Programming External Flash used with STM32 Devices

MDK Tutorial

arm KEIL

AN333, Autumn 2020, v1.0

keil-feedback@arm.com

Abstract

This application note shows a universal approach for programming external flash memory connected to an STM32 microcontroller device with Keil MDK.

An example is presented using the STM32F769I-Discovery board with an STM32F769NIH6 microcontroller and MX25L51245G NOR flash connected over quad-SPI. However, the demonstrated concepts can be similarly applied to other STM32 devices and flash memories.

Contents

Abstract 1
Prerequisites 1
Introduction 2
  Flash programming in Keil MDK 2
Create a flash programming algorithm 3
Debug a flash programming algorithm 9
Access data and execute code from external flash 11
  Memory mapping via scatter file 15
Program external flash memory 16
Summary 17
References and Useful Links 17

Prerequisites

To reproduce the example described in this application note the following components are required:

Components from Arm:

- **Arm Keil MDK**: IDE and debugger used for project development and debug. To create a flash programming algorithm, an MDK-Essential license or above is required. MDK v5.31 is used here.
- **ARM::CMSIS**: CMSIS pack. Version 5.7.0 is used to create the example.
- **Keil::STM32F7xx_DFP**: Device Family Pack (DFP) for STM32F7 devices. Among other items, contains startup and system files as well as example applications. Version 2.13.0 is used.

Components from ST:

- **STM32F769I-Discovery Kit**: target hardware used in the example.
- **STM32CubeMX**: a graphical tool that allows easy configuration of STM32 microcontrollers. Version 6.0.1 with MCU Package for STM32F7 v 1.16.0 is used.

Example project

A ZIP file is available for download at **keil.com/appnotes/docs/apnt_333.asp**. It contains projects implementing flash programming algorithms for several STM32 Discovery boards, as well as Blinky examples showing how to use them in a project.

Copyright © 2020 Arm Ltd. All rights reserved

Introduction

In most cases, on-chip flash memory available on STM32 device is sufficient to run a user application. Loading the program into the device is done using the provided flash programming algorithms.

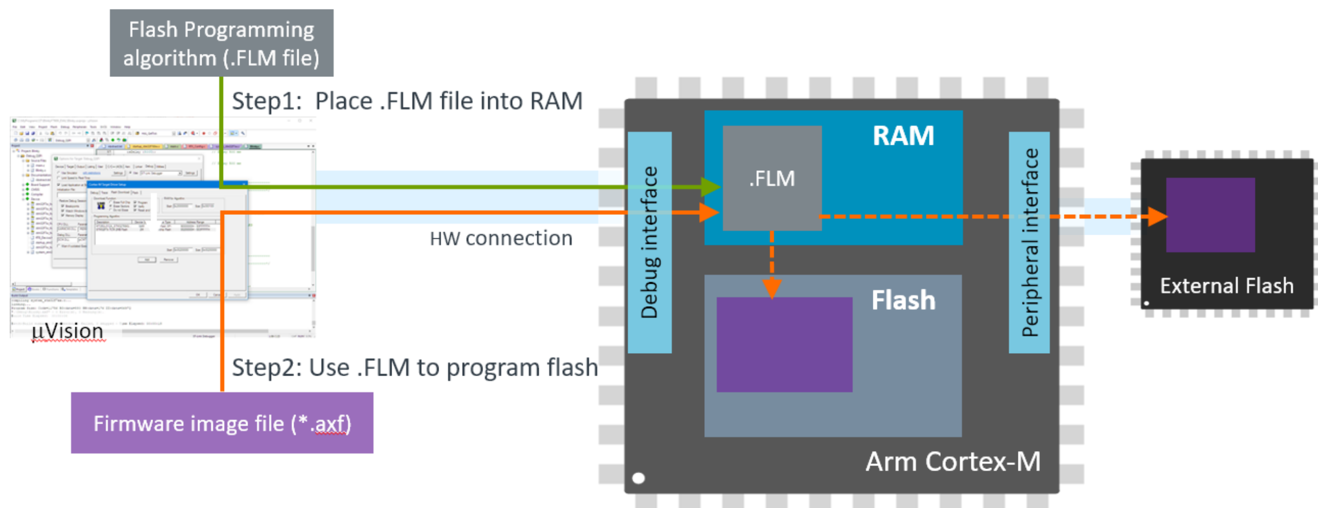
However, in cases when the size of the on-chip flash is not sufficient it may be required to use an external flash to either store constant data on it or to execute the program from it. For example, this can be the case in an embedded system with a display where high-quality graphic content requiring a lot of memory can be kept in external flash.

The process can be split into several distinctive parts explained in corresponding chapters in this document:

- **Create a flash programming algorithm**: explains how to create a special program that can write the required data to the external flash device.
- **Debug a flash programming algorithm**: provides instructions how to set up a project that allows to debug the flash programming algorithm.
- **Access data and execute code from external flash:** gives an example of a program that uses external flash for storing data and program code.
- **Program external flash**: shows how to use the programming algorithm in Keil MDK.

Flash programming in Keil MDK

In order to program on-chip or external flash, Keil MDK relies on flash programming algorithms – a special piece of software that the tool temporarily places into the MCU's RAM and then uses its interface to supply the data and store it in the target flash memory. The flash algorithms for Keil MDK have the extension **FLM**. The Figure below explains the flash programming concept in Keil MDK.



During device programming in the µVision IDE, first the flash programming algorithm for the target ROM area (in form of an FLM file) gets placed into the microcontroller's RAM. After that, using the programming algorithm API, the firmware image file gets written into the corresponding flash memory – this can be on-chip flash, or an external flash device.

Application Note 334: MDK Flash Download explains in detail how flash algorithms are used in MDK to erase, download, and verify the application in the Flash memory.

For the target microcontroller, the algorithms for on-chip flash are typically included in the corresponding Device Family Pack (DFP) and require no modifications from the user. This is also the case for STM32 devices. Example applications provided in the DFP are already configured to use these algorithms for internal flash.

Copyright © 2020 Arm Ltd. All rights reserved

Some STM32 DFPs also contain FLM files that implement flash programming algorithms for external flash devices located on specific development boards such as “Discovery” or “Eval”. However, the provided flash algorithm files will not work on a custom PCB with another device variant and different pinout.

For such custom cases, users need to create a flash algorithm as explained in the next chapter.

Create a flash programming algorithm

This chapter explains the steps required for creating a custom flash programming algorithm that will be able to load a program to an external flash device connected to a target STM32 MCU.

Note: Creating a flash programming algorithm with MDK-Lite is not supported.

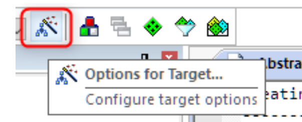
1. Copy the **_Template_Flash** folder from the **ARM:CMSIS** Pack folder (available by default in *C:\Users\<userName>\Pack\ARM\CMSIS\<version>\Device_Template_Flash)* to a new folder. This is a template project for flash programming algorithms. It is used as the basis for implementing the flash programming algorithm for the target system.

2. Remove the read-only protection for the copied directory including all subdirectories and files in it.

3. Rename the folder to reflect the purpose. In our example it is ***STM32F769I_Discovery_QPSI_MX25L51245G***.

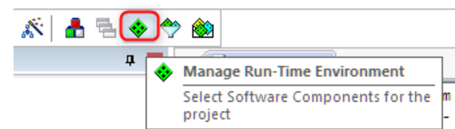
4. Rename the project file **NewDevice.uvprojx** to represent the new Flash ROM device name. In our example it is renamed to ***STM32F769I_Discovery_QPSI_MX25L51245G.uvprojx***.

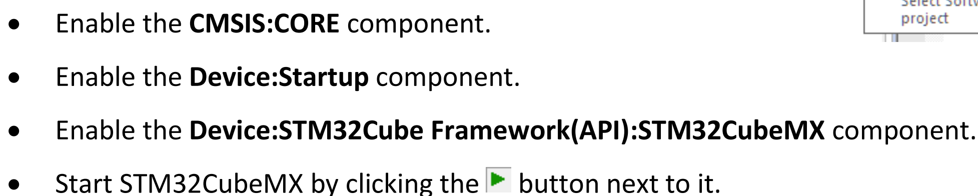
5. Open the project file with µVision.



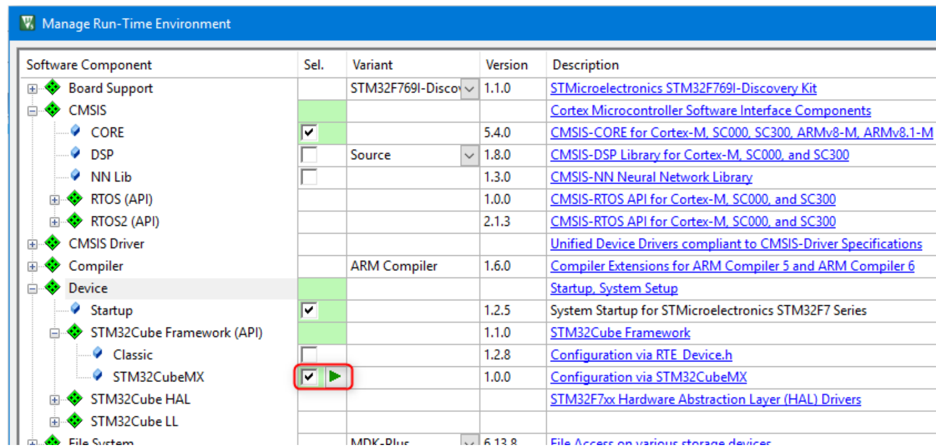
6. Go to **Project – Options for Target (Alt+F7)**:
    - On the **Target** tab, select the target microcontroller device. In our example it is ***STM32F769NIHx***.

      Note: The dialog displays only devices supported by already installed DFPs. If the target device is not available, you need to install its DFP using the **Pack Installer** tool.
    - Verify that in the **Code Generation** area **ARM Compiler** field selects **Use default compiler version 6**.
    - On the **Output** tab:
        - Change the content of the field **Name of Executable** to represent the target device and flash. The name shall not exceed 31 characters as otherwise the file will not be detected by µVision IDE. In our example it is set to ***STM32F769I_Disco_MX25L51245G***.
        - Verify that **Debug Information** flag is set. This is needed to ensure that µVision can call individual functions from the resulting FLM file.
    - Press **OK**.



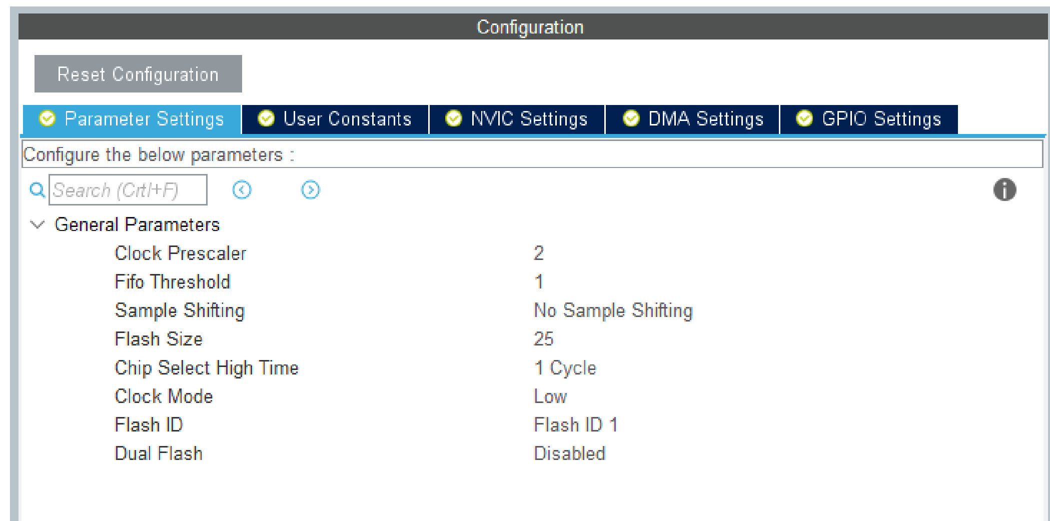
7. Open the **Manage Run-Time Environment** window:
    - Enable the **CMSIS:CORE** component.
    - Enable the **Device:Startup** component.
    - Enable the **Device:STM32Cube Framework(API):STM32CubeMX** component.
    - Start STM32CubeMX by clicking the  button next to it.

Copyright © 2020 Arm Ltd. All rights reserved



8. In STM32CubeMX, follow these steps to configure the MCU connection to the external flash and generate the corresponding code:

- On the **Pinout & Configuration** tab:
    - In the **Categories** view expand the **Connectivity** category and select **QUADSPI** as this is the interface used on the board in our example. This opens **QUADSPI Mode and Configuration**
    - Select the proper **QuadSPI Mode** (in our example **Bank1 with Quad SPI Lines**)
    - In the **Configuration** section – on the **Parameter Settings** tab, set the correct parameters according to the target setup. In our example these are the following:

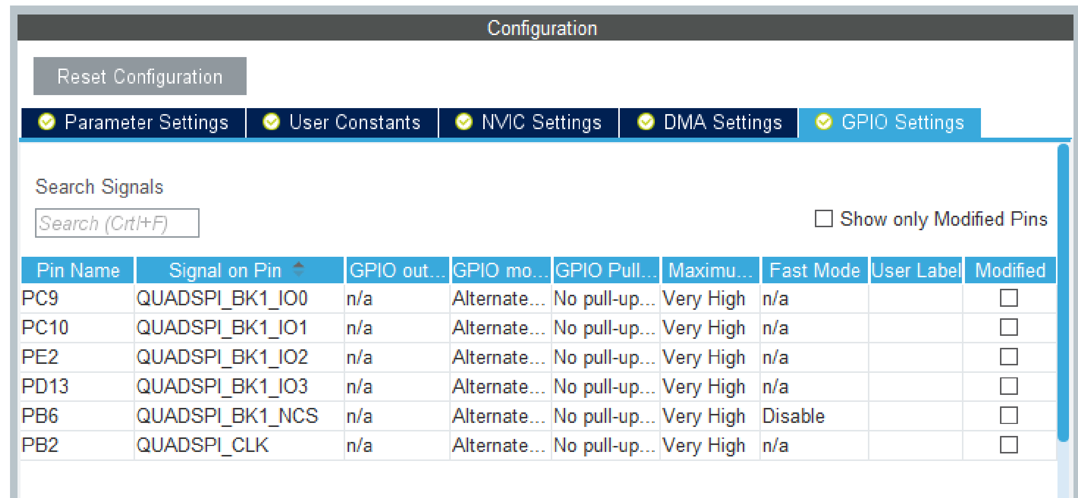


Note: Check device memory datasheet for the correct QSPI parameter settings.

- In the **Pinout View**: select the correct pins that the external flash memory is connected to. The pin configuration can be also observed on the **GPIO Settings** tab in the **Configuration** section.

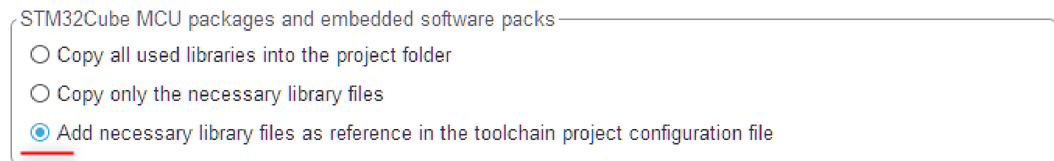
  According to our example's STM32F769I-Discovery schematics, the pins need to be reconfigured as shown below:

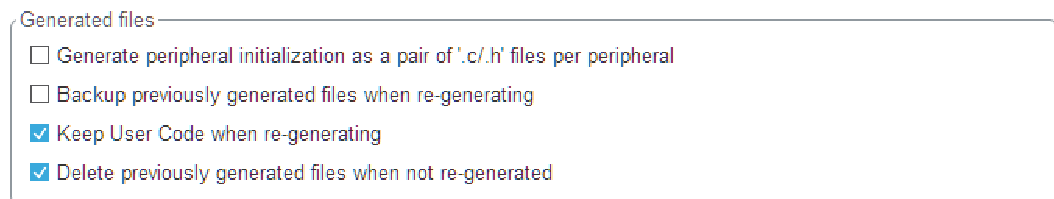
Copyright © 2020 Arm Ltd. All rights reserved



- On the **Clock Configuration** tab: no need to change anything from the default settings.
- On the **Project Manager** tab modify default project options:
  - In the **Project** category select **Do not generate the main()**.

  - In the **Code Generator** category set checkbox **Add necessary library files as reference in the toolchain project configuration file.**





  - In the **Advanced Settings** category for the QUADSPI driver:
    - Enable **Do Not Generate Function Call**
    - Disable **Visibility (Static)**

Generated Function Calls

Rank	Function Name	IP Instance Name	Do Not Generate Function Call	Visibility (Static)
1	MX_GPIO_Init	GPIO	☐	☑
2	SystemClock_Config	RCC	☐	☐
3	MX_QUADSPI_Init	QUADSPI	☑	☐

- Press the **Generate Code** button.
  - If requested, download the software package for the target MCU.
  - Once the code generation is completed, press **Close** in the **Code Generation** dialog.

9. Switch back to the µVision project.

- Press **OK** in the **Manage Run-Time Environment** window

Copyright © 2020 Arm Ltd. All rights reserved

- Click **Yes** in the dialog asking whether to **Import changes** in the project based on the code generated in STM32CubeMX.

10. Implement the flash programming algorithm functions in the file **FlashPrg.c**.

    The file group **Program Functions** contains the **FlashPrg.c** file where high-level functions for flash operations explained in section **Flash programming in Keil MDK** need to be implemented.

    The implementation is quite universal across STM32 devices but is specific for the communication interface used (for example QuadSPI or OctoSPI). It is also independent from the external flash memory device.

    In the application note's ZIP file, you can find the project ***STM32F769I_Discovery_QPSI_MX25L51245G*** that contains the **FlashPrg.c** file implementing the algorithm functions for our target system. Just use this file instead of the default template file.

11. Adapt the device parameters in the file **FlashDev.c**.

    This file specifies the parameters of the target flash memory device such as page size, sectors, total size and others. In our example it needs to have a following content:

```
#include "FlashOS.h"          // FlashOS Structures

struct FlashDevice const FlashDevice  =  {
   FLASH_DRV_VERS,            // Driver Version, do not modify!
   "STM32F769I_Disco_MX25L51245G", // Device Name
   EXTSPI,                    // Device Type
   0x90000000,                // Device Start Address
   0x04000000,                // Device Size in Bytes (64MB)
   0x00001000,                // Programming Page Size: 4Kb (16* page size)
   0x00,                      // Reserved, must be 0
   0xFF,                      // Initial Content of Erased Memory
   10000,                     // Program Page Timeout 1000 mSec
   10000,                     // Erase Sector Timeout 1000 mSec

// Specify Size and Address of Sectors
   0x1000, 0x000000,            // Sector Size 4kB, Sector Num : 16383
   SECTOR_END
};
```

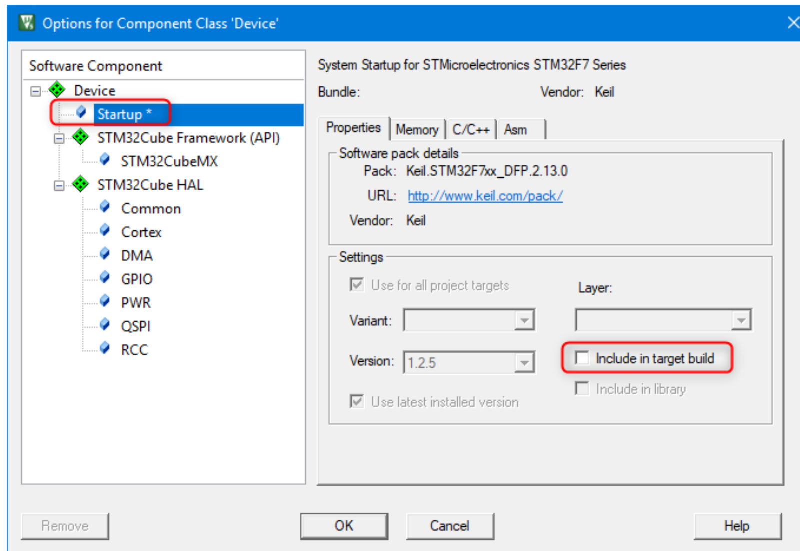
12. Remove the startup file from the build and replace the system file.

    We need to exclude the startup file from our flash programming algorithm so that no vector table gets defined and the algorithm program does not run automatically when loaded. We also need to add back the system file.

    In the µVision **Project** window:

    - Right-click on **Device** group then select **Options for Component Class 'Device'**. In the open dialog select the **Startup** component and uncheck **Include in target build**. Click **OK**.

Copyright © 2020 Arm Ltd. All rights reserved



- In the previous step we have also removed the system file that is required for the build. Thus, we need to add it back.
    - Right-click on the target folder (***Cortex-M*** in our case) and select **Add Group**. A group with name **New Group** is added to the project. Click twice on it and rename to ***System File***.
    - Right-click on the **System File** group and select **Add existing files to group 'System File'** and add a system file from the project's "***./RTE/Device/<DeviceName>/***" directory to it. In our example it is "***./RTE/Device/STM32F769NIHx/system_stm32f7xx.c***" file.

13. Add the driver for the memory connection interface.

    STMicroelectronics provides drivers for various external quad-SPI flash memory devices in a GitHub repository https://github.com/STMicroelectronics/stm32-external-loader/tree/contrib.

    Our example's STM32F769I-Discovery board carries an MX25L51245G flash memory device, so we use the corresponding QSPI Driver from the repository.

    - In the root directory of our µVision project create a folder with the flash device name (e.g. ***MX25L51245G***) and copy **quadspi.c** and **quadspi.h** files from the QSPI driver repository into it.
    - In µVision's **Project** window, right-click on the target folder (***Cortex-M*** in our case) and select **Add Group**. A group with name **New Group** is added to the project. Rename it to ***QUADSPI Memory***.
    - Right-click on the ***QUADSPI Memory*** group and select **Add existing files to group 'QUADSPI Memory'…** then add the existing ***quadspi.c*** file from "***. \QUADSPI Memory***" directory to it.
    - Go to **Project – Options for Target (Alt+F7) – C/C++(AC6)** tab and in the **Include Paths** field add the path to the folder with the memory connection driver. In our case it is ***.\MX25L51245G***.

14. Edit **quadspi.c** and **quadspi.h** files.

    The default QSPI driver files need slight modification to ensure proper usability.

    - Modify **quadspi.c** and add at the top:

    ```
    #include "quadspi.h"
    extern QSPI_HandleTypeDef hqspi;
    ```

    - Modify **quadspi.h**:
        - At the top add: `#include "main.h"`

    Refer to the example project in the application note's ZIP file if you want to double-check.

Copyright © 2020 Arm Ltd. All rights reserved

15. Add HAL Tick functions in **main.c**:

    Default implementations of HAL Tick functions rely on SysTick. However, during the flash programming we want to ensure that no interrupts occur and avoid unnecessary initialization of additional peripherals.

    Thus, new implementations of the HAL Tick functions need to be placed between the sections marked with `/* USER CODE BEGIN 0 */` and `/* USER CODE END 0 */`. This will override weak implementations in the HAL.

    - Add **HAL_InitTick** function:

```
/**
  * Override default HAL_InitTick function
  */
HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority) {
  return HAL_OK;
}
```

    - Add **HAL_GetTick** function:

```
/**
  * Override default HAL_GetTick function
  */
uint32_t HAL_GetTick (void) {
  static uint32_t ticks = 0U;
         uint32_t i;

  /* If Kernel is not running wait approximately 1 ms then increment
     and return auxiliary tick counter value */
  for (i = (SystemCoreClock >> 14U); i > 0U; i--) {
    __NOP(); __NOP(); __NOP(); __NOP(); __NOP(); __NOP();
    __NOP(); __NOP(); __NOP(); __NOP(); __NOP(); __NOP();
  }
  return ++ticks;
}
```

    - Add **HAL_Delay** function:

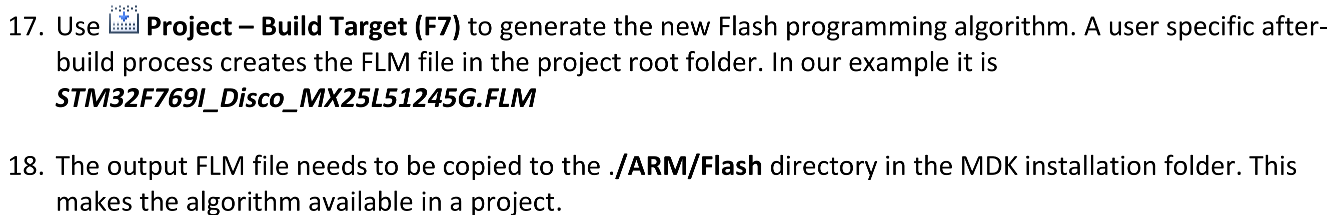
```
/**
  * Override default HAL_InitTick function
  */
void HAL_Delay(uint32_t Delay)
{
  uint32_t tickstart = HAL_GetTick();
  uint32_t wait = Delay;

  /* Add a period to guaranty minimum wait */
  if (wait < HAL_MAX_DELAY)
  {
    wait += (uint32_t)(HAL_TICK_FREQ_DEFAULT);
  }

  while((HAL_GetTick() - tickstart) < wait)
  {
    __NOP();
  }
}
```

16. Edit **main.h**

    Add definitions of **SystemClock_Config()** and **MX_QUADSPI_Init()** functions:

```
/* USER CODE BEGIN EFP */
void SystemClock_Config(void);
void MX_QUADSPI_Init (void);
/* USER CODE END EFP */
```


Copyright © 2020 Arm Ltd. All rights reserved



17. Use **Project – Build Target (F7)** to generate the new Flash programming algorithm. A user specific after-build process creates the FLM file in the project root folder. In our example it is ***STM32F769I_Disco_MX25L51245G.FLM***

18. The output FLM file needs to be copied to the **./ARM/Flash** directory in the MDK installation folder. This makes the algorithm available in a project.

The flash algorithm created in this chapter can be used to place a firmware image to the external flash as explained in chapter **Program external flash memory**. Chapter **Access data and execute code from external flash** shows how to use external flash memory in an application project for storing data and code. If the algorithm does not work see chapter **Debug a flash programming algorithm** that explains how to setup a test project for debug purposes.

Debug a flash programming algorithm

The project from the previous chapter implements the flash programming algorithm however, it does not allow to debug the code if any issues problems occur. This chapter explains how to create a separate test project that can be used for debug purposes.

In our example we place the complete code and data in RAM and start a debug session.

1. Copy the example project created in the chapter **Create a flash programming** algorithm into a separate directory. This will be the basis for the test project with debug capabilities.

2. Ensure the test program is executed from RAM:

   Go to **Project – Options for Target (Alt+F7)**:

   - On the **C/C++ (AC6)** tab, uncheck options **Read-Only Position Independent** and **Read-Write Position Independent**.
   - On the **Linker** tab, click the **Edit** button and modify the linker script so that all code and data are placed in RAM. In our example, it is done using the following:

```
; Linker Control File (scatter-loading)
;

LR_ROM 0x20000000 0x0000F000  {    ; load region size_region
  ER_ROM 0x20000000 0x0000F000  {  ; load address = execution address
    *.o (RESET, +First)
    *(InRoot$$Sections)
    *  (+RO +XO)
  }

  RW_IRAM1 0x2000F000 0x00002000  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

   Refer to chapter Scatter File Syntax in the Linker User's Guide for more information about the scatter file.

Copyright © 2020 Arm Ltd. All rights reserved

- On the **Debug** tab:
  - In the **Use** row select the target debug adapter from the drop-down menu. In our case this is **ST-LINK**.
  - Uncheck **Load Application at Startup** flag.
  - Add an **Initialization File** called *Dbg_RAM.ini* with the following content:

```
/*----------------------------------------------------------------
  Setup()  configure PC & SP for RAM Debug
 *--------------------------------------------------------------*/
FUNC void Setup (void) {
  SP = _RDWORD(0x20020000);         // Setup Stack Pointer
  PC = _RDWORD(0x20020004);         // Setup Program Counter
  _WDWORD(0xE000ED08, 0x20020000); // Setup Vector Table Offset Register
}

/*---------------------------------------------------------------
  OnResetExec() configure PC & SP after reset for RAM Debug
 *--------------------------------------------------------------*/
FUNC void OnResetExec (void) {
  Setup();
}

LOAD %L INCREMENTAL                   // load the application

Setup();                              // Setup for Running

//g, main
```

- On the **Utilities** tab uncheck the option **Update Target before Debugging**.
- Press **OK**.

3. Re-enable the usage of the startup code from the Device Family Pack.
   This reverses step **12** from chapter **Create a flash programming** algorithm. In the **Project** window:

   - Exclude the system file provided by STM32CubeMX:
     - Find the **System File** group.
     - Right-click on the system file located there (in our example *system_stm32f7xx.c*)
     - Select **Options for File 'system_stm32f7xx.c'...**
     - In the dialog window uncheck **Include in Target Build**
     - Press **OK**.
   - Include device startup from DFP:
     - In the **Device** group find the system file (*system_stm32f7xx.c* in our case).
     - Right click on it and select **Options for Component Class 'Device'..**
     - Ensure that **Startup** is selected in the **Software Component** section on the left. Enable **Include in target build**.
     - Press **OK**.

4. Create a file with the test code.

   - In the **Project** window, right-click on the target folder (***Cortex-M*** in our case) and select **Add Group**. A group with name **New Group** is added to the project. Rename it to ***Test Code***.
   - Right-click on the **Test Code** group and select **Add new Item to Group 'Test Code'.** In the dialog select **C File (.c)** option and specify the file name, for example ***FlashTest.c***. Press the **Add** button and close the dialog.

Copyright © 2020 Arm Ltd. All rights reserved

- Populate the ***FlashTest.c*** file with a `main()` function that tests the API functions from the ***FlashPrg.c*** file. It should implement the flow used in µVision as described in the CMSIS-Pack documentation (section **Algorithm Functions**) and **Application Note 334: MDK Flash Download**. For example, testing the Flash Erase operation:

```
#include "RTE_Components.h"
#include CMSIS_device_header
#include "FlashOS.h"

extern struct FlashDevice const FlashDevice;

volatile int ret;                    // Return Code

/* Error handling function */
void stop_on_error(uint32_t cond)
{
  if(cond) {
    __BKPT(0x1); // Error occurred during execution
    while(1){}
  }
}

/*----------------------------------
  Main Function
 *----------------------------------*/
int main(void)
{
  /* Test Flash Erase operation */
  ret = Init(FlashDevice.DevAdr, 0, 1);
  stop_on_error (ret);

  ret = EraseChip();
  stop_on_error (ret);

  ret = UnInit(1);
  stop_on_error(ret);

  while(1){}
}
```

Here the `main()` function initializes external memory, performs full memory erase operation and then uninitializes connection to the flash device. If execution of a flash programming function is unsuccessful, the program is halted at the breakpoint in `stop_on_error()`.

Flash Program and Flash Verify flows can be implemented in a similar way and debugged if returned code is unsuccessful.

Access data and execute code from external flash

This chapter demonstrates how to extend an existing application for storing data constants or program code in the external flash memory.

All steps listed below are universal and could be applied to any project. In our example, we start with a **CMSIS-RTOS2 Blinky with STM32CubeMX** project for the STM32F769I-Discovery board included with the Device Family Pack for the STM32F7 Series.

The application note's ZIP archive contains the modified Blinky example that uses the external flash memory.

1. Use **Pack Installer** to copy the **CMSIS-RTOS2 Blinky with STM32CubeMX** project for the STM32F769I-Discovery board.

Copyright © 2020 Arm Ltd. All rights reserved

CAN RTR (STM32756G-EVAL)	Copy	CAN example that demonstrates Remote Transmission Request (RTR) usage
CMSIS-RTOS Blinky (STM32F746G-Discovery)	Copy	CMSIS-RTOS2 Blinky example
CMSIS-RTOS Blinky (STM32F769I-Discovery)	Copy	CMSIS-RTOS based Blinky example
CMSIS-RTOS Blinky (STM32F769I-EVAL)	Copy	CMSIS-RTOS based Blinky example
CMSIS-RTOS Blinky (STM32756G-EVAL)	Copy	CMSIS-RTOS based Blinky example
CMSIS-RTOS Blinky with STM32CubeMX (STM32F746G-Discovery)	Copy	CMSIS-RTOS based Blinky example with VIO and configured STM32CubeMX
CMSIS-RTOS Blinky with STM32CubeMX (STM32F769I-Discovery)	Copy	CMSIS-RTOS based Blinky example configured with STM32CubeMX
FTP Server IPv4/IPv6 (STM32F746G-Discovery)	Copy	File Server using FTP protocol with SD/MMC Memory Card as storage media

Note that for STM32F2/F4/F7 series you need to copy the Blinky example that has "***with STM32CubeMX***" in its name. Only this project is configurable with STM32CubeMX by default. For other series, the standard CMSIS-RTOS Blinky example supports STM32CubeMX and can be extended for using external flash memory per flow explained in this chapter.

2. Configure the QuadSPI interface using STM32CubeMX.

   To be able to use external flash memory we need to add and configure the QSPI driver interface in the project using STM32CubeMX tool.

   Target hardware is the same as used for the flash algorithm, so just repeat the steps **7**, **8** and **9** from the chapter **Create a flash programming algorithm**.

   Note that in this case the clock setup as well as other peripherals used by the application can be also configured in the STM32CubeMX.

3. Add Quad-SPI driver files to the project.

   The **quadspi.c** and **quadspi.h** files can be fully reused from the flash algorithm project as follows:

   - Copy the .***MX25L51245G***-directory from the flash algorithm project into the root of the Blinky project.
   - In the **Project** window, right-click on the target folder and select **Add Group**. A **New Group** is added to the project. Rename it to ***QUADSPI Memory***.
   - Right-click on the ***QUADSPI Memory*** group and select **Add existing files to group 'QUADSPI Memory'…** and add an existing ***quadspi.c*** file from the .***MX25L51245G*** directory to it.
   - Go to **Project – Options for Target (Alt+F7) – C/C++(AC6)** tab and in the **Include Paths** field add the path to the folder with the memory connection driver. In our case it is **.*MX25L51245G***.

4. Modify **main.h** file:

   Add the definition of the **MX_QUADSPI_Init()** function:

```
/* USER CODE BEGIN EFP */
void MX_QUADSPI_Init (void);
/* USER CODE END EFP */
```

5. Modify **main.c** file:

   - Add `#include "quadspi.h"` under User includes:

```
  …
/* USER CODE BEGIN Includes */
#include "quadspi.h"
/* USER CODE END Includes */
```

   - Enable memory mapped mode in **main()**:

     To seamlessly access data and execute code located in the external flash memory, the memory mapped mode for the QUADSPI needs to be enabled.

Copyright © 2020 Arm Ltd. All rights reserved

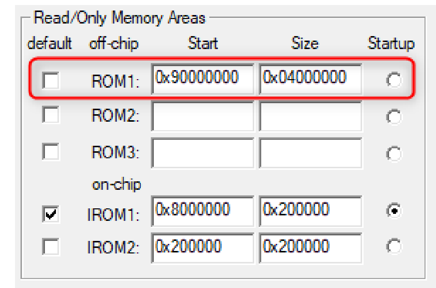
For that, add calls to the corresponding CSP functions in the **main()** function in section `/* USER CODE BEGIN 2 */` as shown below:

```
/* USER CODE BEGIN 2 */
CSP_QUADSPI_Init();
CSP_QSPI_EnableMemoryMappedMode();
…
```

6. Extend the memory layout.

   We need to define the memory area that corresponds to the external flash used in the system, so that required data or code can be linked into it.

   Go to **Project – Options for Target (Alt+F7) – Target** tab. In the **Read/Only Memory Areas,** specify an additional ROM area with the **Start** and **Size** fields that correspond to the external flash memory as specified in the **FlashDev.c** file used in the flash programming algorithm. In our example we use **ROM1** to specify the area for the external flash.



   The **default** checkbox defines which memory is used by default. The **Startup** radio button selects the area used for the startup code.

   Both must be enabled for internal ROM to ensure that device is able to start and initialize the QSPI interface.

7. Add constant data to be placed in the external flash.

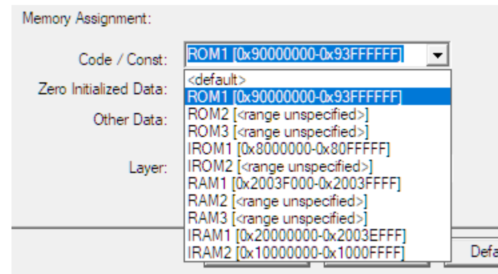
   - In the **Project** window right-click on the **Source** group and then **Add a New Item to Group "Source"…**
   - Select **C File (.c)**, provide a file name (for example ***Data.c***.) and click on **Add**.
   - Add the following content to the empty ***Data.c*** file that has appeared in the Source group:

```
/* Delay constants */
const unsigned int delay_data[10]={100,200,300,400,500,600,700,800,900,1000};
```

8. Map target files on to external flash.

   In the µVision IDE it is possible to specify the target memory for software components, file groups and individual files.

   In our example, we want to place the ***Data.c*** file on external flash memory.



   In the **Project** window, right-click on the ***Data.c*** file and then select **Options for File 'Data.c'**. In the dialog in the **Memory Assignment** area choose from the drop-down menu the target memory for the **Code/Const**. In our example it should be ***ROM1***. Press **OK**.

   Similar approach can be used to place program code into external flash memory (for example **Blinky.c** file).

9. Alternatively, a scatter file can be used for memory mapping. See section **Memory mapping via scatter file**.

   Note that the Quad-SPI driver as well as the startup code (up to the initialization of the memory mapped mode) need to be started out of the internal flash.

Copyright © 2020 Arm Ltd. All rights reserved

10. Update the ***Blinky.c*** to use ***delay_data***.

    We modify the default logic so that when a joystick is pressed the next value from the ***delay_data*** array, stored in the external flash memory, is used as an interval for blinking the LED.

    The example uses some board-specific peripherals such as LEDs and Joystick.

```
…
extern const uint32_t delay_data[10];   // array with delay values
static       uint32_t delay = 0U;       // current delay value

/*----------------------------------------------------------------------------
  thrLED: blink LED
 *---------------------------------------------------------------------------*/
__NO_RETURN void thrLED (void *argument) {

  for (;;) {
    LED_On (0U);                    // Switch LED on
    osDelay (delay);                // Delay
    LED_Off (0U);                   // Switch off
    osDelay (delay);                // Delay
  }
}

/*----------------------------------------------------------------------------
  thrBUT: check button state
 *---------------------------------------------------------------------------*/
__NO_RETURN static void thrBUT(void *argument) {
  uint32_t last  = 0;
  uint32_t state = 0;
  uint32_t index = 0;

  delay = delay_data[index];

  for (;;) {
    state = (Buttons_GetState () & 1U);   // Get pressed button state
    if (state != last){                   // Act only on state changes
      if (state == 1){                    // Act only on new presses
        index++;
        if (index == (sizeof(delay_data) / sizeof(delay_data[0]))){
          index = 0U;
        }
        delay = delay_data[index]; // Obtain next delay value from external flash
      }
    }
    last = state;
    osDelay (100U);
  }
}
```

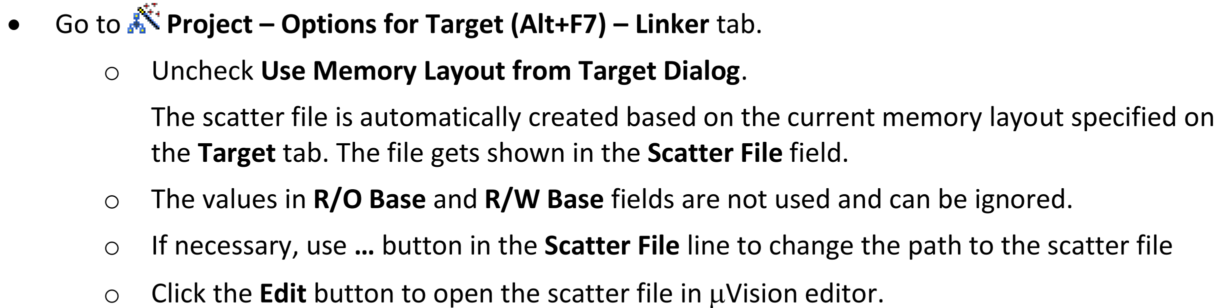
11. Go to **Project – Build Target (F7)** and in the **Build Output** window observe that it is built correctly without errors or warnings.

Now the firmware image can be loaded to the internal and external flash memory as explained in chapter **Program external flash memory**.

Copyright © 2020 Arm Ltd. All rights reserved

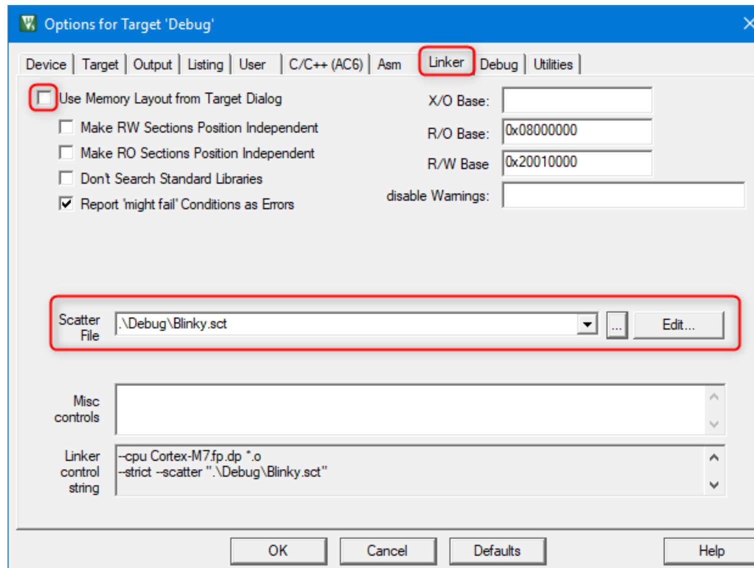
Memory mapping via scatter file

For complex cases, a scatter file can be used to define data and code placement in the memory:



- Go to **Project – Options for Target (Alt+F7) – Linker** tab.
    - Uncheck **Use Memory Layout from Target Dialog**.

      The scatter file is automatically created based on the current memory layout specified on the **Target** tab. The file gets shown in the **Scatter File** field.
    - The values in **R/O Base** and **R/W Base** fields are not used and can be ignored.
    - If necessary, use **…** button in the **Scatter File** line to change the path to the scatter file
    - Click the **Edit** button to open the scatter file in µVision editor.



In our example, the scatter file will have the following content, with Data.c and Blinky.c files mapped to the external data flash (ROM1):

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x08000000 0x00200000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00200000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
   .ANY (+XO)
  }
  RW_IRAM1 0x20021000 0x0005F000{  ; RW data
   .ANY (+RW +ZI)
  }
  RW_RAM1 0x20020000 UNINIT 0x00001000 {
    EventRecorder.o (+ZI)
  }
}

LR_ROM1 0x90000000 0x04000000  {
  ER_ROM1 0x90000000 0x04000000  {  ; load address = execution address
    Data.o (+RO)
    Blinky.o (+RO)
  }
}
```

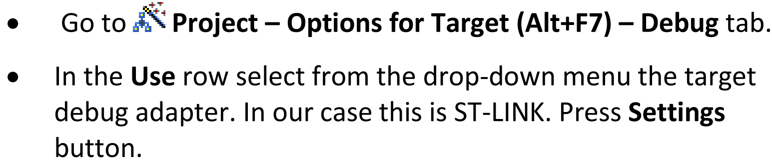
Refer to chapter [Scatter File Syntax](#) in the Linker User's Guide for more information about the scatter file.

Copyright © 2020 Arm Ltd. All rights reserved

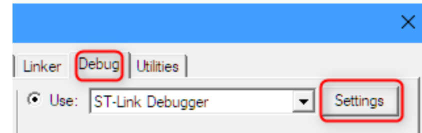
Program external flash memory

This chapter explains how to use the flash programming algorithm in MDK and program a firmware image that requires external memory flash.

1. Open a project that builds so that the code or constants are mapped to the external memory flash. Chapter **Access data and execute code from external flash** explains how to use external flash memory in a program and how to configure the project accordingly.
2. Add the flash programming algorithm to the debug driver settings.



- Go to **Project – Options for Target (Alt+F7) – Debug** tab.
- In the **Use** row select from the drop-down menu the target debug adapter. In our case this is ST-LINK. Press **Settings** button.

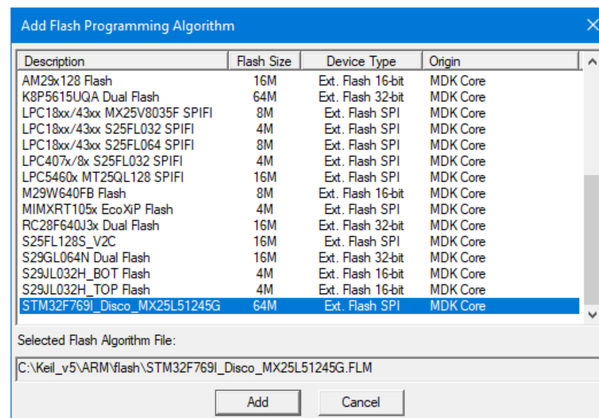


  Note that the settings are applied only for the selected debug driver and need to be repeated if a different debug adapter is used (for example ULINKpro).

- Go to the **Flash Download** tab.
- Press the **Add** button. It opens a list of flash programming algorithms available for selection.

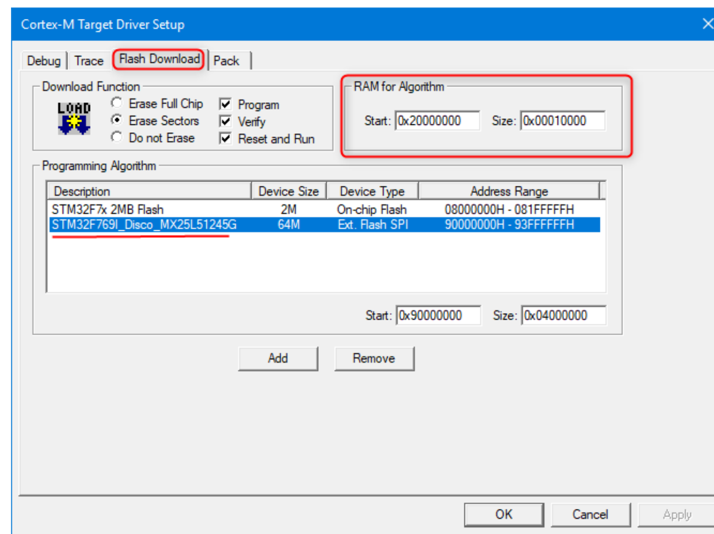
  The **Origin** column shows where the algorithm is located. Some of the algorithms are delivered with the Device Family Pack and others are part of MDK-Core installation.

  Find and select the target flash programming algorithm for your system. Its name matches the string specified in the **FlashDev.c**. When an algorithm is selected you can also see the path to it and verify that correct FLM file gets used. In our example we need to add algorithm *STM32F769I_Disco_MX25L51245G.*



- Press **Add**. The algorithm appears in the list of programming algorithms to be used by the debug adapter.
- Verify the values specified in the **RAM for Algorithm** section. Especially the **Size** value needs to be large enough to ensure there is sufficient space available for the algorithm. In our example it is set to **0x00010000**.

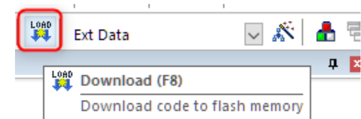
Copyright © 2020 Arm Ltd. All rights reserved



- Press **OK**.

See section [Target Driver Settings](#) in µVision User's Guide explaining details of debug driver setup for flash download.

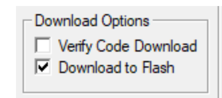
3. Go to **Project – Build Target (F7)** and **Download (F8)** the image file (.axf) to the target.



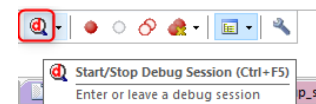
   - Observe status in **Build Output** window. The memory erase, programming and verification shall succeed.

```
Build Output
Load "C:\\MyPrograms\\STM32F769 Discovery Blinky\\Blinky\\Debug\\Blinky.axf"
Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 15:08:25
```

4. It is now also possible to debug the program.



   - Make sure that in the **Options for Target..** dialog, **Debug** tab, **Settings** for the target debugger the **Verify Code Download** is disabled.
   - Go to **Debug – Start/Stop Debug Session (Ctrl+F5).** As configured, the debugger stops in the *main()* function.



Summary

In this application note, we have explained how to use Keil MDK for programming an external flash memory device connected to an STM32 microcontroller. It provided step-by-step guidance for creating and then using custom flash programming algorithms for loading the firmware to an external flash memory device. Additionally, it showed an example demonstrating how to map the project data and code onto the external flash and how then to access it in the application.

References and Useful Links

[1] µVision User's Guide
[2] µVision Application Note 334: MDK Flash Download
[3] Using STM32CubeMX with Keil MDK projects

Copyright © 2020 Arm Ltd. All rights reserved