Installing a NIO driver in MQX for Kinetis SDK

By: Technical Information Center

MQX for KSDK uses default drivers (and HAL functions) just like in bare-metal implementation, however, MQX also provides a uniform way to communicate with Input / Output (I/O) drivers by using a NIO driver, this NIO driver is a wrapper for I/O drivers in order to use POSIX standard. The main reason to use these I/O drivers is to hide the driver-specific functions and use the same APIs (POSIX standard) for all drivers. These APIs consist of basic I/O operations such as open, close, read and write.

An I/O Subsystem (NIO driver in MQX for KSDK) allows to install dynamically different I/O device drivers, after which, any task can open and use them. By default, NIO Serial Driver is one of the drivers implemented in the NIO framework, it means that user can access to serial driver services using top level NIO API, however, it is not limited to this driver only. User can install more NIO drivers and access to different peripherals services using same NIO APIs.

This document describes the way to create a basic NIO driver, the steps needed to install it and then use it with basic I/O operations like open, close, read, write, etc.

FRDM-K64F and Kinetis SDK 1.3 are used to create a SPI NIO driver, however, it is not limited to this specific board, SDK version or even this peripheral.

1 Requirements.

1.1 Install KDS 3.0.0 (Kinetis Design Studio), you can download from www.nxp.com/kds

1.2 Install KSDK 1.3.0 (Kinetis Software Development Kit), you can download from www.nxp.com/ksdk

1.3 Install ***'KSDK_1.3.0_Eclipse_Update'*** you can find the update in ***'<KSDK_1_3_PATH>\tools\eclipse_update'***. The instruction to make the updates are described in chapter 2 of ***'<KSDK_1_3_PATH>\doc\rtos\mqx\MQX RTOS IDE Guides\MQX-KSDK-KDS-Getting-Started.pdf'***

2 Creating source and header files for new NIO Driver

2.1 In *<KSDK_1_3_PATH>\rtos\mqx\mqx\source\nio\drivers* path, it is shown all available NIO drivers, among these drivers, there is a nio_null folder that was created to serve as template for new drivers, so, copy and paste this folder and rename it along with source and header files. In this case, folder was renamed as nio_spi, header and sources files were renamed as nio_spi.h and nio_spi.c respectively.

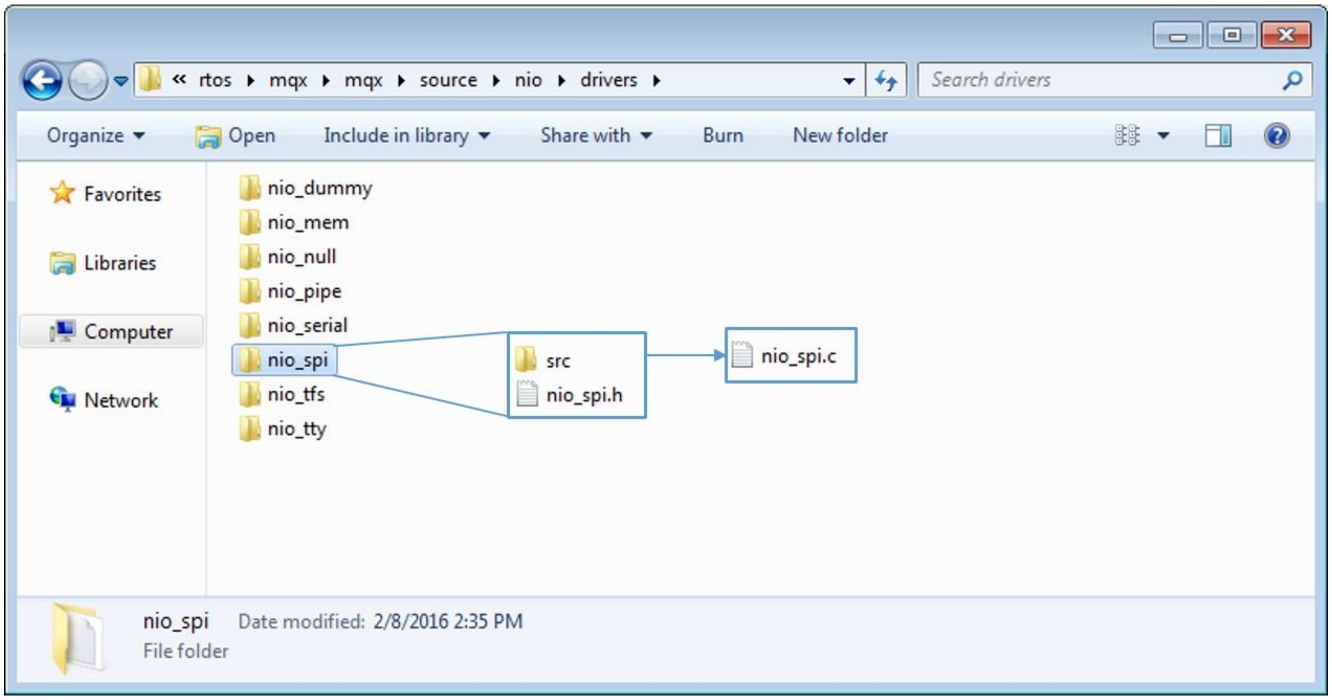
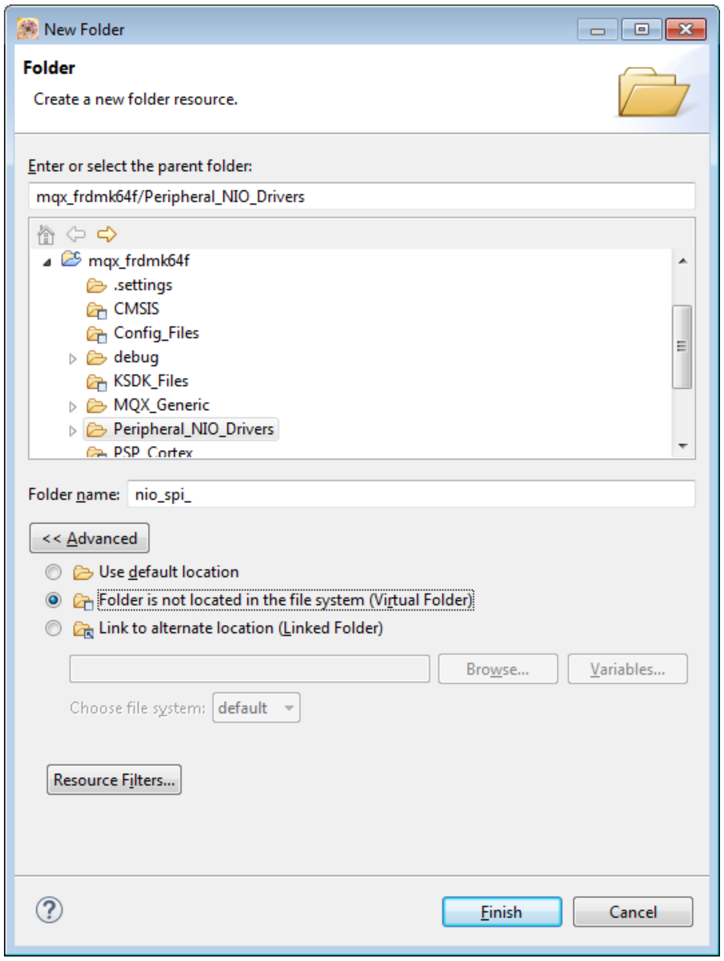


Figure 1- New NIO's folder and files

2.2 Now, add these source and header files to *mqx_<board>* library. In this case, ***mqx_frdmk64f*** is used, so import the project located at: *<KSDK_1_3_PATH>\rtos\mqx\mqx\build\<IDE>\mqx_frdmk64f.* Once it is opened, create a virtual folder into ***mqx_frdmk64f > Peripheral_NIO_Drivers*** and include (link) header and source files for new NIO driver.

Note: For creating a virtual folder, just right click on ***Peripheral_NIO_Drivers*** folder and then select the ***New > Folder*** option. A *New Folder* window will appear, just give a name for this folder and press the **Advanced >>** button. Check the ***Folder is not located in the file system (Virtual Folder)*** option just as shown below:



New Folder

Folder

Create a new folder resource.

Enter or select the parent folder: mqx_frdmk64f/Peripheral_NIO_Drivers

- mqx_frdmk64f
  - .settings
  - CMSIS
  - Config_Files
  - debug
  - KSDK_Files
  - MQX_Generic
  - Peripheral_NIO_Drivers
  - PSP_Cortex

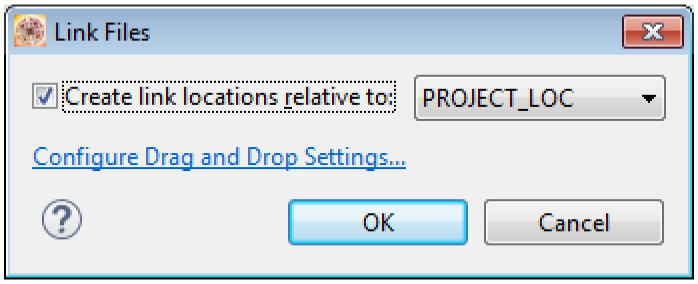
Folder name: nio_spi_

<< Advanced

( ) Use default location
(●) Folder is not located in the file system (Virtual Folder)
( ) Link to alternate location (Linked Folder)

Figure 2 - Creating a virtual folder

Once virtual folder has been created, add NIO source and header files (nio_spi.c and nio_spi.h), this can be done by dragging each file from explorer window into the virtual folder, a new window will appear showing that file will be linked by using PROJECT_LOC variable:



Link Files

[✓] Create link locations relative to: PROJECT_LOC

Figure 3 - Link Files

Finally, project will be seen as follows:

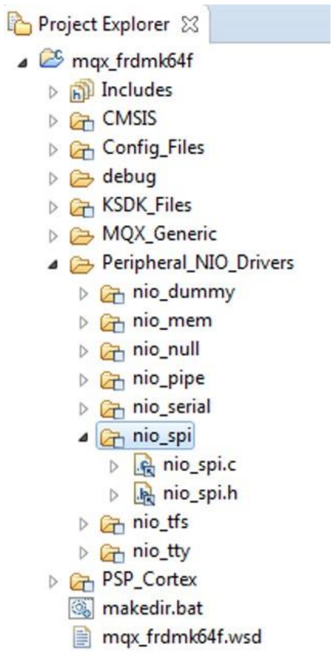
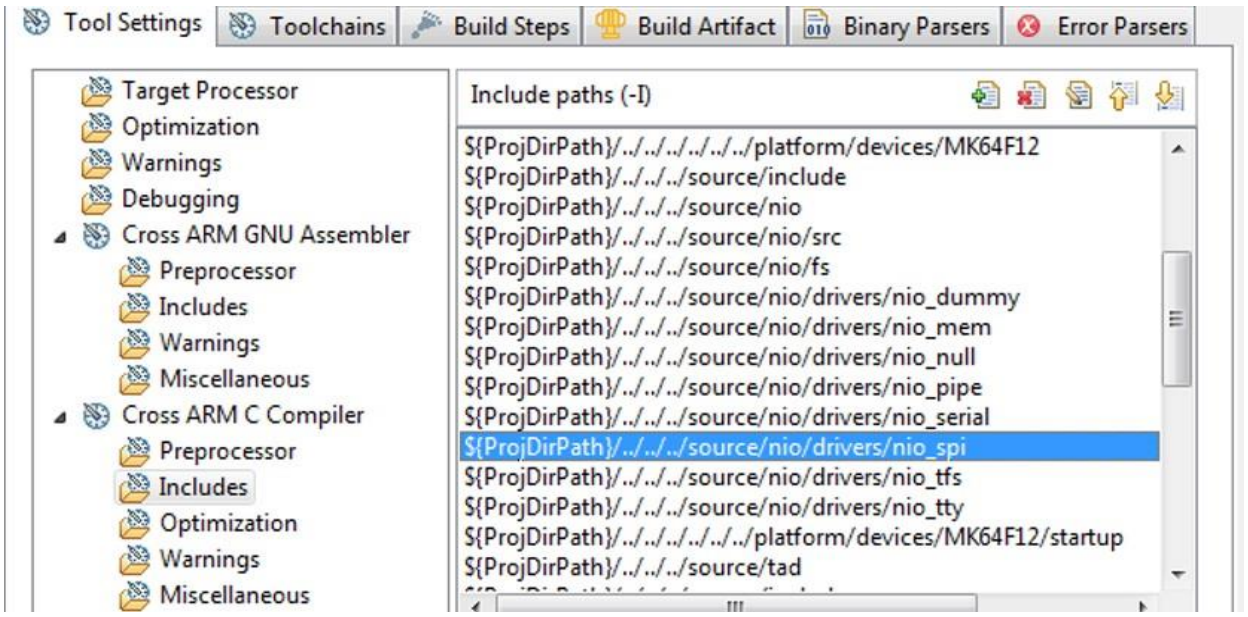


Figure 4 - Add New NIO files into MQX library

2.3 Go to Project's properties (right click on mqx_frdmk64f project) and then to *C/C++ Build > Settings > Cross ARM C Compiler > Includes* and add this path:

${ProjDirPath}/../../../source/nio/drivers/nio_spi



```
${ProjDirPath}/../../../../../platform/devices/MK64F12
${ProjDirPath}/../../../source/include
${ProjDirPath}/../../../source/nio
${ProjDirPath}/../../../source/nio/src
${ProjDirPath}/../../../source/nio/fs
${ProjDirPath}/../../../source/nio/drivers/nio_dummy
${ProjDirPath}/../../../source/nio/drivers/nio_mem
${ProjDirPath}/../../../source/nio/drivers/nio_null
${ProjDirPath}/../../../source/nio/drivers/nio_pipe
${ProjDirPath}/../../../source/nio/drivers/nio_serial
${ProjDirPath}/../../../source/nio/drivers/nio_spi
${ProjDirPath}/../../../source/nio/drivers/nio_tfs
${ProjDirPath}/../../../source/nio/drivers/nio_tty
${ProjDirPath}/../../../../../platform/devices/MK64F12/startup
${ProjDirPath}/../../../source/tad
```

Figure 5 - Add path for NIO driver

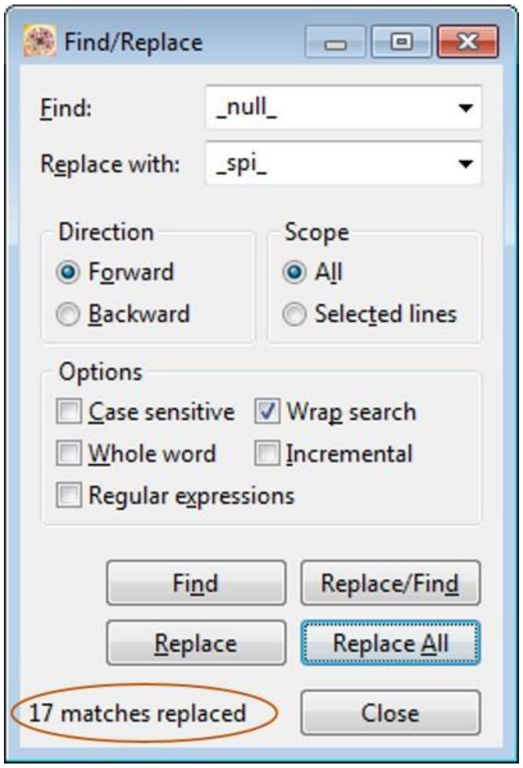
2.4 At this point, we are ready to populate the open, read, control and write functions that are defined in source and header files.

3 Create a NIO Driver

Next steps will guide user to create a NIO driver from the empty template (NIO_NULL), these steps are focused on creating a SPI driver, so naming, functionality and more elements are referred to SPI module, however, user can adapt it to his/her own needs. For full reference of this NIO_SPI driver, please check ***Appendix A: Source and Header files Reference***

3.1 In source file (nio_spi.c) include nio_spi.h header file (`#include "nio_spi.h"`).

3.2 In same source file (nio_spi.c) empty functions for open, read, write and more are already defined, just rename them according to peripheral that is used, for example, change **`nio_null_open`** for **`nio_spi_open`**, **`nio_null_read`** for **`nio_spi_read`** and so on.
To make it easier and faster, go to *Edit* tab and select *Find/Replace (CTRL + F)*, fill the new windows as follows and select the *Replace All* button:



Find: _null_

Replace with: _spi_

Direction: [✓ Forward] Backward

Scope: [✓ All] Selected lines

Options: Case sensitive [✓ Wrap search] Whole word Incremental Regular expressions

17 matches replaced

Figure 6 - Find/Replace Window

After selecting *Replace All* option, 17 matches should be replaced, just as shown in previous figure.

3.3 After renaming these functions, rename the **NIO_DEV_FN_STRUCT** structure as well, change ***nio_null_dev_fn*** for ***nio_spi_dev_fn*** and fill the structure with every function that will be supported for this driver, this structure is located at the end of the nio_spi source file.

```
const NIO_DEV_FN_STRUCT nio_spi_dev_fn = {
    .OPEN = nio_spi_open,
    .READ = nio_spi_read,
    .WRITE = nio_spi_write,
    .LSEEK = NULL,
    .IOCTL = nio_spi_ioctl,
    .CLOSE = nio_spi_close,
    .INIT = nio_spi_init,
    .DEINIT = nio_spi_deinit,
};
```

For every non-supported function, fill it with NULL, for example, the seek function will not be supported in SPI driver but all others will be.

Remember to modify the name for this structure in header file (nio_spi.h) as well:

```
extern const NIO_DEV_FN_STRUCT nio_spi_dev_fn;
```

3.4 It is necessary to create a device context structure that will include all buffers, synchronization methods, variables and more elements that will be accessed for every driver's function. These 2 structures should be included in nio_spi.c file and they are defined in this example as follows:

```
typedef struct {
    uint8_t *buff;      /* Data buffer */
    uint32_t size;      /* Buffer size */
}NIO_SPI_BUFFER_STRUCT;
typedef struct {
    mutex_t lock;                           /* Mutex for SPI driver*/
    uint8_t instance;                       /* Which SPI instance is used */
    dspi_device_t *spi_device_context;      /* Baud rate and data format context */
    dspi_master_state_t *spi_master_state;  /* Settings that are needed in master mode */
    NIO_SPI_BUFFER_STRUCT tx_buff;          /* Buffer structure for transmitter */
    NIO_SPI_BUFFER_STRUCT rx_buff;          /* Buffer structure for receiver */
}NIO_SPI_DEV_CONTEXT_STRUCT;
```

Some elements from these structures might be defined in other header file(s), like `dspi_device_t` and `dspi_master_state_t` that are defined in fsl_dspi_master_driver header file, so, Include all the header files that are needed, for example in nio_spi.h is added: `#include "fsl_dspi_master_driver.h"`

3.5 In header file (nio_spi.h), define an initialization structure that will be passed to INIT function in order to initialize the module. This structure may include definitions to configure the module like bus settings (baud rate, clock settings, bit format) and other parameters such as module instance and more.

```
typedef struct
{
    uint32_t SPI_INSTANCE;                      /*!< Number of used SPI peripheral */
    uint32_t BAUDRATE;                          /*!< SPI baud rate */
    uint8_t BITS_PER_FRAME;                     /*!< number of bits, 8-bit (default) or 16-bit */
    dspi_clock_phase_t CLOCK_PHASE;             /*!< Clock phase */
    dspi_clock_polarity_t CLOCK_POLARITY;       /*!< Clock polarity */
    uint32_t IRQ_PRIORITY;                      /*!< irq priority for SPI */
    uint32_t RX_BUFF_SIZE;                      /*!< Size of rx buffer.*/
    uint32_t TX_BUFF_SIZE;                      /*!< Size of tx buffer.*/
    bool CHIP_SELECT_CONTINUOUS;                /*!< DSPI configuration */
    bool SCK_CONTINUOUS;                        /*!< DSPI configuration */
    dspi_delay_type_t DELAY_TYPE;               /*!< Delay type for SPI */
    uint32_t DELAY_VALUE_NANO_SEC;              /*!< Delay value for SPI */
} NIO_SPI_INIT_DATA_STRUCT;
```

Compared to device context structure, this initialization structure must be included in header file instead of source file in order to be "visible" for other files.

3.6 In source file (nio_spi.c) populate every function that will be supported for this driver. Do it according to driver's functionality, a basic SPI driver is created in order to serve as guidance for other user's drivers with more robustness and complexity.

3.6.1 *INIT* function

This function receives 3 parameters:

- **void** *init_data: Pointer to initialization structure that was declared in 3.5
- **void** **dev_context: pointer to pointer to store device context that will be initialized inside this function.
- **int** *error: Output pointer to return error codes.

Basically, it handles all the initialization data for this driver, it includes:

- Allocate memory for device context structure and any other element that will be used for this driver:
- Initialize any synchronization method like semaphores, mutexes, etc.
- Install interrupts
- Configure the module according to init_data structure that is passed to this function, it should be casted to NIO_<MODULE>_INIT_DATA_STRUCT type that was declared in 3.5.
- Save the device context using the double pointer that was passed to this function. This way, MQX kernel could will save the context for this driver.

For detailed information, consult Appendix A: Source and Header files Reference

3.6.2 *DEINIT* function

This function receives 3 parameters:

- `void` *dev_context: pointer to refer to device context for this driver.
- `int` *error: Output pointer to return error codes.

In this function, De-initialize the module and free all memory that was used.

For detailed information, consult Appendix A: Source and Header files Reference

3.6.3 *IOCTL* function

This function receives 5 parameters:

- `void` *dev_context: pointer to refer to device context for this driver.
- `void` *fp_context: pointer to refer to device data context per FP (File Pointer). This is allocated by low level drivers in open function.
- `int` *error: Output pointer to return error codes.
- `unsigned long int` request: Request to be processed.
- `va_list` ap: Variable arguments that will be used for different requests.

This function will handle all supported features to configure the driver. Usually the GET and SET commands are implemented.

In order to use available IOCTL definitions, please include ioctl.h file (in nio_spi.h file): `#include "ioctl.h".`

For detailed information, consult Appendix A: Source and Header files Reference

3.6.4 *OPEN* function

This function receives 5 parameters:

- `void` *dev_context: pointer to refer to device context for this driver.
- `const char` * dev_name: string that makes reference for this driver.
- `int` flags: Flags to indicate the mode in which driver will be opened (r, r+, w, …)
- `void` **fp_context: Pointer to pointer to store device data context in the File Pointer list.
- `int` *error: Output pointer to return error codes.

The function includes all the logic that allow the driver to be opened to read/write or both functions in different modes (read only, write only, read and write, etc.). It can also include a variable for counting the times that driver has been opened.

In this case, open function is used to configure baud rate and delays values for SPI module, however, it could be left empty as shown in nio_serial.c.

For detailed information, consult Appendix A: Source and Header files Reference

3.6.5 ***CLOSE*** function

This function receives 3 parameters:

- `void` *dev_context: pointer to refer to device context for this driver.
- `void` *fp_context: pointer to refer to device data context per FP (File Pointer). This is allocated by low level drivers in open function.
- `int` *error: Output pointer to return error codes.

It is the counterpart for open function. In this case, this function is left empty.

For detailed information, consult Appendix A: Source and Header files Reference

3.6.6 ***READ*** and ***WRITE*** functions.

These functions receive 5 parameters each:

- `void` *dev_context: pointer to refer to device context for this driver.
- `void` *fp_context: pointer to refer to device data context per FP (File Pointer). This is allocated by low level drivers in open function.
- `const void` * buf: buffer to store/take the data that will be received/written
- `size_t` nbytes: quantity of data to be read/written.
- `int` *error: Output pointer to return error codes.

These functions will handle the reading and writing procedures for the driver. It is designed according to driver's capabilities and could contain buffering handling (circular buffers), signaling events and block the application until reading and writing are completed.

These functions return the quantity of data that were read/written.

For detailed information, consult Appendix A: Source and Header files Reference

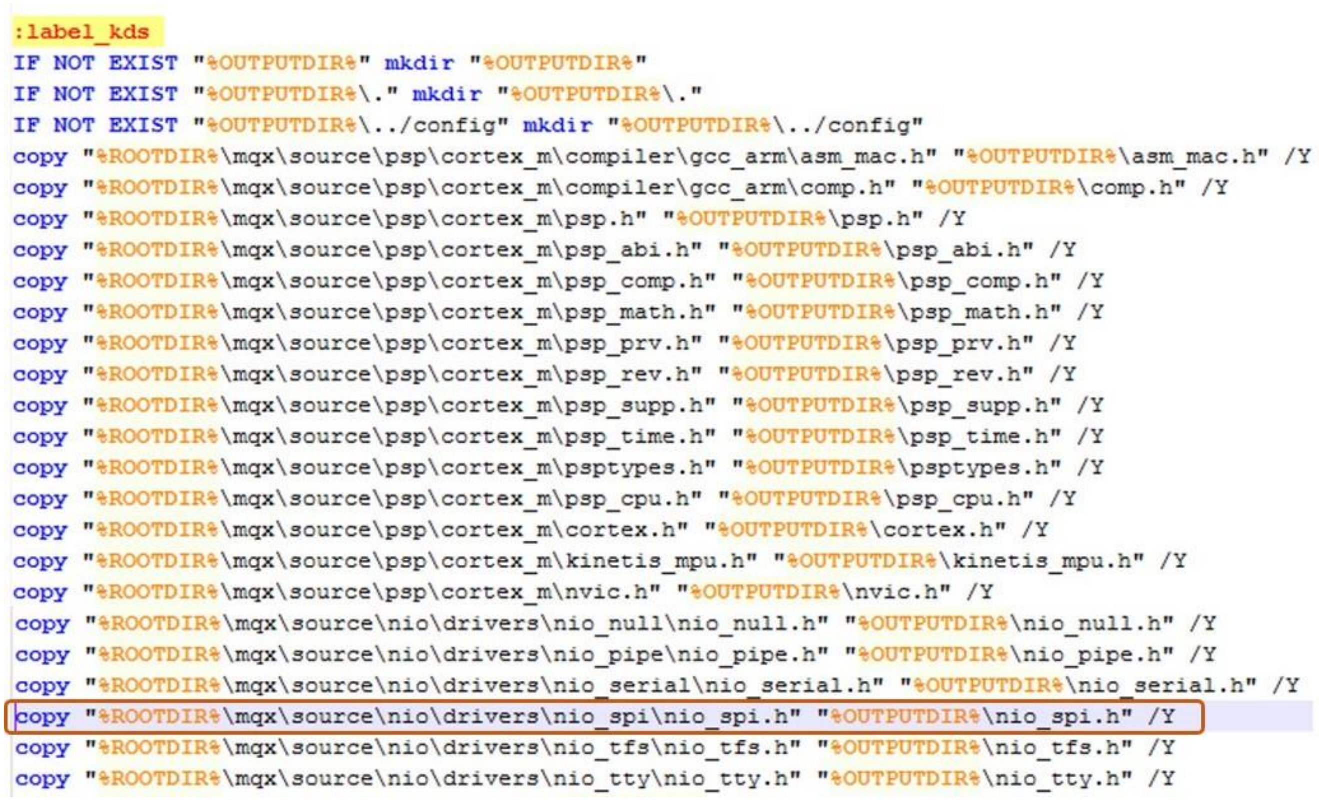
4 Create the MQX library image.

4.1 Every time that MQX library is compiled, header files and libraries (*.a) from MQX project are copied in <KSDK_1_3_PATH>\rtos\mqx\lib\<BOARD>.<IDE>\debug\mqx, so, we need to modify bat file in order to copy the new nio_spi.h file.

4.2 Look for your target's BAT file, they are located at: <KSDK_1_3_PATH>\rtos\mqx\mqx\build\bat , right-click on the file and select "open with" in order to use a text editor to edit the file, then add next line:

```
copy "%ROOTDIR%\mqx\source\nio\drivers\nio_spi\nio_spi.h" "%OUTPUTDIR%\nio_spi.h" /Y
```

This BAT file handle the copy process for all IDEs, so be careful to copy previous line in the right IDE label, just as shown below:



```
:label_kds
IF NOT EXIST "%OUTPUTDIR%" mkdir "%OUTPUTDIR%"
IF NOT EXIST "%OUTPUTDIR%\." mkdir "%OUTPUTDIR%\."
IF NOT EXIST "%OUTPUTDIR%\../config" mkdir "%OUTPUTDIR%\../config"
copy "%ROOTDIR%\mqx\source\psp\cortex_m\compiler\gcc_arm\asm_mac.h" "%OUTPUTDIR%\asm_mac.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\compiler\gcc_arm\comp.h" "%OUTPUTDIR%\comp.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\psp.h" "%OUTPUTDIR%\psp.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\psp_abi.h" "%OUTPUTDIR%\psp_abi.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\psp_comp.h" "%OUTPUTDIR%\psp_comp.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\psp_math.h" "%OUTPUTDIR%\psp_math.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\psp_prv.h" "%OUTPUTDIR%\psp_prv.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\psp_rev.h" "%OUTPUTDIR%\psp_rev.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\psp_supp.h" "%OUTPUTDIR%\psp_supp.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\psp_time.h" "%OUTPUTDIR%\psp_time.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\psptypes.h" "%OUTPUTDIR%\psptypes.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\psp_cpu.h" "%OUTPUTDIR%\psp_cpu.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\cortex.h" "%OUTPUTDIR%\cortex.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\kinetis_mpu.h" "%OUTPUTDIR%\kinetis_mpu.h" /Y
copy "%ROOTDIR%\mqx\source\psp\cortex_m\nvic.h" "%OUTPUTDIR%\nvic.h" /Y
copy "%ROOTDIR%\mqx\source\nio\drivers\nio_null\nio_null.h" "%OUTPUTDIR%\nio_null.h" /Y
copy "%ROOTDIR%\mqx\source\nio\drivers\nio_pipe\nio_pipe.h" "%OUTPUTDIR%\nio_pipe.h" /Y
copy "%ROOTDIR%\mqx\source\nio\drivers\nio_serial\nio_serial.h" "%OUTPUTDIR%\nio_serial.h" /Y
copy "%ROOTDIR%\mqx\source\nio\drivers\nio_spi\nio_spi.h" "%OUTPUTDIR%\nio_spi.h" /Y
copy "%ROOTDIR%\mqx\source\nio\drivers\nio_tfs\nio_tfs.h" "%OUTPUTDIR%\nio_tfs.h" /Y
copy "%ROOTDIR%\mqx\source\nio\drivers\nio_tty\nio_tty.h" "%OUTPUTDIR%\nio_tty.h" /Y
```

Figure 7 - Editing BAT file

4.3 Compile MQX library and notice that nio_spi.h was copied to: <KSDK_1_3_PATH>\rtos\mqx\lib\<BOARD>.<IDE>\debug\mqx.

4.4 At this point, we have successfully added this new NIO driver to MQX library and it is ready to be used for the application. Next steps guide you through the way to use it.

5 Using NIO driver

5.1 Create a new MQX project for KSDK (or use any of the given examples, for example, use the hello_frdmk64f example that is located at: <KSDK_1_3_PATH>\rtos\mqx\mqx\examples\hello\build\kds\hello_frdmk64f) and in main.c (or hello.c file, if you are using hello_frdmk64f example) include nio.h and nio_spi.h header files and then define the initialization structure for the driver (This structure is the one defined in ***3.5***)

```
#include "nio.h"
#include "nio_spi.h"

const NIO_SPI_INIT_DATA_STRUCT nio_spi_default_init =
{
    .SPI_INSTANCE            = BOARD_DSPI_INSTANCE,
    .BAUDRATE                = 350000,
    .BITS_PER_FRAME          = 8,
    .CLOCK_PHASE             = kDspiClockPhase_FirstEdge,       /*!< Clock phase */
    .CLOCK_POLARITY          = kDspiClockPolarity_ActiveHigh,   /*!< Clock polarity */
    .IRQ_PRIORITY            = 4,                               /*!< irq priority for SPI */
    .RX_BUFF_SIZE            = NIO_SPI_MIN_BUFF_SIZE,           /*!< Size of rx buffer.*/
    .TX_BUFF_SIZE            = NIO_SPI_MIN_BUFF_SIZE,           /*!< Size of tx buffer.*/
    .CHIP_SELECT_CONTINUOUS = true,
    .SCK_CONTINUOUS          = false,
    .DELAY_TYPE              = kDspiAfterTransfer,
    .DELAY_VALUE_NANO_SEC    = 500                               /*!< Value in nano seconds */
};
```

5.2 In bsp.h (at: <your_project>/BSP_Files), define a name for this NIO driver:

```
#define BSP_DEFAULT_SPI_CHANNEL  "nio_spi" BSP_MACRO_TO_STRING(BOARD_DSPI_INSTANCE) ":"
```

5.3 Select any task to install the driver, it is done by using:

```
void *res = NULL;
/* Install SPI driver */
res = _nio_dev_install(BSP_DEFAULT_SPI_CHANNEL,
                       &nio_spi_dev_fn,
                       (void*) &nio_spi_default_init,
                       NULL);
assert(NULL != res);
```

First parameter is the name given to the driver (***5.2***), second parameter is the NIO structure that contains the supported functions for the driver (***3.3***) and the third parameter is the initialization structure defined in ***5.1***.

5.4 At this point, driver is installed correctly, now, it is time to open the driver with proper flags:

```
FILE    *spi_driver = NULL;
spi_driver = fopen(BSP_DEFAULT_SPI_CHANNEL, "r+");
```

Remember to validate that spi_driver pointer is different to NULL.

5.5 If validation is successful, then, we can access to the new SPI NIO driver, try using fwrite/fread to send/receive data:

```
#define BUFFER_SIZE     4

uint8_t buffer[BUFFER_SIZE] = {[0 ... (BUFFER_SIZE - 1)] = 0xA5};
int data_written;


data_written = fwrite((void *)buffer, BUFFER_SIZE, 1, spi_driver);
if (data_written != 1) {
    printf("written data %d differs than expected!", data_written);
}
```

5.6 That's it! Now you can use your full-featured NIO driver.

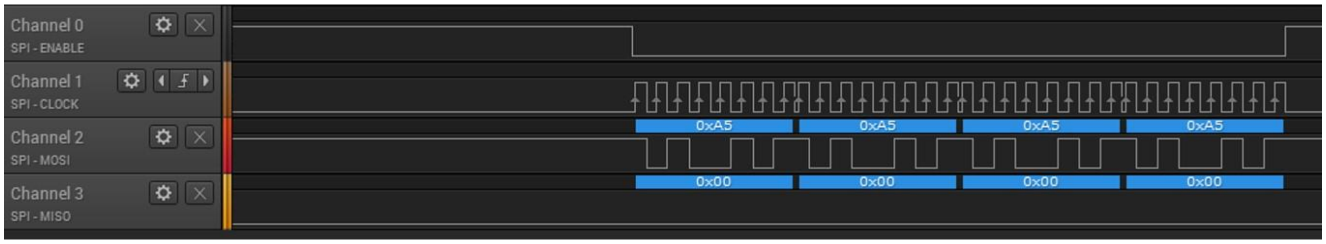


Figure 8 - Using NIO Driver

6 Additional Information

- KSDK and MQX documentation
  <KSDK_1_3_PATH>/docs

- How To: Create a New MQX RTOS for KSDK Project in KDS
  https://community.freescale.com/docs/DOC-103405

- Interrupt handling with KSDK and Kinetis Design Studio
  https://community.freescale.com/docs/DOC-104352

Appendix A: Source and Header files Reference

Source and header files for this new SPI driver as well as application files and bsp.h are shown below.

- *NIO Header File (nio_spi.h)*

```
#ifndef __NIO_SPI_H__
#define __NIO_SPI_H__
#include "nio.h"
#include "fsl_dspi_master_driver.h"
#include <assert.h>
#include "ioctl.h"
extern const NIO_DEV_FN_STRUCT nio_spi_dev_fn;
/* Definitions to configure SPI driver */
#define SPI_USES_BLOCKING_FUNCTIONS     (0) /*!< Uses blocking or non-blocking functions
*/
#define NIO_SPI_MIN_BUFF_SIZE           (4)   /*!< Minimal size of serial buffers */
/* INITIALIZATION STRUCTURE */
typedef struct
{
    uint32_t SPI_INSTANCE;                      /*!< Number of used SPI peripheral */
    uint32_t BAUDRATE;                          /*!< SPI baud rate */
    uint8_t BITS_PER_FRAME;                     /*!< number of bits, 8-bit (default) or 16-bit
*/
    dspi_clock_phase_t CLOCK_PHASE;             /*!< Clock phase */
    dspi_clock_polarity_t CLOCK_POLARITY;       /*!< Clock polarity */
    uint32_t IRQ_PRIORITY;                       /*!< irq priority for SPI */
    uint32_t RX_BUFF_SIZE;                      /*!< Size of rx buffer.*/
    uint32_t TX_BUFF_SIZE;                      /*!< Size of tx buffer.*/
    bool CHIP_SELECT_CONTINUOUS;                /*!< DSPI configuration */
    bool SCK_CONTINUOUS;                        /*!< DSPI configuration */
    dspi_delay_type_t DELAY_TYPE;               /*!< Delay type for SPI */
    uint32_t DELAY_VALUE_NANO_SEC;              /*!< Delay value for SPI */
} NIO_SPI_INIT_DATA_STRUCT;
#ifdef __cplusplus
extern "C" {
#endif
#ifdef __cplusplus
}
#endif
#endif
```

- NIO Source File (nio_spi.c)

```
#include <stdint.h>
#include "nio.h"

#include "nio_spi.h"

/* Buffer structure for Rx / Tx */
typedef struct {
    uint8_t *buff;
    uint32_t size;
}NIO_SPI_BUFFER_STRUCT;

/* Device context structure, this structure contains all the elements needed
 * for every supported function (open, init, read, write, etc). The elements
 * for this structure depends on driver's complexity and robustness */
typedef struct {
    mutex_t lock;
    uint8_t instance;
    dspi_device_t *spi_device_context;
    dspi_master_state_t *spi_master_state;
    NIO_SPI_BUFFER_STRUCT tx_buff;
    NIO_SPI_BUFFER_STRUCT rx_buff;
}NIO_SPI_DEV_CONTEXT_STRUCT;

/* SPI handler when using interrupt mode */
extern void DSPI_DRV_IRQHandler(uint32_t instance);

static int nio_spi_open(void *dev_context, const char *dev_name, int flags, void
**fp_context, int *error) {
    /* Validate that dev_context pointer is different to NULL */
    assert(dev_context != NULL);

    /* Get device context structure from dev_context argument */
    NIO_SPI_DEV_CONTEXT_STRUCT *spi_dev_context = (NIO_SPI_DEV_CONTEXT_STRUCT
*)dev_context;
    uint32_t temp_value;

    /* Configure delay between SS and SCK */
    if ( kStatus_DSPI_Success != DSPI_DRV_MasterSetDelay(spi_dev_context->instance,
                                kDspiPcsToSck, 500, &temp_value))
    {
        /* If error pointer is available, return an error */
        if (error != NULL) {
            *error = NIO_ENXIO;
        }
        return -1;
    }

    /* Configure SPI's bus */
    if ( kStatus_DSPI_Success != DSPI_DRV_MasterConfigureBus(spi_dev_context->instance,
                                                    spi_dev_context->spi_device_context,
                                                                    &temp_value))
    {
        /* If error pointer is available, return an error */
        if (error != NULL) {
            *error = NIO_ENXIO;
        }
        return -1;
    }
    /* Update new baud rate */
    if (temp_value != (spi_dev_context->spi_device_context)->bitsPerSec) {
        (spi_dev_context->spi_device_context)->bitsPerSec = temp_value;
    }
    /* Return no error */
    return 0;
}

static int nio_spi_read(void *dev_context, void *fp_context, void *buf, size_t nbytes, int
*error) {
    /* Validate that dev_context pointer is different to NULL */
    assert(dev_context != NULL);

    /* Get device context structure from dev_context argument */
    NIO_SPI_DEV_CONTEXT_STRUCT *spi_dev_context = (NIO_SPI_DEV_CONTEXT_STRUCT
*)dev_context;
    /* Get buffer context from device context */
    NIO_SPI_BUFFER_STRUCT* rxBuffer = &spi_dev_context->rx_buff;
    NIO_SPI_BUFFER_STRUCT* txBuffer = &spi_dev_context->tx_buff;
    osa_status_t status;

    /* If no data to be read, return no error */
    if (0 == nbytes)
    {
        return 0;
    }
    /* Validate output buffer is different to NULL */
    if (NULL == buf)
    {
        /* If error pointer is available, return an error */
        if (error) {
           *error = NIO_EINVAL;
        }
        return -1;
    }

    /* To simplify this driver, we only accept only transfers with equal or less
     * elements than buffer's space */
    if (nbytes > rxBuffer->size) {
        /* If error pointer is available, return an error */
        if (error) {
            *error = NIO_E2BIG;
        }
        return -1;
    }

    /* Lock Driver access */
    status = OSA_MutexLock(&spi_dev_context->lock, OSA_WAIT_FOREVER);
    if (status != kStatus_OSA_Success)
    {
        /* If error pointer is available, return an error */
        if (error) {
            *error = NIO_EIO;
        }
        return -1;
    }

    /* Fill Tx buffer with 0xFF, like SPI is full-duplex communication, we send
     * "dummy" data in order to generate clock signal and receive data in MISO
     * terminal */
    uint32_t index = 0;
    for (index = 0; index < nbytes; index++) {
        txBuffer->buff[index] = 0xFF;
        rxBuffer->buff[index] = 0;
    }
    /* Perform a read instruction */
#if SPI_USES_BLOCKING_FUNCTIONS
    if (kStatus_DSPI_Success != DSPI_DRV_MasterTransferBlocking(spi_dev_context->instance,
                                                                NULL,
                                                                &txBuffer->buff[0],
                                                                &rxBuffer->buff[0],
                                                                nbytes, 0xFFFF)) {
#else
    if (kStatus_DSPI_Success != DSPI_DRV_MasterTransfer(spi_dev_context->instance, NULL,
                                                                &txBuffer->buff[0],
                                                                &rxBuffer->buff[0],
                                                                nbytes)) {
#endif
        /* If error is received, unlock the peripheral and return and error */
        status = OSA_MutexUnlock(&spi_dev_context->lock);
        if (error) {
            *error = NIO_EIO;
        }
        return -1;
    }
    /* Copy received data to application buffer */
    memcpy((void*)(buf), (const void *)&rxBuffer->buff[0], nbytes);
    /* Unlock mutex */
    status = OSA_MutexUnlock(&spi_dev_context->lock);
    if (status != kStatus_OSA_Success)
    {
        if (error) {
            *error = NIO_EIO;
        }
    }
    /* Return quantity of bytes that were read */
    return nbytes;
}

static int nio_spi_write(void *dev_context, void *fp_context, const void *buf, size_t
nbytes, int *error) {
    /* Validate that dev_context pointer is different to NULL */
    assert(dev_context != NULL);

    /* Get device context structure from dev_context argument */
    NIO_SPI_DEV_CONTEXT_STRUCT *spi_dev_context = (NIO_SPI_DEV_CONTEXT_STRUCT
*)dev_context;
    /* Get buffer context from device context */
    NIO_SPI_BUFFER_STRUCT* rxBuffer = &spi_dev_context->rx_buff;
    NIO_SPI_BUFFER_STRUCT* txBuffer = &spi_dev_context->tx_buff;
    osa_status_t status;

    /* If no data to be written, return no error */
    if (0 == nbytes)
    {
        return 0;
    }
    /* Validate input buffer is different to NULL */
    if (NULL == buf)
    {
        /* If error pointer is available, return an error */
        if (error) {
           *error = NIO_EINVAL;
        }
        return -1;
    }

    /* To simplify this driver, we only accept only transfer with equal or less
     * elements than buffer's space */
    if (nbytes > txBuffer->size) {
        /* If error pointer is available, return an error */
        if (error) {
            *error = NIO_E2BIG;
        }
        return -1;
    }
    /* Lock the Driver access */
    status = OSA_MutexLock(&spi_dev_context->lock, OSA_WAIT_FOREVER);
    if (status != kStatus_OSA_Success)
    {
        if (error) {
            *error = NIO_EIO;
        }
        return -1;
    }

    /* Clear receiving buffer before sending data, this is done in case
     * device sends data while driver is transmitting */
    uint32_t index = 0;
    for (index = 0; index < nbytes; index++) {
        rxBuffer->buff[index] = 0;
    }
    /* Fill internal Tx buffer with data from application */
    memcpy((void*) &txBuffer->buff[0], (const void *)(buf), nbytes);

    /* Perform a write instruction */
#if SPI_USES_BLOCKING_FUNCTIONS
```

```
    if (kStatus_DSPI_Success != DSPI_DRV_MasterTransferBlocking(spi_dev_context->instance,
                                                           NULL,
                                                           &txBuffer->buff[0],
                                                           &rxBuffer->buff[0],
                                                           nbytes, 0xFFFF)) {
#else
    if (kStatus_DSPI_Success != DSPI_DRV_MasterTransfer(spi_dev_context->instance, NULL,
                                                           &txBuffer->buff[0],
                                                           &rxBuffer->buff[0],
                                                           nbytes)) {
#endif
        status = OSA_MutexUnlock(&spi_dev_context->lock);
        /* If error pointer is available, return an error */
        if (error) {
            *error = NIO_EIO;
        }
        return -1;
    }
    /* In case slave send data when master was transmitting, copy received data
     * to application buffer */
    memcpy((void*)(buf), (const void *)&rxBuffer->buff[0], nbytes);

    /* Unlock mutex */
    status = OSA_MutexUnlock(&spi_dev_context->lock);
    if (status != kStatus_OSA_Success)
    {
        if (error) {
            *error = NIO_EIO;
        }
    }
    /* Return quantity of bytes that were written */
    return nbytes;
}

static int nio_spi_ioctl(void *dev_context, void *fp_context, int *error, unsigned long
int request, va_list ap) {
    /* Validate that dev_context pointer is different to NULL */
    assert(dev_context != NULL);

    /* Get device context structure from dev_context argument */
    NIO_SPI_DEV_CONTEXT_STRUCT *spi_dev_context = (NIO_SPI_DEV_CONTEXT_STRUCT
*)dev_context;
    uint32_t *dest = NULL;

    switch (request) {
        case IOCTL_ABORT:
            /* Validate that there is pending transfer */
            if (kStatus_DSPI_Busy ==
                              DSPI_DRV_MasterGetTransferStatus(spi_dev_context->instance,
                                                                        NULL)) {
                /* Abort current transfer */
                if (kStatus_DSPI_Success !=
                              DSPI_DRV_MasterAbortTransfer(spi_dev_context->instance)) {
                    /* If error pointer is available, return an error */
                    if (error) {
                        *error = NIO_ENOTTY;
                        return -1;
                    }
                }
            }
            break;
        case IO_IOCTL_SERIAL_RXBUFF_GET_SIZE:
            /* Validate input arguments */
            dest = va_arg(ap, void*);
            if (NULL == dest)
            {
                if (error) {
                    *error = NIO_EINVAL;
                    return -1;
                }
            }
            /* Return buffer size for RX */
            *dest = spi_dev_context->rx_buff.size;
            break;
        default:
            if (error) {
                *error = NIO_ENOTTY;
            }
            return -1;
            break;
    }
    return 0;
}

static int nio_spi_close(void *dev_context, void *fp_context, int *error) {
    return 0;
}

static int nio_spi_init(void *init_data, void **dev_context, int *error) {

    /* Validate that init_data pointer is different to NULL */
    assert(init_data != NULL);
    /* Validate that dev_context pointer is different to NULL */
    assert(dev_context != NULL);

    /* Get device context structure from dev_context argument */
    NIO_SPI_DEV_CONTEXT_STRUCT *spi_dev_context =
                            (NIO_SPI_DEV_CONTEXT_STRUCT *)dev_context;
    /* Get init data structure from init_data argument */
    NIO_SPI_INIT_DATA_STRUCT *init = (NIO_SPI_INIT_DATA_STRUCT*)init_data;
    /* Get buffer context from device context */
    NIO_SPI_BUFFER_STRUCT *rxBuffer = NULL;
    NIO_SPI_BUFFER_STRUCT *txBuffer = NULL;

    /* Validate SPI instance */
    assert(init->SPI_INSTANCE < SPI_INSTANCE_COUNT);

    /* Allocate SPI device context */
    spi_dev_context = (NIO_SPI_DEV_CONTEXT_STRUCT *)
```

```
                OSA_MemAlloc(sizeof(NIO_SPI_DEV_CONTEXT_STRUCT));
if (NULL == spi_dev_context) {
    if (error != NULL) {
        *error = NIO_ENOMEM;
    }
    return -1;
}
/* Configure Rx Buffer*/
rxBuffer = &spi_dev_context->rx_buff;
rxBuffer->size = init->RX_BUFF_SIZE;
if (rxBuffer->size < NIO_SPI_MIN_BUFF_SIZE) {
    rxBuffer->size = NIO_SPI_MIN_BUFF_SIZE;
}
/* Allocate Rx buffer */
rxBuffer->buff = (uint8_t *) OSA_MemAlloc(rxBuffer->size);
if (NULL == rxBuffer->buff) {
    OSA_MemFree(spi_dev_context);
    if (error != NULL) {
        *error = NIO_ENOMEM;
    }

    return -1;
}

/* Configure Tx Buffer */
txBuffer = &spi_dev_context->tx_buff;
txBuffer->size = init->TX_BUFF_SIZE;
if (txBuffer->size < NIO_SPI_MIN_BUFF_SIZE) {
    txBuffer->size = NIO_SPI_MIN_BUFF_SIZE;
}
/* Allocate Tx buffer */
txBuffer->buff = (uint8_t *) OSA_MemAlloc(txBuffer->size);
if (NULL == txBuffer->buff) {
    OSA_MemFree(rxBuffer->buff);
    OSA_MemFree(spi_dev_context);
    if (error != NULL) {
        *error = NIO_ENOMEM;
    }
    return -1;
}

/* Synchronization objects initialization */
if ( kStatus_OSA_Success != OSA_MutexCreate(&spi_dev_context->lock))
{
    OSA_MemFree(txBuffer->buff);
    OSA_MemFree(rxBuffer->buff);
    OSA_MemFree(spi_dev_context);
    if (error != NULL) {
        *error = NIO_ENOMEM;
    }
    return -1;
}

/* Allocate spi_master_state structure */
```

```
    spi_dev_context->spi_master_state =
            (dspi_master_state_t *)OSA_MemAlloc(sizeof(dspi_master_state_t));
    if (NULL == spi_dev_context->spi_master_state)
    {
        OSA_MutexDestroy(&spi_dev_context->lock);
        OSA_MemFree(txBuffer->buff);
        OSA_MemFree(rxBuffer->buff);
        OSA_MemFree(spi_dev_context);
        if (error != NULL) {
            *error = NIO_ENOMEM;
        }
        return -1;
    }

    /* Allocate spi_device_context element */
    spi_dev_context->spi_device_context =
            (dspi_device_t *)OSA_MemAlloc(sizeof(dspi_device_t));
    if (NULL == spi_dev_context->spi_device_context)
    {
        OSA_MemFree(spi_dev_context->spi_master_state);
        OSA_MutexDestroy(&spi_dev_context->lock);
        OSA_MemFree(txBuffer->buff);
        OSA_MemFree(rxBuffer->buff);
        OSA_MemFree(spi_dev_context);
        if (error != NULL) {
            *error = NIO_ENOMEM;
        }
        return -1;
    }

    /* SPI handler interrupt installation */
    IRQn_Type IRQNumber = g_dspiIrqId[init->SPI_INSTANCE];
    NVIC_SetPriority((IRQn_Type)IRQNumber, init->IRQ_PRIORITY);
    _int_install_isr(IRQNumber, (INT_ISR_FPTR)DSPI_DRV_IRQHandler,
                        (void*)init->SPI_INSTANCE);

    /* Initialize module */
    dspi_master_user_config_t dspi_config;
    /* Initialize module according to init structure */
    dspi_config.isChipSelectContinuous  = init->CHIP_SELECT_CONTINUOUS;
    dspi_config.isSckContinuous         = init->SCK_CONTINUOUS;
    dspi_config.whichPcs                = kDspiPcs0;
    dspi_config.pcsPolarity             = kDspiPcs_ActiveLow;
    dspi_config.whichCtar               = kDspiCtar0;

    if ( kStatus_DSPI_Success !=
            DSPI_DRV_MasterInit(init->SPI_INSTANCE,
                                spi_dev_context->spi_master_state,
                                &dspi_config))
    {
        OSA_MemFree(spi_dev_context->spi_master_state);
        OSA_MutexDestroy(&spi_dev_context->lock);
        OSA_MemFree(txBuffer->buff);
        OSA_MemFree(rxBuffer->buff);
        OSA_MemFree(spi_dev_context);
        if (error != NULL) {
            *error = NIO_ENXIO;
        }
        return -1;
    }

    /* This function involves the DSPI module's delay options to "fine tune" */
    uint32_t calculated_delay = 0;
    if ( kStatus_DSPI_Success !=
            DSPI_DRV_MasterSetDelay(init->SPI_INSTANCE,
                                    init->DELAY_TYPE,
                                    init->DELAY_VALUE_NANO_SEC,
                                    &calculated_delay))
    {
        DSPI_DRV_MasterDeinit(init->SPI_INSTANCE);
        OSA_MemFree(spi_dev_context->spi_master_state);
        OSA_MutexDestroy(&spi_dev_context->lock);
        OSA_MemFree(txBuffer->buff);
        OSA_MemFree(rxBuffer->buff);
        OSA_MemFree(spi_dev_context);
        if (error != NULL) {
            *error = NIO_ENXIO;
        }
        return -1;
    }

    /* Save the bus configuration */
    (spi_dev_context->spi_device_context)->bitsPerSec =
                                                    init->BAUDRATE;
    (spi_dev_context->spi_device_context)->dataBusConfig.bitsPerFrame =
                                                    init->BITS_PER_FRAME;
    (spi_dev_context->spi_device_context)->dataBusConfig.clkPhase =
                                                    init->CLOCK_PHASE;
    (spi_dev_context->spi_device_context)->dataBusConfig.clkPolarity =
                                                    init->CLOCK_POLARITY;
    (spi_dev_context->spi_device_context)->dataBusConfig.direction =
                                                    kDspiMsbFirst;

    uint32_t calculated_baudrate = 0;
    if ( kStatus_DSPI_Success !=
            DSPI_DRV_MasterConfigureBus(init->SPI_INSTANCE,
                                        spi_dev_context->spi_device_context,
                                        &calculated_baudrate))
    {
        DSPI_DRV_MasterDeinit(init->SPI_INSTANCE);
        OSA_MemFree(spi_dev_context->spi_master_state);
        OSA_MutexDestroy(&spi_dev_context->lock);
        OSA_MemFree(txBuffer->buff);
        OSA_MemFree(rxBuffer->buff);
        OSA_MemFree(spi_dev_context);
        if (error != NULL) {
            *error = NIO_ENXIO;
        }
        return -1;
    }

    /* Device context initialization */
    spi_dev_context->instance = init->SPI_INSTANCE;
    /* Once all the initialization is made, save this context to MQX kernel */
    *dev_context = (void*)spi_dev_context;
    return 0;
}

static int nio_spi_deinit(void *dev_context, int *error) {
    /* Validate that dev_context pointer is different to NULL */
    assert(dev_context != NULL);

    /* Get device context structure from dev_context argument */
    NIO_SPI_DEV_CONTEXT_STRUCT *spi_dev_context = (NIO_SPI_DEV_CONTEXT_STRUCT
*)dev_context;
    if (NULL == dev_context)
    {
        if (error) {
            *error = NIO_EINVAL;
        }
        return -1;
    }

    /* De-initialize the module */
    DSPI_DRV_MasterDeinit(spi_dev_context->instance);

    /* Validate pointers */
    assert(&spi_dev_context->lock);
    assert(spi_dev_context->spi_master_state);
    assert(spi_dev_context->rx_buff.buff);
    assert(spi_dev_context->tx_buff.buff);
    assert(dev_context);

    /* Free memory for each resource */
    OSA_MutexDestroy(&spi_dev_context->lock);
    OSA_MemFree(spi_dev_context->spi_master_state);
    OSA_MemFree((&spi_dev_context->rx_buff)->buff);
    OSA_MemFree((&spi_dev_context->tx_buff)->buff);
    OSA_MemFree(dev_context);
    /* Return no error */
    return 0;
}

/* Functions that will be supported for this SPI driver */
const NIO_DEV_FN_STRUCT nio_spi_dev_fn = {
    .OPEN = nio_spi_open,
    .READ = nio_spi_read,
    .WRITE = nio_spi_write,
    .LSEEK = NULL,
    .IOCTL = nio_spi_ioctl,
    .CLOSE = nio_spi_close,
    .INIT = nio_spi_init,
    .DEINIT = nio_spi_deinit,
};
```

- Application source file (hello.c from hello_frdmk64f example)

```
#include <stdio.h>
#include <mqx.h>
#include <bsp.h>

#include "nio.h"
#include "nio_spi.h"

#define BUFFER_SIZE     4

const NIO_SPI_INIT_DATA_STRUCT nio_spi_default_init =
{
    .SPI_INSTANCE            = BOARD_DSPI_INSTANCE,
    .BAUDRATE                = 350000,
    .BITS_PER_FRAME          = 8,
    .CLOCK_PHASE             = kDspiClockPhase_FirstEdge,      /*!< Clock phase */
    .CLOCK_POLARITY          = kDspiClockPolarity_ActiveHigh, /*!< Clock polarity */
    .IRQ_PRIORITY            = 4,                              /*!< irq priority for SPI */
    .RX_BUFF_SIZE            = NIO_SPI_MIN_BUFF_SIZE,          /*!< Size of rx buffer.*/
    .TX_BUFF_SIZE            = NIO_SPI_MIN_BUFF_SIZE,          /*!< Size of tx buffer.*/
    .CHIP_SELECT_CONTINUOUS = true,
    .SCK_CONTINUOUS          = false,
    .DELAY_TYPE              = kDspiAfterTransfer,
    .DELAY_VALUE_NANO_SEC    = 500                             /*!< Value in nano seconds */
};

/* Task IDs */
#define HELLO_TASK  5
#define WORLD_TASK  6

extern void hello_task(uint32_t);
extern void world_task(uint32_t);


const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
   /* Task Index,   Function,   Stack,  Priority, Name,     Attributes,          Param,
Time Slice */
    { WORLD_TASK,   world_task, 700,   9,        "world",  MQX_AUTO_START_TASK, 0,     0
},
    { HELLO_TASK,   hello_task, 700,   8,        "hello",  0,                   0,     0
},
    { 0 }
};

/*TASK*-----------------------------------------------------
*
* Task Name    : world_task
* Comments     :
*    This task creates hello_task and then prints " World ".
*
*END*-----------------------------------------------------*/
```

```
void world_task
   (
      uint32_t initial_data
   )
{
   _task_id hello_task_id;
   FILE    *spi_driver = NULL;
   void *res = NULL;
   uint8_t buffer[BUFFER_SIZE] = {[0 ... (BUFFER_SIZE - 1)] = 0xA5};
   int data_written;

   configure_spi_pins(nio_spi_default_init.SPI_INSTANCE);

   /* Install SPI driver */
   res = _nio_dev_install(BSP_DEFAULT_SPI_CHANNEL, &nio_spi_dev_fn, (void*)
&nio_spi_default_init, NULL);
   assert(NULL != res);

   hello_task_id = _task_create(0, HELLO_TASK, 0);
   if (hello_task_id == MQX_NULL_TASK_ID) {
      printf ("\n Could not create hello_task\n");
   } else {
      printf(" World \n");
   }

   printf("Openning SPI driver...");
   spi_driver = fopen(BSP_DEFAULT_SPI_CHANNEL, "r+");
   if (NULL == spi_driver) {
       printf("[FAIL]\r\n");
       _task_block();
   } else {
       printf("[OK]\r\n");
   }

   printf("Sending %d data: ", BUFFER_SIZE);
   for (data_written = 0; data_written < BUFFER_SIZE; data_written++) {
       printf(" 0x%.2X ",buffer[data_written]);
   }
   printf("\r\n");
   data_written = fwrite((void *)buffer, BUFFER_SIZE, 1, spi_driver);
   if (data_written != 1) {
       printf("written data %d differs than expected!", data_written);
   }
   printf("Received data: ");
   for (data_written = 0; data_written < BUFFER_SIZE; data_written++) {
       printf(" 0x%.2X ",buffer[data_written]);
   }

   while (1) {

   }
   _task_block();
}
```

```
/*TASK*-----------------------------------------------------
*
* Task Name    : hello_task
* Comments     :
*    This task prints " Hello".
*
*END*-----------------------------------------------------*/
void hello_task
   (
      uint32_t initial_data
   )
{

   printf("\n Hello\n");
   _task_block();

}
```

- BSP Header file (bsp.h)

```
#ifndef __bsp_h__
#define __bsp_h__   1

#include "psp.h"
/* SDK */
#include "fsl_os_abstraction.h"
#include "fsl_clock_manager.h"
#include "fsl_hwtimer.h"
#include "fsl_hwtimer_systick.h"

#ifndef PEX_MQX_KSDK
  #include "fsl_gpio_driver.h"
  #include "fsl_hwtimer_pit.h"
  #ifdef HW_ENET_INSTANCE_COUNT
    #include "fsl_enet_hal.h"
  #endif
#endif

/* SDK board configuration file */
#include "board.h"
/* BSP default configuration file */
#include "bsp_config.h"

/*
 * Board Debug Console Settings
 */

#ifndef BOARD_DEBUG_UART_INSTANCE
  #error BOARD_DEBUG_UART_INSTANCE should be defined in board.h file.
#endif
#ifndef BOARD_DEBUG_UART_BAUD
    #error BOARD_DEBUG_UART_BAUD should be defined in board.h file.
#endif

/* Define BSP_DEFAULT_IO_CHANNEL for MQX_init_struct */
#define BSP_MACRO_TO_STRING_(x) #x
#define BSP_MACRO_TO_STRING(x) BSP_MACRO_TO_STRING_(x)

#ifndef BSP_DEFAULT_IO_CHANNEL
    #define BSP_DEFAULT_IO_CHANNEL                             "nio_ser"
BSP_MACRO_TO_STRING(BOARD_DEBUG_UART_INSTANCE) ":"
#endif


#define BSP_DEFAULT_SPI_CHANNEL          "nio_spi"
BSP_MACRO_TO_STRING(BOARD_DSPI_INSTANCE) ":"



#ifdef __cplusplus
extern "C" {
#endif
/* Function declarations */
int _bsp_init(void); /* needs to be exported for lite configuration */

#ifdef __cplusplus
}
#endif

#endif  /* __bsp_h__ */
```