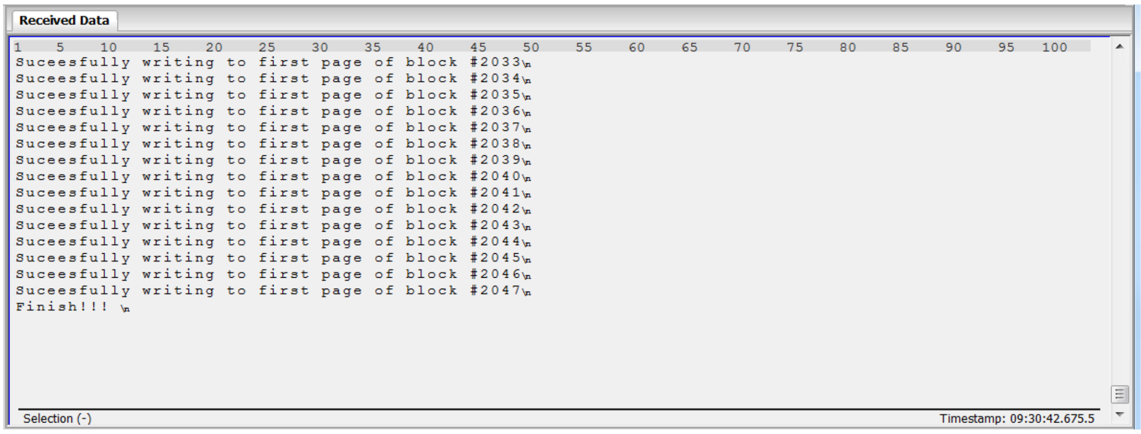
In this example, bad blocks occur at block #3, #1536, #1537, #1561, and #1789.



```
Received Data
Flash ID: 39117 \n
Checking for Bad Block \n
Bad block at block #3 \n
Bad block at block #1536 \n
Bad block at block #1537 \n
Bad block at block #1561 \n
Bad block at block #1789 \n
Total of bad blocks: 5 \n
Please press user button to start the flash test \n
```

Figure 7: Running the demo 1

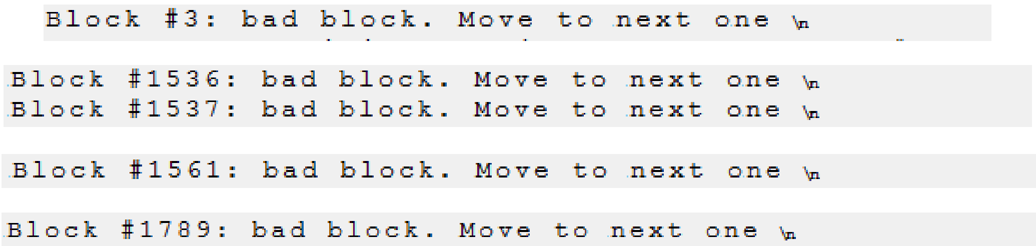
Then press the **blue button** to continue the demo. The controller will check if the current block is bad block, then it erases the data inside the block and write new data to the first page of each block.



```
Received Data
Suceesfully writing to first page of block #2033\n
Suceesfully writing to first page of block #2034\n
Suceesfully writing to first page of block #2035\n
Suceesfully writing to first page of block #2036\n
Suceesfully writing to first page of block #2037\n
Suceesfully writing to first page of block #2038\n
Suceesfully writing to first page of block #2039\n
Suceesfully writing to first page of block #2040\n
Suceesfully writing to first page of block #2041\n
Suceesfully writing to first page of block #2042\n
Suceesfully writing to first page of block #2043\n
Suceesfully writing to first page of block #2044\n
Suceesfully writing to first page of block #2045\n
Suceesfully writing to first page of block #2046\n
Suceesfully writing to first page of block #2047\n
Finish!!! \n
Selection (-)                                    Timestamp: 09:30:42.675.5
```

Figure 8: Running the demo 2

Bad blocks at block number 3, 1536, 1537, 1561, 1789 are reported again during the run so they are not overwritten.



```
Block #3: bad block. Move to next one \n
Block #1536: bad block. Move to next one \n
Block #1537: bad block. Move to next one \n
Block #1561: bad block. Move to next one \n
Block #1789: bad block. Move to next one \n
```

Figure 9: Running the demo 3

Getting started from the scratch

Getting started with CubeMX

Open CubeMX and choose **New Project.**

In the new project window, go to Board Selector tab. In the field **Type of Board**, **Nucleo64** is chosen and in the **MCU Series STM32F4** is picked. Finally in the **Board Lists field, NUCLEO-F401RE** is chosen.

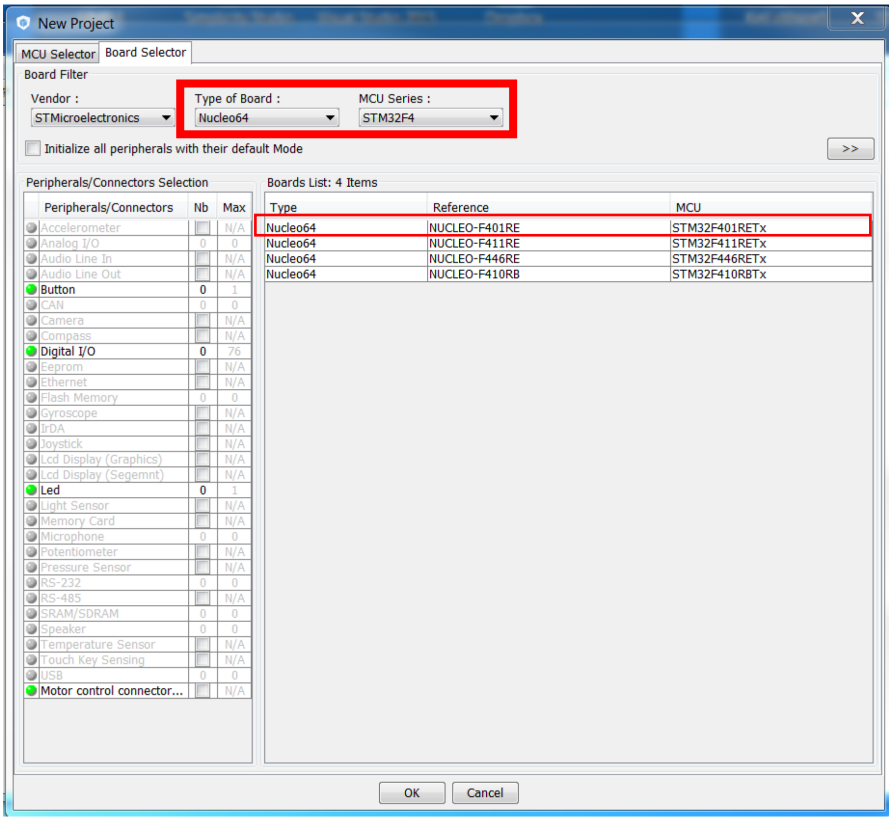


Figure 10: Board selection for new project

By pressing OK, a new window appears for project configuration. First thing need to be done is cleaning the pinout. Choosing the option **Pinout → Clear Pinouts**

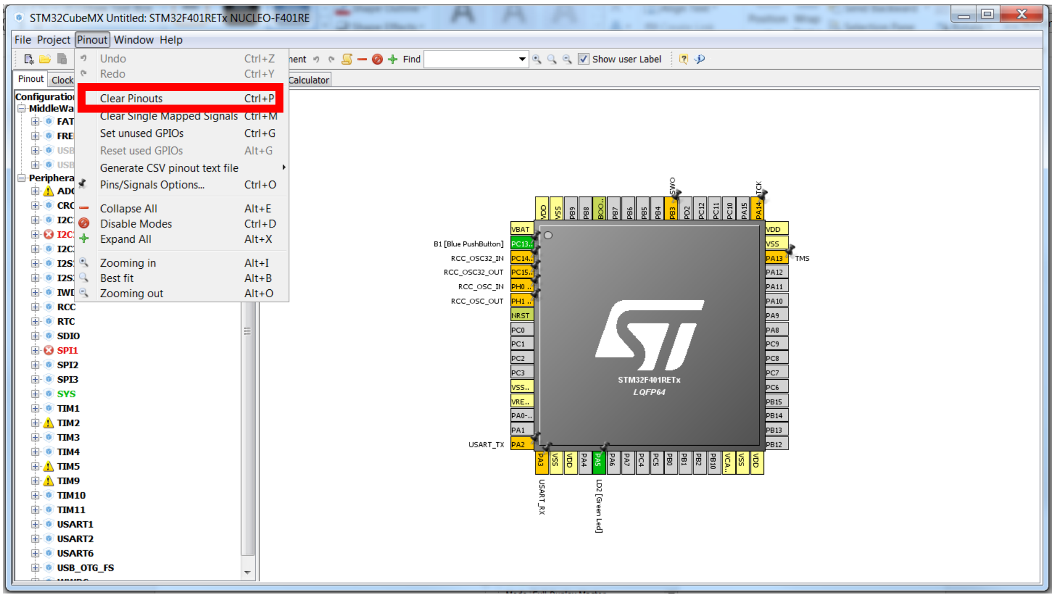


Figure 11: Clearing the pinout

In the **Pinout** tab, go to **SPI1**, we configure *Mode* as *Full-Duplex Master*, and *Hardware NSS Signal* as *Disable*

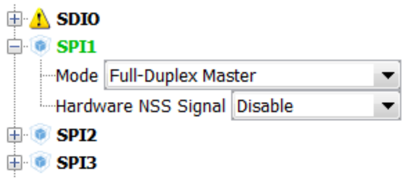


Figure 12: SPI configuration

In the **TIM1** configuration, we set *Clock Source* as *Internal Clock* to activate the General purpose timer 1

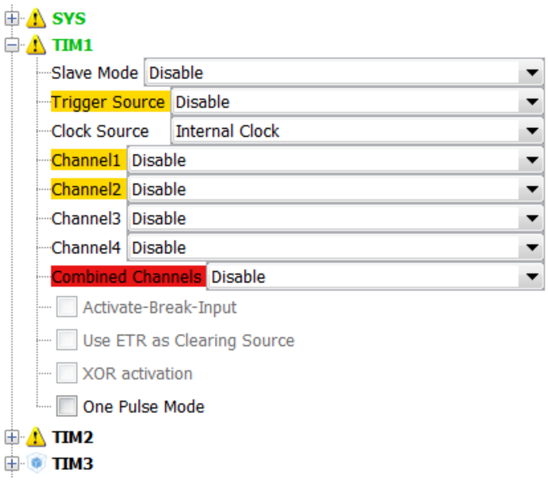


Figure 13: TIM1 configuration

Next, we activate UART by going to **USART2** configuration and set *Mode* as *Asynchronous*, and *Hardware Flow Control (RS232)* as *Disable*

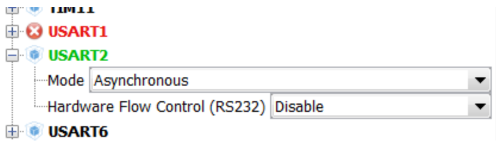


Figure 14: USART2 configuration

In the window on the right, configure pins *PB6, PB4, PA9, PA8, PC7*, and *PB10* as *GPIO_Output* and *PC13* as *GPIO_EXT13*

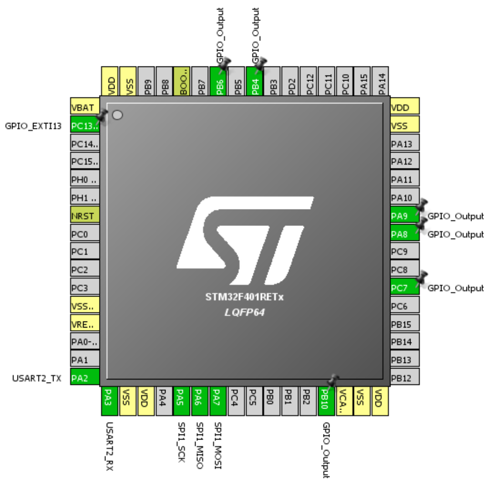


Figure 15: Setting the pins

Move to **Clock Configuration** tab and make sure that the clock is set to 84MHz

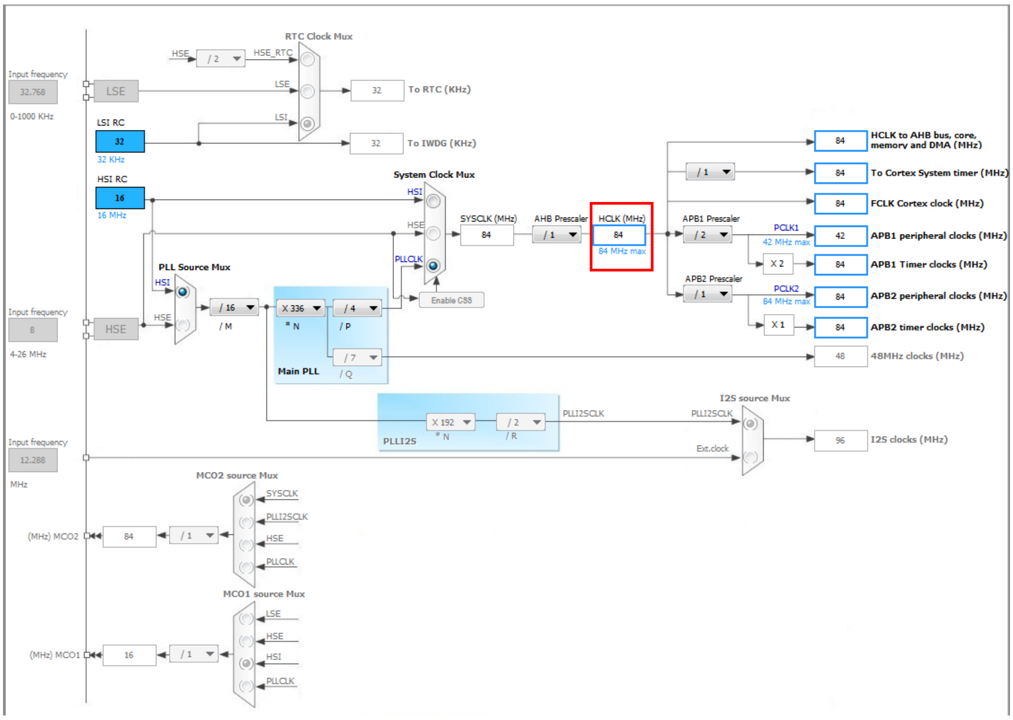


Figure 16: Setting the clock

Move to **Configuration** tab. If we have configured everything correct, we should have the picture as below

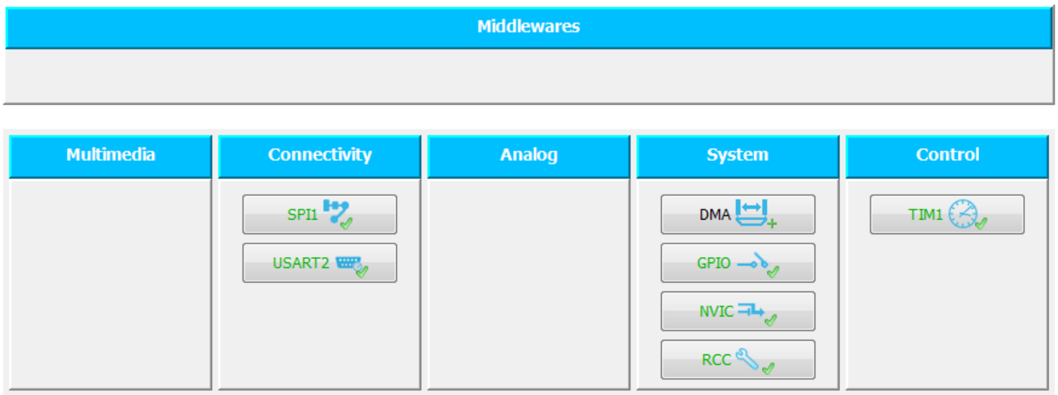


Figure 17: Configuration Tab

We start to configure the SPI by clicking in the tab **SPI1**. The setting in **SPI1** tab is configured as follow

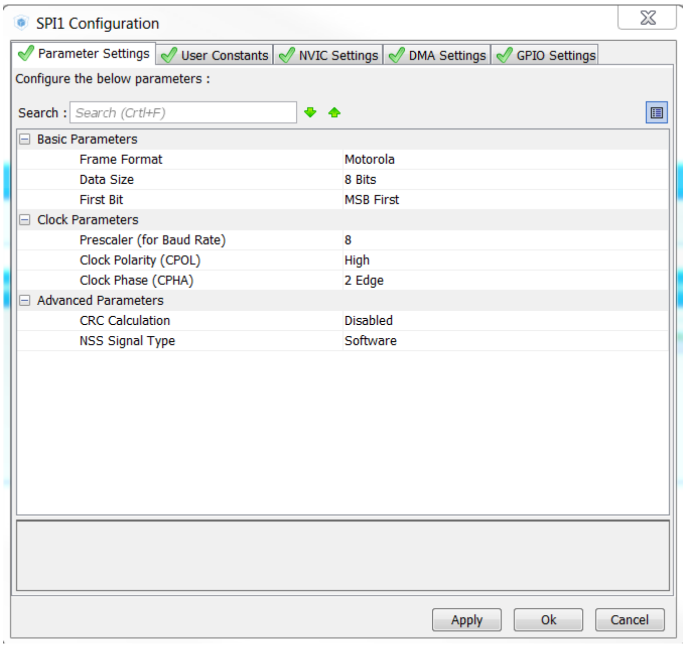


Figure 18: Setting for SPI1

Then the **USART2** is configured as followed:

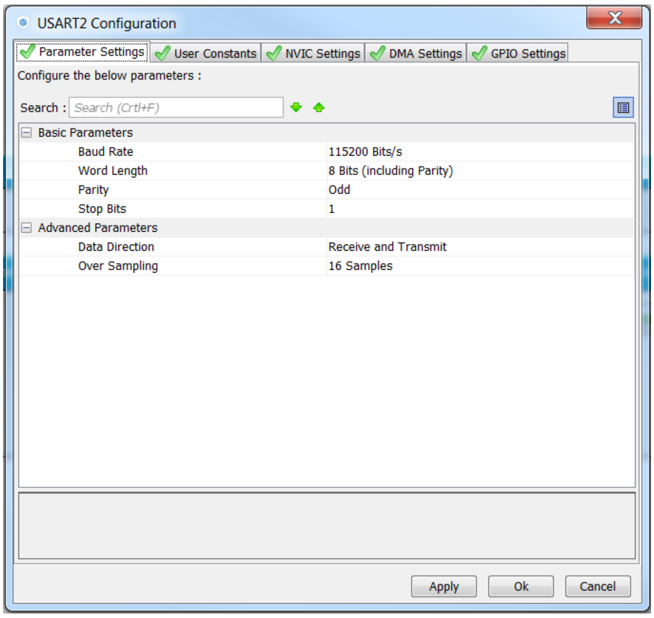


Figure 19: Setting for USART 2

Here is the configuration for the timer:

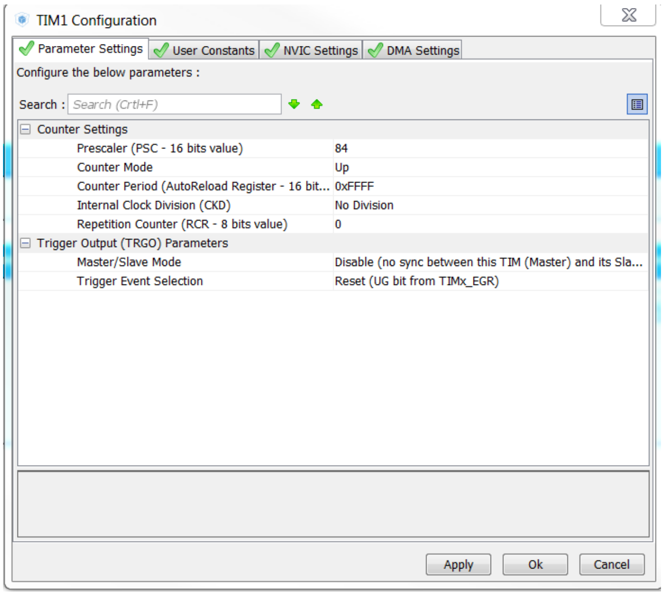


Figure 20: Setting for TIM1

Finally, we go to **NVIC** tab to enable the interrupt for the push button.

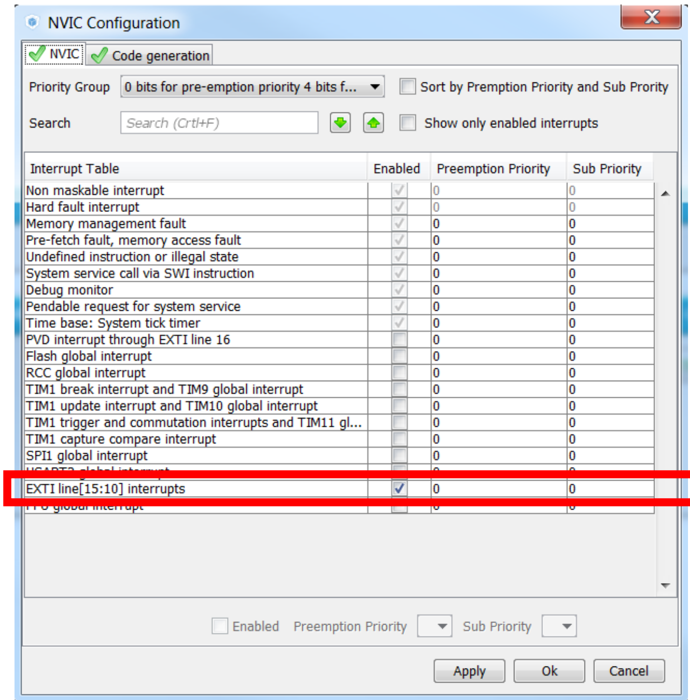


Figure 21: Setting for NVIC

Then we go to **Project → Setting** to configure the final setting for our project before generating it.

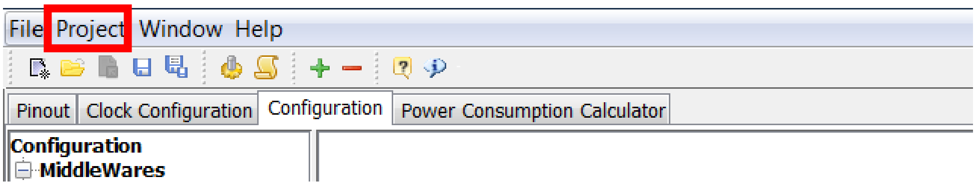


Figure 22: Project setting 1

In the **Project Setting** window, we name our project, choose a location to place it and, the most important, choose the Toolchain/IDE to writing the code (in this case, Keil is used).

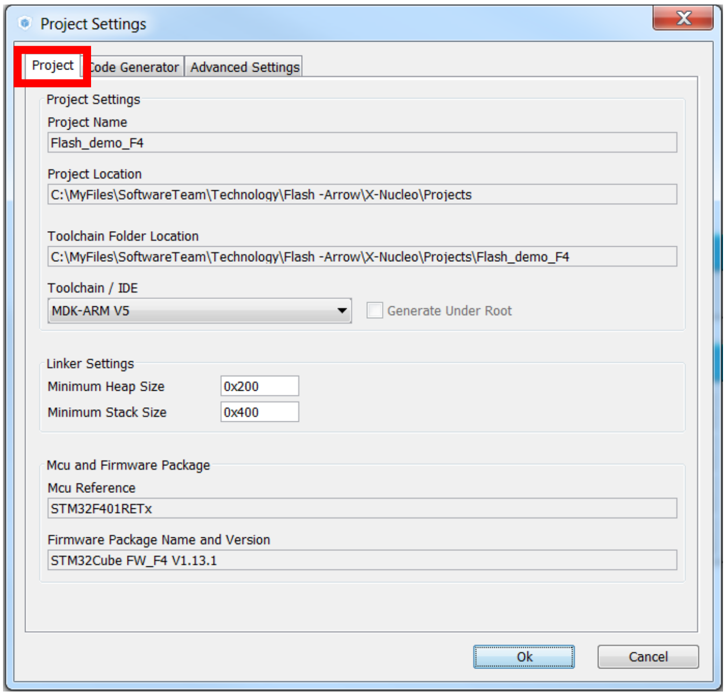


Figure 23: Project setting 2

Press Ok to close the window, then the project can be generated by choosing **Project→ Generate Code** or clicking on the **cog wheel** button

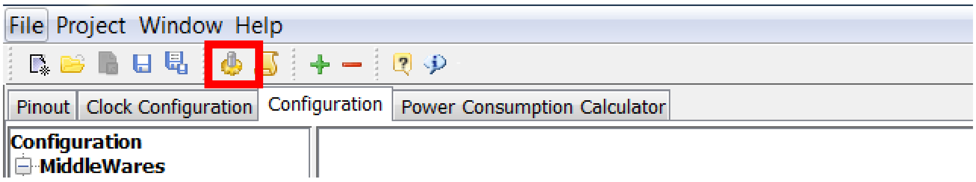


Figure 24: Generate project 1

After the project is successfully generated, a window will pop up to ask for further action. From here we can open the project in preferred IDE

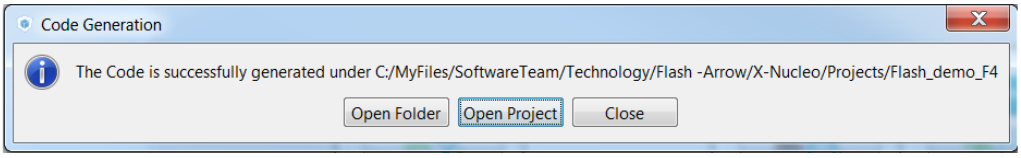


Figure 25: Generate Project 2

Editing the code

Keil is used in this guide but it can be easily tailored to other IDE.

First thing we do is adding all the necessary source file into the project, which is *TC58_FPP_CMD.c* in this case. Please also remember to add the header files to the *Inc folder*. All the related source and header files can be found in the Toshiba_NAND_Flash Library folder.

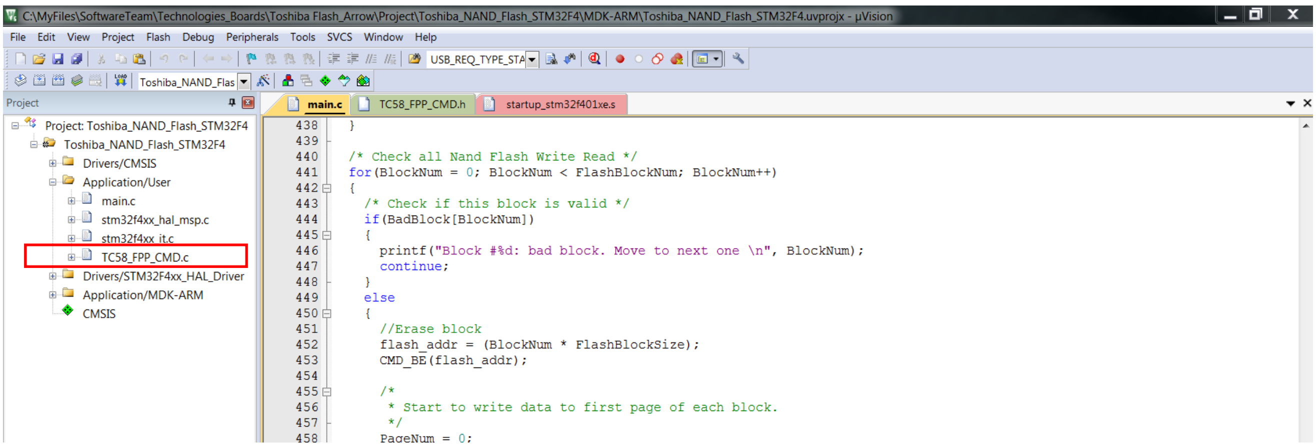


Figure 26: Adding additional source file

First step is done, we go to the main.c and enter these lines between the */* USER CODE BEGIN Includes */* and */* USER CODE END Includes */*

```
/* USER CODE BEGIN Includes */
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "TC58_FPP_CMD.h"
/* USER CODE END Includes */
```

Figure 27: Include and define section

Those codes include all the headers that needed. Next, we add all the needed variables to the *USER CODE BEGIN PV* section.

```
/* USER CODE BEGIN PV */
/* Private variables ---------------------------------------------------------*/
/* Enable printf on Keil and IAR*/
#ifdef __GNUC__
  #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
  #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */

#define  TRANS_BAD_LENGTH 4224
#define  FLASH_TARGET_ADDR  0x00000000
#define  Error_inc(x)  x = x + 1;

uint8_t   st_reg1 = 0;
uint8_t   memory_addr[TRANS_BAD_LENGTH];
uint8_t   memory_addr_cmp[TRANS_BAD_LENGTH];
uint8_t   memory_addr_t[TRANS_BAD_LENGTH];
uint8_t   memory_addr_t_cmp[TRANS_BAD_LENGTH];
uint8_t pTxData[16];
uint8_t pRxData[16];
uint8_t BadBlock[FlashBlockNum];

char buff[50];

uint16_t BlockNum;
uint16_t PageNum;
uint16_t  i=0;
uint16_t  col_address = 0;
uint16_t BadBlockTot = 0;
uint16_t BadBlockCnt;
uint16_t  flash_id = 0;
uint16  error_cnt = 0;

uint32_t PageAddress;
uint32_t BlockAddress;
uint32_t FlashAddressTest;
uint32_t FlashAddressShift1;
uint32_t FlashAddressShift2;
uint32_t flash_addr;

HAL_StatusTypeDef status = HAL_OK;
ReturnMsg message = Flash_Success;
/* USER CODE END PV */
```

Figure 28: Private variables section

Then we add the code for the function prototype to the *USER CODE BEGIN PFP* section. The names of the functions self-explaining their purposes. *FlashTest* tests the operation of the flash memory and *SearchBadBlock* looks for the bad block in the flash.

```
/* USER CODE BEGIN PFP */
/* Private function prototypes -----------------------------------------------*/
void SearchBadBlock(void);
void FlashTest(void);
/* USER CODE END PFP */
```

Figure 29: Function prototype section

Inside the main function, we navigate to the USER CODE BEGIN 2 section and add this block.

```
/* USER CODE BEGIN 2 */
__HAL_DBGMCU_FREEZE_TIM1();
HAL_TIM_Base_Start_IT(&htim1);
Initial_Spi();
```

Figure 30: Peripherals initialize

The block above initializes the timer and SPI and also starts the timer. After that, we add the following code to the application. This block reads the address of the flash memory and compares it with the pre-defined one. If there is a mismatch, the program will print the error message and go to the *Error_Handler* function, which is an infinite loop. If there is no error, it will run the *SearchBadBlock* function.

```
/* Read flash id */
CMD_RDID((uint16*)&flash_id );

/* Read flash id */
CMD_RDID((uint16*)&flash_id );

/* Compare to expected value */
if( flash_id != FlashID )
{
  Error_inc(error_cnt);
  printf("Wrong Flash ID \n");
  Error_Handler();
}
printf("Flash ID: %d \n", flash_id);


if(!error_cnt)
{
  printf("Checking for Bad Block \n");
  SearchBadBlock();
  printf("Total of bad blocks: %d \n", BadBlockTot);
}

printf("Please press user button to start the flash test \n");
/* USER CODE END 2 */
```

Figure 31: Check ID and check bad blocks

Scrolling down to the USER CODE BEGIN 4, we add this block to the program. This block allows us to use the *printf* function to write the data to the UART communication.

```
/* USER CODE BEGIN 4 */

/* Enable printf on System Workbench*/
int _write(int file, char *ptr, int len)
{
  HAL_UART_Transmit(&huart2, (uint8_t *)ptr, len, 10);
  return len;
}

/* Enable printf on IAR or Keil*/
PUTCHAR_PROTOTYPE
{
  HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, 0xFFFF);
  return ch;
}
```

Figure 32: Enable using printf

Next thing we add is the body of the function *SearchBadBlock*. According to the datasheet, information about the bad block is stored in column 0 or 4096 of the first and second page of the block. Therefore, this function scans through the column 4096 of the first page of each block and checks if they are equal to 0xFF. If it is not equal to 0xFF then we have a bad block.

```
void SearchBadBlock(void)
{
  BadBlockCnt = 0;
  for(BadBlockCnt = 0; BadBlockCnt < FlashBlockNum; BadBlockCnt++)
  {
    /* Check First Byte Spare Area of Page0 Block N */
    flash_addr = BadBlockCnt * FlashBlockSize;
    FlashAddressTest = (BadBlockCnt << 6);
    FlashAddressShift1 = (flash_addr >> 12);
#if 0
    if(flash_addr >= 0x10000000)
    {
      BadBlock[BadBlockCnt] = 0;
    }
#endif
    BadBlock[BadBlockCnt] = 0;

    /* Read flash memory data to memory buffer */
    message = CMD_READ( flash_addr );
    if(message != Flash_Success)
    {
      while(1)
        printf("Read error \n");
    }
    col_address = 0;
    memset(memory_addr, 0, TRANS_BAD_LENGTH);
    message = CMD_READ_CACHE( col_address, memory_addr, TRANS_BAD_LENGTH, 0 );
    if(message != Flash_Success)
    {
      while(1);
    }

    if(memory_addr[FlashPageSize + 1] != 0xFF)
    {
      BadBlockTot++;
      BadBlock[BadBlockCnt] = 1;
      printf("Bad block at block #%d \n", BadBlockCnt);
      continue;
    }
  }
}
```

Figure 33: CheckBadBlock function

After that we add the body of the function *FlashTest*. The main task of this function is erasing the block, writing data into the first page of the first block, then reading the data again and comparing them with the data it has written before to see if there is a match.

```
void FlashTest(void)
{
  /* Clear the block protection bit*/
  CMD_GET_FEATURE( 0xa0, &st_reg1 );
  if (st_reg1 & 0x38)
  {
    CMD_SET_FEATURE( 0xa0, (st_reg1&0x87) );
  }

  /* Check all Nand Flash Write Read */
  for(BlockNum = 0; BlockNum < FlashBlockNum; BlockNum++)
  {
    /* Check if this block is valid */
    if(BadBlock[BlockNum])
    {
      printf("Block #%d: bad block. Move to next one \n", BlockNum);
      continue;
    }
    else
    {
      //Erase block
      flash_addr = (BlockNum * FlashBlockSize);
      CMD_BE(flash_addr);

      /*
       * Start to write data to first page of each block.
       */
      PageNum = 0;
      col_address = 0;
      flash_addr = (BlockNum * FlashBlockSize) + (PageNum * FlashPageSize);

      /* Read flash memory data to memory buffer */
      CMD_READ(flash_addr);
      CMD_READ_CACHE(col_address, memory_addr_t_cmp, FlashPageSize, 0);
      if(memory_addr_t_cmp[0] != 0xFF)
      {
        memory_addr_t_cmp[0] = 0;
      }

      /* Write one page at time */
      memory_addr_t[0] = (BlockNum & 0xFF);
      memory_addr_t[1] = ((BlockNum >> 8) & 0xFF);
      memory_addr_t[2] = PageNum;
      memory_addr_t[3] = 0x0F;
      for(i = 4; i < FlashPageSize; i+=2)
      {
        memory_addr_t[i] = (i & 0xFF);            /* generate sequential byte data */
        memory_addr_t[i+1] = ((i >> 8) & 0xFF);   /* generate sequential byte data */
      }

      /* Program data to flash memory */
      CMD_PP_LOAD(col_address, memory_addr_t, FlashPageSize, 0);
      CMD_PROGRAM_EXEC(flash_addr);

      /* Read flash memory data to memory buffer */
      CMD_READ(flash_addr);
      CMD_READ_CACHE(col_address, memory_addr_t_cmp, FlashPageSize, 0);

      /* Compare Data Write with data Readed */
      if(memcmp(memory_addr_t, memory_addr_t_cmp, FlashPageSize) != 0)
      {
        Error_inc( error_cnt );
        printf("Error writing to first page of block #%d\n", BlockNum);
      }
      else
      {
        printf("Suceesfully writing to first page of block #%d\n", BlockNum);
      }
    }
  }
  printf("Finish!!! \n");
}
```

Figure 34: FlashTest function

From the code block above, the first thing the function *FlashTest* does is clearing the protection bit. Then it will run through block by block, check if the block is bad block. If current block is not a bad block, the function will generate random data, write it to the first page of the block, read the data again and compare with the original one. If the data is matching, the function will move to the next block. If the data is not matching, it will prompt an error message.

Finally we add the call back function for the push button. Whenever the button is pressed, the call back function calls the *FlashTest*.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  FlashTest();
}
```

Figure 35: Call back function for push button

Now we can build our program, download it into the board and our application is ready to run.

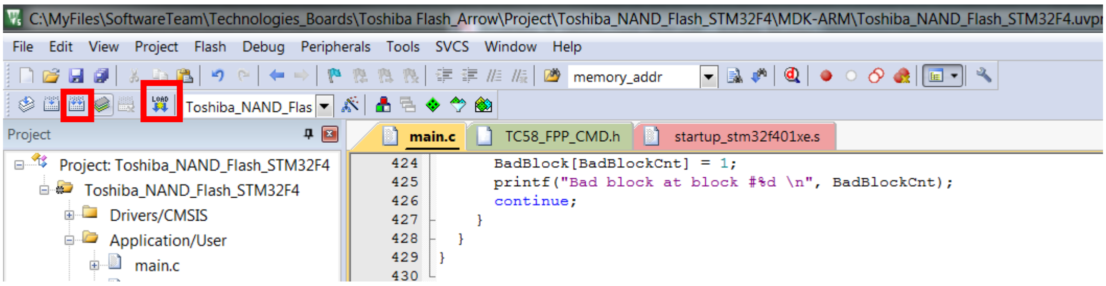


Figure 36: Build and Download

THE END