arm CHINA

Application Note

Using the QCU Driver of STAR

Version 1.0
Document ID: ACN-02202201-001
Non-Confidential

Release Information

Document History

Issue	Date	Confidentiality	Change
A	**25/02/2022**	**Non-Confidential**	**Initial release**

Contents

Non-Confidential Proprietary Notice **1**

1 About this document **4**
1.1 References 4
1.2 Terms and abbreviations 4
1.3 Conventions and feedback 4
1.3.1 Feedback on this product 5
1.3.2 Feedback on documentation 5
1.3.3 Other information 5

2 Introduction **6**
2.1 CMSIS 6
2.2 STAR DFP 6
2.3 QCU 6

3 Before you begin **8**

4 QCU driver introduction **9**

5 Using the example project **11**

6 Changing Flash device **15**

1 About this document

This Application Note is intended for developers, programmers, and users who use the Arm China STAR *Device Family Pack* (DFP). This Application Note gives you a basic understanding of the *QSPI Controller Unit* (QCU) driver in STAR and provides guidance on how to use the example project which is using the QCU driver.

1.1 References

Reference	Document number	Title
[1]	00903001_0100_00	Arm China Star Technical Reference Manual

1.2 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
CMSIS	Cortex Microcontroller Software Interface Standard
DFP	Device Family Pack
QCU	QSPI Controller Unit
QSPI	Quad Serial Peripheral Interface
MCC	Motherboard Configuration Controller

1.3 Conventions and feedback

The following describes the typographical conventions and how to give feedback:

Convention	Meaning
monospace	denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
monospace	denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
monospace italic	denotes arguments to commands and functions where the argument is to be replaced by a specific value.
monospace bold	denotes language keywords when used outside example code.
italic	highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for Arm China processor signal names.

1.3.1 Feedback on this product

If you have any comments and suggestions about this product, contact your supplier and give:

- Your name and company.
- The serial number of the product.
- Details of the release you are using.
- Details of the platform you are using, such as the hardware platform, operating system type and version.
- A small standalone sample of code that reproduces the problem.
- A clear explanation of what you expected to happen, and what actually happened.
- The commands you used, including any command-line options.
- Sample output illustrating the problem.
- The version string of the tools, including the version number and build numbers.

1.3.2 Feedback on documentation

If you have comments on the documentation, e-mail errata@armchina.com. Give:

- The title.
- The number, [Document ID Value], [Issue].
- If viewing online, the topic names to which your comments apply.
- If viewing a PDF version of a document, the page numbers to which your comments apply.
- A concise explanation of your comments.

Arm China also welcomes general suggestions for additions and improvements.

1.3.3 Other information

- Arm Glossary, http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html.

2 Introduction

2.1 CMSIS

The *Cortex Microcontroller Software Interface Standard* (CMSIS) is a vendor-independent hardware abstraction layer for microcontrollers.

The CMSIS defines generic tool interfaces and enables consistent device support.

The CMSIS provides:

- Simple software interfaces to processor and peripherals.
- A common approach to interface to peripherals, real-time operating systems, and middleware components.

2.2 STAR DFP

For CMSIS compliant toolchains such as Keil MDK and IAR EW, additional software components and support for microcontroller devices are provided by software packs.

A DFP is one of the CMSIS software packs. It indicates that a software pack contains support for microcontroller devices.

A DFP provides essential support for the software targets on a specific device, such as startup, system, linker scripts, and debug configuration.

The STAR processor is the first processor in the Arm China STAR series processor family.

STAR is a fully featured microcontroller class processor based on the Armv8-M mainline architecture with Arm TrustZone technology (depending on the actual core).

In STAR CMSIS DFP v1.3.0 and later, there are example projects of STAR application. These example projects can help you quickly build projects and run the application software and then get a better understanding of how to use STAR.

2.3 QCU

The STAR QCU provides a mechanism to load executing programs directly from external Flash memory instead of boot-up from embedding Flash memory. It provides a low-cost and simple method to implement SoC integration.

QCU provides the necessary functionality to a host to communicate with a serial Flash device through the SPI. The unit supports most common serial Flash device instructions, such as read, program, erase, and other custom instructions. The communication with Flash devices is used by commands, which includes five phases—Instruction, Address, Alternate byte, Dummy and Data. Any of these phases can be configured to be skipped, but at least one of them needs to be present.

QCU is highly flexible and can be configured to support a large number of SPI Flash memories. QCU also supports newer serial Flash devices with densities up to 256MB.

QCU is a specialized communication interface targeting single, dual, or quad SPI Flash memories.

QCU can work in one of the following modes:

- **Direct Read Access mode**: The external Flash memory is mapped to the device address space and is seen by the system as if it was an internal memory. Direct Read Access mode can be used to both access and directly execute code from external Flash memory, and it supports Flash memory XIP mode. After power-on, QCU changes to default Direct Read Access mode to boot up. The operation mode is accessed through AHB-Bus. When in Direct Read mode, any other modes can be inserted at any time.
- **Indirect mode**: All the operations are performed using the registers. This mode allows software to access the internal TX FIFO and RX FIFO directly. The level of FIFO can be configurable. It is used to access the volatile and non-volatile configuration registers, the legacy SPI status registers, other status and protection registers and the Flash ROM content. It is recommended that this mode is used to erase and configure the serial Flash device. When a transaction request is from Q-AHB, the transaction will be waiting until exiting the current Indirect mode into Direct mode.
- **Inactive mode**: This mode is used to disable QSPI-Bus. It offers a *clean* state to set up the QSPI's related registers, such as division clock index, command mode, and command type. When a transaction request is from Q-AHB, an error is returned.

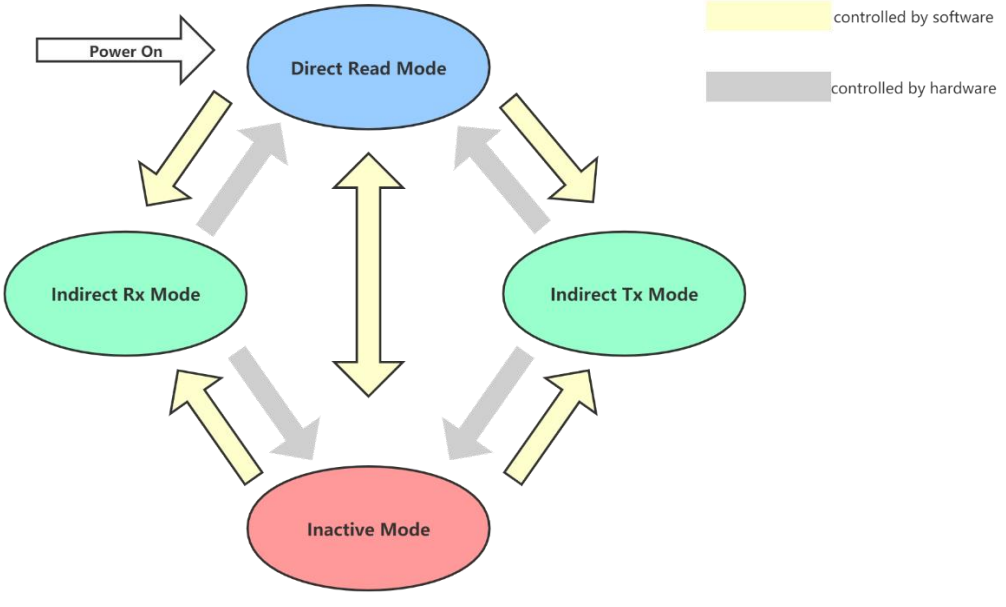


Figure 2-1 QCU modes

The initial state is Direct Read Mode after power-on or cold reset. You can configure QSPI to other states by setting the control register. Indirect Mode is a special state. When the transaction in Indirect Mode is done, hardware will be responsible for changing the state back to the previous one (Inactive or Direct Read, depending on the mode which it enters from). The status register is used to indicate whether the indirect transaction is done.

The Direct Read Mode and Indirect Mode use different sets of registers to construct respective SPI communication. These settings take effect only after the control register is configured, which means that you must confirm the target mode, Direct or Indirect mode, then write the corresponding registers. When the control register is configured, the QCU will load Mode-specific parameters according to the value of the control register.

3 Before you begin

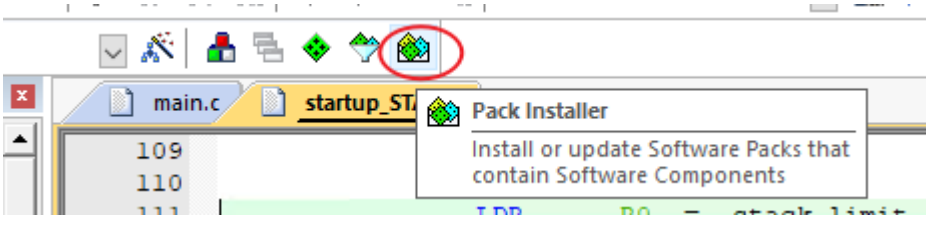
The example project of the application software runs on an MPS3 FPGA board.

Before using the example project, you need to:

- Ensure that you have an MPS3 FPGA board and have a STAR-based device implemented with QCU on the board.
- Check the STAR CMSIS DFP version.

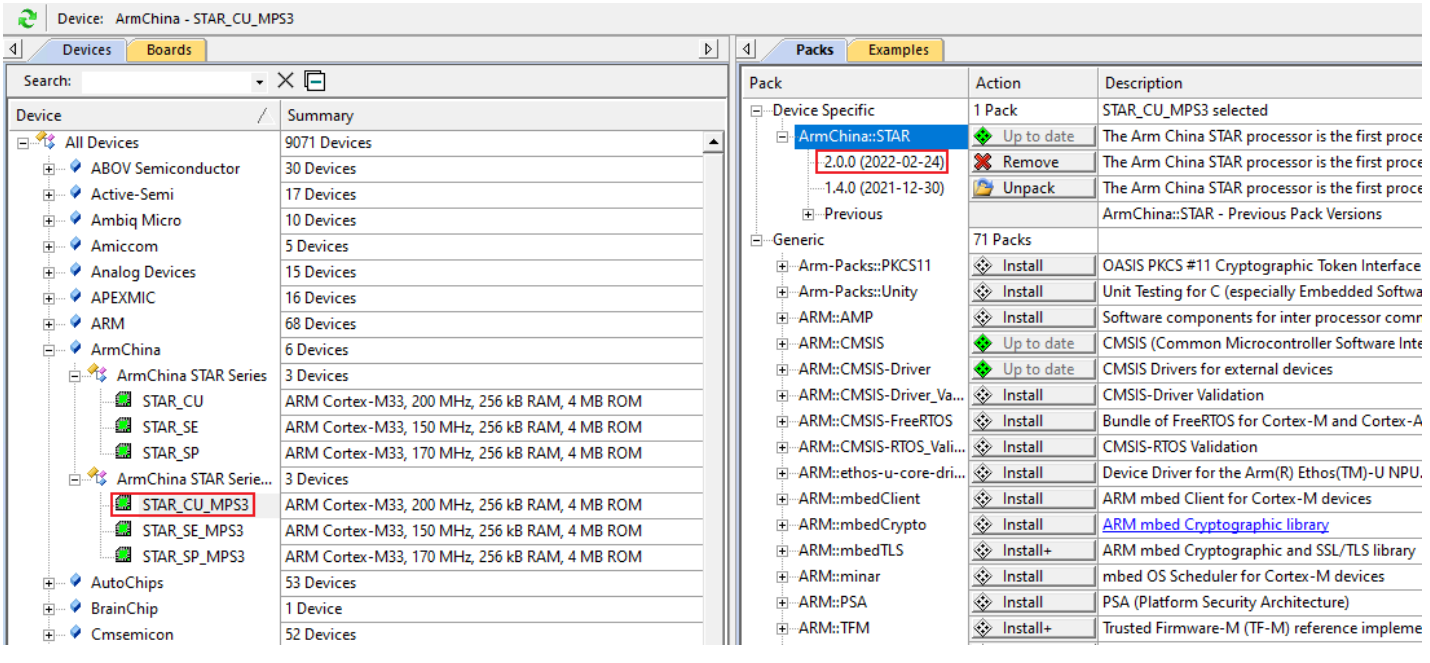
To check the STAR CMSIS DFP version:

1. Start MDK.
2. On the toolbar, click the **Pack Installer** icon.



3. On the **Devices** tab, select a device (for example, STAR_CU_MPS3) and check the version of the installed pack.

   As shown in the following figure, the version of the ArmChinaSTAR pack should be 2.0.0 or later.



4 QCU driver introduction

The QCU driver is a set of functions in form of software snippets which can be used to drive STAR-internal (embedded) QSPI controller to access the STAR-external serial interface device typically such as a serial Flash device. The following figure shows the software architecture with the QCU driver:

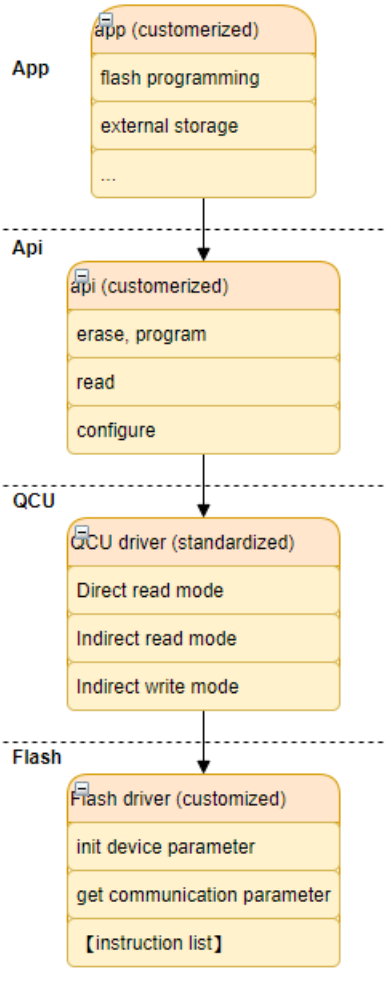


Figure 4-1 Software architecture with the QCU driver

The operations supported by the QCU driver are as follows:

- **Initiate**

  int32_t QCU_Init(GetInitParameter_t GetInitParameter)

  Initialize the control register of QCU according to the device, such as clock prescaler, Flash memory size, and SCK mode.

- **Indirect write**

  int32_t QCU_Write(uint8_t cmd, QCU_CommData_Typedef *comm_data, bool xip_mode, GetParameter_t GetParameter)

Send the data to the device or configure registers of the device in QCU indirect mode. In indirect write mode, all operations are performed using the registers. Software is allowed to access serial Flash memory through the internal TX FIFO directly.

- **Indirect read**

  int32_t QCU_Read_Indirect(uint8_t cmd, QCU_CommData_Typedef *comm_data, GetParameter_t GetParameter)

  Read the memory data or status of the device using the QCU registers in indirect mode. In indirect read mode, software is allowed to access serial Flash memory through the internal RX FIFO directly.

- **Direct read**

  int32_t QCU_Read_Direct(uint8_t cmd, bool xip_mode, GetParameter_t GetParameter)

  This function configures the direct read mode registers of QCU. After calling the QCU_SetOPMode() function to change the OPMODE, the registers will be updated. The external Flash memory is mapped to the device address space and is seen by the system as if it was an internal memory.

- **Set OPMODE**

  int32_t QCU_SetOPMode(uint8_t op_mode, bool xip_mode)

  This function sets the operation mode of QCU. You need to ensure that the parameters of registers such as *OMCR* and *RMCR* are configured correctly before calling this function.

- **Get OPMODE**

  uint8_t QCU_GetOPMode(void)

  This function gets the operation mode of QCU.

Before you proceed, ensure that the following preparations are completed:

- Familiar with QCU operation mode.
- Familiar with the command sequence and features of the serial Flash device that you are using.
- Coding Flash driver to provide the operation parameters for the QCU driver.

After that, you can communicate with the Flash device.

5 Using the example project

An example project, which demonstrates how to use the QCU driver, is available in STAR DFP v2.0.0.

In this example, the hardware platform is MPS3, and on which an MCU system with STAR-r1 inside is implemented. The image of the example software code will be pre-loaded into Flash by the MCC of the MPS3 board.

STAR QCU hardware is in Direct Read Access mode by default. Therefore, after power-up, the CPU can fetch the code from Flash through the QCU which will send the default read command to the device, then the example software has chance to configure the QCU and Flash to High-Speed read SQI mode with XIP. The example will show the performance difference between default mode and High-Speed read SQI mode with XIP running the matrix-matrix multiplication. Figure 5-1 shows the workflow of the example project:

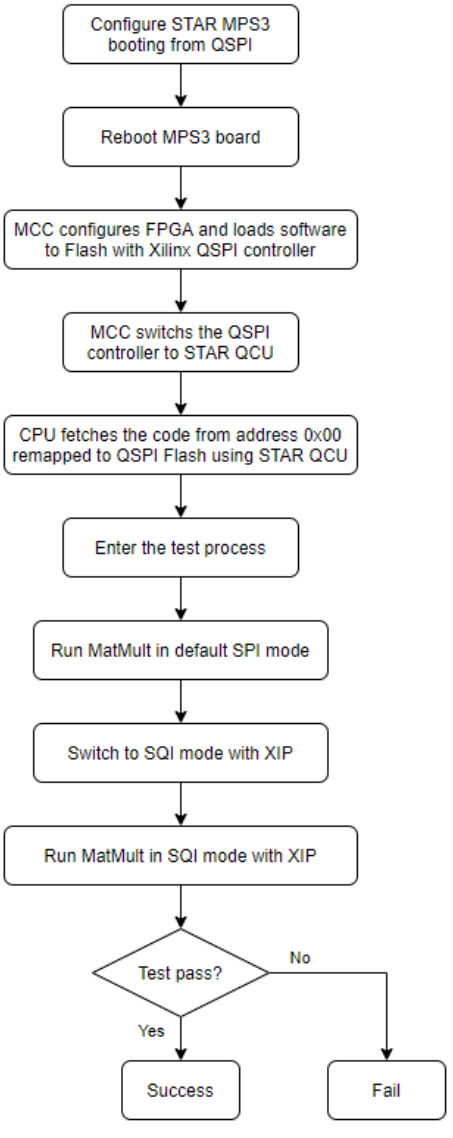
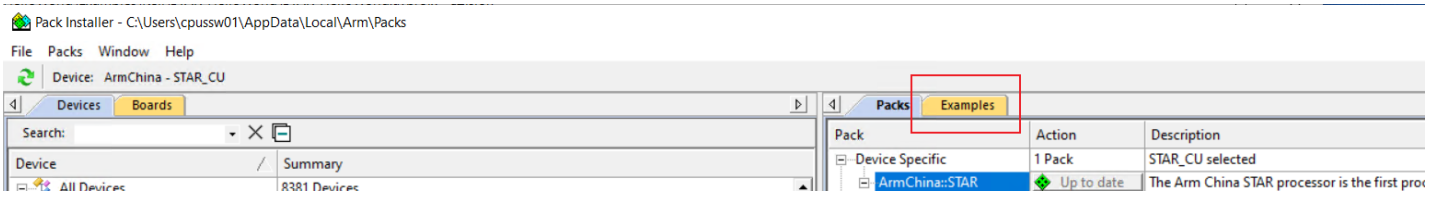


Figure 5-1 Workflow of the example project

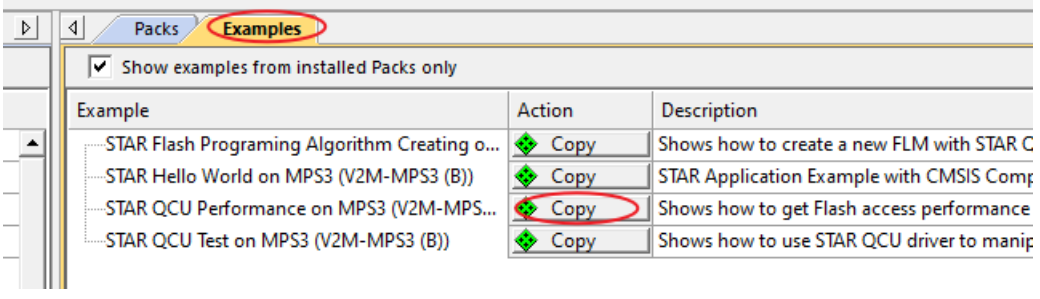
Take the following steps to use this example project:

1. In the Pack Installer, click the **Examples** tab.



2. On the **Examples** tab, select the example that you want to use and click **Copy**.

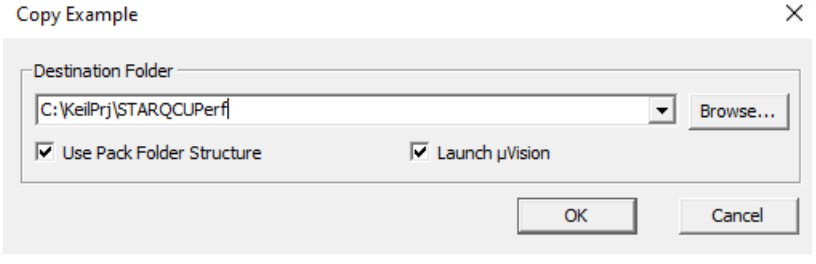
   In STAR CMSIS DFP v2.0.0, there is an example named **STAR QCU Performance on MPS3** available. This example project will load the code into the Flash chip on MPS3, and the CPU fetches the code from Flash using QCU.



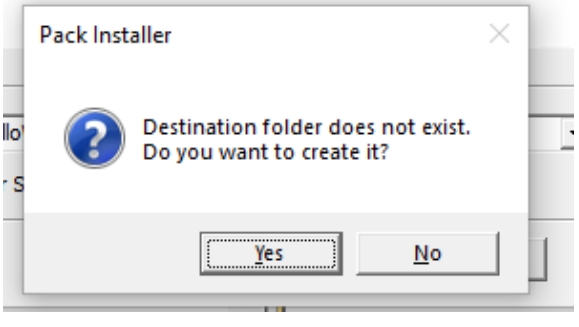
   Note:

   Sometimes the Copy button is disabled in gray because updates are needed for certain packs. You can check the progress bar to confirm this situation. When the progress reaches 100%, the Copy button will be enabled.

3. In the **Copy Example** dialog box that appears, specify the destination folder path to save the project, and then click **OK**.



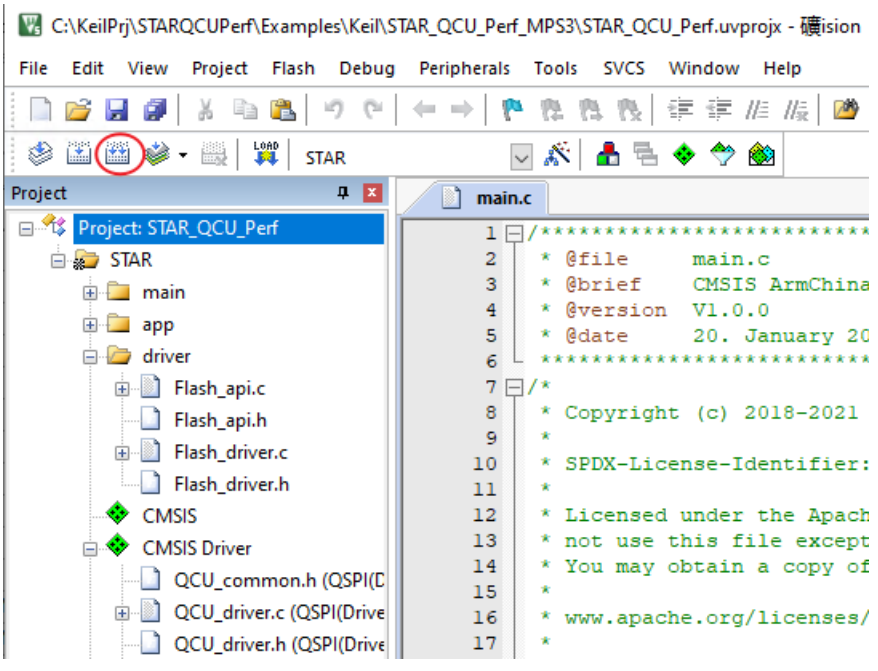
   If the destination folder does not exist, click **Yes** to create it.



   A project is created in the destination folder.

The µVision will start automatically and open the created project. In the Project pane, you can see all the required files.

4. Click the **Rebuild** icon to recompile and build the project.



5. Copy the STAR_QCU.axf file from $C:\KeilPrj\STARQCUPerf\Objects\ to $V2M-MPS3\SOFTWARE of the MPS3 SD card.

6. Configure the text file $F:\MB\HBI0309C\AN524\an524_v2.txt to let STAR_QCU.axf be able to boot from Flash.

```
REMAP: QSPI
REMAPVAL: 1
XIPMODE: SPI
TOTALSYSCONS: 4
SYSCON: 0x000 0x00000001
SYSCON: 0x008 0x00000000
SYSCON: 0x018 0x00000000
SYSCON: 0x01c 0x00000000
```

7. Reboot the MPS3 board.

8. Run the built software.
   You can see the running log in the UART terminal window and observe the difference between default SPI mode and quad I/O with XIP mode.

```
COM14 - PuTTY

==========TEST QCU XIP performance start==========

QCU driver initialization...

==========TEST: Default SPI mode run matrix multiplciation==========
Run the calculation and counting...
Matrix multiplciation count is 2772444 in SPI mode without XIP.

==========TEST: SQI mode with XIP run matrix multiplciation==========
Change to SQI direct read mode with XIP...
Run the calculation and counting...
Matrix multiplciation count is 733449 in SQI mode with XIP.

==========TEST QCU XIP performance finish==========
QCU change to SPI mode.

STAR: QCU with XIP test is successful!
```

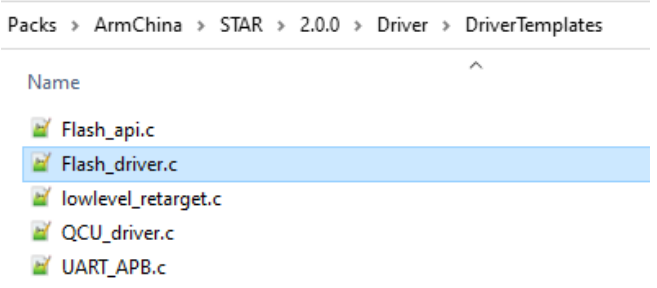
Based on this project, you can start to use the QCU driver and try more in your STAR-based application software.

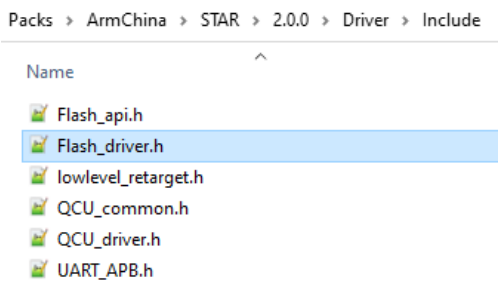
6 Changing Flash device

The QCU driver can work in three modes to suit different Flash devices including indirect write, indirect read, and direct read mode. It supports most operations of Flash, such as Flash configuration, erase chip/sector, program page, read register, and enable XIP mode.

With this example project, you can change the Flash device to another one by completing the following steps:

1. Adapt the Flash instruction list, related macros (such as command, busy, and status bit), and enumeration in Flash driver files.






```
/* Macros declaration */
/* Change macros according to your device */
#define BUSY_BIT_POS 0x01
#define WEL_BIT_POS  0x02
#define IOC_BIT_POS  0x02
#define BUSY_TIMEOUT 0xFFFF
#define FLASH_ADDR_MAX 0x800000
```

```
#define CMD_WREN                    (0x06U)
#define CMD_PP                      (0x02U)
#define CMD_EQIO                    (0x38U)
#define CMD_RSTQIO                  (0xFFU)
#define CMD_RDSR1                   (0x05U)
#define CMD_SDOR_INDIRECT           (0x3BU)
#define CMD_SQIOR_INDIRECT          (0xEBU)
#define CMD_READ_DIRECT             (0x03U)
#define CMD_SDIOR_DIRECT            (0xBBU)
#define CMD_SQIOR_DIRECT            (0xEBU)

static const QCU_Comm_Typedef Instr_List[]={
/* WREN */              {DDRMODE_DISABLE, IDMODE_WRITE, XIP_BIT_CLEAR, NUMDC_DEFAULT, RXDLY_DEFAULT, DSIZE_8BITS, DMODE_NO_DATA, \
                         ABSIZE_ALT_BYTES_8BITS, ABMODE_NONE, ADSIZE_8BITS, ADMODE_NONE, IMODE_SINGLE_LINE, CMD_WREN, ALT_BYTE_DEFAULT}, \
/* PP */                {DDRMODE_DISABLE, IDMODE_WRITE, XIP_BIT_CLEAR, NUMDC_DEFAULT, RXDLY_DEFAULT, DSIZE_8BITS, DMODE_SINGLE_LINE_DATA, \
                         ABSIZE_ALT_BYTES_8BITS, ABMODE_NONE, ADSIZE_24BITS, ADMODE_SINGLE_LINE, IMODE_SINGLE_LINE, CMD_PP, ALT_BYTE_DEFAULT}, \
/* EQIO */              {DDRMODE_DISABLE, IDMODE_WRITE, XIP_BIT_CLEAR, NUMDC_DEFAULT, RXDLY_DEFAULT, DSIZE_8BITS, DMODE_NO_DATA, \
                         ABSIZE_ALT_BYTES_8BITS, ABMODE_NONE, ADSIZE_8BITS, ADMODE_NONE, IMODE_SINGLE_LINE, CMD_EQIO, ALT_BYTE_DEFAULT}, \
/* RSTQIO */            {DDRMODE_DISABLE, IDMODE_WRITE, XIP_BIT_CLEAR, NUMDC_DEFAULT, RXDLY_DEFAULT, DSIZE_8BITS, DMODE_NO_DATA, \
                         ABSIZE_ALT_BYTES_8BITS, ABMODE_NONE, ADSIZE_8BITS, ADMODE_NONE, IMODE_SINGLE_LINE, CMD_RSTQIO, ALT_BYTE_DEFAULT}, \
```

Note that the commands are divided into three types according to QCU modes and the command enumeration must be consistent with the instruction list.

```
/* The order of enum must be the same as Instr_List */
typedef enum {
    WRITE_ENABLE = 0,
    PROGRAM_PAGE,
    ENA_QIO,
    RST_QIO,

    INDIRECT_WRITE_CMD_MAX
} Indirect_Write_CMD;

typedef enum {
    READ_SR1 = INDIRECT_WRITE_CMD_MAX,
    SDOR_IND,
    SQIOR_IND,

    INDIRECT_READ_CMD_MAX
} Indirect_Read_CMD;

typedef enum {
    READ_DIR = INDIRECT_READ_CMD_MAX,
    SDIOR_DIR,
    SQIOR_DIR,

    DIRECT_READ_CMD_MAX
} Direct_Read_CMD;
```

2. Adapt Flash API functions.

   Although most of the properties are similar, different features within various types of Flash devices exist, such as the number of status registers, the way to enter XIP mode, and supportive instruction sequence.

   For example, W25Q64JV has three status registers which can be read continuously or respectively. N25Q032A should be configured through registers to activate or terminate XIP mode.

   Therefore, you need to check the features of the Flash and adapt some Flash API functions. You can also add the new functions in the Flash API layer to support more Flash operations based on the QCU driver.

   In a word, you can design new Flash API functions calling the QCU driver for different Flash devices with corresponding adaptation of the Flash driver layer.

Take W25Q64JV Flash for example, there are two Write Enable instructions 06H and 50H for the non-volatile and volatile status register bits. In this case, the instruction list in Flash_driver.c is as follows:

```
#define CMD_WREN                    (0x06U)
#define CMD_WREN_VOLA               (0x50U)
static const QCU_Comm_Typedef Instr_List[]={
/* WREN */              {DDRMODE_DISABLE, IDMODE_WRITE, XIP_BIT_CLEAR, NUMDC_DEFAULT, RXDLY_DEFAULT, DSIZE_8BITS, DMODE_NO_DATA, \
                         ABSIZE_ALT_BYTES_8BITS, ABMODE_NONE, ADSIZE_8BITS, ADMODE_NONE, IMODE_SINGLE_LINE, CMD_WREN, ALT_BYTE_DEFAULT}, \
/* WREN_VOLA */         {DDRMODE_DISABLE, IDMODE_WRITE, XIP_BIT_CLEAR, NUMDC_DEFAULT, RXDLY_DEFAULT, DSIZE_8BITS, DMODE_NO_DATA, \
                         ABSIZE_ALT_BYTES_8BITS, ABMODE_NONE, ADSIZE_8BITS, ADMODE_NONE, IMODE_SINGLE_LINE, CMD_WREN_VOLA, ALT_BYTE_DEFAULT}, \
```

The enumeration type definition in Flash_driver.h is as follows:

```
/* The order of enum must be the same as Instr_List */
typedef enum {
    WRITE_ENABLE = 0,
    WRITE_ENABLE_VOLA,
```

In addition, Write Enable for the volatile status register instruction (50H) will not set the *Write Enable Latch* (WEL) bit.

The default Write Enable function is as follows:

```
/**
  \fn          int32_t Api_WriteEn (void)
  \brief       Configure the Write-Enable bit of flash to allow write operations to occur.
  \return      \ref execution_status
*/
int32_t Api_WriteEn(void)
{
    int32_t ret = ARM_DRIVER_OK;
    uint8_t val = 0;

    Api_CfgFlash(WRITE_ENABLE, QCU_COMM_NO_DATA, NULL, False);
    ret = Api_WaitBusy(False);
    if (ARM_DRIVER_OK != ret) {
        return ret;
    }
    val = Api_ReadSR(False);
    if (!(val & WEL_BIT_POS)) {
        /* Failed WREN configuration */
        return ARM_DRIVER_ERROR;
    }

    return ARM_DRIVER_OK;
}
```

A new Flash API function should be added without WEL bit check for Write Enable (50H) as follows:

```
/**
  \fn          int32_t Api_WriteEnVola (void)
  \brief       Write-Enable configuration for write the volatile status register.
  \return      \ref execution_status
*/
int32_t Api_WriteEnVola(void)
{
    int32_t ret = ARM_DRIVER_OK;
    uint8_t val = 0;

    Api_CfgFlash(WRITE_ENABLE_VOLA, QCU_COMM_NO_DATA, NULL, False);
    ret = Api_WaitBusy(False);
    if (ARM_DRIVER_OK != ret) {
        return ret;
    }

    return ARM_DRIVER_OK;
}
```

Then you can call this function when writing the status register to change the bit values of volatile status register.

Similarly, you can make adaption in Flash and QCU drivers for other features that are not covered in the example project.