KSDK SPI Master and Slave example using the FRDM-K64F

By: Technical Information Center

1 Introduction

Kinetis SDK (KSDK) is a Software Development Kit that provides comprehensive software support for Freescale Kinetis devices. It is intended to provide complete software support when using Kinetis devices reducing the developing time process.

This document describes the use of the dSPI module, present on the Kinetis devices, using the KSDK drivers based on a simple SPI Master-Slave configuration example.

For more information about KSDK visit:
www.freescale.com/KSDK

Content

2 Serial Peripheral Interface.

The serial peripheral interface (SPI) module provides a synchronous serial bus for communication between an MCU and an external peripheral device.

The module can support different configurations changing the polarity or the state of the lines when the bus is idle, and much more. For better understanding of the characteristics and functionality of the module please refer to the specific Kinetis device reference manual.

2.1 SPI signals.

The SPI peripheral has different signals that are connected when using Master or Slave mode, the next table presents such signals.

Signal	Master mode	Slave mode	I/O
PCS0/$\overline{SS}$	Peripheral Chip Select 0 (O)	Slave Select (I)	I/O
PCS[1:3]	Peripheral Chip Selects 1–3	(Unused)	O
PCS4	Peripheral Chip Select 4	(Unused)	O
PCS5/ $\overline{PCSS}$	Peripheral Chip Select 5 /Peripheral Chip Select Strobe	(Unused)	O
SCK	Serial Clock (O)	Serial Clock (I)	I/O
SIN	Serial Data In	Serial Data In	I
SOUT	Serial Data Out	Serial Data Out	O

Table 1. SPI signals

3 SPI configuration example.

The next information and images are related with the SPI configuration examples FRDM-K64_SPI_MASTER_KSDK_example and FRDM-K64_SPI_SLAVE_KSDK_example (both for KDS IDE). The next sections will discuss the philosophy of the examples showing the configuration code.

3.1 Configure SPI pins.

The KSDK includes board support files for each supported hardware platform. These files are located in the *<install_dir>/boards*, look for the board folder; each folder contains same information related to the specific board. The pin mux folder contains functions to configure the pins present on the board for each peripheral.

The **`configure_spi_pins`** function initializes the pin signal for the peripheral instances present on the board. In the example the SPI0 is used calling:

```
configure_spi_pins(HW_SPI0);
```

The next signals are configured:

SPI0	Alternative 7
PTD0	SPI0_PCS0 (Chip Select)

PTD1	SPI0_SCK (Serial Clock)
PTD2	SPI0_SOUT (Serial Data Out)
PTD3	SPI0_SIN (Serial Data In)

The same SPI configuration is used for the Master and Slave mode.

3.2 SPI IRQ Handler.

The SPI KSDK driver implements just one ISR callback for Master and Slave mode. The IRQ handler is included in the fsl_dspi_irq.c file; it can be found in: <install_dir> \KSDK_1.1.0\platform\drivers\src\dspi.

The handler implements a different algorithm whether the mode is Master or Slave.

3.3 SPI Master Configuration.

Here the SPI master configuration is discussed: the necessary structure configurations and the drivers needed.

NOTE:

The notation RM in refers that more information can about can be found in the device specific reference manual

- **Driver Support**: Including the SPI support is necessary when using the drivers. Include the file:
  `#include "fsl_dspi_master_driver.h"`
- **Master User Configuration structure**: Creating a user configuration structure allows the user to set the common parameters for the SPI peripheral. This instruction is passed before to the `DSPI_DRV_MasterInit` function. A basic user configuration includes:
  - `dspi_master_user_config_t userConfig;`
    Create a structure for the user configuration.
  - `userConfig.isChipSelectContinuous = false;`
    Select if the PCSn (Chip Select) signal is returned to their inactive state between transfers or if it keeps asserted. (SPIx_PUSHR[CONT] RM)
  - `userConfig.isSckContinuous = false;`
    It disables or enables the continuous operations of the SCK (Serial Clock) signal. (SPIx_MCR[CONT_SCKE] RM)
  - `userConfig.pcsPolarity = kDspiPcs_ActiveLow;`
    Select if the Active value of the PCSx is low or high. (SPIx_MCR[PCSIS] RM)
  - `userConfig.whichCtar = kDspiCtar0;`
    Select which SPI CTARx is being used. For master opration it can be CTAR0 or CTAR1. (Number of CTARs section RM).
  - `userConfig.whichPcs = kDspiPcs0;`
    Select the device specific chip select. In this case the PCS0 (PTD0) is used. (SPIx_PUSHR[PCS]).
- **Master Bus Configuration structure**: Creating a data structure for the bus configuration allows the user to set properly the parameters to allow the communication.
  - `dspi_device_t spiDevice;`
    Create a structure for the SPI device bus settings.

- `spiDevice.dataBusConfig.bitsPerFrame = 8;`
  Configure the bits sent per frame through the bus. It allows a max number of 16 for master mode and 32 for slave mode.
- `spiDevice.dataBusConfig.clkPhase = kDspiClockPhase_SecondEdge;`
  Select when the data is captured, if leading edge or the second edge. (SPIx_CTARn[CPHA] RM).
- `spiDevice.dataBusConfig.clkPolarity = kDspiClockPolarity_ActiveHigh;`
  Select the inactive values of the SCK line selecting low or high state. (SPIx_CTARn[CPOL] RM).
- `spiDevice.dataBusConfig.direction = kDspiMsbFirst;`
  Select whether the LSB or MSB of the frame is transferred first. (SPIx_CTARn[LSBFE] RM).
- `spiDevice.bitsPerSec = 50000;`
  Select the bits per seconds transmitted. This selection sets the properly baudrate value for the peripheral.

- **Master Configuration Drivers:** As seen above there are two configurations done by the master: configuration of the peripheral and configuration of the bus. Next are listed the two drivers needed to complete the process.
  - `DSPI_DRV_MasterInit(HW_SPI0, &dspiMasterState, &userConfig);`
    The driver sets the SPI module with the parameters configured in the user configuration structure.
    Parameters:
    Instance →select the SPI instance, in this case SPI0 is used.
    dSPIstate →pointer to a data state structure. It needs to be created by the user to allocate memory for it.
    userConfig →pointer to the user configuration structure mentioned before.
  - `DSPI_DRV_MasterConfigureBus(HW_SPI0, &spiDevice, &calculatedBaudRate);`
    Configure the SPI port physical parameters to access a device on the bus.
    Parameters:
    Instance →select the SPI instance, in this case SPI0 is used.
    device →pointer to the bus configuration structure.
    calculatedBaudRate →pointer to the variable where the Calculated Baud Rate value will be stored. The user can check the value to determinate if the desired value is close enough to the one set by the driver.

3.4 SPI Master Transfer.

The SPI KSDK driver includes blocking and non blocking functions to implement the data transfers. In this example the blocking function was used.

- `DSPI_DRV_MasterTransferBlocking(HW_SPI0,`
  `NULL,`
  `&spiSourceBuffer,`
  `&spiSinkBuffer,`
  `1,`
  `1000);`

Sends the data contained in the source buffer and stores the incoming data in the sink buffer returning a specific error value.

Parameters:

Instance →select the SPI instance, in this case SPI0 is used.

device →pointer to the bus configuration structure. It can be set to NULL if the bus configuration was done before.

sendBuffer →Pointer to the source buffer.

receiveBuffer →Pointer to the sink buffer

transferByteCount →Number of bytes that will be transmitted.

timeout → Value handled by the OSA that determinates how long the function will block the continuity of the code. It can be set to OSA_WAIT_FOREVER to indicate that function will be blocking until a returned error code.

3.5 SPI Slave Configuration.

The SPI Slave configuration is very similar to the master configuration.

- **Slave User Configuration structure:** For the SPI Slave configuration it is only necessary to set the user configuration structure using the same parameters as in master configuration.
  An extra parameter in the user configuration for the slave can be set:
  - `slaveUserConfig.dummyPattern = DSPI_DEFAULT_DUMMY_PATTERN;`
    Select the value that will be send by the slave when the transmit buffer does not have any data.
- **Slave Configuration Driver**: The slave user configuration has to be passed to the initialization driver to complete the process.
  - **`DSPI_DRV_SlaveInit`**`(HW_SPI0, &dspiSlaveState, &slaveUserConfig);`
    The driver sets the SPI module with the parameters configured in the user configuration structure.
    Parameters:
    Instance →select the SPI instance, in this case SPI0 is used.
    dSPIstate →pointer to a data state structure. It needs to be created by the user to allocate memory for it.
    slaveConfig →pointer to the user configuration structure mentioned before.

3.6 SPI Slave Transfer.

As in master mode the slave mode drivers include blocking and non blocking transfer functions. The structure of the function is very similar to the master transfer function.

- **`DSPI_DRV_SlaveTransferBlocking`**`( HW_SPI0,`
  `NULL,`
  `&spiSourceBuffer,`
  `&spiSinkBuffer,`
  `1,`
  `OSA_WAIT_FOREVER);`

  Stores the incoming data in the sink buffer and sends the data contained in the source buffer returning a specific error value.

Parameters:

Instance →select the SPI instance, in this case SPI0 is used.

sendBuffer →Pointer to the source buffer.

receiveBuffer →Pointer to the sink buffer

transferByteCount →Number of bytes that will be transmitted.

timeout → Value handled by the OSA that determinates how long the function will block the continuity of the code. It can be set to OSA_WAIT_FOREVER to indicate that function will be blocking until a returned error code.

4 SPI Master-Slave example.

To test the code two FRDM-K64Fs are used. The philosophy of the example is the next:

The Master board is configured to send one of three possible messages to the Slave device every time the SW2 is pressed, confirming the Master is calling the transfer function the Blue led is toggled. The possible values are 0x52 (R), 0x47 (G) or 0x42 (B) and are sent in that order.

The Slave board is receiving the data from the Master, if it receives R it will turn the Red led and is the same process for G or B. Every time the Slave receives data it writes to the data buffer a byte that will be sent in the next transfer.

The Boards configuration is showed in the image below.

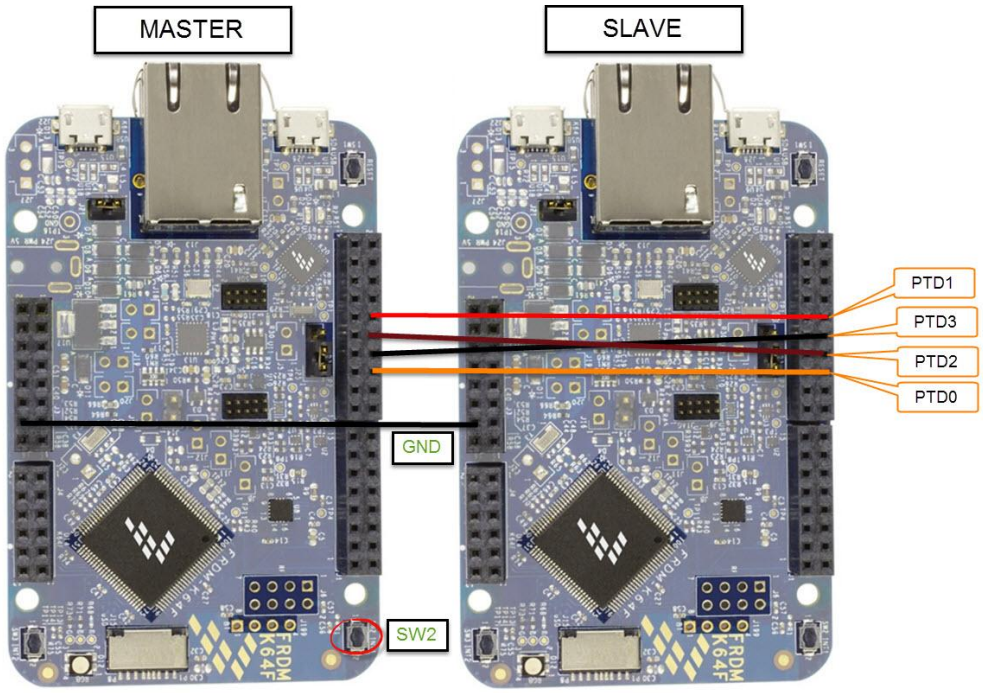
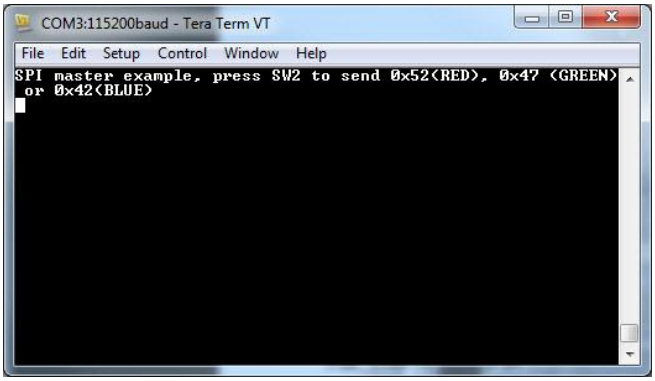


Figure 1. Boards Configuration

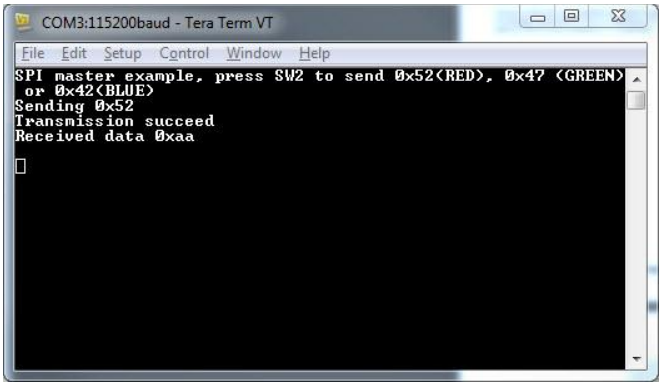
4.1 Running the Example.

Once one board is running the master code and the other one is running the slave code and are correctly wired the procedure is the next:

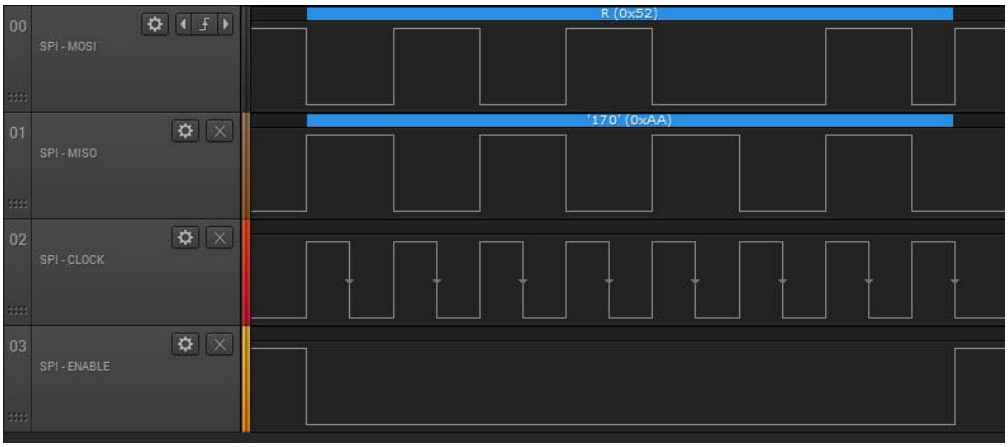
1. Run master board, the next is shown in the terminal (115200 buadrate, 8 bit data, no parity and 1 stop bit) of the COMx port of Master board:



2. Run the slave board, you will see all the RGB led on for a second and then off. It indicates the program is running.
3. Press SW2 and see the Blue led on the master board being toggled. The slave board will receive the message and turn the corresponding led. This is the message of the terminal of the master board.

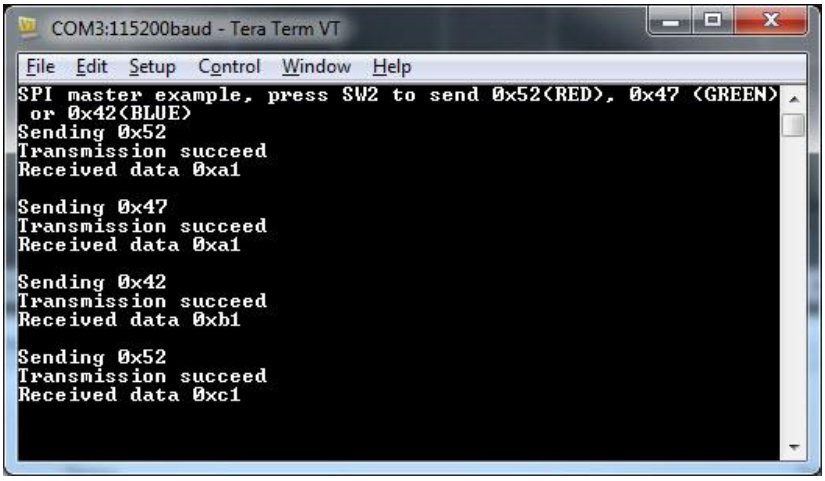


The terminal shows the sent message and the received data. Here is image of the signals viewed on a logic analyzer.



KSDK SPI Master and Slave example using the FRDM-K64F

4. Press again the SW2 and a different message will be sent.



5 References.

- K64 Reference Manual
- Getting Started with Kinetis SDK (KSDK): <install_dir>\KSDK_1.1.0\doc
- Writing my first KSDK Application in KDS – Hello World and Toggle LED with Interrupt.

6 Example Main Code.

6.1 Master main code

```
#include "fsl_device_registers.h"
#include "fsl_os_abstraction.h"
#include "fsl_dspi_master_driver.h"
#include "board.h"

#define RED    0x52
#define GREEN  0x47
#define BLUE   0x42

bool pushflag = false;
uint8_t spiSourceBuffer = RED;
uint8_t spiSinkBuffer;

uint8_t RGBdataOut[3] = {RED, GREEN, BLUE};
uint8_t RGBcount = 0;
int main(void)
{
    /* Write your code here */
        hardware_init();
        dbg_uart_init();
        OSA_Init();

        GPIO_DRV_Init(switchPins, ledPins);

        printf("SPI master example, press SW2 to send 0x52(RED), 0x47 (GREEN) or 0x42(BLUE)
        \n\r");
```

```
        configure_spi_pins(HW_SPI0);
        dspi_master_state_t dspiMasterState; // simply allocate memory for this
        // configure the members of the user config //
        dspi_master_user_config_t userConfig;
        userConfig.isChipSelectContinuous = false;
        userConfig.isSckContinuous = false;
        userConfig.pcsPolarity = kDspiPcs_ActiveLow;
        userConfig.whichCtar = kDspiCtar0;
        userConfig.whichPcs = kDspiPcs0;  //Selects the Chip select
        // init the DSPI module //
        DSPI_DRV_MasterInit(HW_SPI0, &dspiMasterState, &userConfig);
        // Define bus configuration.
        uint32_t calculatedBaudRate;
        dspi_device_t spiDevice;
        spiDevice.dataBusConfig.bitsPerFrame = 8;
        spiDevice.dataBusConfig.clkPhase = kDspiClockPhase_SecondEdge;
        spiDevice.dataBusConfig.clkPolarity = kDspiClockPolarity_ActiveHigh;
        spiDevice.dataBusConfig.direction = kDspiMsbFirst;
        spiDevice.bitsPerSec = 50000;
        // configure the SPI bus //
        DSPI_DRV_MasterConfigureBus(HW_SPI0, &spiDevice, &calculatedBaudRate);

    for (;;) {

        if(GPIO_DRV_ReadPinInput(kGpioSW1) == 0)    //is switch pressed?
        {
               pushflag = true;
        }
        OSA_TimeDelay(250);
        if(pushflag)
        {
               pushflag = false;
               GPIO_DRV_TogglePinOutput(BOARD_GPIO_LED_BLUE);
               printf ("Sending 0x%x \n\r",spiSourceBuffer & 0xFF);
               //SEND DATA
            dspi_status_t Error = DSPI_DRV_MasterTransferBlocking(HW_SPI0,
                                                                  NULL,
                                                                  &spiSourceBuffer,
                                                                  &spiSinkBuffer,
                                                                  1,
                                                                  1000);
            if (Error == kStatus_DSPI_Success)
            {
               printf ("Transmission succeed  \n\r");
               printf ("Received data 0x%x \n\n\r",spiSinkBuffer & 0xFF);
               RGBcount++;
               if(RGBcount > 2 ){RGBcount = 0;}
               spiSourceBuffer = RGBdataOut[RGBcount];
            }
        }
    }
    return 0;
}
```

6.2 Slave main code

```
#include "fsl_device_registers.h"
#include "fsl_os_abstraction.h"
#include "fsl_dspi_slave_driver.h"
#include "fsl_dspi_hal.h"
#include "board.h"

/*RGB values*/
#define RED    0x52
#define GREEN  0x47
#define BLUE   0x42

#define RED_LED                     0
#define GREEN_LED           1
#define BLUE_LED            2
#define ALL_ON              3
#define ALL_OFF                     4

void Turn_Led (uint8_t Led);

uint8_t spiSourceBuffer = 0xAA;
uint8_t spiSinkBuffer;

int main(void)
{

    /* Write your code here */
       // declare which module instance you want to use
       hardware_init();
       dbg_uart_init();
       OSA_Init();

       GPIO_DRV_Init(switchPins, ledPins);

       Turn_Led (ALL_ON);
       OSA_TimeDelay(1000);
       Turn_Led (ALL_OFF);

       configure_spi_pins(HW_SPI0);

       dspi_slave_state_t dspiSlaveState;
       dspi_slave_user_config_t slaveUserConfig;
       slaveUserConfig.dataConfig.clkPhase = kDspiClockPhase_SecondEdge;
       slaveUserConfig.dataConfig.clkPolarity = kDspiClockPolarity_ActiveHigh;
       slaveUserConfig.dataConfig.bitsPerFrame = 8;
       slaveUserConfig.dataConfig.direction = kDspiMsbFirst;
       slaveUserConfig.dummyPattern = DSPI_DEFAULT_DUMMY_PATTERN;

       DSPI_DRV_SlaveInit(HW_SPI0, &dspiSlaveState, &slaveUserConfig);

    for (;;) {

       dspi_status_t Error = DSPI_DRV_SlaveTransferBlocking(HW_SPI0,
                                                            &spiSourceBuffer,
                                                            &spiSinkBuffer,
                                                            1,
                                                            OSA_WAIT_FOREVER);
```

```
if (Error == kStatus_DSPI_Success)
                {
                        switch(spiSinkBuffer)
                        {
                                        case((uint8_t)RED):
                                                Turn_Led (RED_LED);
                                                spiSourceBuffer = 0xA1;
                                                break;
                                        case((uint8_t)GREEN):
                                                Turn_Led (GREEN_LED);
                                                spiSourceBuffer = 0xB1;
                                                break;
                                        case((uint8_t)BLUE):
                                                Turn_Led (BLUE_LED);
                                                spiSourceBuffer = 0xC1;
                                                break;
                                        case((uint8_t)0xFF):
                                                Turn_Led (ALL_ON);
                                                break;
                        }
                }

    }
    return 0;
}

void Turn_Led (uint8_t Led)
{
        switch(Led)
        {
                case 0:
                        GPIO_DRV_ClearPinOutput(BOARD_GPIO_LED_RED);
                        GPIO_DRV_SetPinOutput(BOARD_GPIO_LED_GREEN);
                        GPIO_DRV_SetPinOutput(BOARD_GPIO_LED_BLUE);
                        break;
                case 1:
                        GPIO_DRV_SetPinOutput(BOARD_GPIO_LED_RED);
                        GPIO_DRV_ClearPinOutput(BOARD_GPIO_LED_GREEN);
                        GPIO_DRV_SetPinOutput(BOARD_GPIO_LED_BLUE);
                        break;
                case 2:
                        GPIO_DRV_SetPinOutput(BOARD_GPIO_LED_RED);
                        GPIO_DRV_SetPinOutput(BOARD_GPIO_LED_GREEN);
                        GPIO_DRV_ClearPinOutput(BOARD_GPIO_LED_BLUE);
                        break;
                case 3:
                        GPIO_DRV_ClearPinOutput(BOARD_GPIO_LED_RED);
                        GPIO_DRV_ClearPinOutput(BOARD_GPIO_LED_GREEN);
                        GPIO_DRV_ClearPinOutput(BOARD_GPIO_LED_BLUE);
                        break;
                case 4:
                        GPIO_DRV_SetPinOutput(BOARD_GPIO_LED_RED);
                        GPIO_DRV_SetPinOutput(BOARD_GPIO_LED_GREEN);
                        GPIO_DRV_SetPinOutput(BOARD_GPIO_LED_BLUE);
                        break;
        }
}
```