Freescale Semiconductor
Application Note

Document Number:AN4652
Rev. 0, 01/2013

An I2C Driver Based on Interrupt and Blocking Mechanism for MQX

by: Guo Jia

Contents

1 Introduction

This application note implements an I2C driver on MQX based on interrupt and blocking mechanism, which masked the details of the I2C module operations and simplifies the application code to a great extent. It covers both the master and slave modes. The code is validated on the K60N512-TWR board.

2 How the driver is designed

This driver is fully driven by interrupt. Following is the operation of driver in master and slave modes.

- Master mode: When it is working as a master and sending data, the application task passes the address and length of buffer to driver and after that, it is blocked. After the driver finishes transmitting the data in the buffer, it resumes the application task. To reduce memory consumption, the driver does not make its own buffer and copy the data from application to it. The driver uses the buffer at application layer directly. In the same way, when the master is receiving, it passes the address and length of buffer to the driver, and then, it is pended by the driver until driver finishes the reading operation. See Figure 1.

- Slave mode: There are some differences when it is working as a slave. A slave is working in a passive way, not as a master does. So, when it is sending data, it keeps blocked until the master reads it. When it is reading, it will keep blocked until the master sends data to it. See Figure 1.

See the following figure for the illustration of the design.

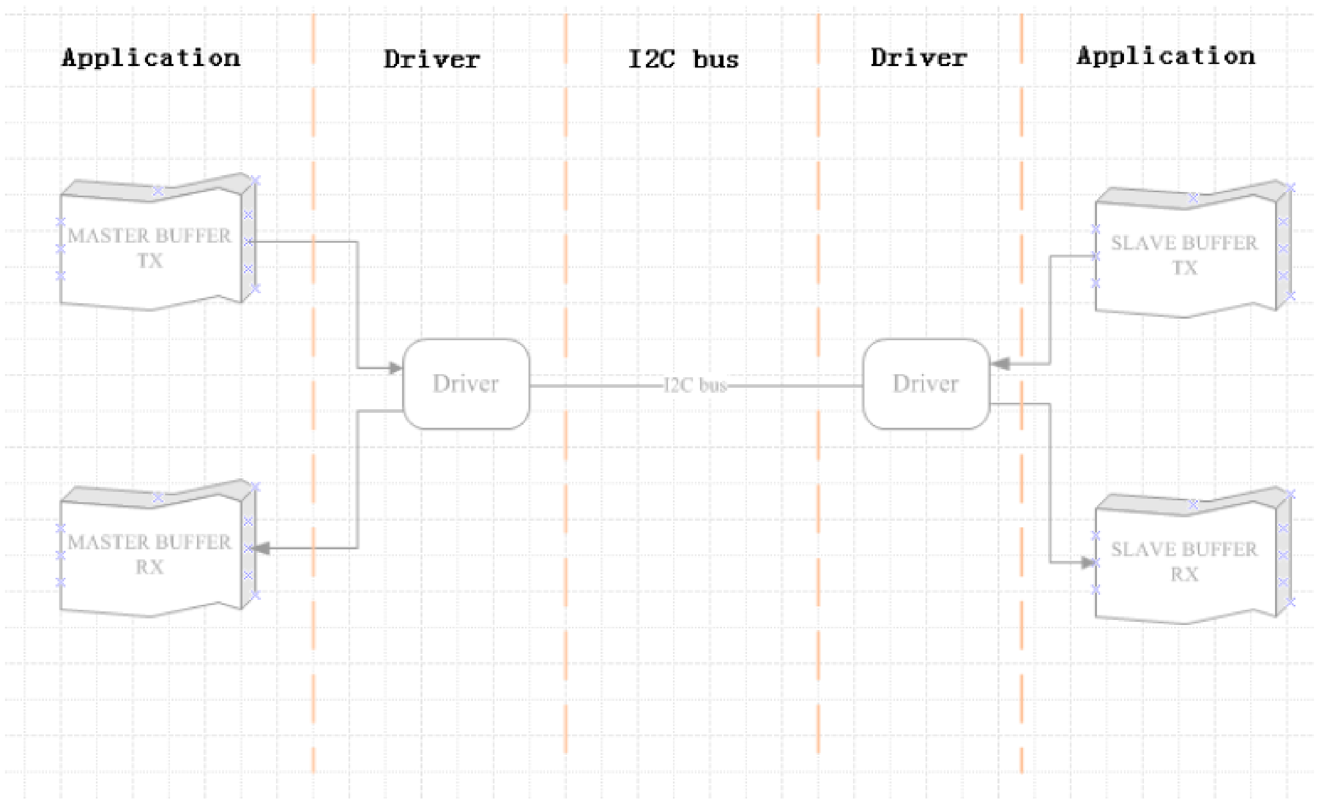
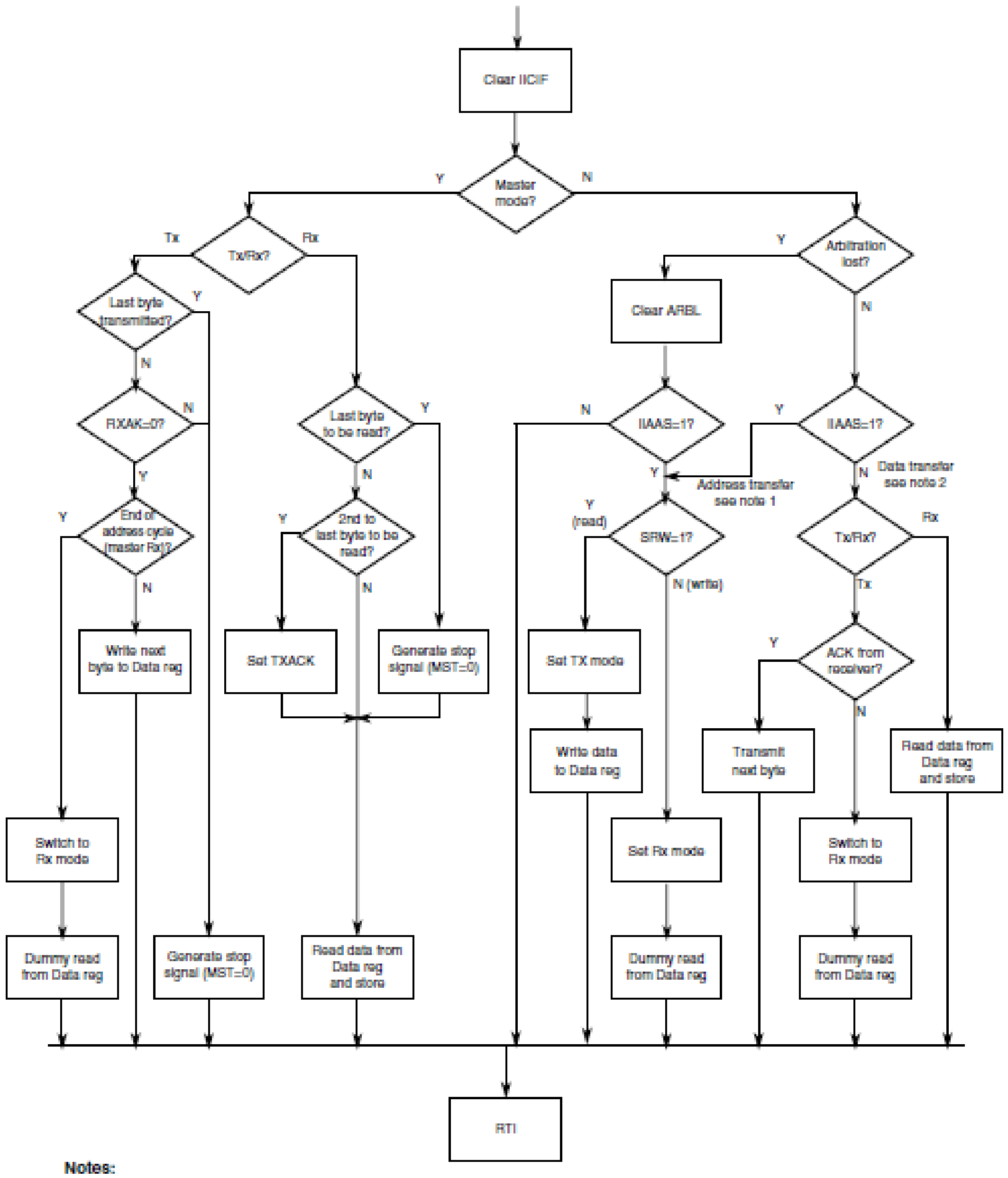


Figure 1. Illustration of the design

3 Program flow

Figure 2 shows the program flow chart for the operation of the I2C driver in both Master and Slave modes. For more information, see K60P120M100SF2RM: K60 Reference Manual, available on **freescale.com**.



Notes:
1. If general call is enabled, check to determine if the received address is a general call address (0x00). If the received address is a general call address, the general call must be handled by user software.
2. When 10-bit addressing addresses a slave, the slave sees an interrupt following the first byte of the extended address. Ensure that for this interrupt, the contents of the Data register are ignored and not treated as a valid data transfer.

Figure 2. Typical I2C interrupt routine

4 Communication time sequence

4.1 Typical time sequence

The communication time sequence is referred from the MMA7660 device, and this is a typical sequence which can meet the requirement of a lot of applications. For other sequence with special requirement, the driver needs to be modified a little to adapt to the new application. See Figure 3 for master writing sequence and Figure 4 for master reading sequence.

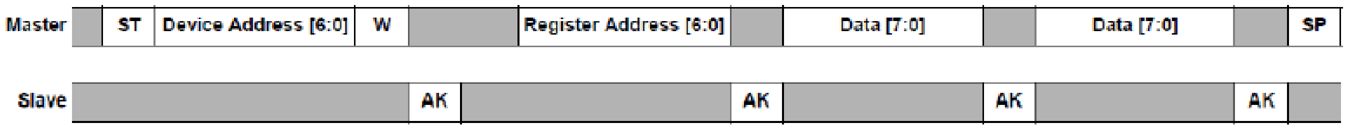


Figure 3. Writing sequence

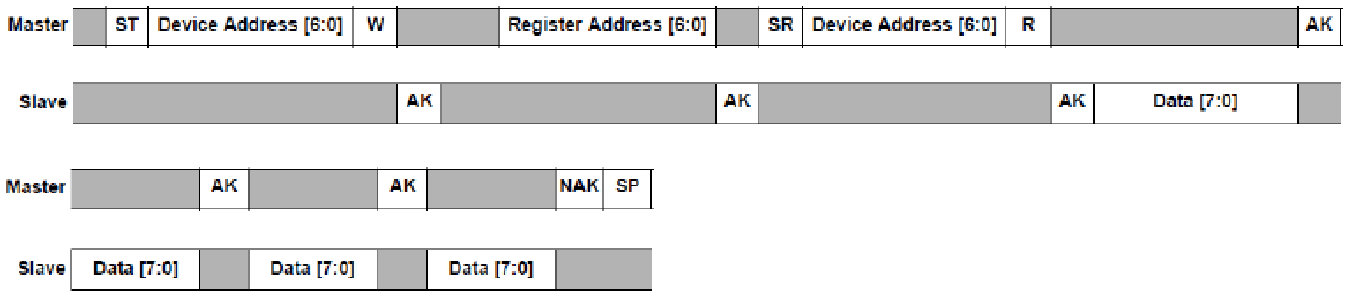


Figure 4. Reading sequence

4.2 Modification to the typical time sequence

As on some Kinetis device, when I2C module is working in slave state, it cannot generate an interrupt when it receives a STOP signal sent by the master. So, the driver does not know when to give the data to the application task. To solve this issue, one command is occupied. When the register address is 0xFF, the slave assumes it to be a STOP signal and passes the data to task.

5 Key function and macro explanation

The following table describes a list of all the macros and the explains the function of each of these.

Function/Macro name	Explanation
_ki2c_isr	The entry function for I2C interrupt
_isr_ii2c_slave	The interrupt service routine for slave
_isr_ii2c_master	The interrupt service routine for master
_ki2c_int_fb_tx	The interface with application, for sending data, mapped to **fwrite**.
_fb_tx_master	For master, sending data
_fb_rx_master	For master, receiving data
_ki2c_int_fb_rx	The interface with application, for receiving data, mapped to **fread**.
_fb_tx_slave	For slave, sending data

Table continues on the next page...

Function/Macro name	Explanation
_fb_rx_slave	For slave, receiving data
#define DEBUG_ENABLE	This is a macro which enables to dump the information for debugging. For normal usage, it is masked.

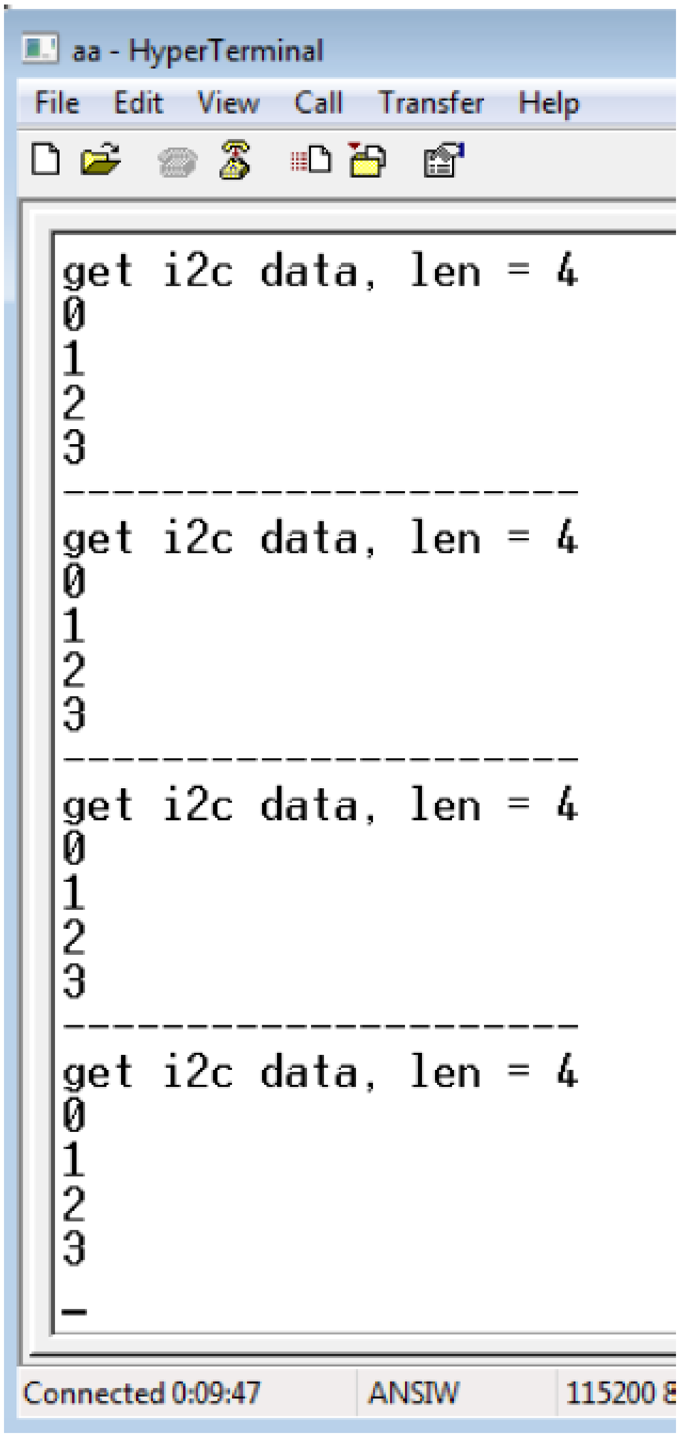
6 Demo code for using this driver

The following subsections provide the demo code for using the I2C driver in Master and Slave modes.

6.1 Demo code for master

```
struct STRU_I2C_BUFFER
{
    char dst_addr;
    char reg_addr;
    char data[100]; // Here is the buffer for sending or
                // receiving data
};

struct STRU_I2C_BUFFER i2c_buf_rx;
struct STRU_I2C_BUFFER i2c_buf_tx;
int_32 i2c_fb_test_master(void)
{
    uint_32 param;
    int i;
    int len;

    file_iic0 = fopen("ii2c0fb:", NULL);
    if (file_iic0 == NULL)
    {
        printf("\nOpen the IIC0 driver failed!!!\n");
        return IIC_ERR;
    }

    param = 100000;
    ioctl(file_iic0, IO_IOCTL_I2C_SET_BAUD, &param);

    i2c_buf_tx.dst_addr = 0x50; // The I2C slave address
    i2c_buf_tx.reg_addr = 0;    // You may view this as a register
                                // address or a command to slave.
    i2c_buf_rx.dst_addr = 0x50; // The same as above, for receiving

    i2c_buf_rx.reg_addr = 0;    // The same as above, for receiving


    for(i=0;i<8;i++)            // initialize data
        i2c_buf_tx.data[i] = 0xb0+i;

    while(1)
    {
        printf("---------------------\n");
        // send 4 bytes to slave
        len = fwrite(&i2c_buf_tx, 1, 4, file_iic0);
        if( len < 4 )
            printf("send failed, len = %d \n", len);
        else
            printf("send ok, len = %d \n", len);
        // clear receiving buffer
        memset(i2c_buf_rx.data, 0, 4);
        // receive 4 bytes from slave
        len = fread(&i2c_buf_rx, 1, 4, file_iic0);
        printf("get i2c data, len = %x\n", len);
        for(i=0; i<len; i++)
            printf("%x \n", i2c_buf_rx.data[i]);
        _time_delay(10);
    }
}
```

Figure 5 shows the running result for this demo.



```
get i2c data, len = 4
0
1
2
3
---------------------
get i2c data, len = 4
0
1
2
3
---------------------
get i2c data, len = 4
0
1
2
3
---------------------
get i2c data, len = 4
0
1
2
3
_
```

Figure 5. Console output for master reading

6.2 Demo code for slave

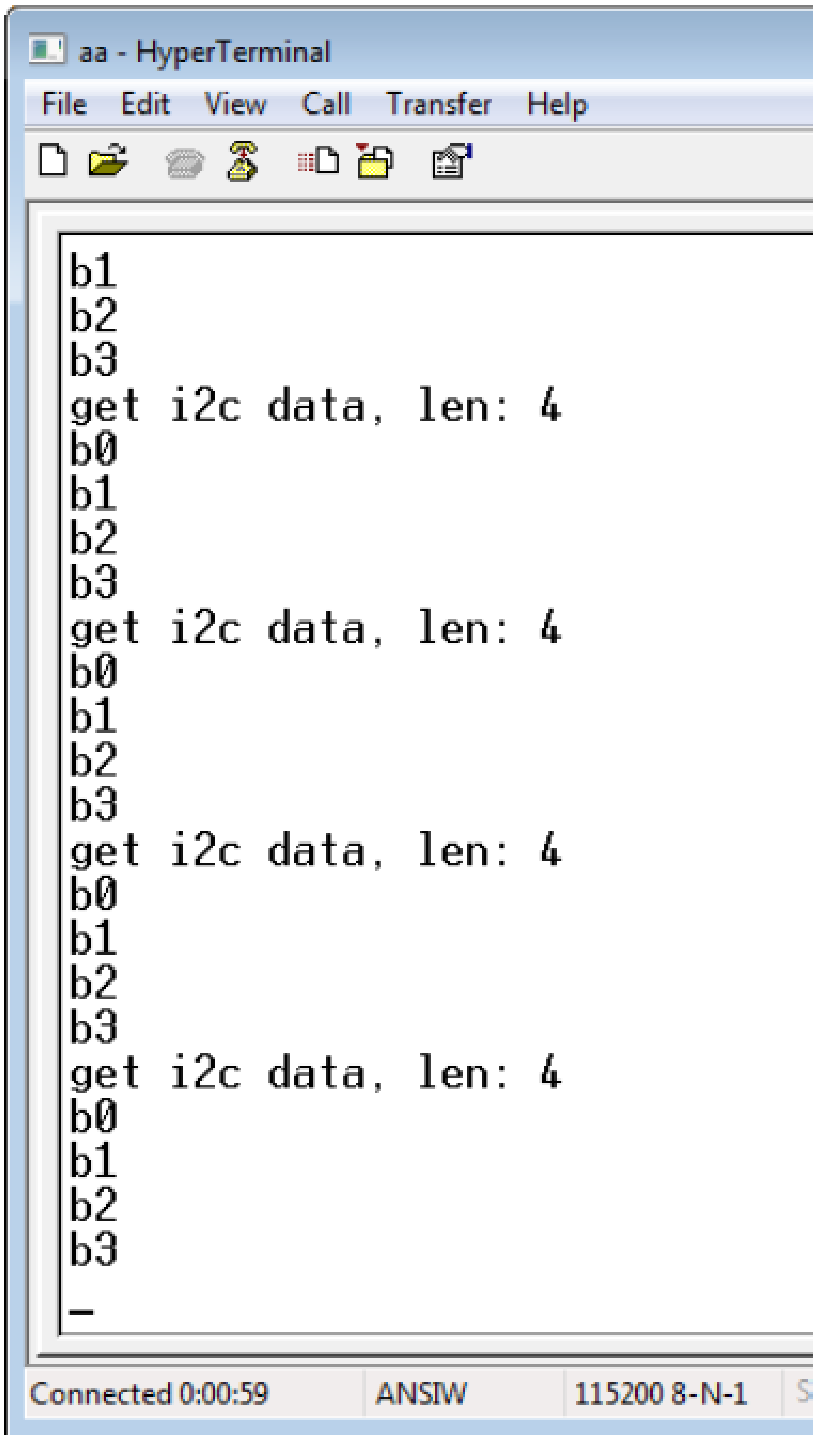
In this demo code, two tasks are created for sending and receiving separately. But this is not a must; the user can send and receive data in the same task too.

NOTE

As the driver shares the same buffer with the task, make sure that the data must not be modified by other tasks or interrupts when the driver is working.

```
#define I2C0_SLAVE_ADDRESS 0x50
char buf_i2c_rx[256];
char buf_i2c_tx[256];

void task_slave_rx(uint_32 initial_data)
{
    uint_32 param;
    uint_32 len;
    int i;

    printf("I2C slave demo for FB. ******************\n");
    file_iic0 = fopen("ii2c0fb:", NULL);

    // Set to slave mode with specified slave address
    param = I2C0_SLAVE_ADDRESS;
    ioctl(file_iic0, IO_IOCTL_I2C_SET_SLAVE_MODE, &param);

    _task_create(0, TASK_I2C_SLAVE_TX, 0);

    while(1)
    {
        // clearing and reading
        memset(buf_i2c_rx, 0, 256);
        len = fread(buf_i2c_rx, 1, 256, file_iic0);
        if(len > 0)
        {
            // show data received
            printf("get i2c data, len: %d\n", len);
            for(i=0;i<len;i++)
                printf("%x\n", buf_i2c_rx[i]);
        }
        else
            printf("read failed.\n");
     }
}

void task_slave_tx(uint_32 initial_data)
{
    int i;
    int len;
    for(i=0;i<256;i++)
        buf_i2c_tx[i] = i;

    while(1)
    {
        len = fwrite(buf_i2c_tx, 1, 4, file_iic0);
        if(len < 4)
            printf("send fail. len = %d \n", len);
        else
            printf("send OK. len = %d \n", len);
    }
}
```

Figure 6 shows the running result for this demo.



```
b1
b2
b3
get i2c data, len: 4
b0
b1
b2
b3
get i2c data, len: 4
b0
b1
b2
b3
get i2c data, len: 4
b0
b1
b2
b3
get i2c data, len: 4
b0
b1
b2
b3
_
```

Figure 6. Console output for slave reading

7 How to install this driver

To install the driver, please follow these steps:

1. Copy the file **i2c_int_k_fb.c** released with this application note (contained in AN4655SW.zip, on **freescale.com**) to the folder <mqx_install>\mqx\source\io\i2c\int and add it to the BSP project. See the following figure.

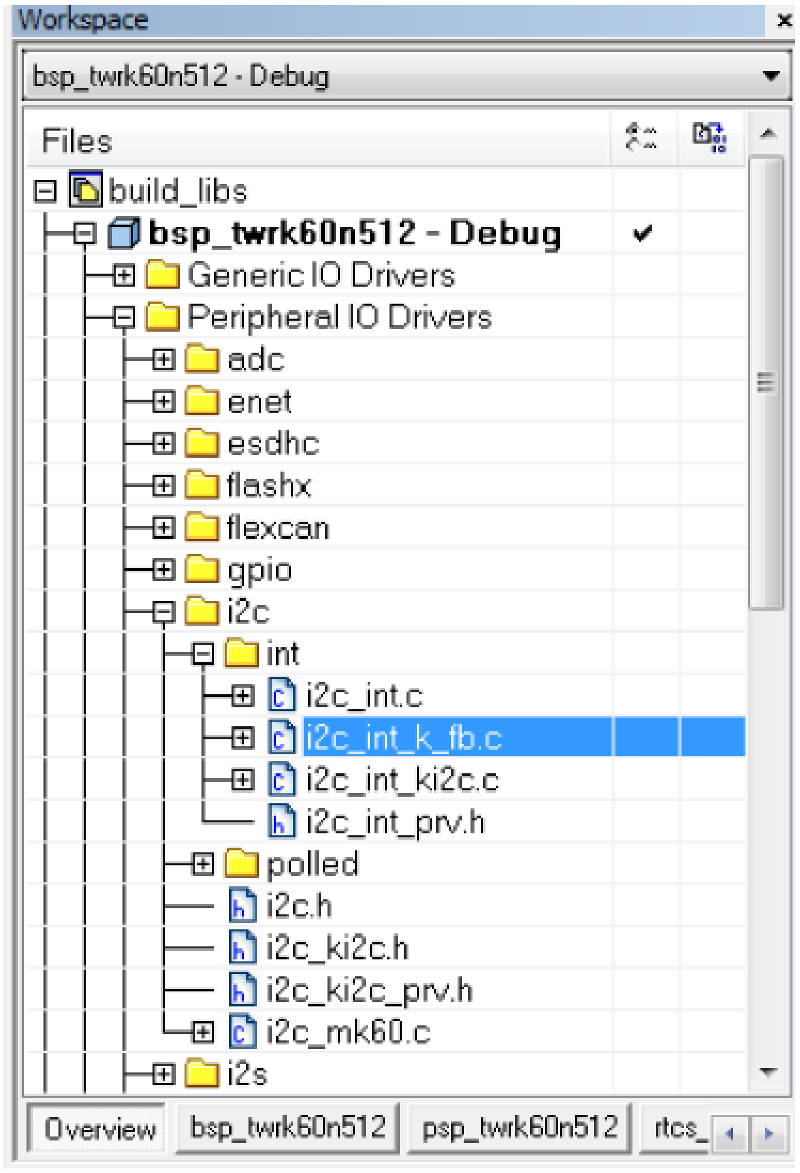


Figure 7. Add i2c_int_k_fb.c to BSP project

2. Add the following line to **i2c_ki2c.h**:

```
extern uint_32 _ki2c_int_fb_install (char_ptr, KI2C_INIT_STRUCT_CPTR);
```

3. In <mqx_install>\mqx\source\bsp\twrk60n512\init_bsp.c, add the following lines, there's a demo file in the software AN4652SW.zip associated with this application note.

```
#if BSPCFG_ENABLE_II2C0_FB
    _ki2c_int_fb_install("ii2c0fb:", &_bsp_i2c0_init);
#endif
#if BSPCFG_ENABLE_II2C1_FB
    _ki2c_int_fb_install("ii2c1fb:", &_bsp_i2c1_init);
#endif
```

4. In <mqx_install>\config\twrk60n512\user_config.h, add the following lines:

```
#define BSPCFG_ENABLE_II2C0_FB    1
#define BSPCFG_ENABLE_II2C1_FB    0
```

5. Then, rebuild it, and the driver is available in application project.

8 Conclusion

In this application note, an I2C driver based on interrupt and blocking mechanism for MQX is introduced. At the beginning, its mechanism is introduced, then the program flow and time sequence is shown. In order to make it easy for reader, key function explanation, demo code, and steps for installment are also discussed.

9 References

The following reference documents are available on **freescale.com.**

- K60P120M100SF2RM: K60 Reference Manual
- AN3902: How to Develop I/O Drivers for MQX

Document Number: AN4652
Rev. 0, 01/2013