ECE353: Introduction to Microprocessor Systems

I2C

Contents [hide]

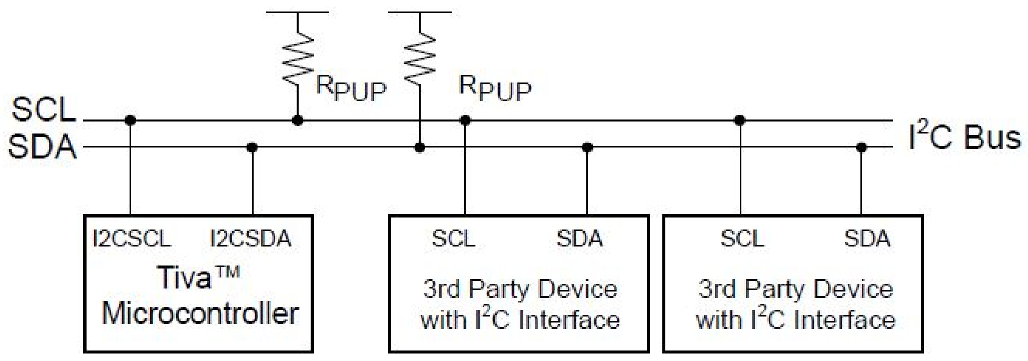
1 Overview
2 Pin Construction
3 Signal Descriptions
4 Data Format
  4.1 Start Condition
  4.2 Stop Condition
  4.3 I2C Device Addresses
  4.4 7-bit Vs. 8-bit Addresses
5 TM4C123 GPIO Configuration
6 TM4C123 I2C Peripheral Configuration
7 TM4C123 I2C Write Byte
8 TM4C123 I2C Read Byte
9 TM4C123 I2C Functions

Overview

The Inter-Integrated Circuit, or I2C, bus is an interface found on most microprocessors and is heavily used in embedded systems design. The I2C bus is used to communicate with peripheral devices in much the same way as the SPI and UART interfaces.

The key feature that differentiates I2C from UART and SPI is that the I2C bus can support multiple master and slave devices on the same bus. The ability to support multiple devices on a single bus allows us to reduce the number of external pins on a microprocessor, reducing the cost and size of the device.

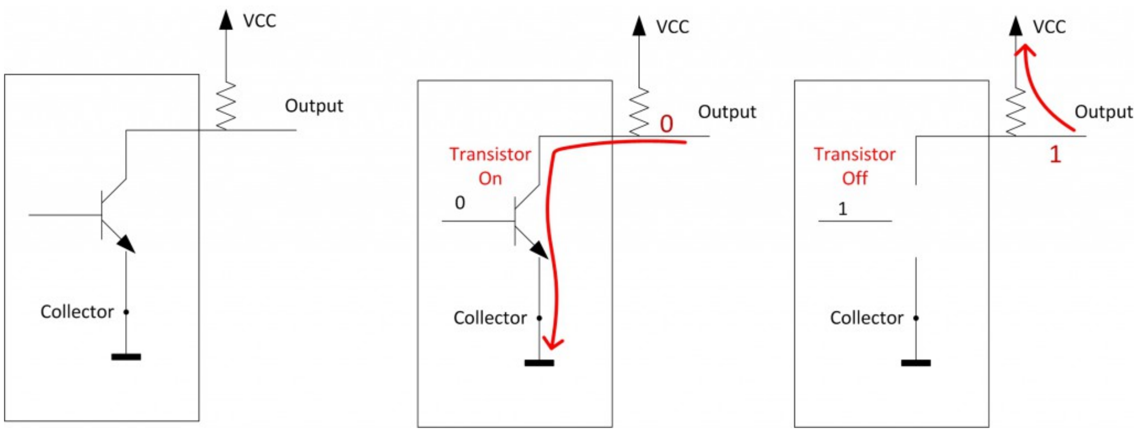
Figure 16-2. I²C Bus Configuration


RPUP
RPUP
SCL
SDA
I²C Bus
I2CSCL
I2CSDA
Tiva™ Microcontroller
SCL
SDA
3rd Party Device with I²C Interface
SCL
SDA
3rd Party Device with I²C Interface

Source: TM4C123GH6PM Data Sheet, section 16.3

Pin Construction

The I2C bus consists of two signals: the data (SDA) and clock (SCL) lines. These pins are configured as open-drain (or open-collector) pins. An open-drain pin is constructed using a single transistor that connects the output pin to ground when the transistor is turned on. When the transistor is turned off, the output pin is left unconnected and a pull-up resistor pulls the line up to VCC (logic 1).



An open-drain pin make it a good choice for a bidirectional data bus where multiple devices many initiate a data transfer. If one device attempts to place logic 1 on the line while another device attempts to drive logic 0, the pull-up resistor acts to limit the current and prevents a short circuit.

Standard (totem pole) GPIO pins would not prevent a short circuit. Standard GPIO pins actively drive logic 1 and 0 on a signal without the use of a pull-up resistor. If two different devices were to drive opposite logic values on the signal, this would result in a short circuit that could damage both devices. This is the reason that standard GPIO pins are not used for bi-directional data transmissions.

I2C bus typologies can be very complex. A single I2C bus can have multiple slaves and multiple masters. We will only deal with most basic bus topology: a single master and multiple slave devices.

Signal Descriptions

“The I2C bus uses only two signals: SDA and SCL, named I2CSDA and I2CSCL on TM4C123GH6PM microcontrollers. SDA is the bi-directional serial data line and SCL is the bi-directional serial clock line. The bus is considered idle when both lines are High.

Every transaction on the I2C bus is nine bits long, consisting of eight data bits and a single acknowledge bit. The number of bytes per transfer is unrestricted, but each data byte has to be followed by an acknowledge bit, and data must be transferred MSB first. When a receiver cannot receive another complete byte, it can hold the clock line SCL Low and force the transmitter into a wait state. The data transfer continues when the receiver releases the clock SCL”

Source: TM4C123GH6PM Data Sheet, section 16.3.1

SDA is allowed to change only when SCL is low. When SCL is high, the SDA must not change.

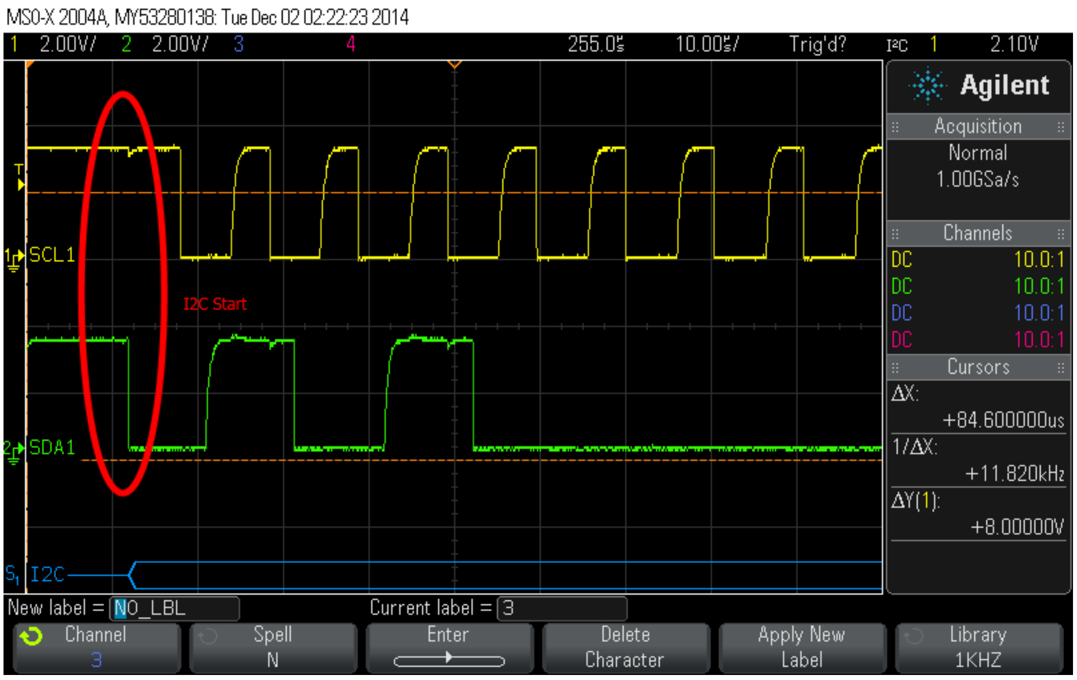
Data Format

Data transmitted over the I2C interface uses many of the same concepts used in SPI and UART interfaces. Data is transmitted most significant bit first. Data sent over I2c is framed by start and stop conditions. Data can be read and written from an external device by specifying the internal addresses of registers in the peripheral device.

A key difference with the I2C interface is that each byte of data has to be acknowledged, or ACKed, by the receiving device in order for the data transaction to complete. The ACK is necessary when many devices are present on the same I2C bus. The ACK indicates to the transmitter that it is successfully communicating with another device and can maintain ownership of the bus until the data transaction completes.

Start Condition

Both the SDA and SCL lines are pulled high when there is no activity on the bus. In order to signal the beginning of a data transaction, the device initiating the data transfer will pull the SDA line low while the SCL is still high.



Acknowledgements

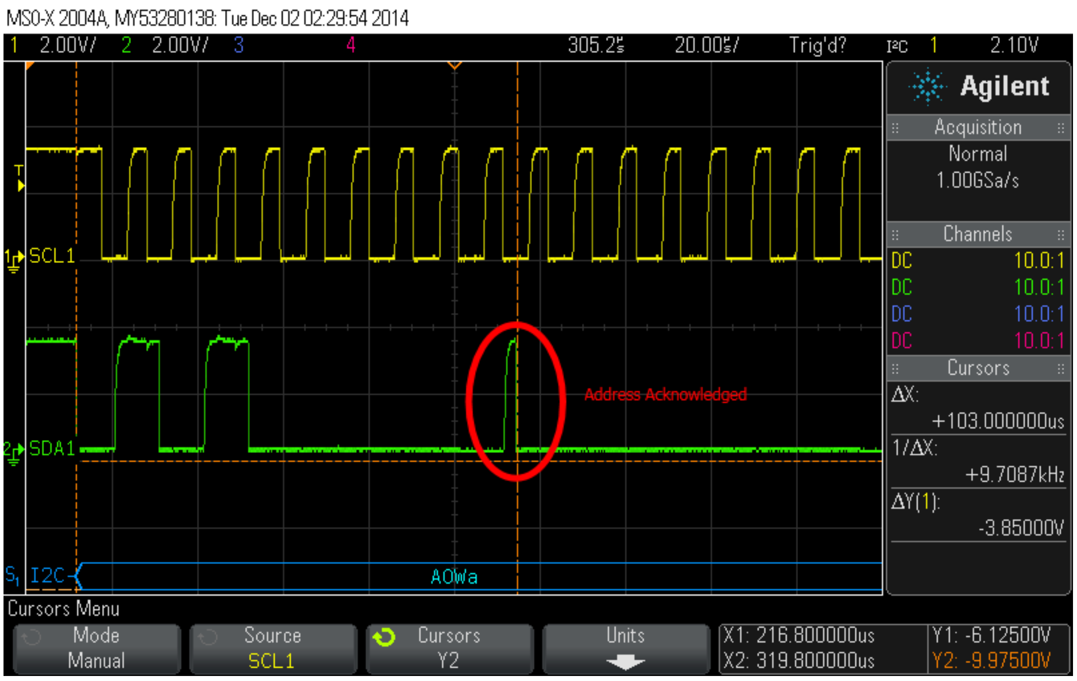
All bus transactions have a required acknowledge clock cycle that is generated by the master. During the acknowledge cycle, the transmitter (which can be the master or slave) releases the SDA line. To acknowledge the transaction, the receiver must pull down SDA during the acknowledge clock cycle.

When a slave receiver does not acknowledge the slave address, SDA must be left High by the slave so that the master can generate a STOP condition and abort the current transfer.

If the master device is acting as a receiver during a transfer, it is responsible for acknowledging each transfer made by the slave. Because the master controls the number of bytes in the transfer, it signals the end of data to the slave transmitter by not generating an acknowledge on the last data byte. The slave transmitter must then release SDA to allow the master to generate the STOP or a repeated START condition.

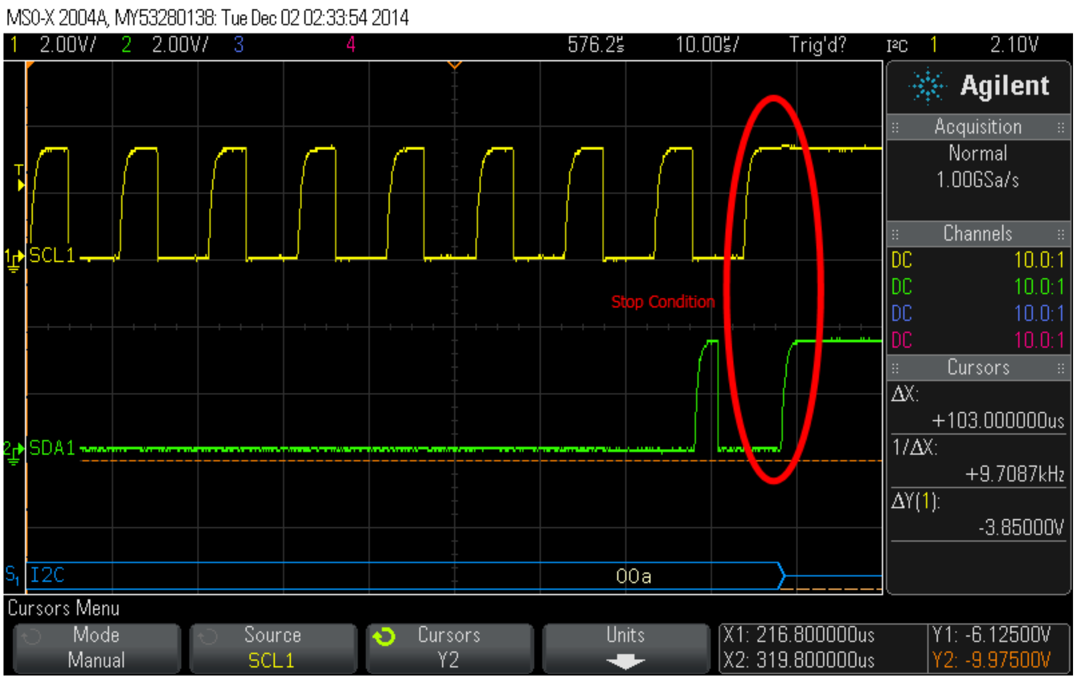
Source: TM4C123GH6PM Data Sheet, section 16.3.1.4

The image below shows an ACK. An ACK is generated when the data line is released on the 9th clock cycle by the transmitter. An ACK occurs if the receiver pulls SDA low before the next positive edge of SCL. If the SDA is not pulled low by the next clock cycle, the data transmission is aborted.



Stop Condition

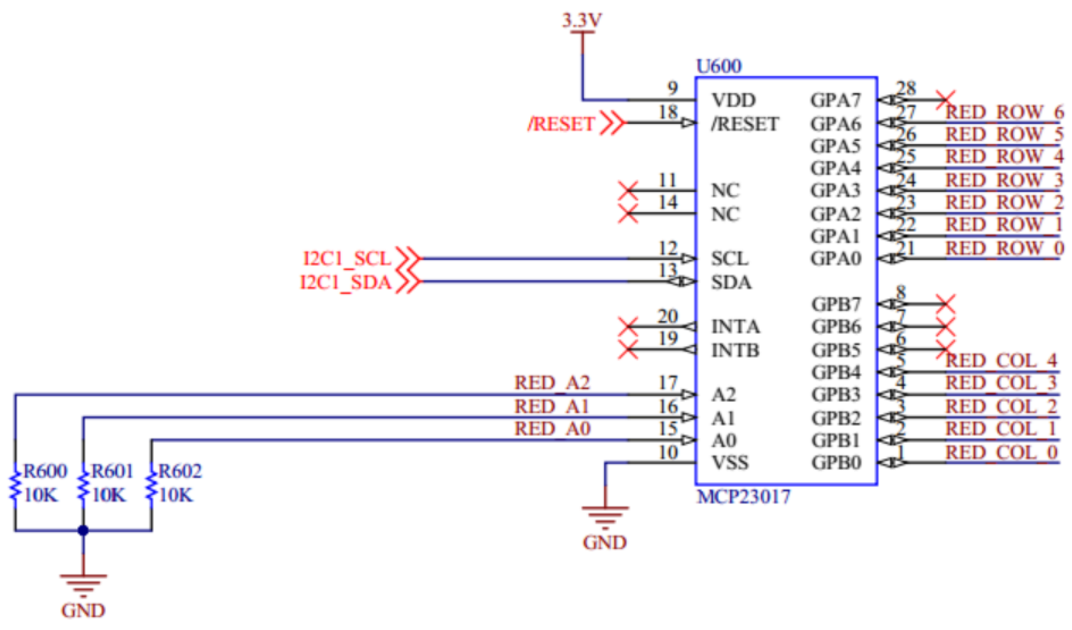
The master indicates that a transaction will end by generating a stop condition. A stop condition occurs when SCL is high and SDA makes a low to high transition.



I2C Device Addresses

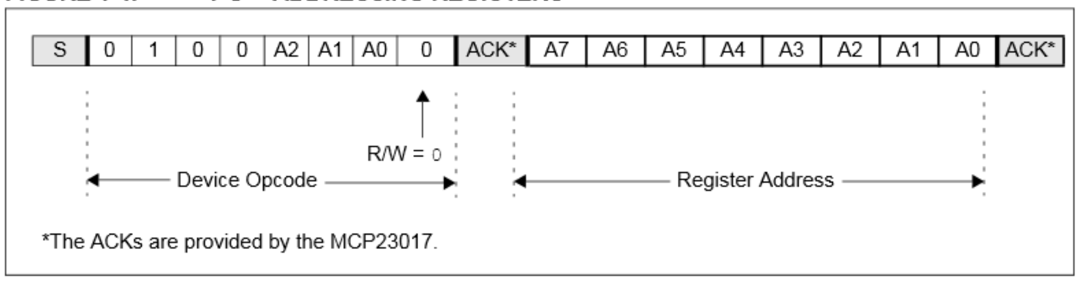
Because the I2C bus is not a point-to-point connection, there needs to be a mechanism to differentiate which pair of devices are currently transmitting data. To solve this problem, each device is assigned an address on the I2C bus. When the master device wants to send/receive data from a specific device, the first byte transmitted is the address of the slave device being accessed.

The address of the slave device consists of a device ID found in the data sheet of the device. In addition to the device ID, many devices have one or more hardware address pins that allow you to support multiple of the same slave device on the same I2C bus. Each hardware address pin will be set to an address using pull-up/down resistors to set a unique address for each device.



We will use the schematic snippet above to determine the I2C address of the MCP23017 IO expander. Chapter 5 of the MCP23017 data sheet gives us the following information about how to determine the address.

FIGURE 1-4: I^2C™ ADDRESSING REGISTERS



Source: MCP23017

Using this figure, we see that the upper 4 bits of the address are 0100_2. Bits 3-1 are set using the values supplied on the hardware address pins. Using the schematic, these bits will be set to 000_2. Bit 0 always indicates if the device is being read or written. In order to write to the MCP23017 above, the master would start a transaction with a byte equal to 0×40. In order to read from the MCP23017 above, the master device would transmit 0×41. When multiple devices are placed on the I2C bus, the hardware designer must ensure that two devices do not have conflicting addresses.

7-bit Vs. 8-bit Addresses

One common mistake that embedded software developers encounter when using I2C is specifying an incorrect address for a device on the I2C bus. The confusion arises because some data sheets list a 7-bit address and others list an 8-bit address. So what is the difference between the two?

An 8-bit address includes the read/write bit. Using the 8th bit to specifying an address results in one logical address for writing to the device and a different address to read from the device. The addresses shown above for the MCP23017 above are 8-bit address.

A 7-bit address omits the read/write bit. When specifying the 7-bit address, we only use bits 7-1 of the 8-bit address. If we use bits 7-1 from the example above, the 7-bit address becomes 0×20. It is understood that the 7-bit address is shifted left by 1 position and the read/write bit is set appropriately.

In the end, both representations of the I2C address will result in the same I2C address being placed on SDA.

TM4C123 GPIO Configuration

```
gpio_enable_port(I2C_GPIO_BASE);

// Configure SCL
gpio_config_digital_enable(I2C_GPIO_BASE, I2C_SCL_PIN);
gpio_config_alternate_function(I2C_GPIO_BASE, I2C_SCL_PIN);
gpio_config_port_control(I2C_GPIO_BASE, I2C_SCL_PIN_PCTL);

// Configure SDA
gpio_config_digital_enable(I2C_GPIO_BASE, I2C_SDA_PIN);
gpio_config_open_drain(I2C_GPIO_BASE, I2C_SDA_PIN);
gpio_config_alternate_function(I2C_GPIO_BASE, I2C_SDA_PIN);
gpio_config_port_control(I2C_GPIO_BASE, I2C_SDA_PIN_PCTL);
```

TM4C123 I2C Peripheral Configuration

```
//*************************************************************************
//
//*************************************************************************
bool initializeI2CMaster(uint32_t base_addr)
{

  myI2C = (I2C0_Type *) base_addr;

  // Validate that a correct base address has been passed
    // Turn on the Clock Gating Register
    switch (base_addr)
    {
      case I2C0_BASE :
          SYSCTL->RCGCI2C  |= SYSCTL_RCGCI2C_R0;
```

```
            while ((SYSCTL->PRI2C & SYSCTL_PRI2C_R0) == 0);    /* wait until
            break;
        case I2C1_BASE :
            SYSCTL->RCGCI2C |= SYSCTL_RCGCI2C_R1;
            while ((SYSCTL->PRI2C & SYSCTL_PRI2C_R1) == 0);    /* wait until
            break;
        case I2C2_BASE :
            SYSCTL->RCGCI2C |= SYSCTL_RCGCI2C_R2;
            while ((SYSCTL->PRI2C & SYSCTL_PRI2C_R2) == 0);    /* wait until
            break;
        case I2C3_BASE :
            SYSCTL->RCGCI2C |= SYSCTL_RCGCI2C_R3;
            while ((SYSCTL->PRI2C & SYSCTL_PRI2C_R3) == 0);    /* wait until
            break;
        default:
            return false;
    }

    // Enable the I2C port as master
    myI2C->MCR = I2C_MCR_MFE;

    // Set the clock speed to be 100Kpbs assuming a 50MHz clock
    // TPR = (System Clock/(2*(SCL_LP + SCL_HP)*SCL_CLK))-1;
    // TPR = (50MHz/(2*(6+4)*100000))-1
    myI2C->MTPR = 0x18;

    return true;
}
```

TM4C123 I2C Write Byte

The following snippet of code writes 1 byte of data to the IODIRA and IODIRB registers in the MCP23017. In addition to the control byte (I2C Address + write), we need to transfer two bytes of data. The first is the address we are writing to and the second is the value we wish to write.

The START condition is initiated with a MCS_START flag and the STOP condition is initiated with a I2C_MCS_STOP flag.

```
 i2cSetSlaveAddr(I2C_BASE, 0x20, I2C_WRITE);

// Send the IODIRA Address
i2cSendByte( I2C_BASE, 0x00, I2C_MCS_START | I2C_MCS_RUN);

// Set PortA to be outputs
i2cSendByte( I2C_BASE, 0x00, I2C_MCS_RUN | I2C_MCS_STOP);

// Send the IODIRB Address
i2cSendByte( I2C_BASE, 0x01, I2C_MCS_START | I2C_MCS_RUN);

// Set PortB to be outputs
i2cSendByte( I2C_BASE, 0x00, I2C_MCS_RUN | I2C_MCS_STOP);
```

TM4C123 I2C Read Byte

The following code would read one byte of data from the IODIRA register of the MCP23017. When reading data, we need to first write the address of the register that we intend to read. After that byte of data has been transmitted, we need to issue a restart condition.

The restart condition is initiated with the same I2C Address, but sets the read bit to a 1. Once the restart operation has been sent, a single byte of data will be read.

```c
i2c_status_t status;

// Set I2C address
i2cSetSlaveAddr(I2C_BASE, 0x20, I2C_WRITE);

// Send the IODIRA Address
i2cSendByte( I2C_BASE, 0x00, I2C_MCS_START | I2C_MCS_RUN);

// Set I2C address,
i2cSetSlaveAddr(I2C_BASE, 0x20, I2C_READ);

// Issue Re-Start, Read IODIR Register
status = i2cGetByte(
                     I2C_BASE,
                     data,
                     I2C_MCS_START | I2C_MCS_RUN | I2C_MCS_STOP
                    );
```

TM4C123 I2C Functions

```c
//*************************************************************************
//*************************************************************************
i2c_status_t i2cSetSlaveAddr(
  uint32_t baseAddr,
  uint8_t slaveAddr,
  i2c_read_write_t readWrite
)
{
  I2C0_Type *myI2C;
  if( i2cVerifyBaseAddr(baseAddr) == false)
  {
    return I2C_INVALID_BASE;
  }

  myI2C = (I2C0_Type *) baseAddr;

  // Set the slave address to transmit data
  myI2C->MSA = (slaveAddr << 1) | readWrite;

  return I2C_OK;
}

//*************************************************************************
//*************************************************************************
i2c_status_t i2cStop(
  uint32_t baseAddr
)
{
  I2C0_Type *myI2C;
  if( i2cVerifyBaseAddr(baseAddr) == false)
  {
    return I2C_INVALID_BASE;
  }

  myI2C = (I2C0_Type *) baseAddr;

  // Stop the interface
  myI2C->MCS = I2C_MCS_STOP;

  return I2C_OK;
}

//*************************************************************************
//*************************************************************************
bool
```

```
I2CMasterBusy(uint32_t i2c_base)
{
  I2C0_Type *myI2C;

  if( i2cVerifyBaseAddr(i2c_base) == false)
  {
    return false;
  }

    myI2C = (I2C0_Type *) i2c_base;

    if(myI2C->MCS & I2C_MCS_BUSY)
    {
        return(true);
    }
    else
    {
        return(false);
    }
}

//*********************************************************************
//*********************************************************************
bool
I2CMasterAdrAck(uint32_t i2c_base)
{
    I2C0_Type *myI2C;
    uint32_t status;
  if( i2cVerifyBaseAddr(i2c_base) == false)
  {
    return false;
  }

  myI2C = (I2C0_Type *) i2c_base;

  status = myI2C->MCS;
  if((status & I2C_MCS_ADRACK)!= 0)
  {
      return(false);
  }
  else
  {
      return(true);
  }
}

//*********************************************************************
//*********************************************************************
bool
I2CMasterDatAck(uint32_t i2c_base)
{
    I2C0_Type *myI2C;
    uint32_t status;
  if( i2cVerifyBaseAddr(i2c_base) == false)
  {
    return false;
  }

  myI2C = (I2C0_Type *) i2c_base;

  status = myI2C->MCS;
  if((status & I2C_MCS_DATACK)!= 0)
  {
      return(false);
  }
  else
  {
      return(true);
  }
```

```
}

//*************************************************************************
//*************************************************************************
i2c_status_t i2cSendByte(
  uint32_t baseAddr,
  uint8_t data,
  uint8_t masterControlSettings
)
{
  I2C0_Type *myI2C;

  if( i2cVerifyBaseAddr(baseAddr) == false)
  {
    return I2C_INVALID_BASE;
  }
  myI2C = (I2C0_Type *) baseAddr;

   // Write the upper address to the data register
  myI2C->MDR    = data;

  // Start the transaction
  myI2C->MCS = masterControlSettings;

  // Wait for the device to be free
  while ( I2CMasterBusy(baseAddr)) {};

    // Check for error conditions
  if ( myI2C->MCS & (I2C_MCS_ERROR | I2C_MCS_ARBLST) )
  {
      return I2C_ARBLST;
  }
  else if ( myI2C->MCS & I2C_MCS_ERROR )
  {
    myI2C->MCS = I2C_MCS_STOP;
    return I2C_BUS_ERROR;
  }
  else if ( myI2C->MCS & I2C_MCS_DATACK )
  {
    return I2C_NO_ACK;
  }
  else
  {
    return I2C_OK;
  }
}

//*************************************************************************
//*************************************************************************
i2c_status_t i2cGetByte(
  uint32_t baseAddr,
  uint8_t *data,
  uint8_t masterControlSettings
)
{
  I2C0_Type *myI2C;

  if( i2cVerifyBaseAddr(baseAddr) == false)
  {
    return I2C_INVALID_BASE;
  }

 myI2C = (I2C0_Type *) baseAddr;

  // Start the transaction
  myI2C->MCS = masterControlSettings;

  // Wait for the device to be free
  while ( I2CMasterBusy(baseAddr)) {};

  // Check for error conditions
  if ( myI2C->MCS & I2C_MCS_ERROR  )
  {
    myI2C->MCS = I2C_MCS_STOP;
    return I2C_BUS_ERROR;
  }
  else
  {
    *data = myI2C->MDR;
    return I2C_OK;
  }
}
```