Lecture 12: I2C and SPI Communications

ME/AE 6705

Introduction to Mechatronics

Dr. Jonathan Rogers

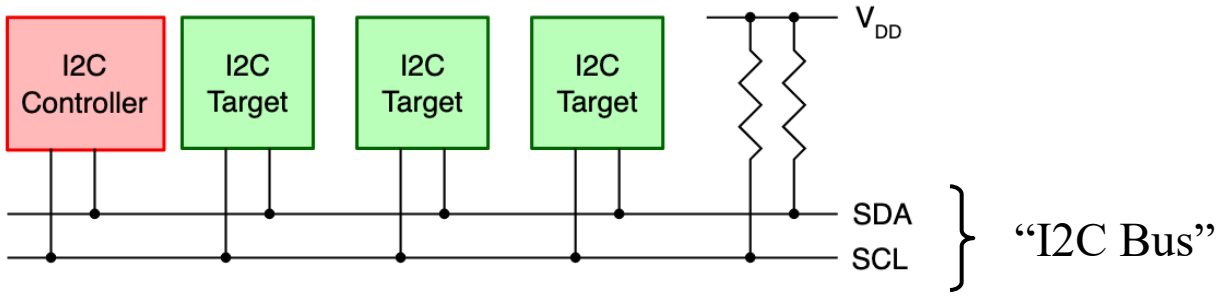
Georgia Tech

Lesson Objectives

- Understand I2C and SPI protocols
- Understand the tradeoffs between using UART, I2C, and SPI protocols
- Be able to implement I2C communications with MSPM0 with multiple Target devices
- Be able to implement SPI communications with MSPM0

I²C Protocol

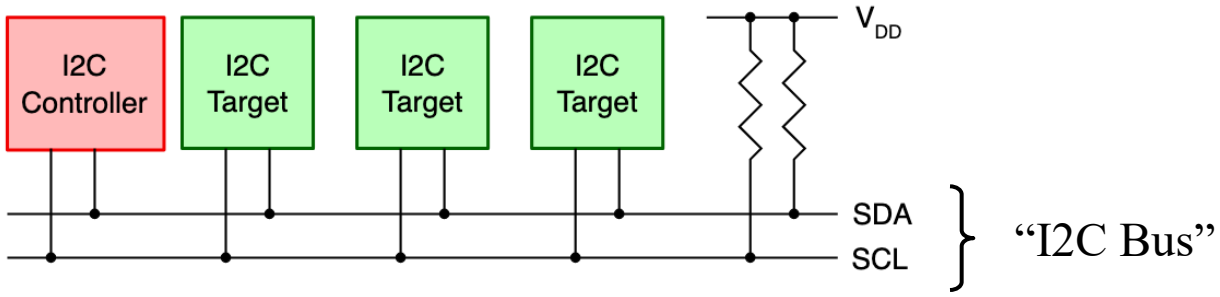
- I²C = Inter-Integrated Circuit
  - Extremely common serial communication protocol used extensively to interface with sensors, actuators, and storage devices
  - Developed by Phillips Semiconductors in 1980's
- Primary advantage: Allows communications with multiple devices on same 2-wire bus



Georgia Tech

I²C Protocol

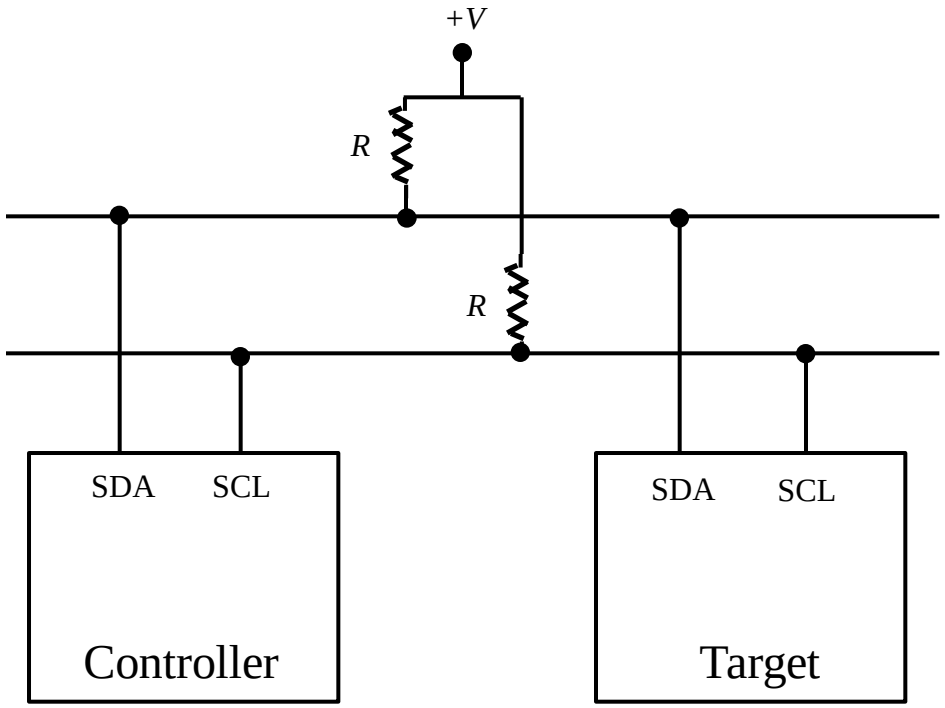
- Terminology
  - **SDA**: Data wire. Data bits flow along this line.
  - **SCL**: Clock wire. This transmits clock signal which is used to synchronize data transfer.
  - **Controller**: Device (MCU, sensor, etc) on I2C bus which controls clock line (SCL). Usually only be one per bus.
  - **Target**: Device (MCU, sensor, etc) on I2C that monitors SCL and communicates on SDA.



Georgia Tech

I²C Wiring

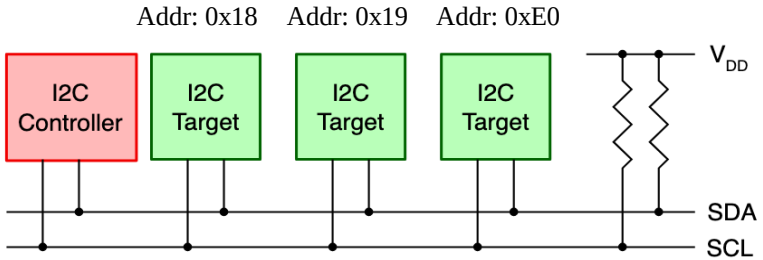
- When implementing I2C bus, pull-up resistor is needed on each line to set the logic level being used (either 3.3V or 5V).
- This is an Open Collector circuit (see Lecture 2)
- On TI Launchpad, the SCL and SDA lines are already implemented using an open collector circuit, so do not need to implement $+V$ connections
- May need to when using other Controller/Target devices



Georgia Tech

I²C Controller and Target Devices

- Only Controller device can initiate data transfer
- Targets respond to Controller request
- Data can be sent from Controller to Target, or from Target to Controller (bus can transfer data in either direction)
- Each Target on bus is addressable via a unique “address” value assigned to that Target



Georgia Tech

I²C (or SPI) Target Devices

- Most sensors or actuators have multiple registers that can be written or read via I2C (or SPI)
  - When reading or writing to these devices, need to specify not only Target **address** but also **register address** on that Target
- For example, Microchip MCP9808 temp sensor:

bit 7-4 **W:** Writable bits

Write '0'.

Bits 7-4 must always be cleared or written to '0'. This device has additional registers that are reserved for test and calibration. If these registers are accessed, the device may not perform according to the specification.

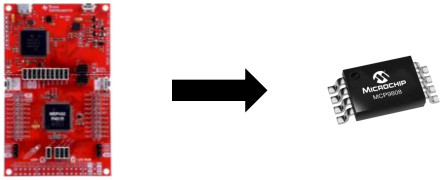
bit 3-0 **Pointer bits**

0000 = RFU, Reserved for Future Use (Read-Only register)
0001 = Configuration register (CONFIG)
0010 = Alert Temperature Upper Boundary Trip register (T_{UPPER})
0011 = Alert Temperature Lower Boundary Trip register (T_{LOWER})
0100 = Critical Temperature Trip register (T_{CRIT})
0101 = Temperature register (T_A)
0110 = Manufacturer ID register
0111 = Device ID/Revision register
1000 = Resolution register
1xxx = Reserved(1)

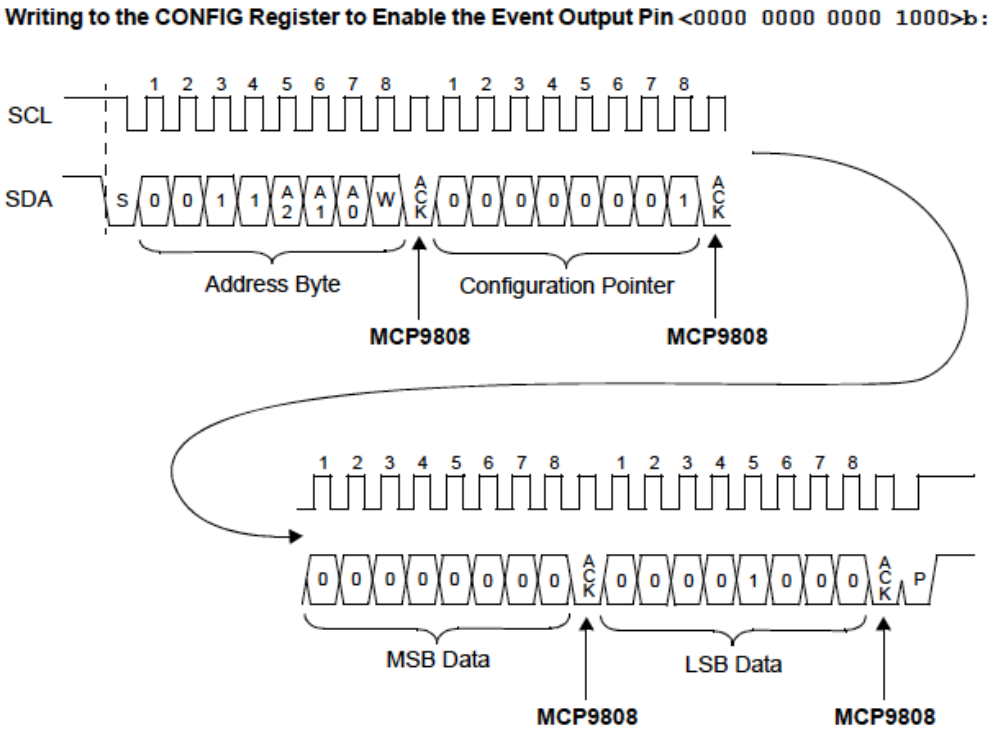
Device configuration register is 0x01

Measured temperature stored in register 0x05

I²C Protocol

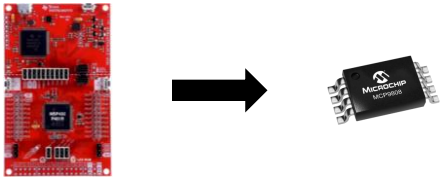


- Complete protocol (writing data to Target device)
  - Example is for writing 2 bytes of data to MCP9808 Configuration register (which is 16 bits)

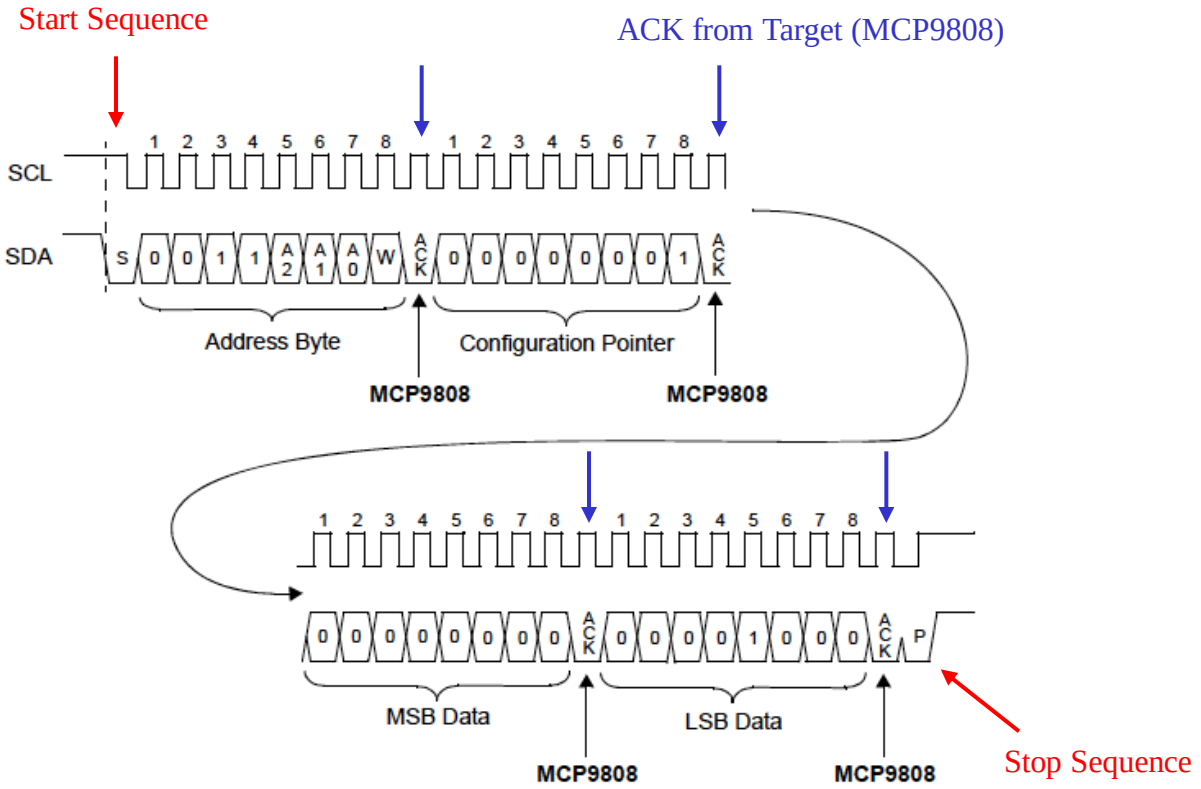


Georgia Tech

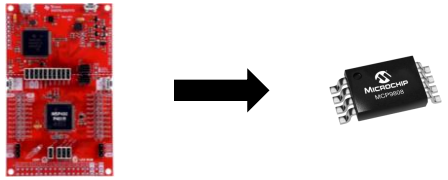
I²C Protocol



- Complete protocol (writing data to Target device)
  - Example is for writing 2 bytes of data to MCP9808 Configuration register (which is 16 bits)



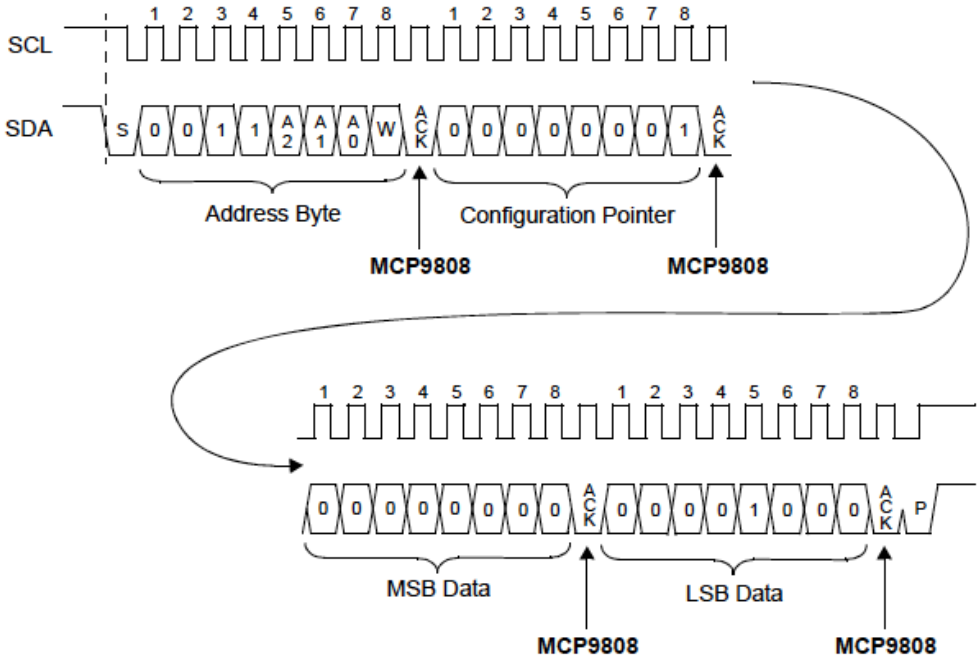
I²C Protocol



- Complete protocol (writing data to Target device)
  - Example is for writing 2 bytes of data to MCP9808 Configuration register (which is 16 bits)

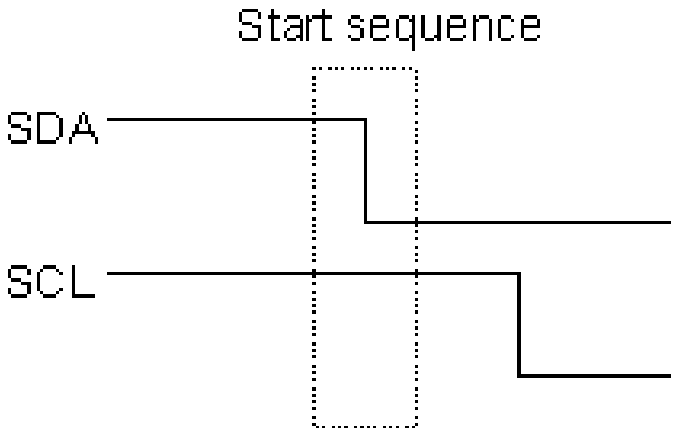
Process:

1. Controller transmits Target address byte
2. Controller transmits register address
3. Controller transmits high byte of data
4. Controller transmits low byte of data



I²C Protocol

- Start sequence used by Controller to initiate a transaction with a Target device
  - Marks beginning of transaction

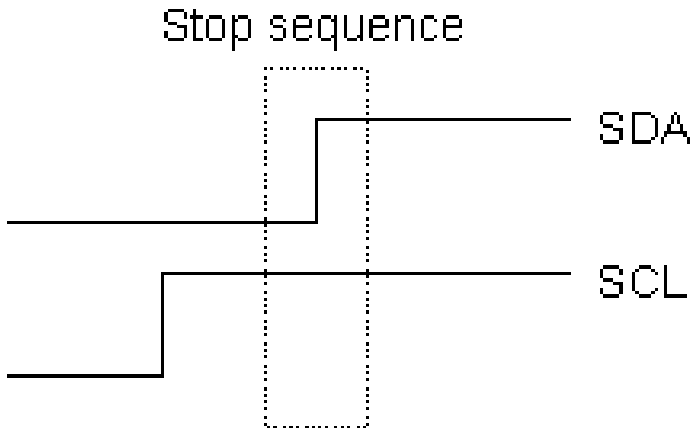


- SDA line changes from high to low while SCL (clock line) held high
- Only time (besides stop command) that SDA is allowed to change while SCL remains constant

Georgia Tech

I²C Protocol

- Stop sequence indicates the end of a transaction between Controller and Target device

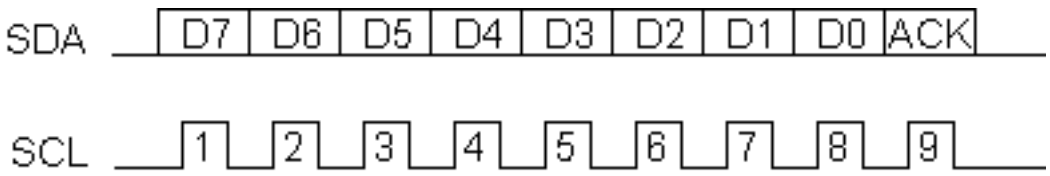


- SDA line changes from low to high while SCL (clock line) held high
- Only time (besides start command) that SDA is allowed to change while SCL remains constant

Georgia Tech

I²C Protocol – Transmitting Data

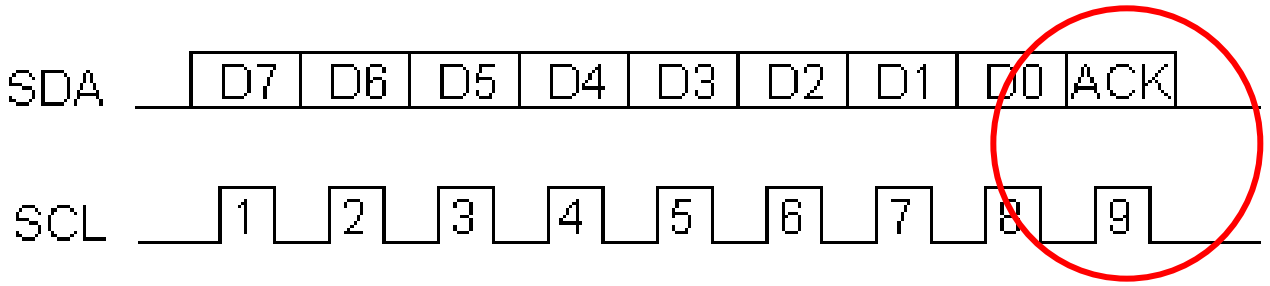
- To send data, SDA line holds bit value
  - 0 for low, 3.3V or 5V for high
- SCL clocked from low to high to low during transmission of single bit
- Clock speeds of **100 kHz** or **400 kHz** usually used
  - Controller generates this clock signal, Targets respond



All I2C communication is done 1 byte at a time.

I²C Protocol – Transmitting Data

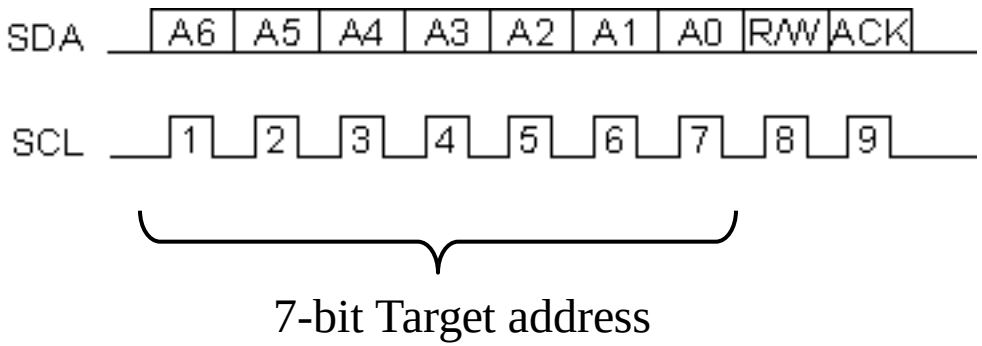
- When 1 byte is sent, 9 bits are actually transmitted
- Last bit is either **“ACK”** or **“NAK”**
  - If bit 9 is low, this signals an ACK – means receiving device has received data and is ready to accept next byte
  - If bit 9 is high, this signals a NAK – means receiving device cannot accept further data and Controller should end transaction



This bit is needed because actual size of data transfer is never transmitted explicitly.

I²C Protocol – Address Byte

- All I2C address are either 7-bit or 10-bit
- Majority are 7-bit (10-bit is very rare)
- When transmitting address byte, Controller also transmits whether operation is read or write using bit 8



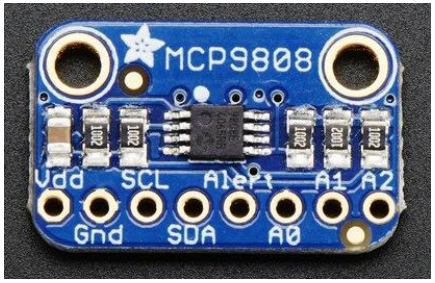
→ Set last bit to 1 when reading from Target device

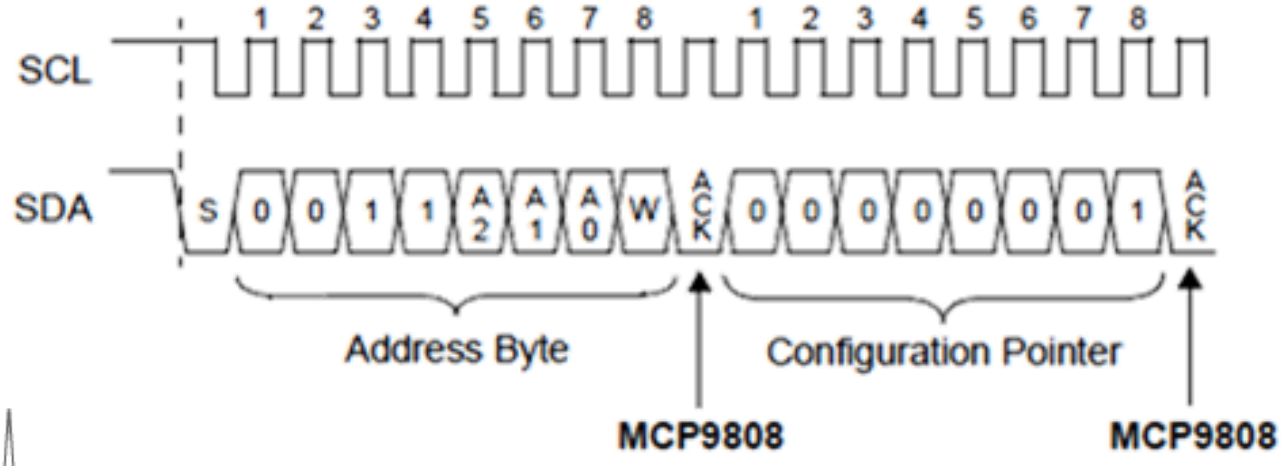
→ Set last bit to 0 when writing to Target device

Georgia Tech

I²C Protocol – Address Byte

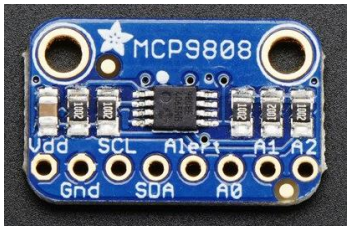
- Example: MCP9808
  - Default address is 0x18
  - By driving A0, A1, A2 pins high, can change address (between 0x18 and 0x1F)





I²C Protocol – Address Byte

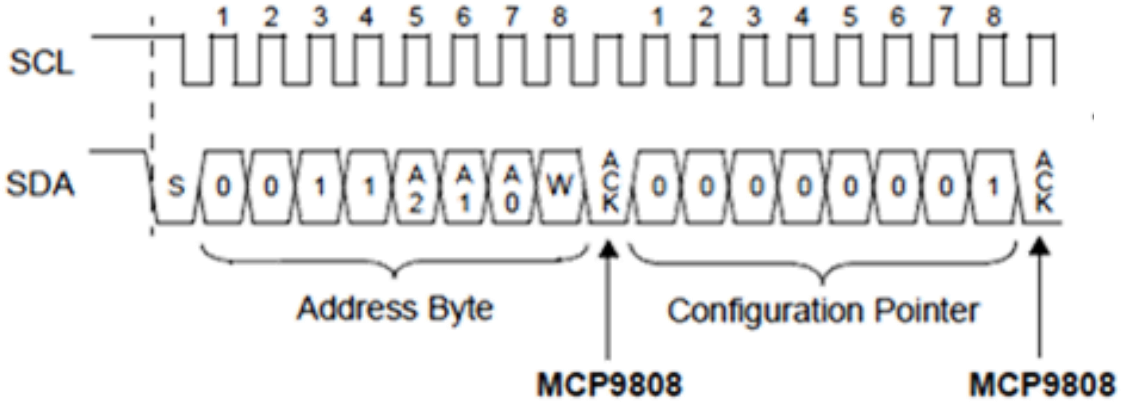
- Example: MCP9808
  - When Controller sends out address byte, it is shifted left to make room for R/W bit



So if address is 0x18, Controller actually sends:

➡ 0x30 for write operation

➡ 0x31 for read operation

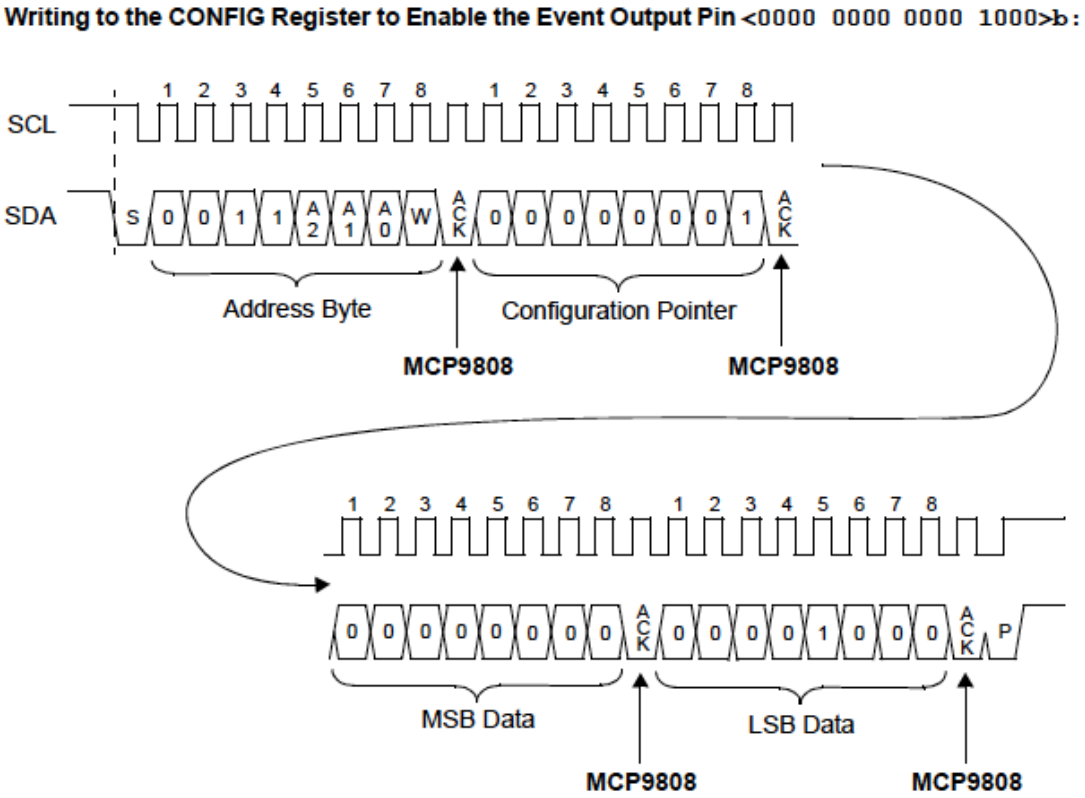


Note: This shifting operation is abstracted from programmer when using Driverlib.

I²C Protocol – Write Operation

- Steps Controller performs to write to Target
  - Example: Writing two bytes to configuration register

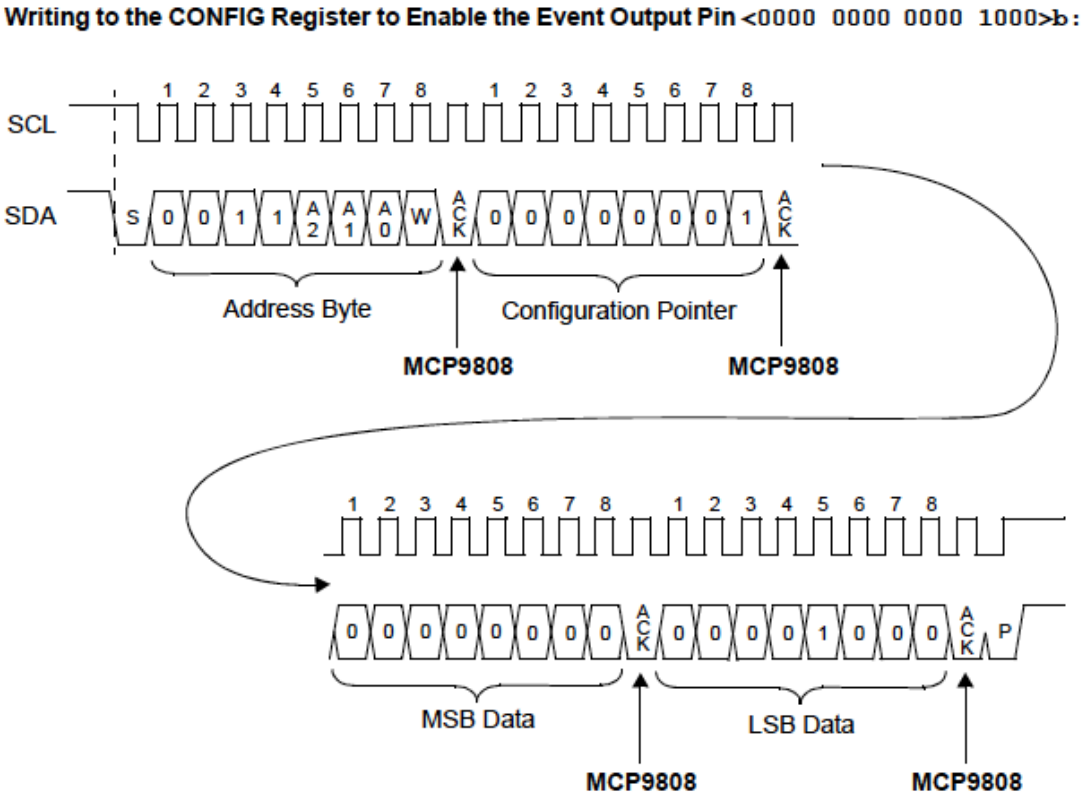
1. Send start sequence (clear SDA while holding SCL high)



I²C Protocol – Write Operation

- Steps Controller performs to write to Target
  - Example: Writing two bytes to configuration register

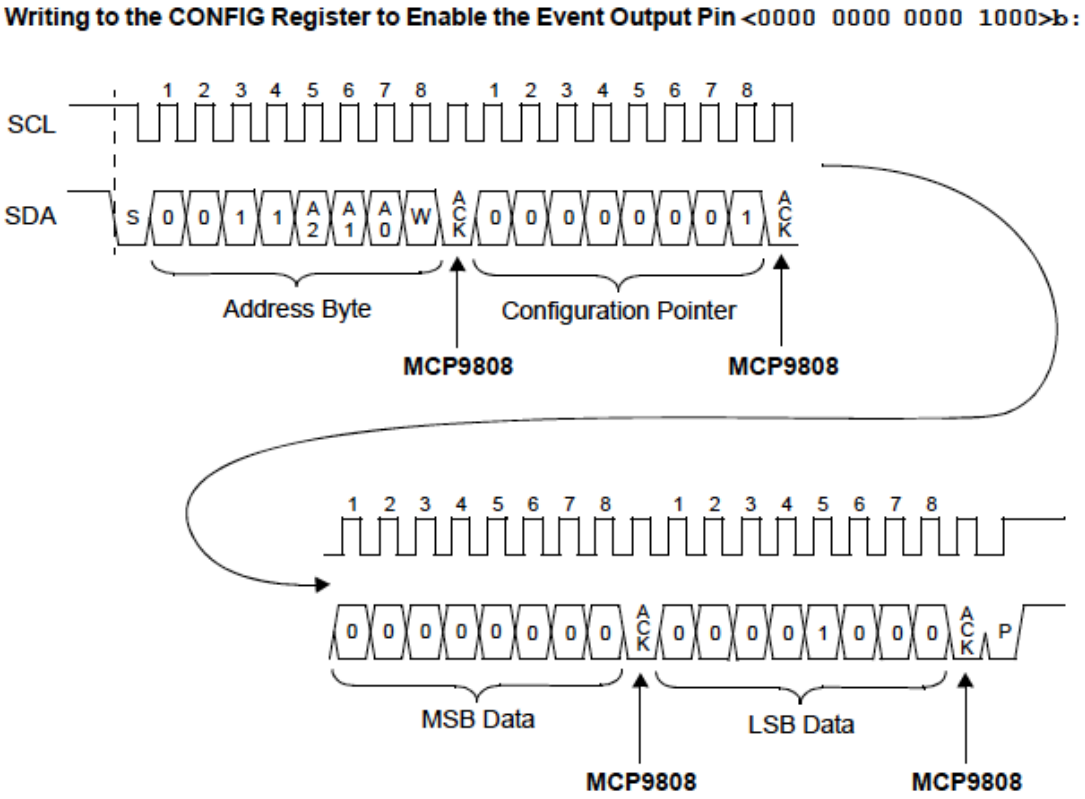
1. Send start sequence (clear SDA while holding SCL high)
2. Send address of Target with read/write bit cleared (byte is I2C address, left shifted one bit with LSB cleared)



I²C Protocol – Write Operation

- Steps Controller performs to write to Target
  - Example: Writing two bytes to configuration register

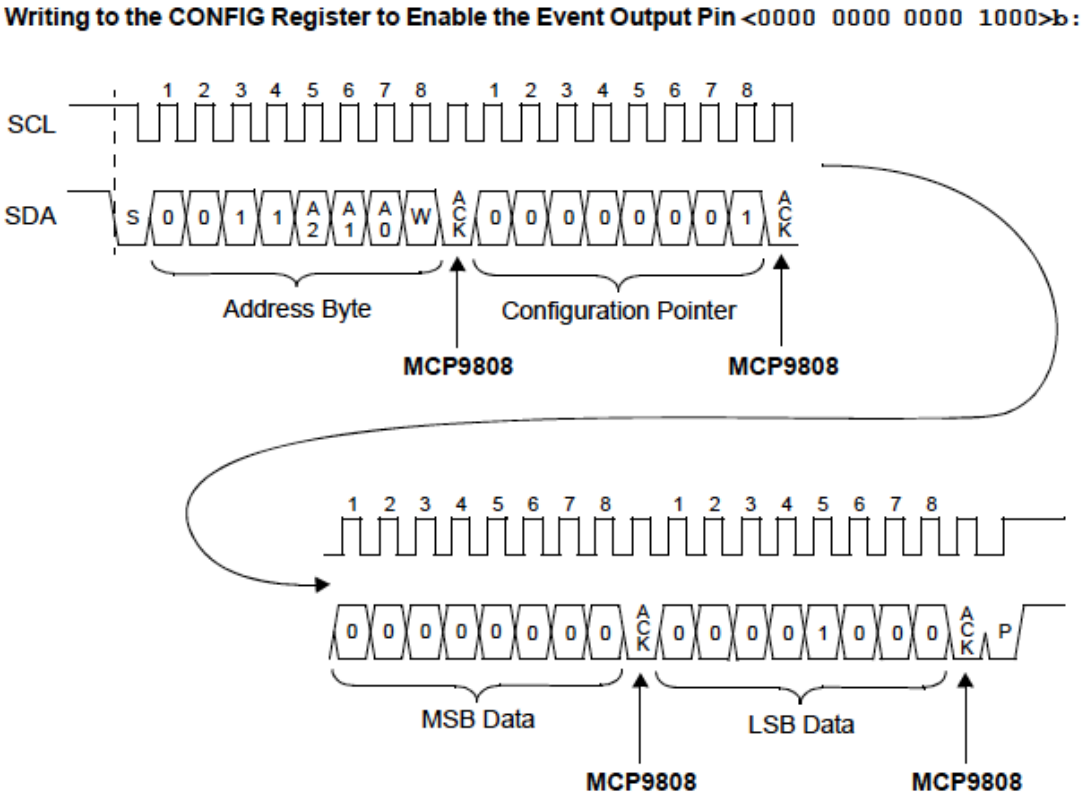
1. Send start sequence (clear SDA while holding SCL high)
2. Send address of Target with read/write bit cleared (byte is I2C address, left shifted one bit with LSB cleared)
3. Send address of register you want to write to



I²C Protocol – Write Operation

- Steps Controller performs to write to Target
  - Example: Writing two bytes to configuration register

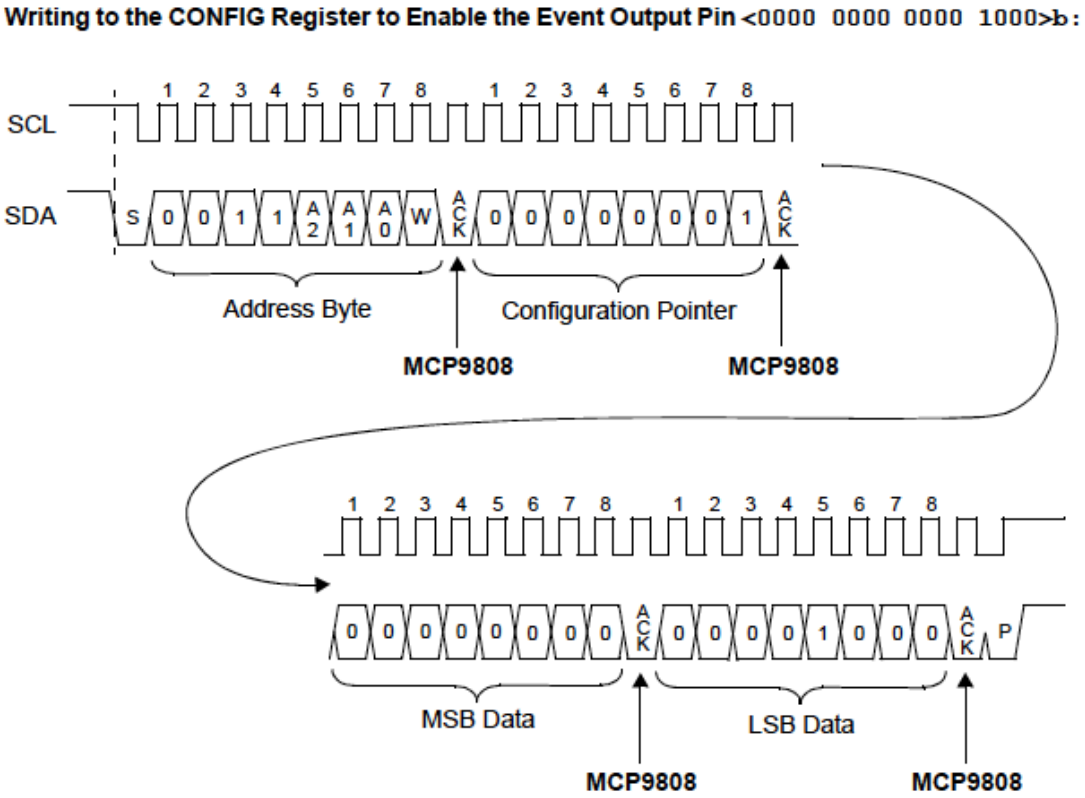
1. Send start sequence (clear SDA while holding SCL high)
2. Send address of Target with read/write bit cleared (byte is I2C address, left shifted one bit with LSB cleared)
3. Send address of register you want to write to
4. Send data, in order of high byte to low byte (see spec sheet of Target device)



I²C Protocol – Write Operation

- Steps Controller performs to write to Target
  - Example: Writing two bytes to configuration register

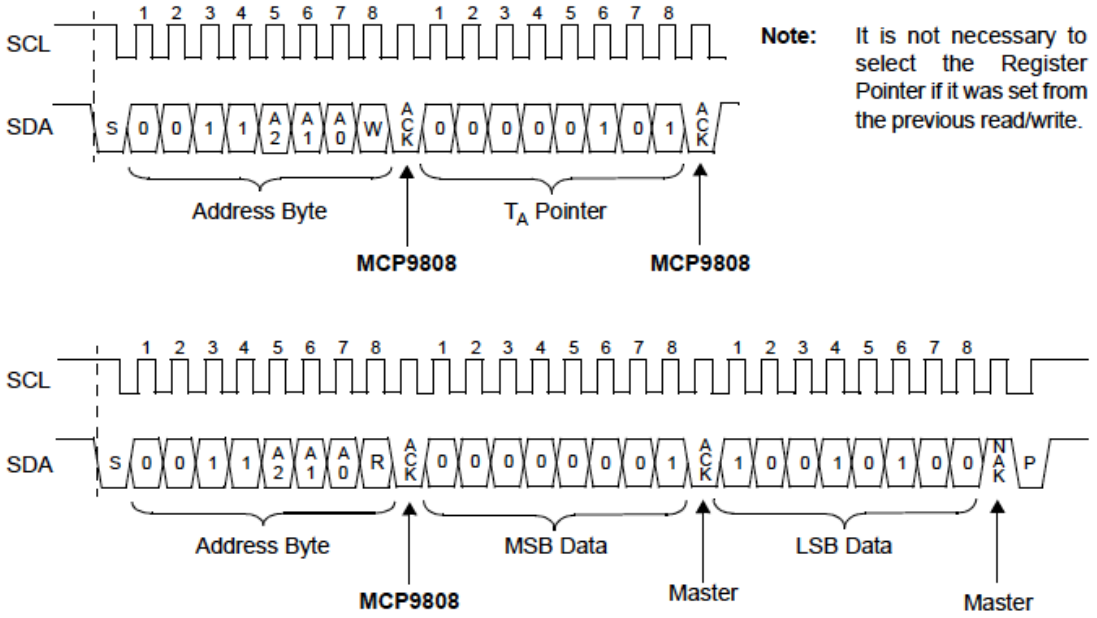
1. Send start sequence (clear SDA while holding SCL high)
2. Send address of Target with read/write bit cleared (byte is I2C address, left shifted one bit with LSB cleared)
3. Send address of register you want to write to
4. Send data, in order of high byte to low byte (see spec sheet of Target device)
5. Send stop sequence (set SDA high while holding SCL high)



I²C Protocol – Read Operation

- Steps Controller performs to read from Target
  - Example: Reading 2-byte register from Target device

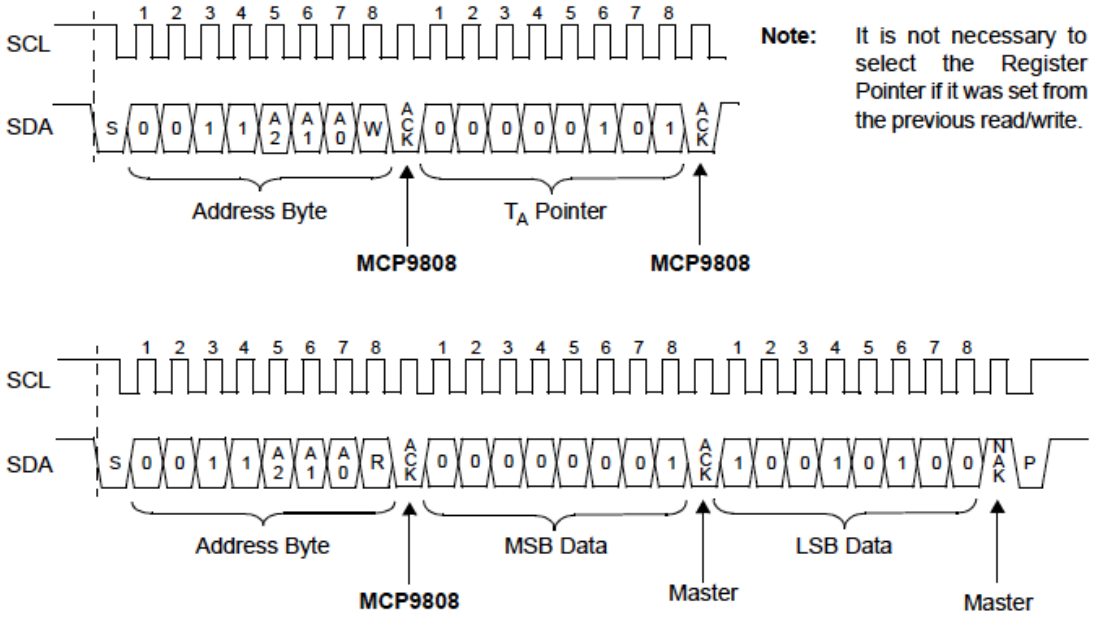
1. Send start sequence (clear SDA while holding SCL high)



I²C Protocol – Read Operation

- Steps Controller performs to read from Target
  - Example: Reading 2-byte register from Target device

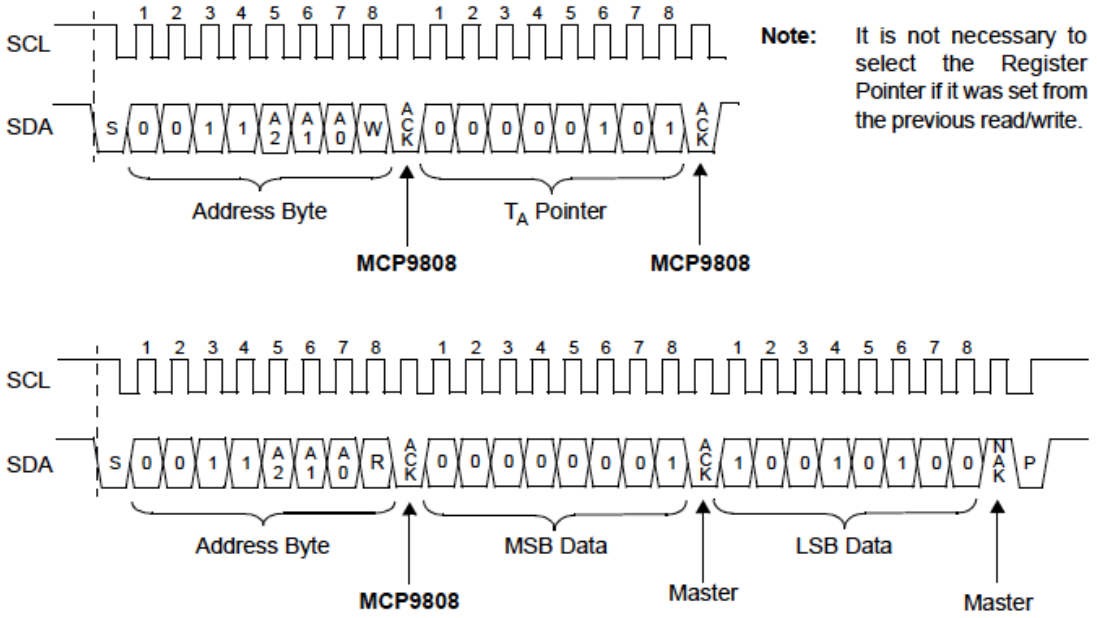
1. Send start sequence (clear SDA while holding SCL high)
2. Send address of Target with read/write bit cleared (byte is I2C address, left shifted one bit with LSB cleared)



I²C Protocol – Read Operation

- Steps Controller performs to read from Target
  - Example: Reading 2-byte register from Target device

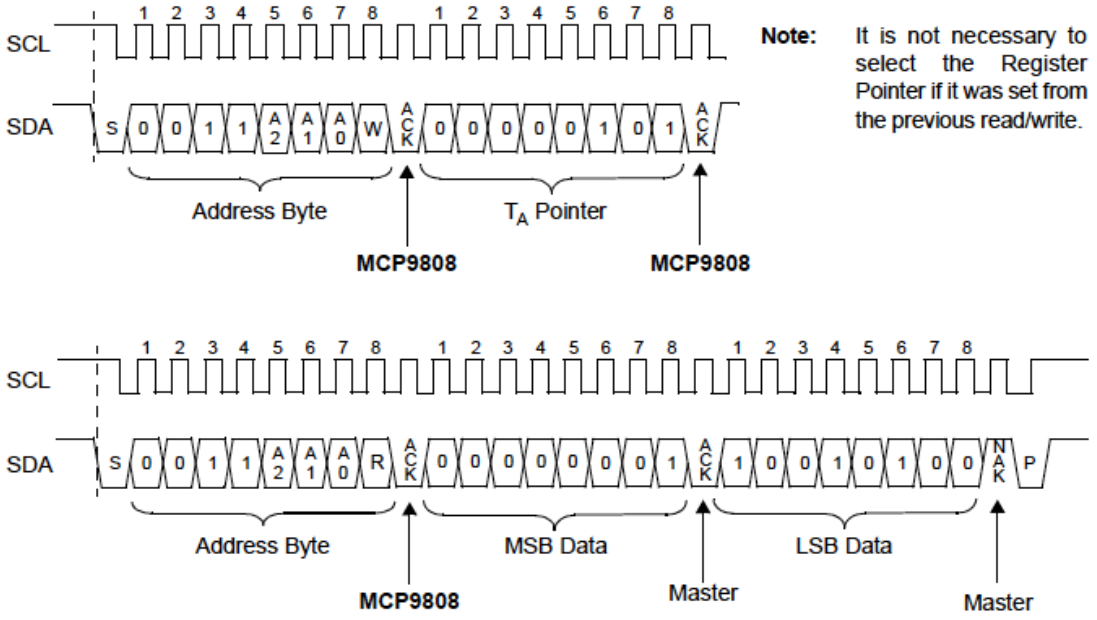
1. Send start sequence (clear SDA while holding SCL high)
2. Send address of Target with read/write bit cleared (byte is I2C address, left shifted one bit with LSB cleared)
3. Send address of register you want to read from



I²C Protocol – Read Operation

- Steps Controller performs to read from Target
  - Example: Reading 2-byte register from Target device

1. Send start sequence (clear SDA while holding SCL high)
2. Send address of Target with read/write bit cleared (byte is I2C address, left shifted one bit with LSB cleared)
3. Send address of register you want to read from
4. Send start sequence (clear SDA while holding SCL high)

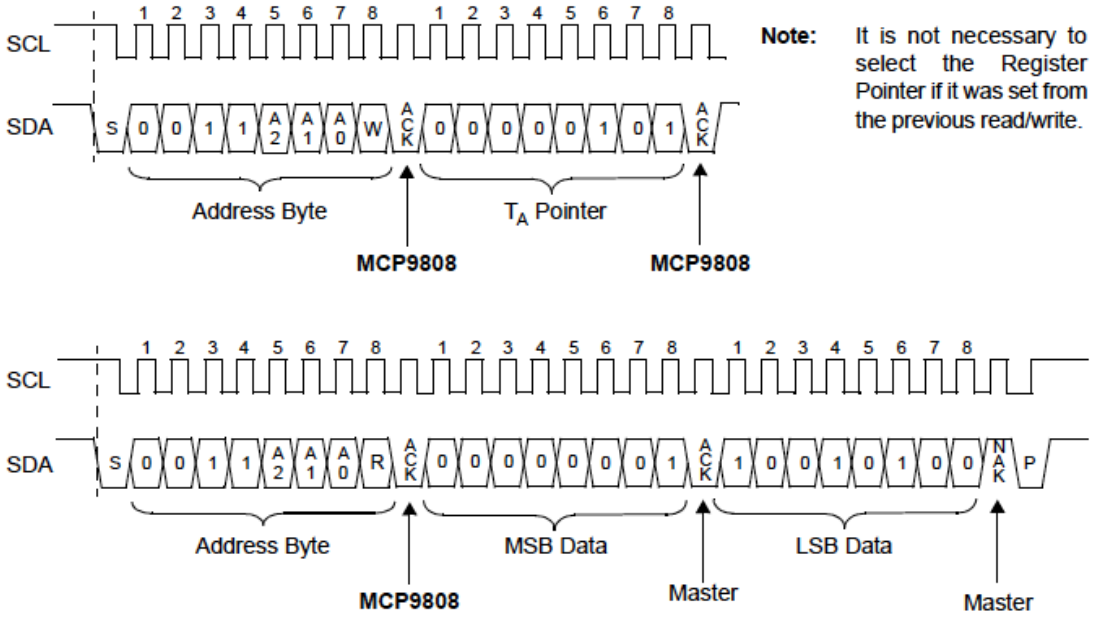


This is sometimes called a "Restart".

I²C Protocol – Read Operation

- Steps Controller performs to read from Target
  - Example: Reading 2-byte register from Target device

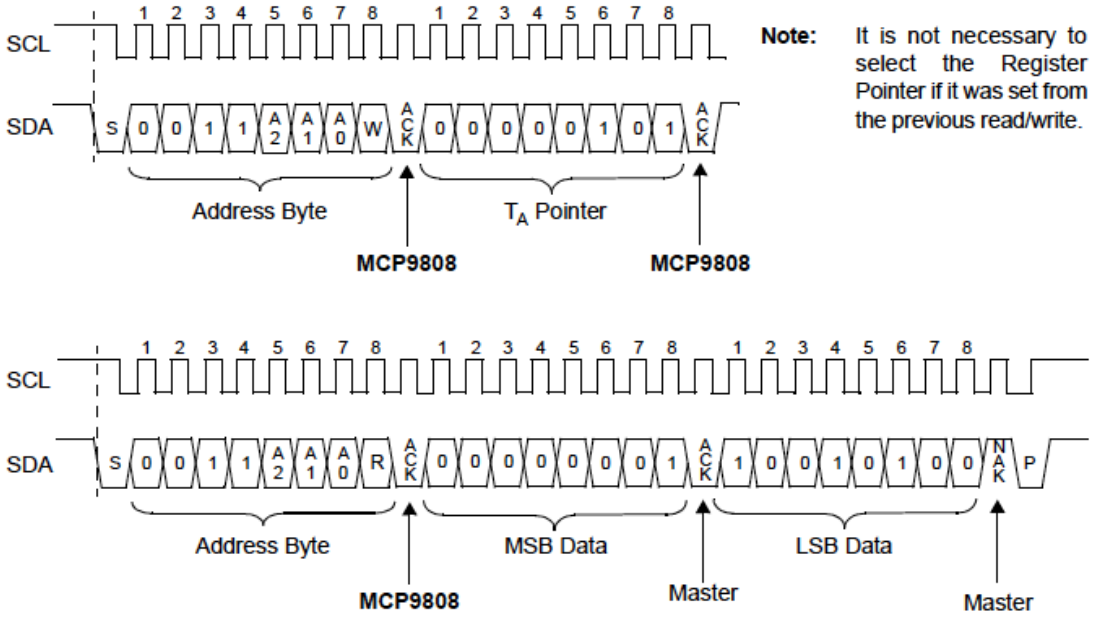
1. Send start sequence (clear SDA while holding SCL high)
2. Send address of Target with read/write bit cleared (byte is I2C address, left shifted one bit with LSB cleared)
3. Send address of register you want to read from
4. Send start sequence (clear SDA while holding SCL high)
5. Send address of Target with read/write bit set (byte is I2C address, left shifted one bit with LSB set)



I²C Protocol – Read Operation

- Steps Controller performs to read from Target
  - Example: Reading 2-byte register from Target device

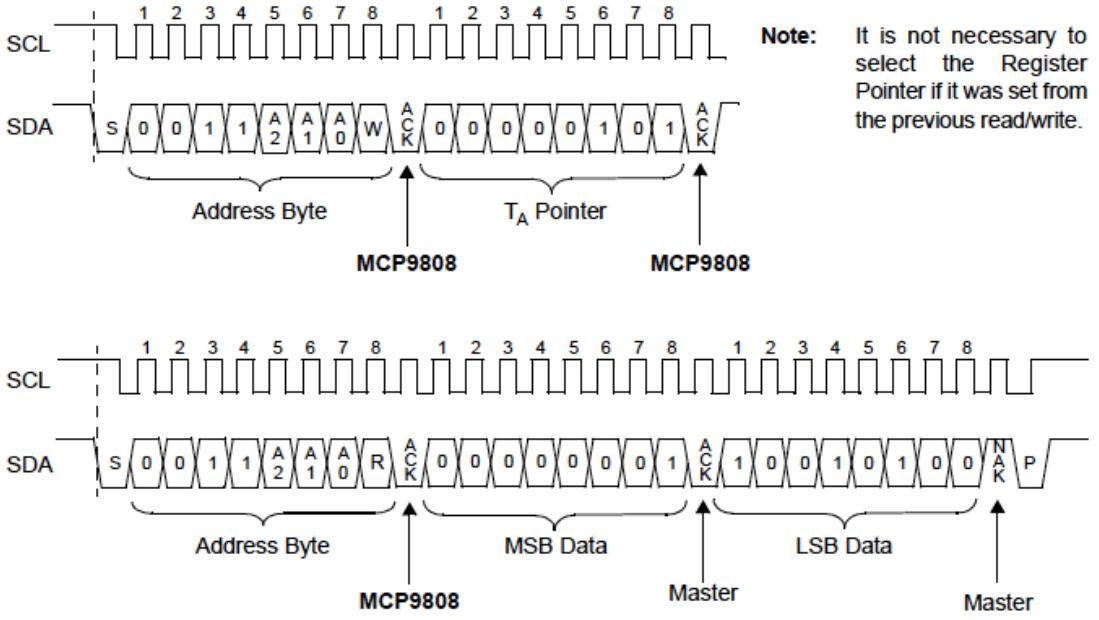
1. Send start sequence (clear SDA while holding SCL high)
2. Send address of Target with read/write bit cleared (byte is I2C address, left shifted one bit with LSB cleared)
3. Send address of register you want to read from
4. Send start sequence (clear SDA while holding SCL high)
5. Send address of Target with read/write bit set (byte is I2C address, left shifted one bit with LSB set)
6. Read data in order of high byte to low byte (see spec sheet of Target device)



I²C Protocol – Read Operation

- Steps Controller performs to read from Target
  - Example: Reading 2-byte register from Target device

1. Send start sequence (clear SDA while holding SCL high)
2. Send address of Target with read/write bit cleared (byte is I2C address, left shifted one bit with LSB cleared)
3. Send address of register you want to read from
4. Send start sequence (clear SDA while holding SCL high)
5. Send address of Target with read/write bit set (byte is I2C address, left shifted one bit with LSB set)
6. Read data in order of high byte to low byte (see spec sheet of Target device)
7. Send stop sequence



I²C on the MSPM0

I2C registers on MSPM0

Note: Eight 1-byte “FIFO” registers which hold data to be transmitted, and data received.

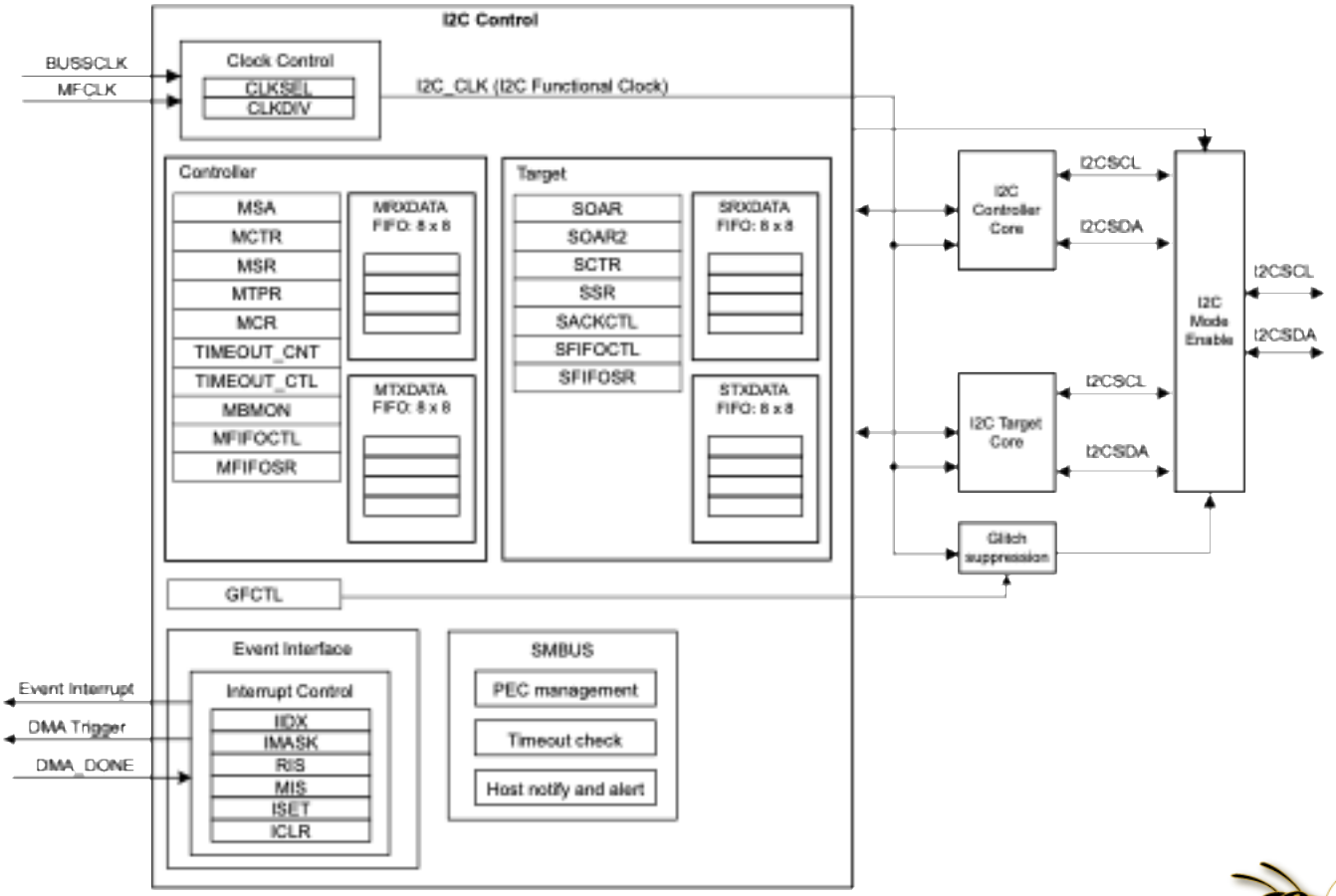


Figure 18-1. I2C Functional Block Diagram

I²C on the MSPM0

- While I2C can be implemented on MSPM0 by using registers directly, we will only discuss use of Driverlib here
- Driverlib provides (somewhat intuitive) interface

Used when operating MSP as I2C Controller

```
DL_I2C_enableController(I2C_Regs *i2c)

uint16_t DL_I2C_fillControllerTXFIFO(I2C_Regs *i2c, uint8_t *buffer,
                                     uint16_t count);

void DL_I2C_startControllerTransfer(I2C_Regs *i2c, uint32_t targetAddr,
                                    DL_I2C_CONTROLLER_DIRECTION direction,
                                    uint16_t length)

uint8_t DL_I2C_receiveControllerData(I2C_Regs *i2c)

uint32_t DL_I2C_getControllerStatus(I2C_Regs *i2c)
```

Used when operating MSP as I2C Target

```
uint8_t DL_I2C_fillTargetTXFIFO (I2C_Regs *i2c, uint8_t *buffer, uint8_t
                                 count)
void DL_I2C_transmitTargetDataBlocking (I2C_Regs *i2c, uint8_t data)
uint8_t DL_I2C_receiveTargetDataBlocking (I2C_Regs *i2c)
```

I²C on the MSPM0

- To configure I2C Controller, first need to set up clock

```
void DL_I2C_setClockConfig  (I2C_Regs * i2c, DL_I2C_ClockConfig * config)
```

Argument 1: I2C module

```
I2C0
I2C1
I2C2
I2C3
```

Argument 2: Clock Configuration Struct

Defined in dl_i2c.h

```
typedef struct DL_I2C_ClockConfig
{
    DL_I2C_CLOCK clockSel ;
    DL_I2C_CLOCK_DIVIDE divideRatio
};
```

```
DL_I2C_setTimerPeriod (I2C_Regs *i2c, uint8_t period)
```

("period" value determined by whether you want to select 100 kbs or 400 kbs…some math involved)

Georgia Tech

I²C on the MSPM0

- Sending data on I2C

Fill transmit buffer:

```
uint16_t DL_I2C_fillControllerTXFIFO(I2C_Regs *i2c, uint8_t *buffer, uint16_t
                                     count);
```

Transmit data:

```
void DL_I2C_startControllerTransfer(I2C_Regs *i2c, uint32_t targetAddr,
                                    DL_I2C_CONTROLLER_DIRECTION_TX, uint16_t length)
```

Use this function to check to make sure transmission is complete:

```
uint32_t DL_I2C_getControllerStatus(I2C_Regs * i2c)
```

Note: To call different Targets on same bus, call targetAddr each time prior to read-write calls.

Georgia Tech

I²C on the MSPM0

- Receiving data on I2C

Send command to target to initiate receive data:

```
void DL_I2C_startControllerTransfer(I2C_Regs *i2c, uint32_t targetAddr,
                    DL_I2C_CONTROLLER_DIRECTION_RX, uint16_t length)
```

Read data from I2C receive FIFO registers:

```
uint8_t  DL_I2C_receiveControllerData (I2C_Regs *i2c))
```

Use this function to check whether there is any more data to be read in receive FIFO:

```
bool DL_I2C_isControllerRXFIFOEmpty(I2C_Regs * i2c)
```

Georgia Tech

I^2C Summary

- Flexible interface – can communicate with up to 128 peripheral devices with only 2 wires!
  - Perhaps most common sensor interface for mechatronic devices
- Not nearly as easy to use as UART
  - Nuances of START, STOP, ACK, NAK can complicate functionality
  - Bus can also hang if protocol not configured right
  - Need to add delays throughout process to allow I2C targets to communicate back

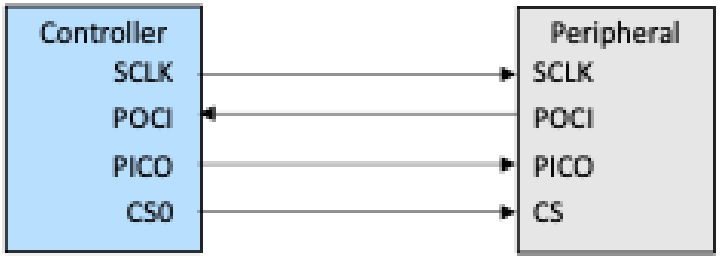
Georgia Tech

I²C Summary

- Will post a basic MSPM0 I2C code on Canvas when available
- Use my code as a template and:
  1. Customize for particular sensors/processors you are using
  2. Add checks so to protect against bus hanging
  3. Consider using interrupts for I2C reads and writes
- See Driverlib documentation for functions that can be used for all of the above

SPI Communications

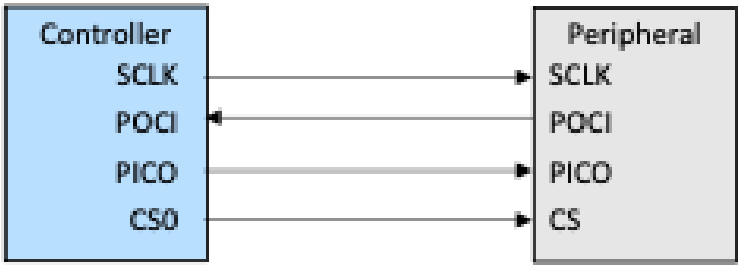
- SPI = Serial Peripheral Interface
- Features:
  - Extremely fast protocol often used for streaming continuous data
    - *Unlike I2C, data can flow continuously without being interrupted by address bytes, start/stop bits, etc.*
  - Data rates up to 10 Mbps (a lot faster than I2C)
  - One bus can handle (theoretically) unlimited number of Peripherals



Georgia Tech

SPI Protocol

- Terminology
  - **SCLK**: Clock wire. This transmits clock signal which is used to synchronize data transfer.
  - **PICO**: Peripheral in Controller out wire. Transfers data from Controller to Peripheral device.
  - **POCI**: Peripheral out Controller in wire. Transfers data from Peripheral to Controller device.
  - **CS0**: Peripheral (Chip) select wire. Activates this Peripheral for communication.

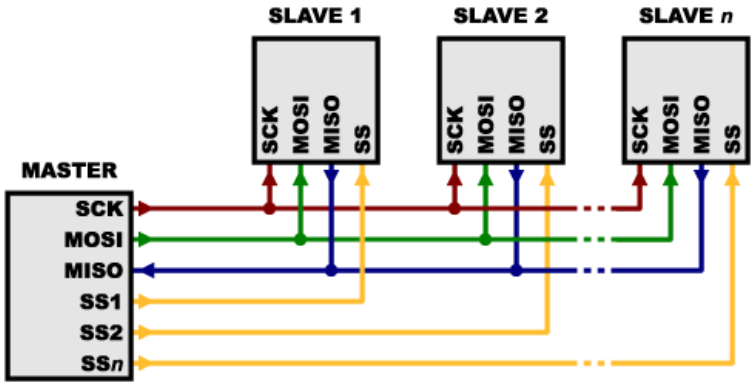


Georgia Tech

SPI Controller-Peripheral Configuration

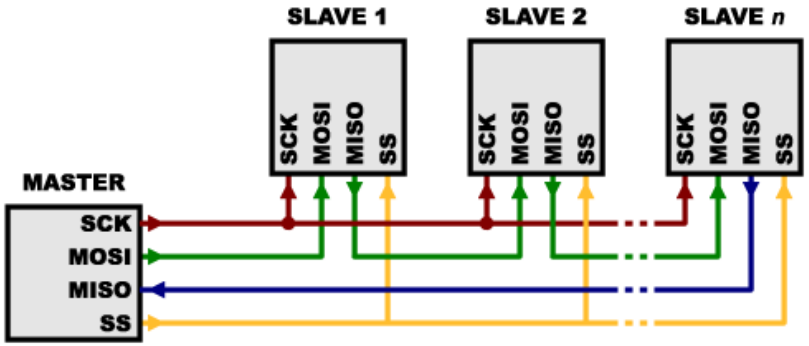
- Multiple Peripherals can be located on same SPI bus

General Configuration



- Use multiple Peripheral select lines
- Set CS*x* line to low to read from Target *x*

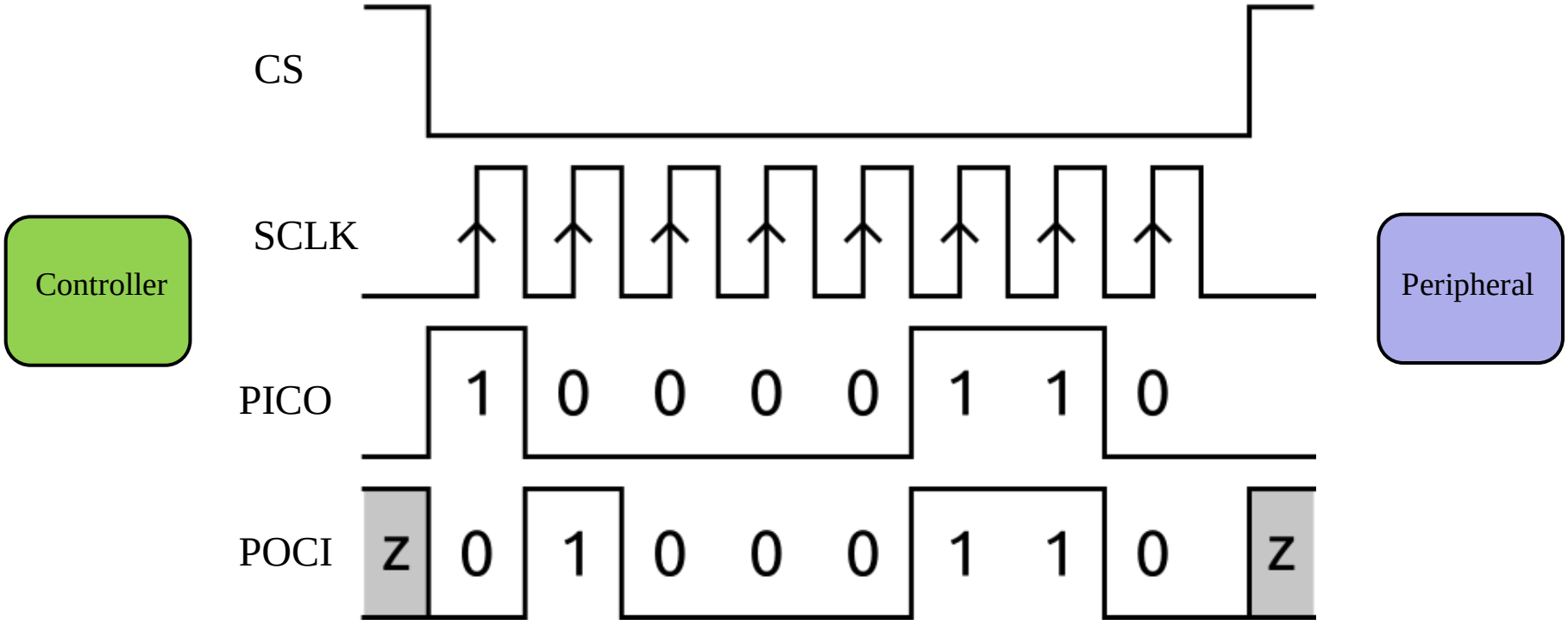
Daisy-Chained Configuration



- Use single Peripheral select line
- Data passed from Controller through each Peripheral sequentially
- Used only when sending data out from Controller exclusively
- POCI line on Controller typically not used

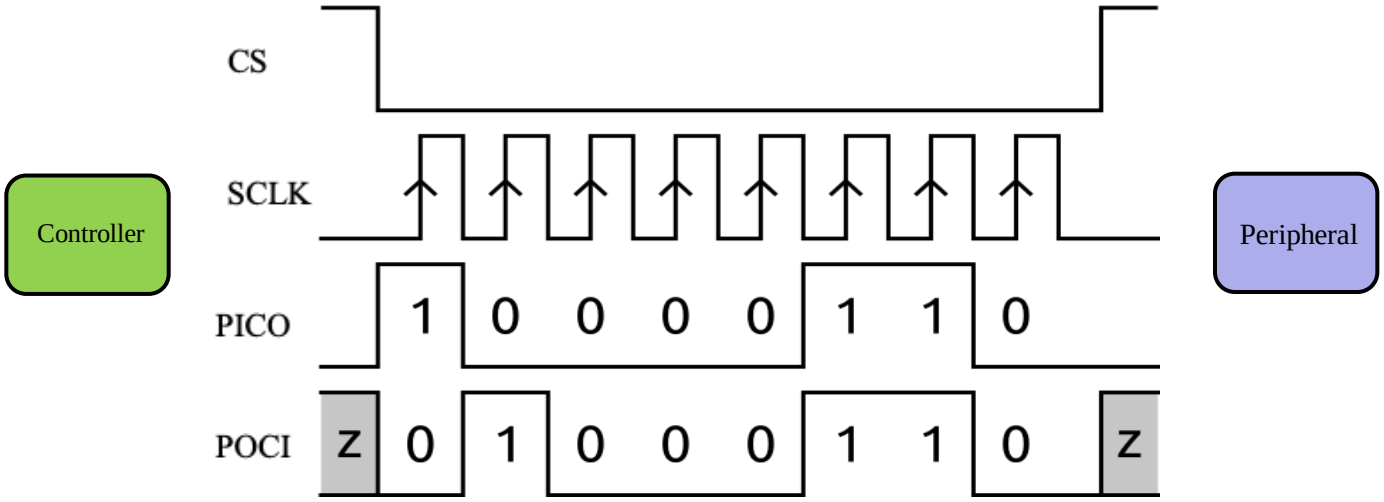
SPI Protocol

- Example SPI transmission of 1 byte of data



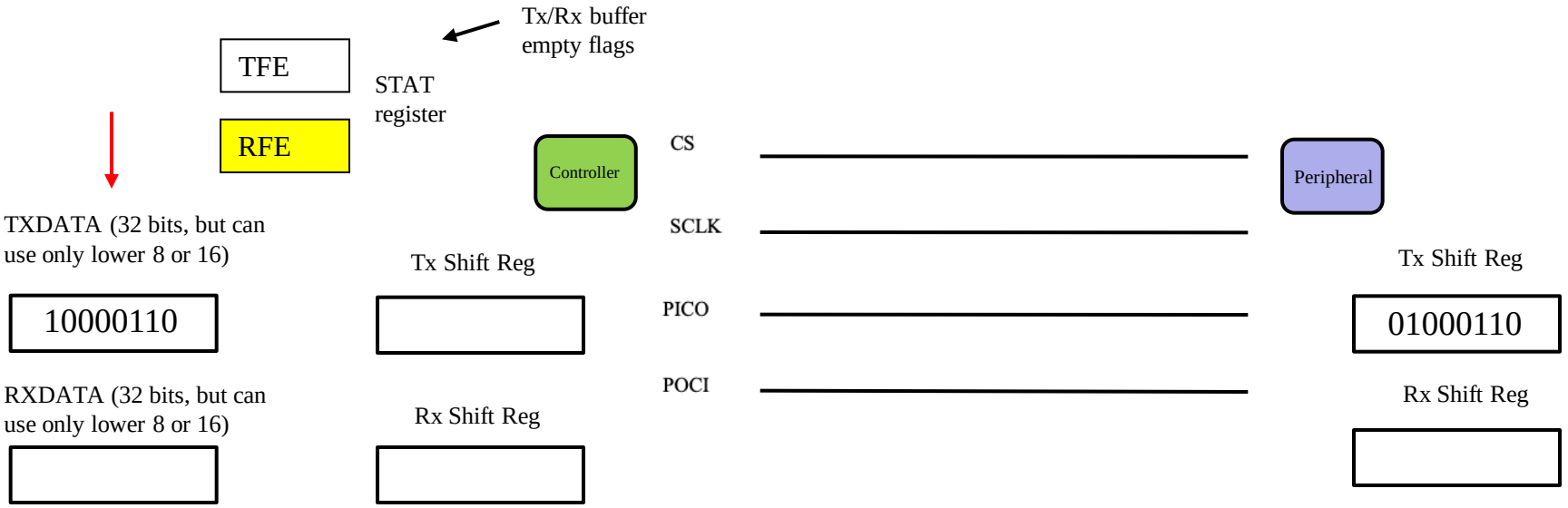
Georgia Tech

SPI Protocol



- SPI communication is full duplex. This means that data is always being transferred from Controller to Peripheral, AND from Peripheral to Controller.
- There are no “read only” or “write only” operations – a bit is always transmitted in both directions when a clock pulse is generated on SCLK
- If Controller just wants to receive data from Peripheral, Controller can just transmit “dummy” value like 0x00. Peripheral will know to ignore it.

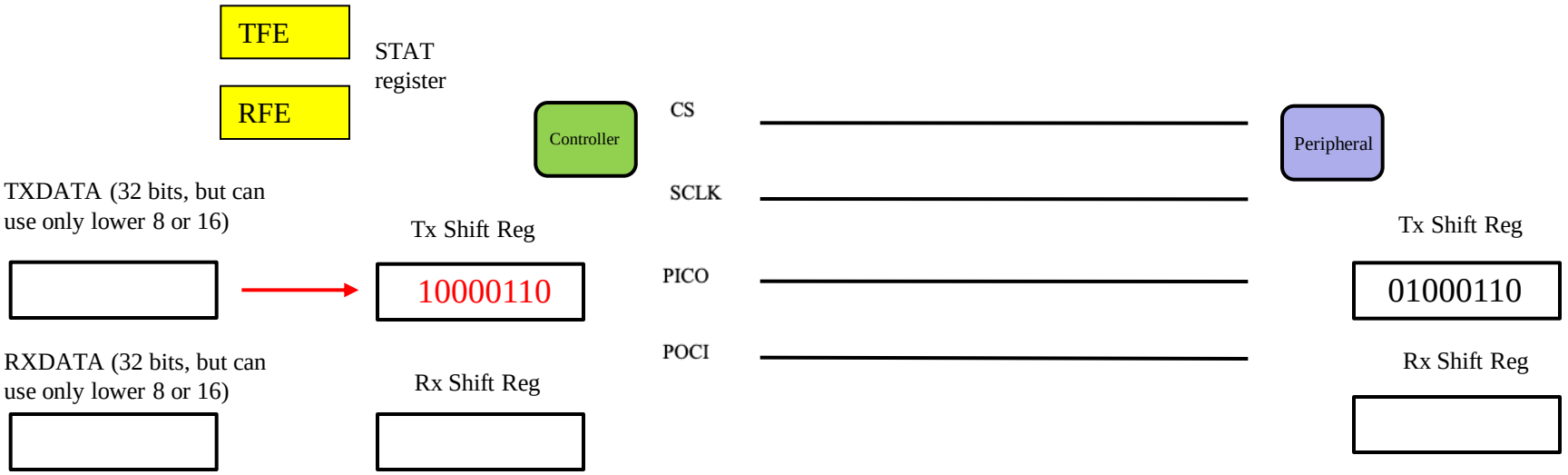
SPI Protocol



Step 1: Controller puts data into transmit buffer register

(Note: Some data is located in Tx Shift Register of Target. This could be "real" data or "dummy" data.)

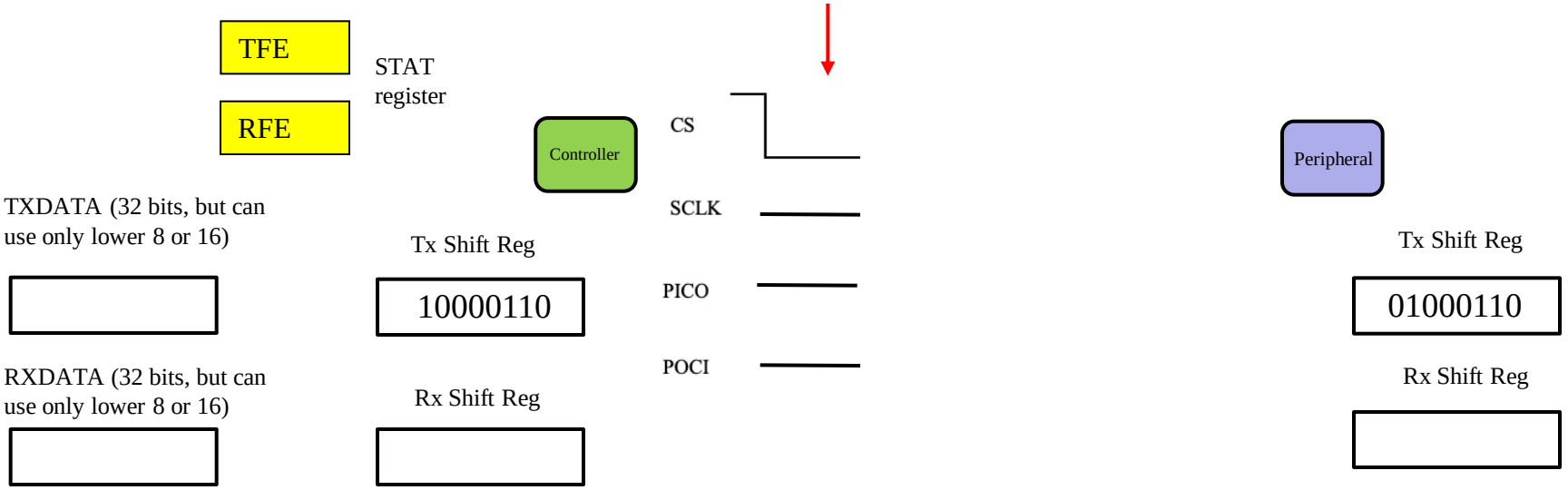
SPI Protocol



Step 2: Data is transferred to Tx Shift Register on Controller.

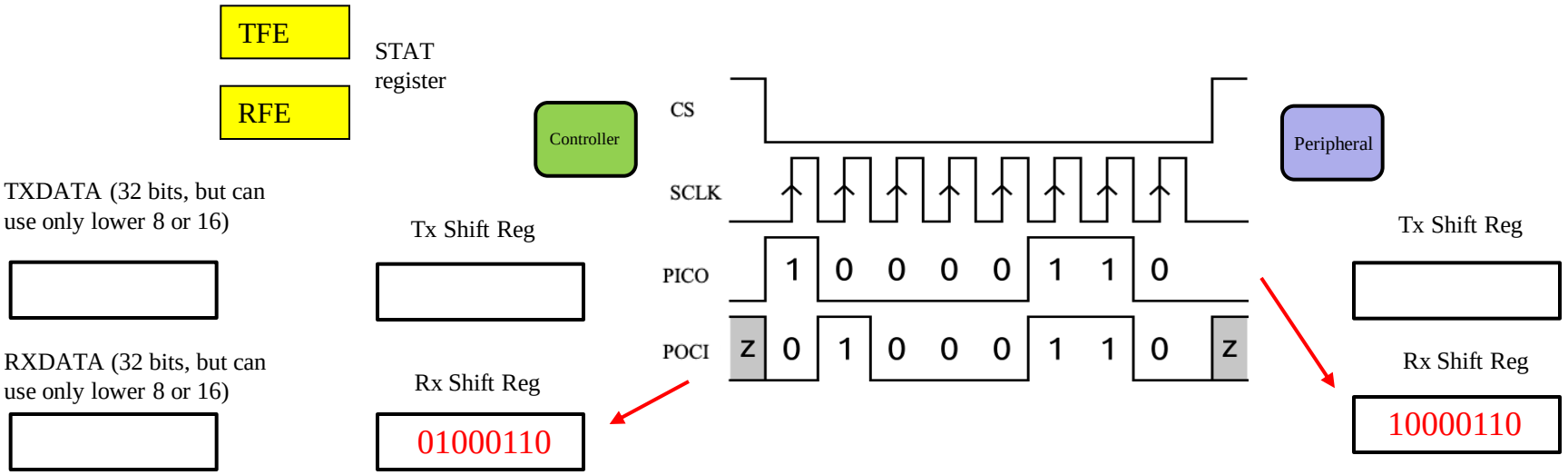
This causes TFE bit to be set in STAT register (on MSPM0), signifying that a new byte can be placed in TXDATA.

SPI Protocol



Step 3: Controller lowers CS line to zero to indicate start of transmission

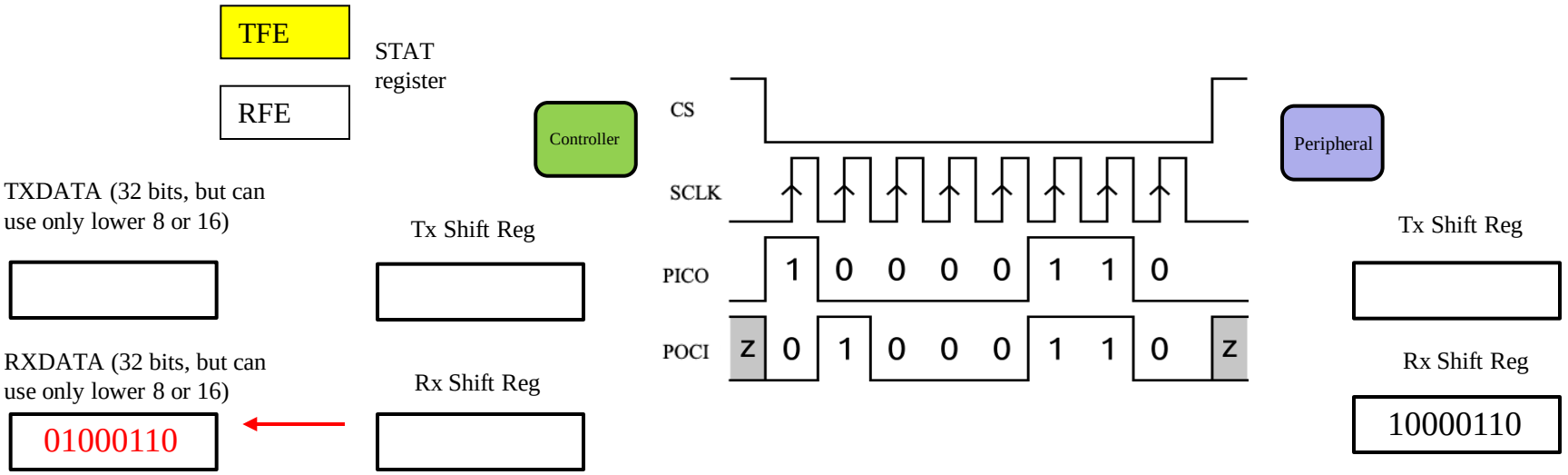
SPI Protocol



Step 4: Controller generates 8 clock pulses, which clocks data into Rx shift registers on both Controller and Peripheral devices

Georgia Tech

SPI Protocol



Step 5: Data is transferred from Receive shift register to Receive Buffer for retrieval by software. Receive FIFO empty flag (RFE) flag is set denoting that new data is ready to be retrieved.

Step 6. Controller raises CS line to 1 indicating end of transmission.

Note: Controller can leave CS line high if wanting to initiate another transmission.

Georgia Tech

SPI on the MSPM0

- Key SPI-related registers

SPIx->TXDATA	32-bit register that holds data waiting to be moved to transmit shift register. Writing to this clears TXEMPTY (transmit interrupt flag) in RIS register.
SPIx->RXDATA	32-bit register that holds the last received word from receive shift register. Reading from this clears RXFULL (receive interrupt flag) in RIS register.
SPIx->STAT	32-bit register that holds status flags for SPI that can be used for polling. Bits 5-31 do nothing. Bit 5 (BUSY) can be checked to see if transmission still in progress, TFE bit to see if TXDATA is empty, RFE bit to see if RXDATA empty.

Georgia Tech

SPI on the MSPM0

- To configure SPI Controller, use these functions

Initialize SPI parameters:

```
void DL_SPI_init (SPI_Regs *spi, DL_SPI_Config *config)
```

Argument 1: SPI instance

```
SPI0
SPI1
SPI2
SPI3
```

Defined in dl_spi.h

Argument 2: Configuration Struct

```
typedef struct DL_SPI_Config
{
    DL_SPI_MODE mode ;
    DL_SPI_FRAME_FORMAT frameFormat ;
    DL_SPI_PARITY   parity ;
    DL_SPI_DATA_SIZE dataSize ;
    DL_SPI_BIT_ORDER bitOrder ;
    DL_SPI_CHIP_SELECT chipSelectPin
};
```

Georgia Tech

SPI on the MSPM0

- To SPI, use these functions

Set bit rate (see DL documentation for how to compute constant SCR):

```
void DL_SPI_setBitRateSerialClockDivider (SPI_Regs *spi, uint32_t SCR)
```

Enable the SPI module:

```
void DL_SPI_enable (SPI_Regs *spi)
```

If setting up interrupt (optional):

```
void DL_SPI_enableInterrupt (SPI_Regs *spi, uint32_t interruptMask)
```

mask field set to either:

```
DL_SPI_INTERRUPT_TX,
DL_SPI_INTERRUPT_RX,
DL_SPI_INTERRUPT_TX_EMPTY,
DL_SPI_INTERRUPT_RX_FULL
```

Georgia Tech

SPI on the MSPM0

- Key transmission and receive functions
  - Can send data of varying sizes

Transmitting data:

```
DL_SPI_transmitData8 (SPI_Regs *spi, uint8_t data)

DL_SPI_transmitData16 (SPI_Regs *spi, uint16_t data)

DL_SPI_transmitData32 (SPI_Regs *spi, uint32_t data)
```

These functions initiate the following sequence:

1. Places data in the Transmit buffer register.
2. The SPI module immediately transfers byte from buffer to transmit shift register.
3. SPI module generates 8/16/32 clock pulses and transmits data on PICO line.

Georgia Tech

SPI on the MSPM0

- Key transmission and receive functions
  - Can receive data of varying sizes

Receiving data:

```
uint8_t DL_SPI_receiveData8 (SPI_Regs *spi)

uint16_t DL_SPI_receiveData16 (SPI_Regs *spi)

uint32_t DL_SPI_receiveData32 (SPI_Regs *spi)
```

- This function ONLY grabs data from the Receive Buffer register. No actual SPI transfer is initiated by this call.
- Thus you MUST check to make sure receive data interrupt set before calling this function.
- No clock pulses are generated by calling this function. Only way to initiate data transfer is by calling `DL_SPI_transmitData8/16/32(…)` function.

Georgia Tech

SPI on the MSPM0

- To check if interrupt flag is set, use following function

```
uint32_t DL_SPI_getRawInterruptStatus (SPI_Regs *spi, uint32_t interruptMask)
```

Argument 2 is either: `DL_SPI_INTERRUPT_TX`, `DL_SPI_INTERRUPT_RX`, `DL_SPI_INTERRUPT_TX_EMPTY`, `DL_SPI_INTERRUPT_RX_FULL`

Returns bitmask of which of the interrupt flags are current high.

SPI on the MSPM0

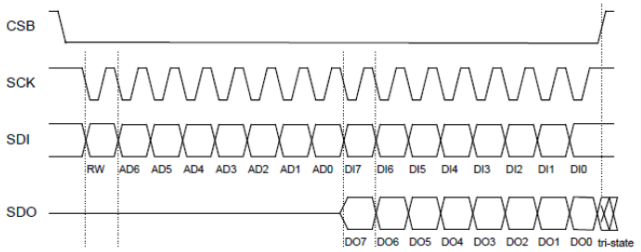
- SPI configuration function

```
void initializeI2C(void)
{
  /* Configure pins for SPI mode */
  DL_GPIO_initPeripheralOutputFunction(IOMUX_PINCM26, 3);  // SCLK
  DL_GPIO_initPeripheralOutputFunction(IOMUX_PINCM25, 3);  // PICO
  DL_GPIO_initPeripheralInputFunction(IOMUX_PINCM24, 3);   // POCI
  DL_GPIO_initPeripheralOutputFunction(IOMUX_PINCM23, 3);  // CS0

  /* Configuring SPI clock */
  DL_SPI_setClockConfig(SPI_0_INST, (DL_SPI_ClockConfig *) &gSPI_0_clockConfig);

      /* Configuring SPI in Controller mode */
      DL_SPI_init(SPI1, (DL_SPI_Config *) &gSPI_0_config);

      /* Configuring SPI clock speed (use math from DL documentation)*/
      DL_SPI_setBitRateSerialClockDivider(SPI1, 31);

      /* Enable SPI module */
      DL_SPI_enable(SPI1);
}
```

```
static const DL_SPI_Config gSPI_0_config = {
    .mode       = DL_SPI_MODE_CONTROLLER,
    .frameFormat = DL_SPI_FRAME_FORMAT_MOTO4_POL0_PHA0,
    .parity     = DL_SPI_PARITY_NONE,
    .dataSize   = DL_SPI_DATA_SIZE_8,
    .bitOrder   = DL_SPI_BIT_ORDER_MSB_FIRST,
    .chipSelectPin = DL_SPI_CHIP_SELECT_0,
};
```

```
static const DL_SPI_ClockConfig gSPI_0_clockConfig = {
    .clockSel    = DL_SPI_CLOCK_BUSCLK,
    .divideRatio = DL_SPI_CLOCK_DIVIDE_RATIO_1
};
```

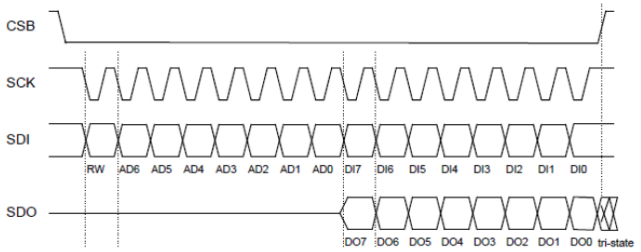
SPI on the MSPM0



- Write function
  - This writes an 8-bit value to a certain register on the Peripheral device

```
void spi_peripheral_write8(uint8_t reg, uint8_t value) {

  reg = reg & ~0x80;    // Clear high bit to write to this register

  // Transmit Target register address
  DL_SPI_transmitData8(SPI1, reg);

  // Transmit write value
  DL_SPI_transmitData8(SPI1, value);

}
```

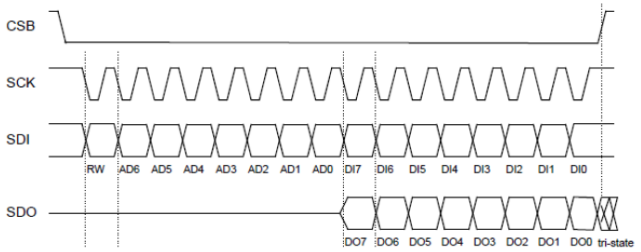
SPI on the MSPM0



- Read function
  - This reads an 8-bit value from a certain register on the Peripheral device

```
uint8_t spi_peripheral_read8(uint8_t reg) {

  uint8_t RXData = 0;
  reg = 0x80 | reg; // Set high bit to read from this register

 // Make sure transmit buffer is clear
  while ( DL_SPI_getRawInterruptStatus (SPI1, DL_SPI_INTERRUPT_TX_EMPTY) == 0);

  DL_SPI_transmitData8(SPI1, reg);              // Transmit register address

  // Send clock pulses.  This will push Target to send data to Controller on PICO line.
  DL_SPI_transmitData8(SPI1, 0x00);

  // Make sure data is available
  while (DL_SPI_getRawInterruptStatus (SPI1, DL_SPI_INTERRUPT_RX_FULL) == 0);

  RXData = DL_SPI_receiveData8(SPI1);     // Pick up received byte in SPI receive data register

  return RXData;
}
```

SPI on the MSPM0



- Read function
  - This reads an 8-bit value from a certain register on the Peripheral device

```
uint8_t spi_peripheral_read8(uint8_t reg) {

  uint8_t RXData = 0;
  reg = 0x80 | reg; // Set high bit to read from this register

 // Make sure transmit buffer is clear
  while ( DL_SPI_getRawInterruptStatus (SPI1, DL_SPI_INTERRUPT_TX_EMPTY) == 0);

  DL_SPI_transmitData8(SPI1, reg);              // Transmit register address

  // Send clock pulses.  This will push Target to send data to Controller on PICO line.
  DL_SPI_transmitData8(SPI1, 0x00);

  // Make sure data is available
  while (DL_SPI_getRawInterruptStatus (SPI1, DL_SPI_INTERRUPT_RX_FULL) == 0);

  RXData = DL_SPI_receiveData8(SPI1);     // Pick up received byte in SPI receive data register

  return RXData;
}
```

Note that Controller actually has to write data in order to read data

SPI Summary

- SPI is a synchronous communication protocol that has unique advantages:
  - Hardware needed to communicate is very simple (i.e., can be made very cheap) compared to UART, I2C
  - Low overhead, supports very high speeds
  - Can implement full duplex (simultaneous read-write communication)
  - Can use multiple Peripherals on same SPI bus

SPI Summary

- Will post basic MSPM0 I2C and SPI codes on Canvas
  - Or you can find Driverlib examples
- Use my code as a template and customize for particular sensors/processors you are using
- See Driverlib documentation for functions that can be used for all of the above

Serial Communications Summary

Protocol	Typical Speeds	Sync / Async	No. Pins Required	Full Duplex	Multiple Targets
UART	8 Mbit/s	Asynchronous	2 (Tx, Rx)	No	No
I^2C	400 kbit/s or 1 Mbit/s	Synchronous	2 (SDA, SCL)	No	Yes
SPI	25 Mbit/s	Synchronous	3 or 4 (POCI, PICO, SCLK, CS)	Yes	Yes

- I2C and SPI used commonly for on-chip communication between devices
- UART commonly used to communicate between multiple devices not on same chip (multiple MCUs, etc)

Georgia Tech