

Agenda



1. Module Overview
2. On-chip Interconnections and Dependencies
3. Hardware Configuration
4. Software Configuration
5. Example Use Case
6. I2C Frequently Asked Questions (FAQs)

In this presentation, we'll cover:

- An overview of the module
- The on-chip interconnections and inter-module dependencies
- Hardware and software configurations
- An example use case for reference
- And a few frequently asked questions



First, let's take a brief look at the I2C protocol and overall features of the Kinetis MCU I2C module.

I2C Protocol Overview

- Only two bus lines are required with possible ground connection:
 - **SDA**: Data line
 - **SCL**: Clock line
 - GND
- SCL and SDA pins of each device on the bus should have an open drain configuration, so external pull-ups are required.
- Simple master/slave relationships between components
 - Each slave has a unique bus address
- No strict baud rate requirements:
 - Master generates the clock signal
- I2C is a true multi-master bus
 - Provides arbitration and collision detection



External Use | 3



I2C Protocol Overview

I2C is a widely used communication protocol to transfer data between devices.

It uses only 2 bus lines: SCL (or clock) and SDA (or data). Note that your system may also require a ground connection if a master and slave device do not reference the same ground connection.

Data transfer is made in a simple master-slave relationship between components with each slave possessing its own, unique bus address.

The protocol does not define a strict baud rate, but the standard transfer rates are up to 1 MHz. The master device determines the frequency.

It is possible to have multiple I2C devices in a multi-master or multi-slave bus configuration with arbitration loss and collision detection.

I2C Module Main Features

- Compatible with the I2C bus specification
- Multimaster operation
- Software programmable for one of 64 clock frequencies
- Arbitration lost interrupt with automatic mode switch from master to slave
- General call recognition
- 10-bit addressing
- Support for SMBUS specification, version 2
- Programmable input glitch filter
- Range slave address support
- Supports clock stretching



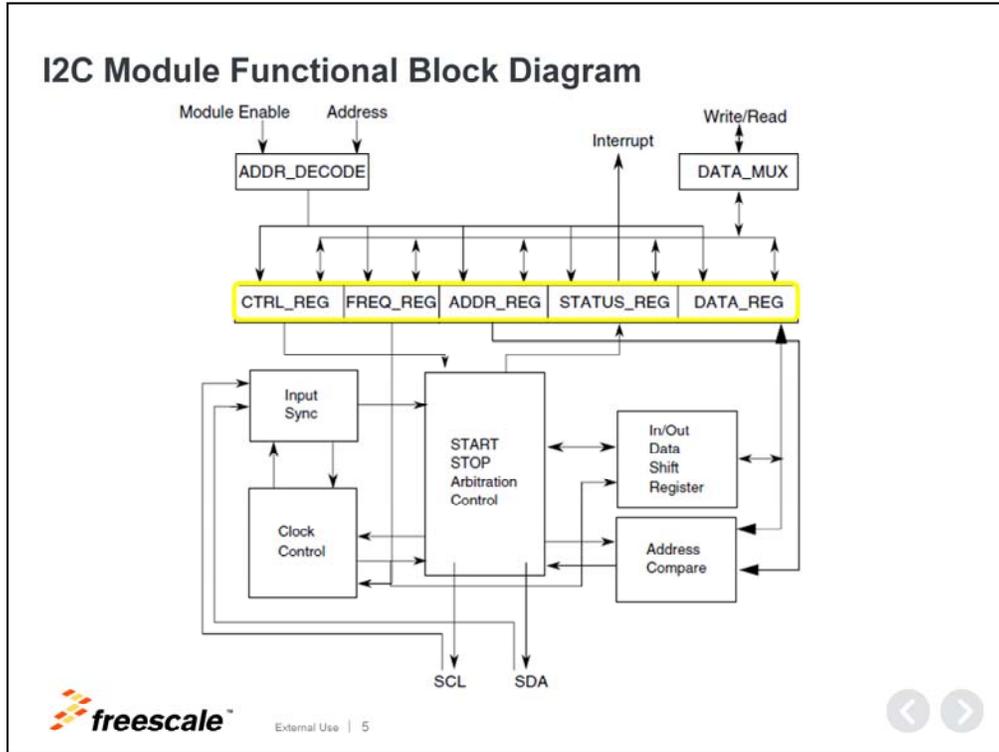
External Use | 4



I2C Module Main Features

The key features of the I2C module in Kinetis MCUs include:

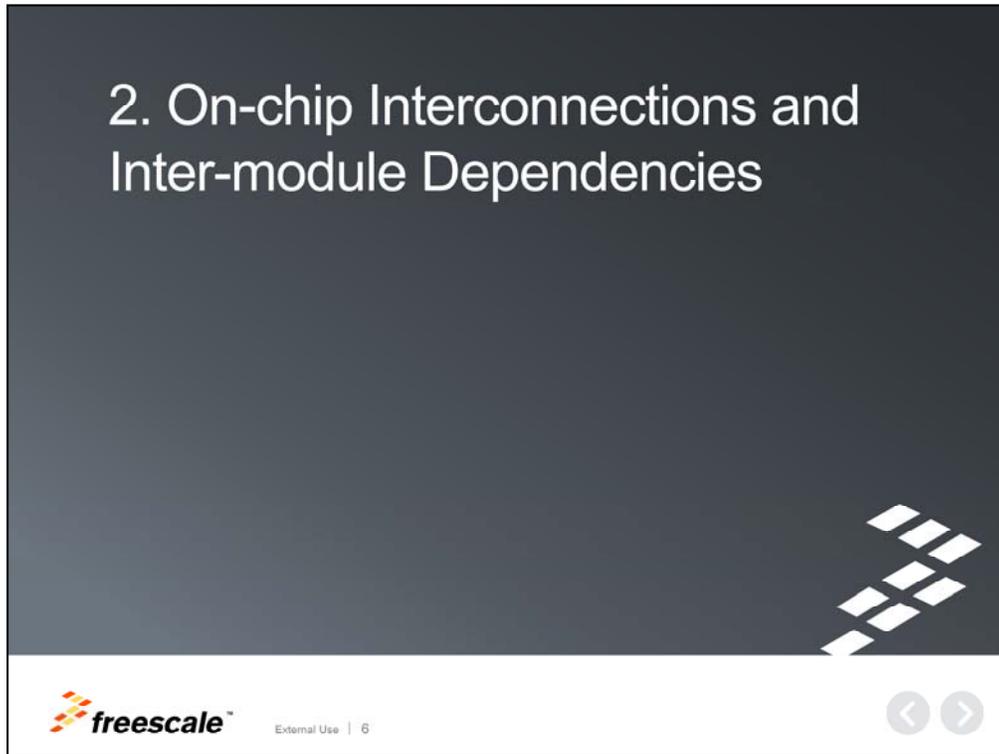
- I2C bus specification support
- Support for 10-bit addressing scheme
- SMBUS version 2 specification support
- Programmable input filter to absorb and eliminate glitches on the bus
- and the ability to wake up the MCU with a slave address match



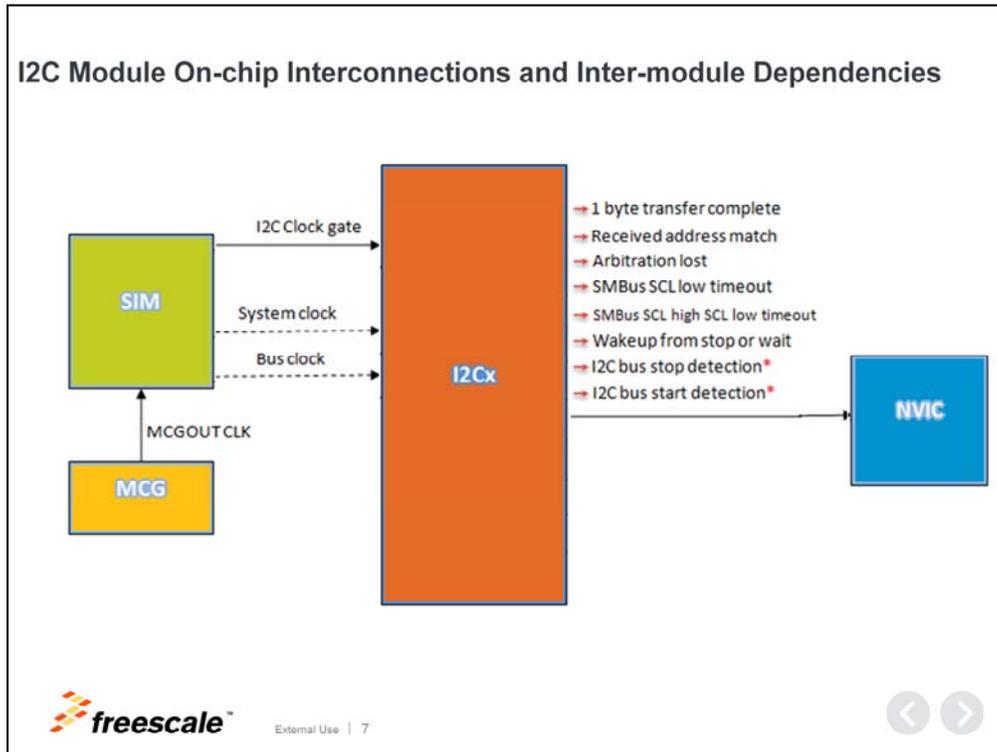
I2C Module Functional Block Diagram

This is a general block diagram of the I2C module. The most relevant parts are the register blocks – outlined in yellow here. The registers provide status flags that serve as an interface to configure the module. The data register provides an interface for the application to send or receive data to or from the I2C bus.

2. On-chip Interconnections and Inter-module Dependencies



Next, let's discuss on-chip interconnection and inter-module dependencies.



I2C Module On-chip Interconnections and Inter Module Dependencies

This diagram displays the I2C interconnections and its relationship with other peripherals or systems.

On left, we see the path for the I2C module source clock. The clock is initially generated by the MCG module, passing then to the SIM module which delivers an equal or divided frequency. Note the dotted lines for system and bus clocks; this means that the I2C clock source will depend on the specific Kinetis device and the specific I2C module instance. Please refer to your Kinetis MCU reference manual for help determining the exact clock source.

From the SIM module, it is very important to enable the I2C clock gate before accessing the I2C registers.

On the right, the possible interrupt triggers are displayed. The triggers marked with a red star are not present in all Kinetis devices. Refer to your device specific reference manual for details. All of the interrupts in an I2C module instance share a single interrupt vector mapped to the NVIC module. There are two required steps to enable interrupts:

1. Enable the desired interrupt triggers in the I2C module, and
2. Enable the common interrupt vector in the NVIC module for the corresponding I2C module instance

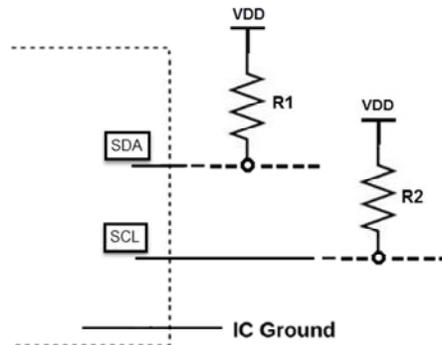
3. Hardware Configuration



Now let's examine a basic hardware setup.

I2C Hardware Configuration

- The I2C pins in Kinetis devices are either pseudo open drain or true open drain.
- In either case, external pull ups for the SDA and SCL lines are required



I2C Hardware Configuration

An important hardware component of the I2C bus are the pull-up resistors. There should be one pull-up resistor per I2C line. This is because the I2C specification requires I2C devices to have open drain output stages. Usually the lines of the bus are pulled up to the same voltage level of the MCUs' VDD. If the Kinetis device has true open drain pins, then it is permissible to pull the pins up to 5 volts.

4. Software Configuration



External Use | 10



In this section, we will examine the I2C driver provided by the Kinetis Software Development Kit (or KSDK).

Kinetic Software Development Kit (KSDK) I2C Master Driver

Driver configuration

```

i2c_master_state_t master;
i2c_device_t device =
{
    .address = 0x7FU,
    .baudRate_kbps = 400
};

```

Init/Deinit and baudrate configuration

```

i2c_status_t I2C_DRV_MasterInit(uint32_t instance, i2c_master_state_t * master);
i2c_status_t I2C_DRV_MasterDeinit(uint32_t instance);
void I2C_DRV_MasterSetBaudRate(uint32_t instance, const i2c_device_t * device);

```

Transmission APIs

```

i2c_status_t I2C_DRV_MasterSendDataBlocking(uint32_t instance, const i2c_device_t * device, const uint8_t * cmdBuff,
uint32_t cmdSize, const uint8_t * txBuff, uint32_t txSize, uint32_t timeout_ms);
i2c_status_t I2C_DRV_MasterSendData(uint32_t instance, const i2c_device_t * device, const uint8_t * cmdBuff,
uint32_t cmdSize, const uint8_t * txBuff, uint32_t txSize);
i2c_status_t I2C_DRV_MasterGetSendStatus(uint32_t instance, uint32_t *bytesRemaining);
i2c_status_t I2C_DRV_MasterAbortSendData(uint32_t instance);

```

Reception APIs

```

i2c_status_t I2C_DRV_MasterReceiveDataBlocking(uint32_t instance, const i2c_device_t * device, const uint8_t * cmdBuff,
uint32_t cmdSize, uint8_t * rxBuff, uint32_t rxSize, uint32_t timeout_ms);
i2c_status_t I2C_DRV_MasterReceiveData(uint32_t instance, const i2c_device_t * device, const uint8_t * cmdBuff,
uint32_t cmdSize, uint8_t * rxBuff, uint32_t rxSize);
i2c_status_t I2C_DRV_MasterGetReceiveStatus(uint32_t instance, uint32_t *bytesRemaining);

```


External Use | 11


Kinetic Software Development Kit (KSDK) I2C Master Driver

The I2C master driver functions require a structure variable of type “i2c_master_state”. This should be a persistent structure, so it is recommended to make it global, static or defined in the main function.

A simple structure variable of type “i2c_device” is also needed. This variable should be initialized with the I2C slave address and the desired baud rate.

APIs initialize and de-initialize the driver, as well as set the baud rate.

The transmission APIs are for data transmission. There are blocking and non-blocking calls.

- Blocking calls do not exit the function until the I2C transfer is complete.
- Non-blocking functions simply start the I2C transfer and then exit the function.

Finally there are APIs for data reception, which also have blocking or non-blocking variants.

Kinetis Software Development Kit (KSDK) I2C Slave Driver

Driver configuration

```

i2c_slave_state_t slave;

i2c_slave_user_config_t userConfig =
{
    .address      = 0x7FU,
    .slaveCallback = NULL,
    .callbackParam = NULL,
    .slaveListening = false,
    .startStopDetect = false,
};
#ifdef FSL_FEATURE_I2C_HAS_START_STOP_DETECT
i2c_slave_user_config_t userConfig =
{
    .address      = 0x7FU,
    .slaveCallback = NULL,
    .callbackParam = NULL,
    .slaveListening = false,
    .startStopDetect = false,
};
#endif

```

Init/Deinit and get handler

```

i2c_status_t I2C_DRV_SlaveInit(uint32_t instance, const i2c_slave_user_config_t * userConfigPtr,
                               i2c_slave_state_t * slave);
i2c_status_t I2C_DRV_SlaveDeinit(uint32_t instance);
i2c_slave_state_t * I2C_DRV_SlaveGetHandler(uint32_t instance);

```


External Use | 12


Kinetis Software Development Kit (KSDK) I2C Slave Driver

The slave driver requires a structure variable of type “i2c_slave_state” and a structure of type “i2c_slave_user_config”. The “i2c_slave_user_config” variable holds the slave address, slave callback, callback parameter, slave listening mode, and start/stop detection depending on the device.

There are simple APIs for initialization, de-initialization and to retrieve a pointer to the slave’s state structure.

Kinetis Software Development Kit (KSDK) I2C Slave Driver – Continued

Transmission APIs

```
i2c_status_t I2C_DRV_SlaveSendData(uint32_t instance, const uint8_t * txBuff, uint32_t txSize);  
i2c_status_t I2C_DRV_SlaveSendDataBlocking(uint32_t instance, const uint8_t * txBuff,  
                                           uint32_t txSize, uint32_t timeout_ms);  
i2c_status_t I2C_DRV_SlaveGetTransmitStatus(uint32_t instance, uint32_t *bytesRemaining);  
i2c_status_t I2C_DRV_SlaveAbortSendData(uint32_t instance, uint32_t *txSize);
```

Reception APIs

```
i2c_status_t I2C_DRV_SlaveReceiveData(uint32_t instance, uint8_t * rxBuff, uint32_t rxSize);  
i2c_status_t I2C_DRV_SlaveReceiveDataBlocking(uint32_t instance, uint8_t * rxBuff, uint32_t rxSize,  
                                              uint32_t timeout_ms);  
i2c_status_t I2C_DRV_SlaveGetReceiveStatus(uint32_t instance, uint32_t *bytesRemaining);  
i2c_status_t I2C_DRV_SlaveAbortReceiveData(uint32_t instance, uint32_t *rxSize);
```



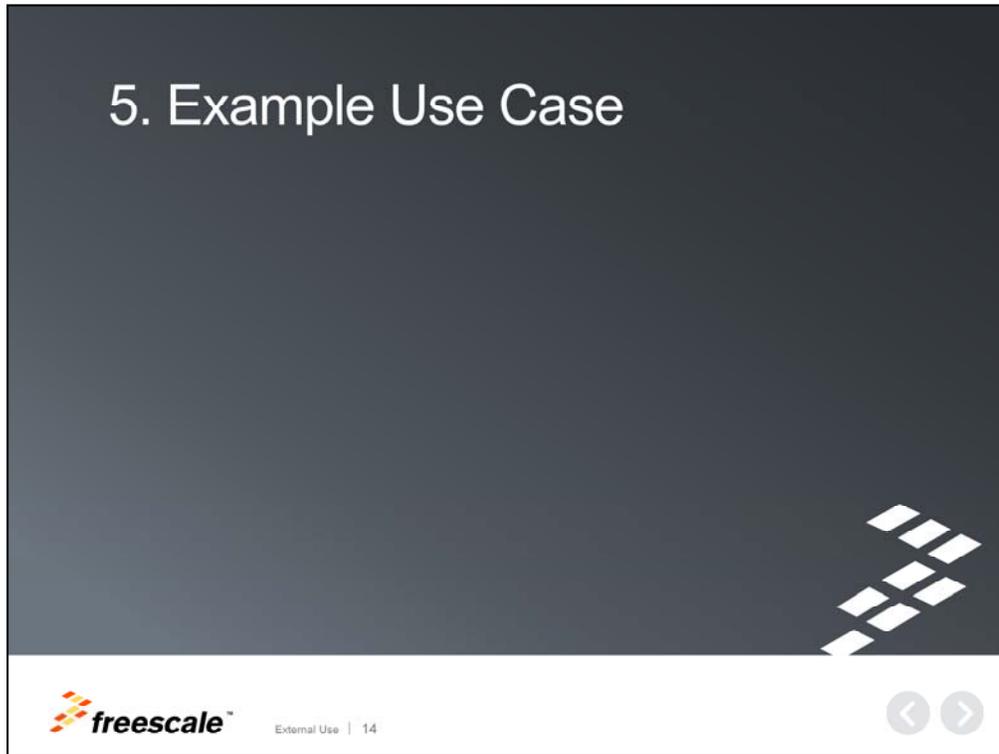
External Use | 13



Kinetis Software Development Kit (KSDK) I2C Slave Driver – Continued

These are the data transmission APIs.

and APIs for data reception.



Let's take a look at an I2C example available with Kinetis SDK.

KSDK I2C Example – Master Code

```
#define DATA_LENGTH 64

// Buffer store data to send to slave
uint8_t txBuff[DATA_LENGTH] = {0};
// Buffer store data to receive from slave
uint8_t rxBuff[DATA_LENGTH] = {0};

bool i2c_compare(uint8_t *txBuff, uint8_t *rxBuff, uint32_t count)
{
    uint32_t i;
    for (i = 0; i < count ; i++)
    {
        if (txBuff[i] != rxBuff[i])
        {
            return false;
        }
    }
    return true;
}

int main(void)
{
    uint32_t count = 0;
    uint32_t i = 0;
    i2c_master_state_t master;
    uint32_t bytesRemaining;

    i2c_device_t device =
    {
        .address = 0x7F,
        .baudRate_kbps = 400
    };
};
```



External Use | 15



Kinetis SDK I2C Example – Master Code

The next three slides are an example of basic master – slave communication. The code in this slide is the I2C master project.

The application consists of continuous iterations of the master sending 1 byte or more to the slave, along with a command byte indicating to the slave how many bytes to expect. Then the master waits for the slave to echo the bytes it received from the master. The communication continues with the master incrementing the number of bytes sent or received within each iteration.

```
hardware_init();
OSA_Init();

I2C_DRV_MasterInit(BOARD_I2C_COMM_INSTANCE, &master);

// Initialize data to send
for(i = 0; i < DATA_LENGTH; i++)
{
    txBuff[i] = i + 1;
}

count = 1;

// Loop for transfer
while(1)
{
    // Wait user press any key
    GETCHAR();

    // Master send 1 byte CMD and data to slave
    I2C_DRV_MasterSendData(BOARD_I2C_COMM_INSTANCE, &device, (const uint8_t*)&count, 1, (const
uint8_t*)txBuff, count);

    // Wait until finish send
    while(I2C_DRV_MasterGetSendStatus(BOARD_I2C_COMM_INSTANCE, &bytesRemaining) != kStatus_I2C_Success){}

    // Delay to wait slave received data
    OSA_TimeDelay(25);

    // Clear rxBuff
    for(i = 0; i < count; i++)
    {
        rxBuff[i] = 0;
    }
}
```

External Use | 16

This is the continuation of the I2C master code.

```
// Master receive count byte data from slave
I2C_DRV_MasterReceiveData(BOARD_I2C_COMM_INSTANCE, &device, NULL, 0, rxBuff, count);

// Wait until finish receive
while(I2C_DRV_MasterGetReceiveStatus(BOARD_I2C_COMM_INSTANCE, &bytesRemaining) != kStatus_I2C_Success){

/* Compare to check result */
if(i2c_compare((uint8_t*)txBuff, rxBuff, count) != true)
{
    break;
}

if(++count > DATA_LENGTH)
{
    count = 1;
}
OSA_TimeDelay(2);
}

I2C_DRV_MasterDeinit(BOARD_I2C_COMM_INSTANCE);

return 0;
}
```

External Use | 17

... and the final section of the master code.

Kinetis SDK I2C Example – Slave Code

```
#define DATA_LENGTH 64

int main(void)
{
    uint32_t count = 0;
    uint32_t i = 0;
    uint8_t dataBuff[DATA_LENGTH] = {0};
    i2c_slave_state_t slave;

    i2c_slave_user_config_t userConfig =
    {
        .address = 0x7F,
        .slaveCallback = NULL,
        .callbackParam = NULL,
        .slaveListening = false,
        #if FSL_FEATURE_I2C_HAS_START_STOP_DETECT
        .startStopDetect = false,
        #endif
        #if FSL_FEATURE_I2C_HAS_STOP_DETECT
        .stopDetect = false,
        #endif
    };

    hardware_init();
    OSA_Init();

    I2C_DRV_SlaveInit(BOARD_I2C_COMM_INSTANCE, &userConfig, &slave);
}
```



External Use | 18



Kinetis SDK I2C Example – Slave Code

In the next two slides is the code for an I2C slave application. The first byte received by the slave indicates the incoming number of bytes. Then the slave receives the specified number of bytes in a buffer and sends them back to the master. The communication continues by the slave receiving and echoing as many bytes as indicated by the master.

```
// Loop transfer
while(1)
{
    // Slave receive buffer from master
    I2C_DRV_SlaveReceiveData(BOARD_I2C_COMM_INSTANCE, (uint8_t*)&count, 1);

    /* Wait until transfer is succesful */
    while(I2C_DRV_SlaveGetReceiveStatus(BOARD_I2C_COMM_INSTANCE, NULL) != kStatus_I2C_Success);

    // Clear receive buffer
    for(i = 0; i < count; i++)
    {
        dataBuff[i] = 0;
    }

    // Slave receive buffer from master
    I2C_DRV_SlaveReceiveData(BOARD_I2C_COMM_INSTANCE, dataBuff, count);

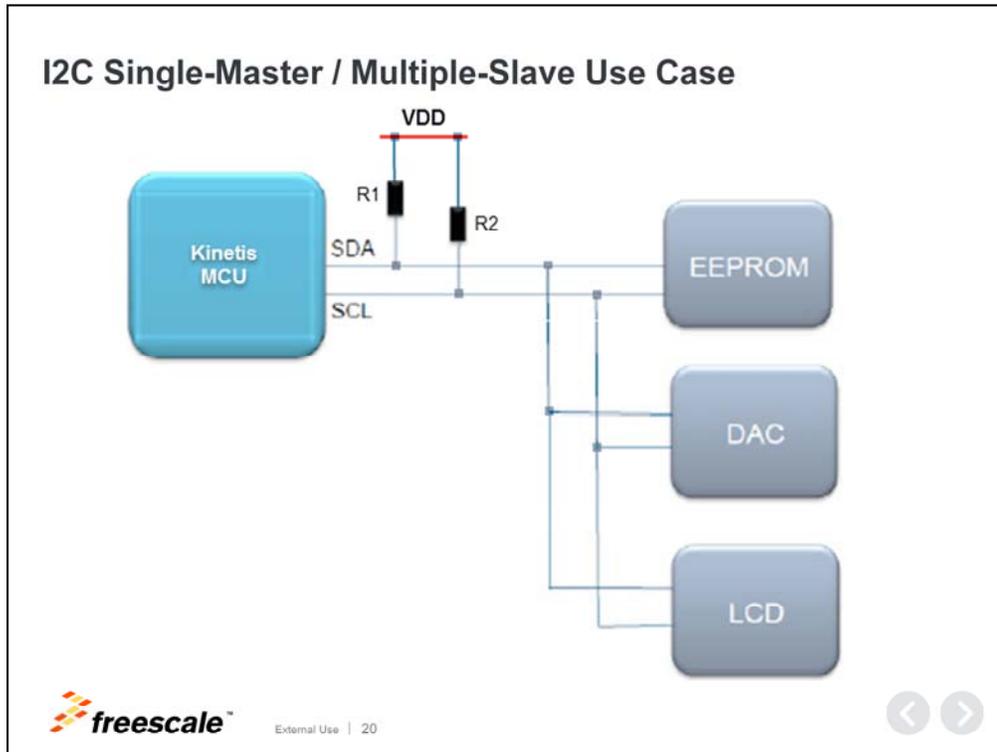
    /* Wait until transfer is succesful */
    while(I2C_DRV_SlaveGetReceiveStatus(BOARD_I2C_COMM_INSTANCE, NULL) != kStatus_I2C_Success);

    // Slave send buffer received from master
    I2C_DRV_SlaveSendData(BOARD_I2C_COMM_INSTANCE, dataBuff, count);

    /* Wait until transfer is succesful */
    while(I2C_DRV_SlaveGetTransmitStatus(BOARD_I2C_COMM_INSTANCE, NULL) != kStatus_I2C_Success);

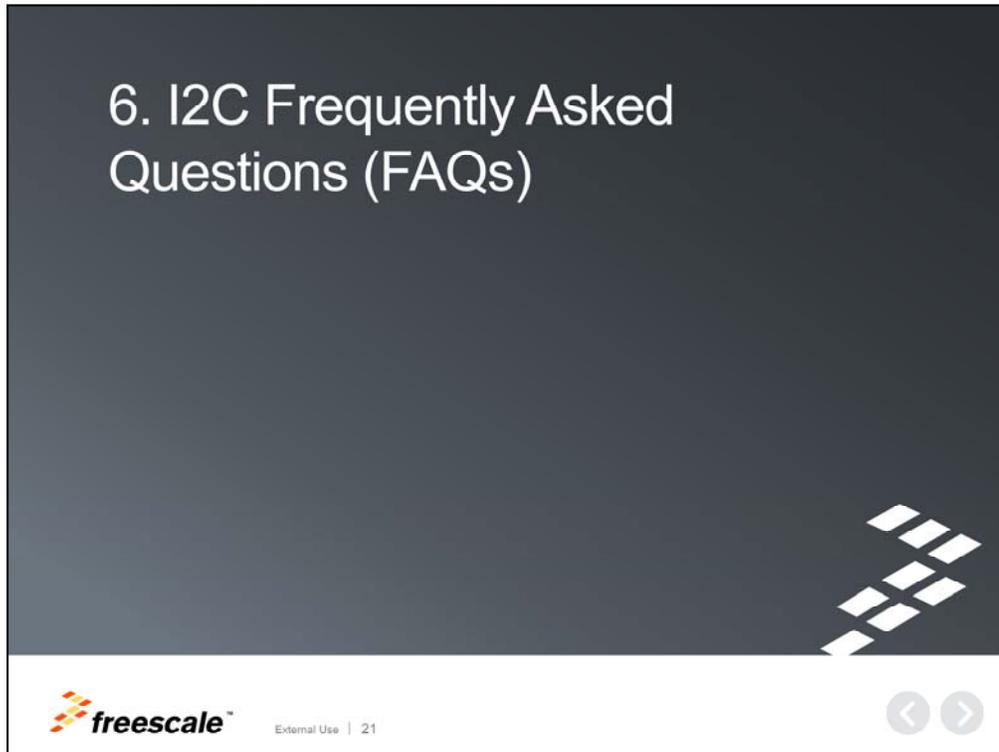
    OSA_TimeDelay(1);
}
}
```

And here is the remaining piece of the slave code.



I2C Single-Master / Multiple-Slave Use Case

This diagram represents one use case with one I2C master and three I2C slave devices. In this case, the three slaves are an EEPROM memory device, a DAC, and a LCD device. Please note the use of pull-up resistors.



Now, let's review a few FAQs that arise when using the I2C module.

I2C FAQs

Q: What is the maximum achievable I2C frequency?

A: The module is targeted for up to 1 MHz frequencies depending on the Kinetis device

Q: Why is the low level signal on SDA or SCL not completely pulled down to 0 Volts?

A: For Kinetis MCUs, you may need to configure the I2C pins for open drain mode by setting the PORTx_PCR[ODE] bit. Another cause for this may be a poor ground connection between your master device and slave device.



External Use | 22



I2C FAQs

Question: What is the maximum bus frequency of the I2C module?

Answer: The module is targeted for up to 1 MHz frequencies depending on the Kinetis device.

Question: Why is the low level signal on SDA or SCL not completely pulled down to 0 Volts?

Answer: For Kinetis MCUs, you may need to configure the I2C pins for open drain mode by setting the PORTx_PCR [ODE] bit. Another cause for this may be a poor ground connection between your master device and slave device.

I2C FAQs – Continued

Q: What is the difference between pseudo open drain and true open drain pins?

A: Pseudo open drain pins are compatible with I2C specification, but these pins cannot be pulled up to a voltage higher than VDD.

True open drain pins are also compatible with I2C specification and can be pulled up to 5V.

I2C FAQs – Continued

Question: What is the difference between pseudo open drain and true open drain pins?

Answer: Pseudo open drain pins are compatible with I2C specification, but these pins cannot be pulled up to a voltage higher than VDD.

True open drain pins are also compatible with I2C specification and can be pulled up to 5V.

Resources

- App Notes ...
 - [AN4803](#) I²C Non-Blocking Communication
 - [AN4342](#) Using I2C for Kinetis
- Website: Freescale.com/Kinetis
- Community: community.freescale.com/community/Kinetis



External Use | 24



Resources

This concludes our presentation on the I2C module for Kinetis MCUs.

For additional references, please visit the application notes listed here.

We also invite you to visit us on the web at Freescale.com/Kinetis and check out our community page.



www.Freescale.com

© 2015 Freescale Semiconductor, Inc. | *External Use*