



香港中文大學

The Chinese University of Hong Kong

CENG2400 Embedded System Design

Lecture 09:

Serial Communication (II)

Ming-Chang YANG

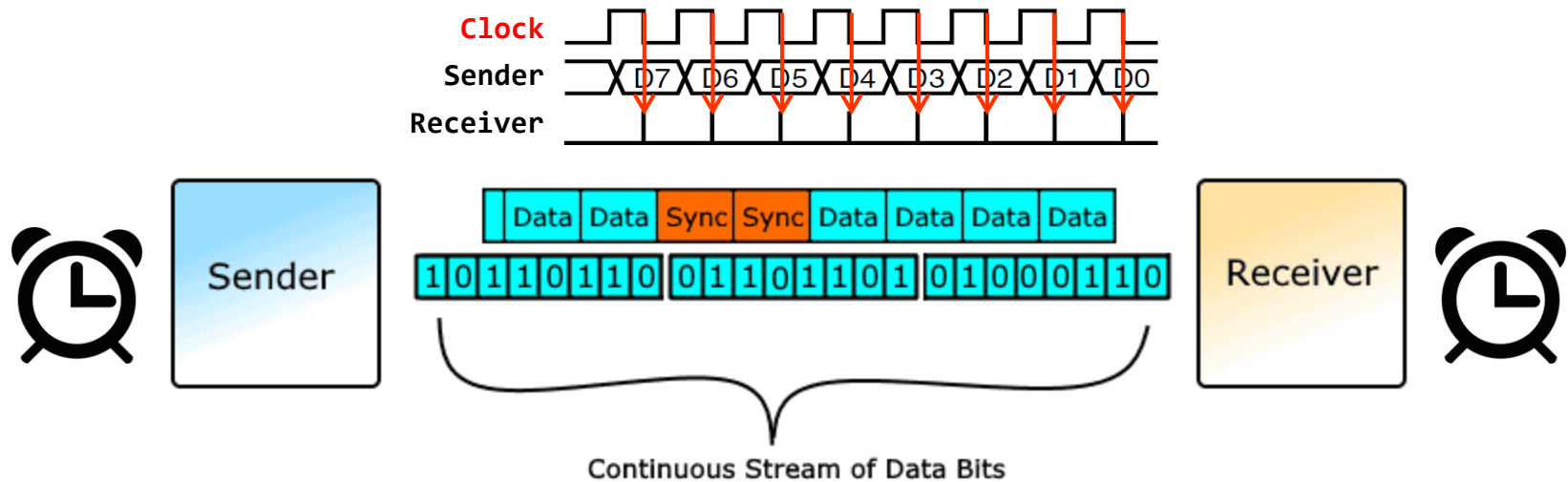
mcyang@cse.cuhk.edu.hk

Thanks to Prof. Q. Xu and Drs. K. H. Wong, Philip Leong, Y.S. Moon, O. Mencer, N. Dulay, P. Cheung for some of the slides used in this course!

Recall: Serial Communication (1/2)



- Serial comm. can be **synchronous** or **asynchronous**:
 - Synchronous Transmission**: The sender and receiver are synchronized using an **external synchronization clock**.

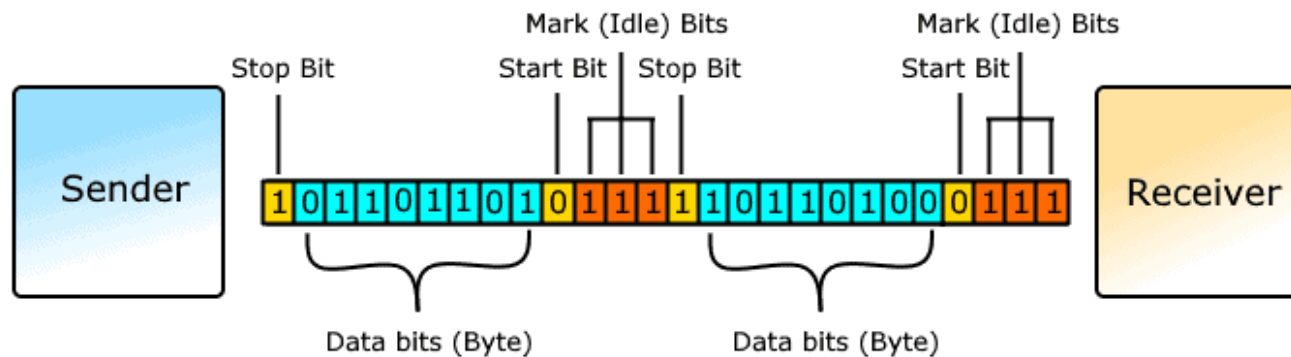


- An **extra channel** is typically employed to transmit the **clock signal**.
 - It provides **synchronous clock pulses** that define a **constant time interval** for data transmission.
- Common **synchronous** serial communication protocols: **Serial Peripheral Interface (SPI)** and **Inter-Integrated Circuit Bus (I²C)**.

Recall: Serial Communication (2/2)

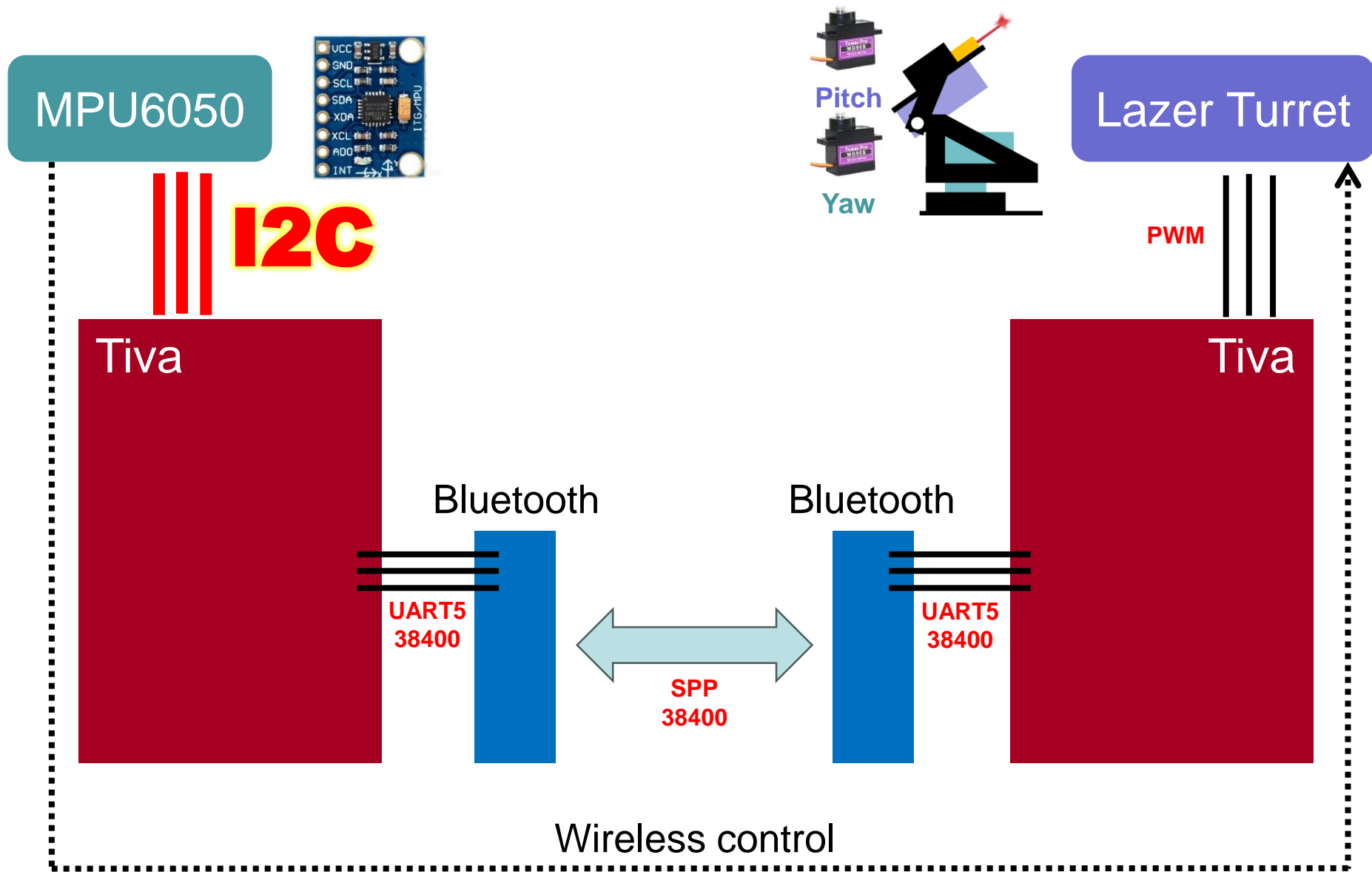


- Serial comm. can be **synchronous** or **asynchronous**:
 - Asynchronous Transmission**: Additional bits are introduced to identify the beginning and end of the data.



- As a result, it **does not** need any external synchronization clock, but the sender and receiver must follow the same **baud rate**.
 - Baud rate**: the number of symbols transmitted per second.
 - Higher baud rates allow **faster transmission** but are more **error-prone**.
- Common **asynchronous** serial communication protocols: **Universal Asynchronous Receiver-Transmitter (UART)**, **Universal Serial Bus (USB)**, and **Bluetooth** (wireless!).

FP: Motion-Controlled Lazer Turret

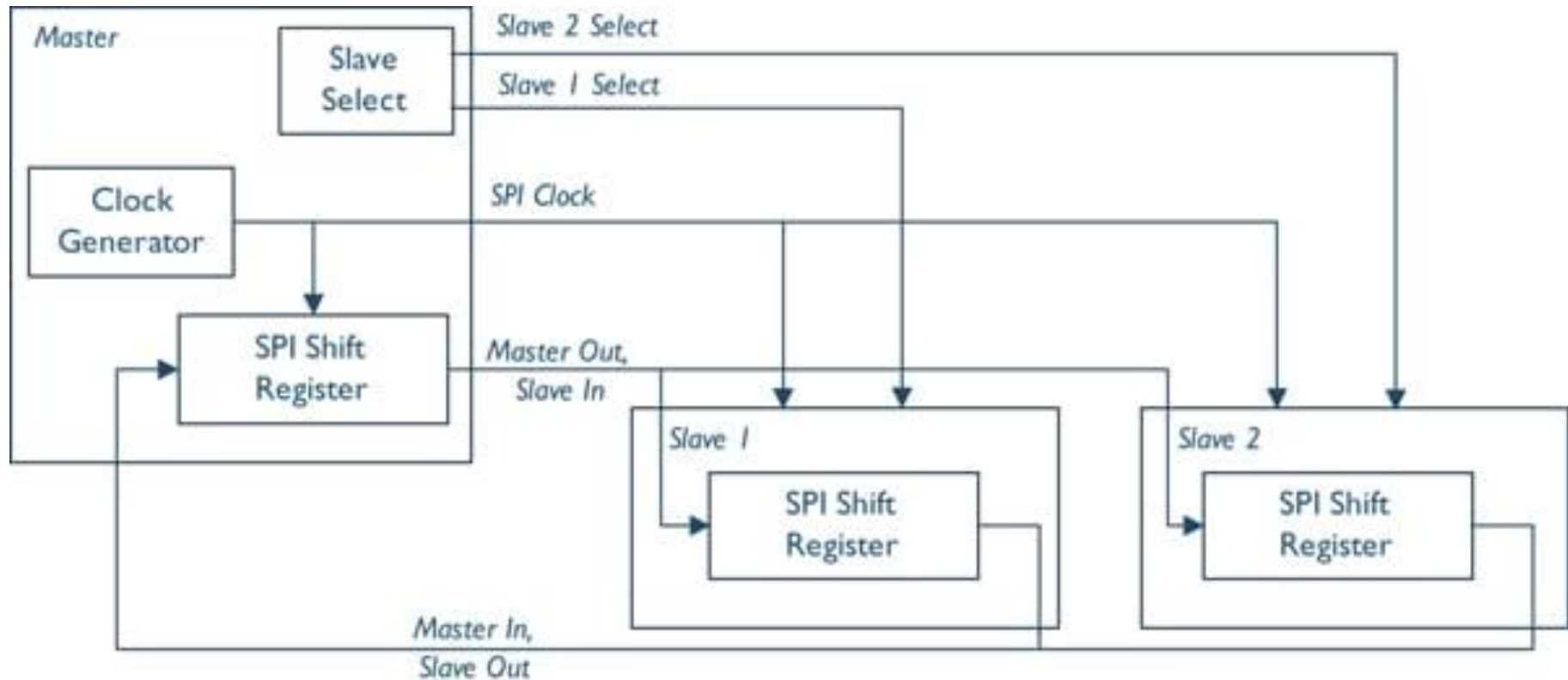


- **Serial Peripheral Interface (SPI)**
 - Logic Signals & Data Transmission
 - Clocking: Polarity and Phase
 - SPI on Tiva™ & in TivaWare™ Library
- **Inter-Integrated Circuit Bus (I²C)**
 - Message Format
 - Start and Stop Conditions, Data Validity, and Acknowledgement
 - I²C on on Tiva™ & in TivaWare™ Library
- **Bluetooth (Notes)**
 - Serial Port Profile (SPP)

Serial Peripheral Interface (SPI)



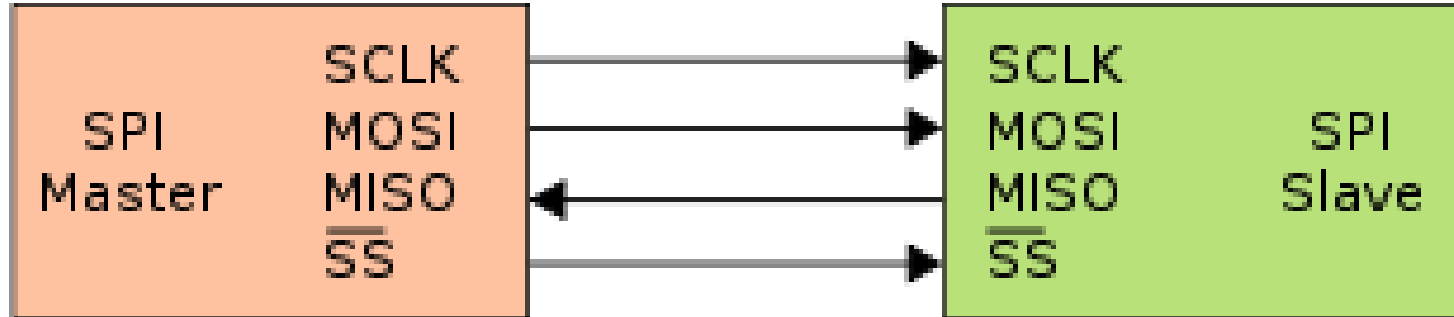
- **SPI** is a type of **synchronous** serial communication with a master and one or more slaves:
 - Typically, the MCU is the **master** and peripheral devices are **slaves** (e.g., ADCs, accelerometers, LCD controllers, etc.).



SPI Logic Signals

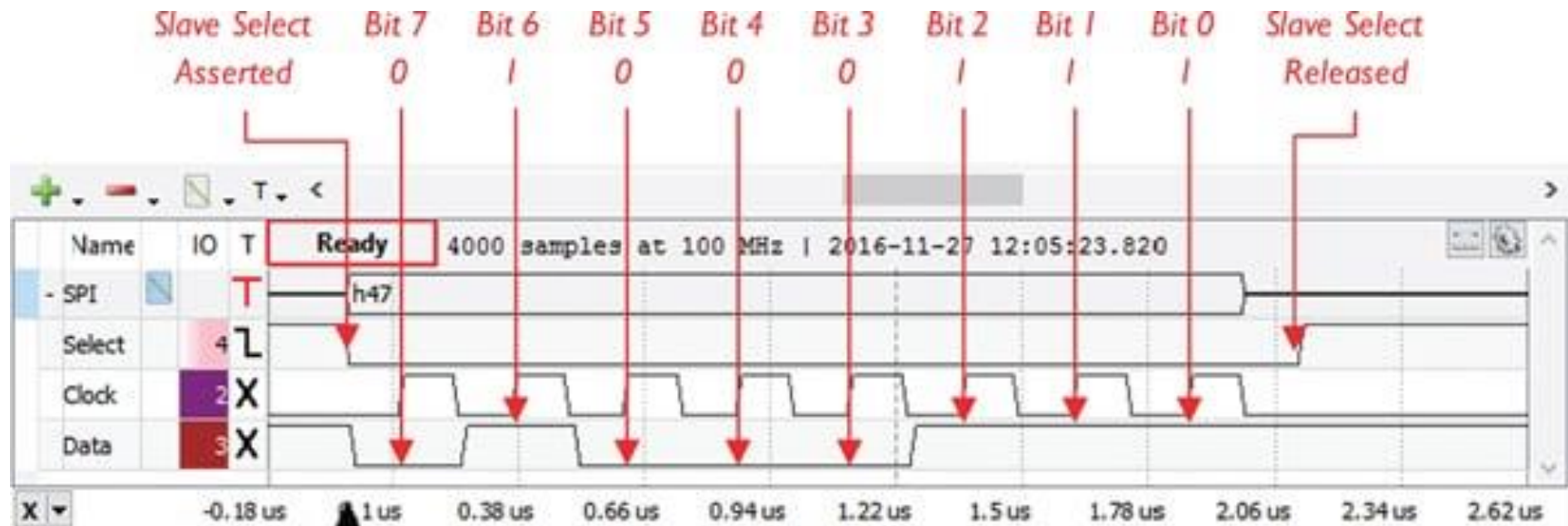


- The SPI connection has **four** logic signals:



- ① **System Clock** indicates **when** data is to be sampled.
 - ② **Master-Out Slave-In** is the **master output** and **slave input**.
 - ③ **Master-In Slave-Out** is the **master input** and **slave output**.
 - ④ **Slave Select/Chip Select** is the **select line** of the slave targeted for communication.
- *Note: If bidirectional communication between the master and slaves is not needed, only one data line is needed (MOSI or MISO), depending on data transfer direction.*

SPI Data Transmission

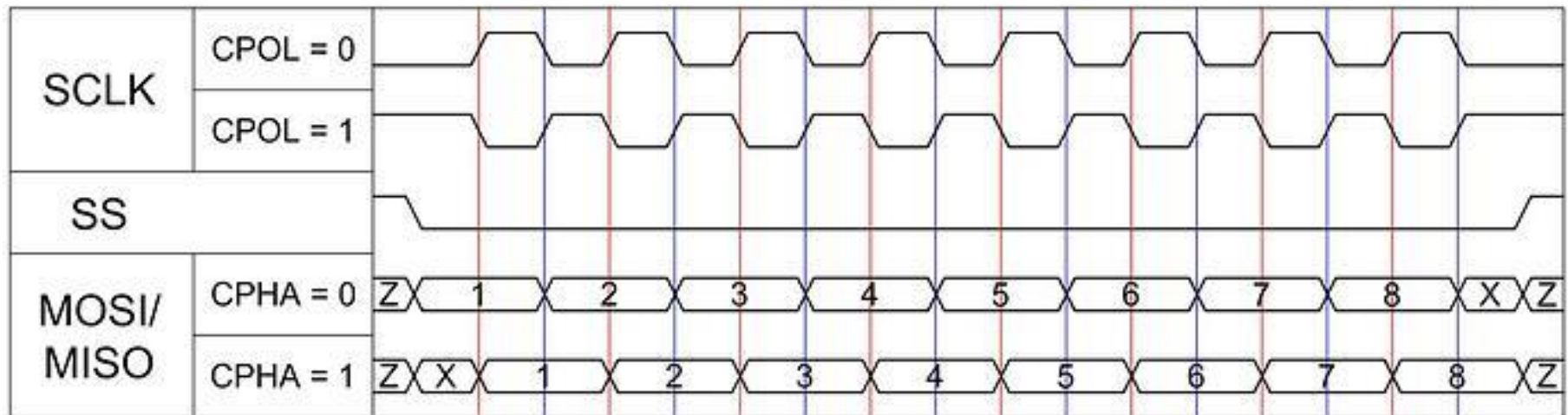


- **M** selects a particular **S** by asserting its **slave select**.
- **M** asserts the **clock** signal to indicate when its data output signal is valid and should be sampled by **S**.
 - Each clock pulse exchanges one bit (usually **MSB first**).
 - At the same time, **M** can receive data from **S** (*not shown*).
- Finally, the **slave select** can be released.

SPI Clocking: Polarity and Phase



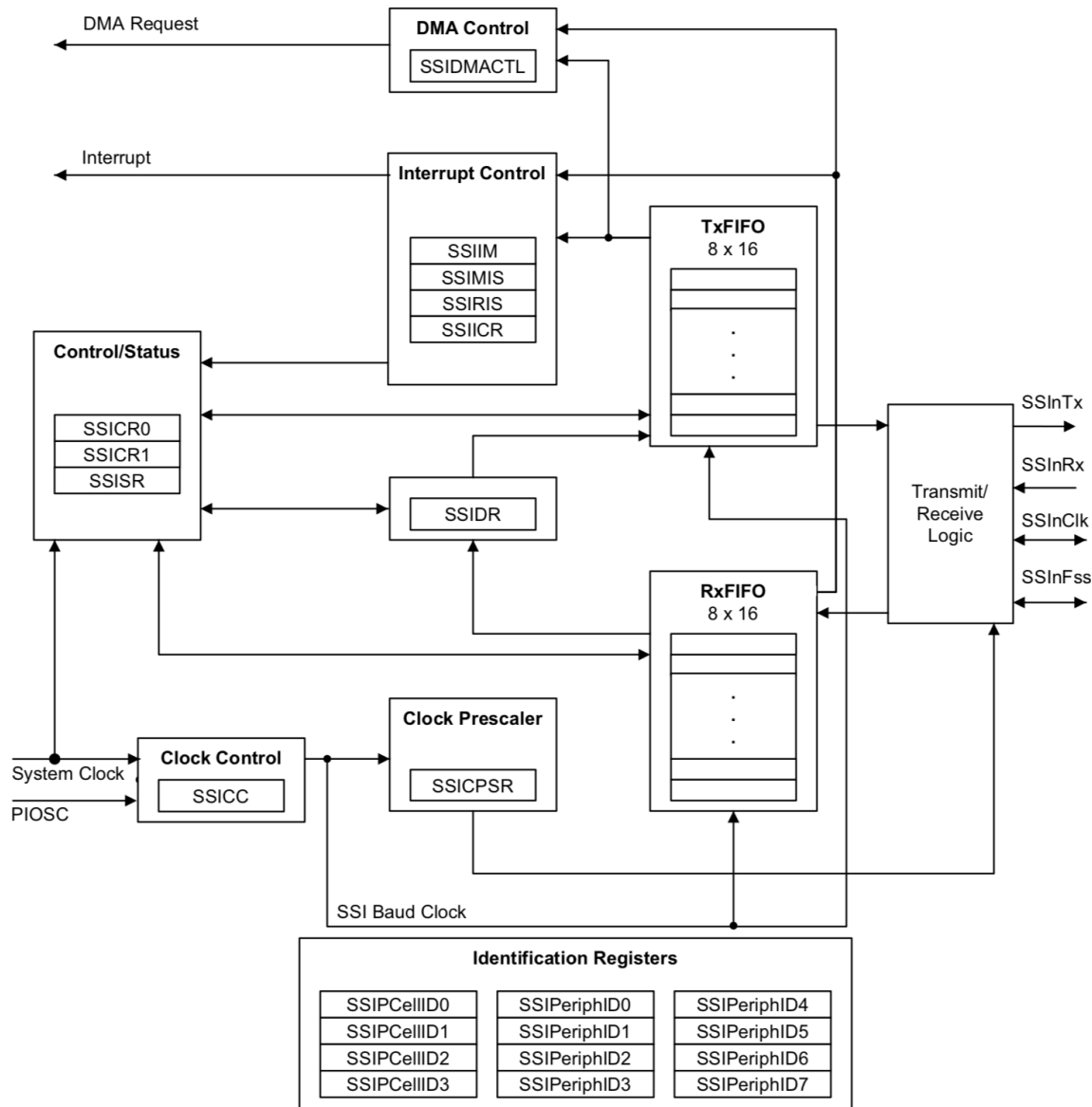
- There are **four** different modes of SPI, based on the relationship between the **clock** and the **data**.
 - The **clock polarity (CPOL)** determines whether the clock signal is **active-high** (idle-low) or **active-low** (idle-high).
 - The **clock phase (CPHA)** determines when the slave starts transmitting valid data on the MISO signal (often **MSB first**).
 - CPHA = 0**: Data is captured on the **first** clock edge;
 - CPHA = 1**: Data is captured on the **second** clock edge.



– *Note: Master and slave must be in the same clocking mode.*

Synchronous Serial Interface on Tiva™

- Tiva™ has four **SSI** modules.
 - Programmable **interface operation**;
 - Freescale **SPI**, MICROWIRE, or Texas Instruments synchronous serial interfaces.
 - Programmable **master or slave operation**;
 - Programmable **clock bit rate and prescaler**;
 - Standard FIFO-based or End-of-Transmission interrupts;
 - Etc.



Bit Rate Generation (TM4C123GH6PM)



- **SSI** supports a **programmable bit rate clock divider and prescaler** to generate the serial output clock:

$$SSInClk = SysClk / (CPSDVSR * (1 + SCR))$$

- It is derived by dividing down the input clock (*SysClk*).
 - The clock is first divided by an **even prescale value** *CPSDVSR* from 2~254, programmed in the **SSI Clock Prescale (SSICPSR)** register.
 - It is further divided by a value from 1~256, which is $(1 + SCR)$, where *SCR* is programmed in the **SSI Control 0 (SSICR0)** register.
- Bit rates are supported to **2 MHz** and **higher**.

Prescaler: A counting circuit used to reduce a high frequency signal to a lower frequency by integer division.

SSI Clock Prescale (SSICPSR)



- It specifies the **division factor** which is used to derive the *SSInClk* from the system clock.

SSI Clock Prescale (SSICPSR)

SSI0 base: 0x4000.8000

SSI1 base: 0x4000.9000

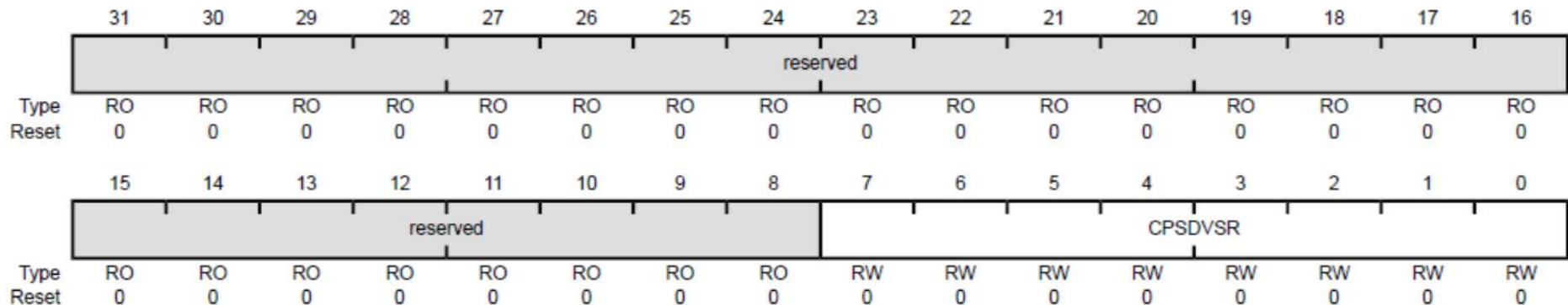
SSI2 base: 0x4000.A000

SSI3 base: 0x4000.B000

Offset 0x010

Type RW, reset 0x0000.0000

$$SSInClk = SysClk / (CPSDVSR * (1 + SCR))$$



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	CPSDVSR	RW	0x00	SSI Clock Prescale Divisor This value must be <u>an even number from 2 to 254</u> , depending on the frequency of <i>SSInClk</i> . The LSB always returns 0 on reads.

SSI Control 0 (SSICR0) (1/2)



- It contains bit fields that control various functions within the SSI module.
 - Such as *protocol mode*, *clock rate*, *clock phase*, *clock polarity*, and *data size*.

SSI Control 0 (SSICR0)

SSI0 base: 0x4000.8000

SSI1 base: 0x4000.9000

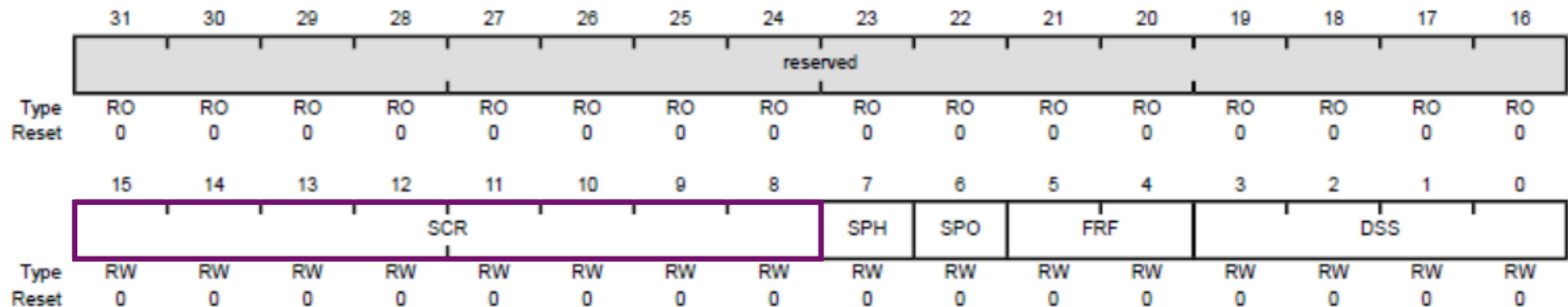
SSI2 base: 0x4000.A000

SSI3 base: 0x4000.B000

Offset 0x000

Type RW, reset 0x0000.0000

$$SSInClk = SysClk / (CPSDVSR * (1 + SCR))$$



SSI Control 0 (SSICR0) (2/2)



Bit/Field	Name	Type	Reset	Description
31:16	reserved	RO	0x0000	Software should not rely on the value of a reserved bit.
15:8	SCR	RW	0x00	SSI Serial Clock Rate: This bit field is used to generate the transmit and receive bit rate of the SSI. (Note: It is a value from 0-255.)
7	SPH	RW	0	SSI Serial Clock Phase: The SPH control bit selects the clock edge that captures data and allows it to change state. (Note: This bit is only applicable to the Freescale SPI.) 0: Data is captured on the first clock edge transition. 1: Data is captured on the second clock edge transition.
6	SPO	RW	0	SSI Serial Clock Polarity 0: A steady state Low value is placed on the SSInClk pin. 1: A steady state High value is placed on the SSInClk pin when data is not being transferred.
5:4	FRF	RW	0x0	SSI Frame Format Select 0x0: Freescale SPI Frame Format 0x1: Texas Instruments Synchronous Serial Frame Format 0x2: MICROWIRE Frame Format 0x3: Reserved
3:0	DSS	RW	0x0	SSI Data Size Select 0x0-0x2: Reserved 0x3: 4-bit data 0x4: 5-bit data ... 0xF: 16-bit data

Class Exercise 9.1



- Assume the system clock frequency is **16 MHz** and the content of **SSICPSR** register is **0x20**.
- Determine the content of **SSICR0** register for a bit rate of **100 Kbps** if we want the SPI with **8-bit** data size with clock polarity to be **1** and clock phase to be **0**.



24 Synchronous Serial Interface (SSI)

Introduction	467
API Functions	467
Programming Example	481

24.1 Introduction

The Synchronous Serial Interface (SSI) module provides the functionality for synchronous serial communications with peripheral devices, and can be configured to use either the Motorola® SPI™ or the Texas Instruments® synchronous serial interface frame formats. In addition, some devices also can be configured to use the National Semiconductor® Microwire format. The size of the data frame is also configurable, and can be set to be between 4 and 16 bits, inclusive.

The SSI module performs serial-to-parallel data conversion on data received from a peripheral device, and parallel-to-serial conversion on data transmitted to a peripheral device. The TX and RX paths are buffered with internal FIFOs, allowing up to eight 16-bit values to be stored independently.

The SSI module can be configured as either a master or a slave device. As a slave device, the SSI module can also be configured to disable its output, which allows a master device to be coupled with multiple slave devices.

The SSI module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the SSI module's input clock. Some Tiva devices can use the PIOC as the serial bit clock. Bit rates are generated based on the input clock and the maximum bit rate supported by the connected peripheral.

For parts that include a DMA controller, the SSI module also provides a DMA interface to facilitate data transfer via DMA.

This driver is contained in `driverlib/ssi.c`, with `driverlib/ssi.h` containing the API declarations for use by applications.

24.2 API Functions

Functions

- void [SSIAAdvDataPutFrameEnd](#) (uint32_t ui32Base, uint32_t ui32Data)
- int32_t [SSIAAdvDataPutFrameEndNonBlocking](#) (uint32_t ui32Base, uint32_t ui32Data)
- void [SSIAAdvFrameHoldDisable](#) (uint32_t ui32Base)
- void [SSIAAdvFrameHoldEnable](#) (uint32_t ui32Base)
- void [SSIAAdvModeSet](#) (uint32_t ui32Base, uint32_t ui32Mode)
- bool [SSIBusy](#) (uint32_t ui32Base)
- uint32_t [SSIClockSourceGet](#) (uint32_t ui32Base)
- void [SSIClockSourceSet](#) (uint32_t ui32Base, uint32_t ui32Source)
- void [SSIConfigSetExpClk](#) (uint32_t ui32Base, uint32_t ui32SSIClk, uint32_t ui32Protocol, uint32_t ui32Mode, uint32_t ui32BitRate, uint32_t ui32DataWidth)
- void [SSIDataGet](#) (uint32_t ui32Base, uint32_t *pui32Data)

- int32_t [SSIDataGetNonBlocking](#) (uint32_t ui32Base, uint32_t *pui32Data)
- void [SSIDataPut](#) (uint32_t ui32Base, uint32_t ui32Data)
- int32_t [SSIDataPutNonBlocking](#) (uint32_t ui32Base, uint32_t ui32Data)
- void [SSIDisable](#) (uint32_t ui32Base)
- void [SSIDMADisable](#) (uint32_t ui32Base, uint32_t ui32DMAFlags)
- void [SSIDMAEnable](#) (uint32_t ui32Base, uint32_t ui32DMAFlags)
- void [SSIEnable](#) (uint32_t ui32Base)
- void [SSIIntClear](#) (uint32_t ui32Base, uint32_t ui32IntFlags)
- void [SSIIntDisable](#) (uint32_t ui32Base, uint32_t ui32IntFlags)
- void [SSIIntEnable](#) (uint32_t ui32Base, uint32_t ui32IntFlags)
- void [SSIIntRegister](#) (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t [SSIIntStatus](#) (uint32_t ui32Base, bool bMasked)
- void [SSIIntUnregister](#) (uint32_t ui32Base)
- void [SSILoopbackDisable](#) (uint32_t ui32Base)
- void [SSILoopbackEnable](#) (uint32_t ui32Base)

24.2.1 Detailed Description

The SSI API is broken into several groups of functions. Each of those groups is addressed below.

The configuration of the SSI module is managed by the [SSIConfigSetExpClk\(\)](#) function, while state is managed by the [SSIEnable\(\)](#) and [SSIDisable\(\)](#) functions. The DMA interface is enabled or disabled by the [SSIDMAEnable\(\)](#) and [SSIDMADisable\(\)](#) functions. The SSI baud clock is managed by the [SSIClockSourceGet\(\)](#) and [SSIClockSourceSet\(\)](#) functions.

Data handling is performed by the [SSIDataPut\(\)](#), [SSIDataPutNonBlocking\(\)](#), [SSIDataGet\(\)](#), and [SSIDataGetNonBlocking\(\)](#) functions.

Interrupts from the SSI module are managed using the [SSIIntClear\(\)](#), [SSIIntDisable\(\)](#), [SSIIntEnable\(\)](#), [SSIIntRegister\(\)](#), [SSIIntStatus\(\)](#), and [SSIIntUnregister\(\)](#) functions.

The [SSIConfig\(\)](#), [SSIDataNonBlockingGet\(\)](#), and [SSIDataNonBlockingPut\(\)](#) APIs from previous versions of the peripheral driver library have been replaced by the [SSIConfigSetExpClk\(\)](#), [SSIDataGetNonBlocking\(\)](#), and [SSIDataPutNonBlocking\(\)](#) APIs. Macros have been provided in `ssi.h` to map the old APIs to the new APIs, allowing existing applications to link and run with the new APIs. It is recommended that new applications utilize the new APIs in favor of the old ones.

24.2.2 Function Documentation

24.2.2.1 SSIAAdvDataPutFrameEnd

Puts a data element into the SSI transmit FIFO as the end of a frame.

Prototype:

```
void  
SSIAAdvDataPutFrameEnd(uint32_t ui32Base,  
                        uint32_t ui32Data)
```

Parameters:

ui32Base specifies the SSI module base address.

Programming Example



- It shows how to **configure** the SSI module as a master device, and how to do a simple **send** of data.

```
// Enable the SSI0 peripheral
SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
// Wait for the SSI0 module to be ready.
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_SSI0)){
// Configure the SSI
SSISetExpClk(SSIO_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE0,
SSI_MODE_MASTER, 2000000, 8);
// Enable the SSI module.
SSIEnable(SSIO_BASE);

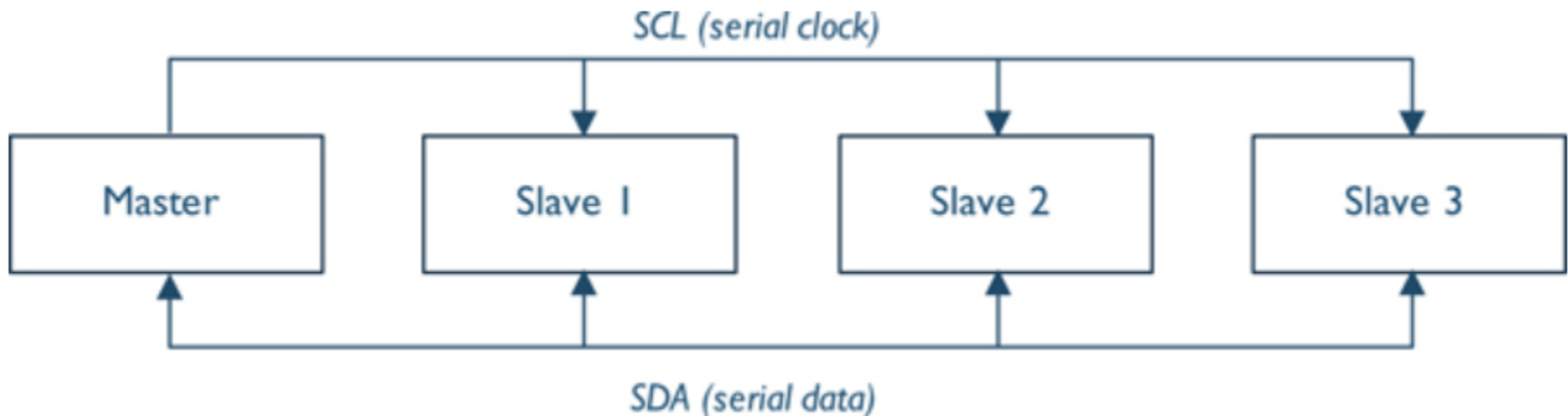
// Send some data.
char *pcChars = "SSI Master send data.";
int32_t i32Idx = 0;
while(pcChars[i32Idx]) {
    SSIDataPut(SSIO_BASE, pcChars[i32Idx]);
    i32Idx++;
}
```



- **Serial Peripheral Interface (SPI)**
 - Logic Signals & Data Transmission
 - Clocking: Polarity and Phase
 - SPI on Tiva™ & in TivaWare™ Library
- **Inter-Integrated Circuit Bus (I²C)**
 - Message Format
 - Start and Stop Conditions, Data Validity, and Acknowledgement
 - I²C on on Tiva™ & in TivaWare™ Library
- **Bluetooth (Notes)**
 - Serial Port Profile (SPP)

Inter-Integrated Circuit Bus (I²C or I2C)

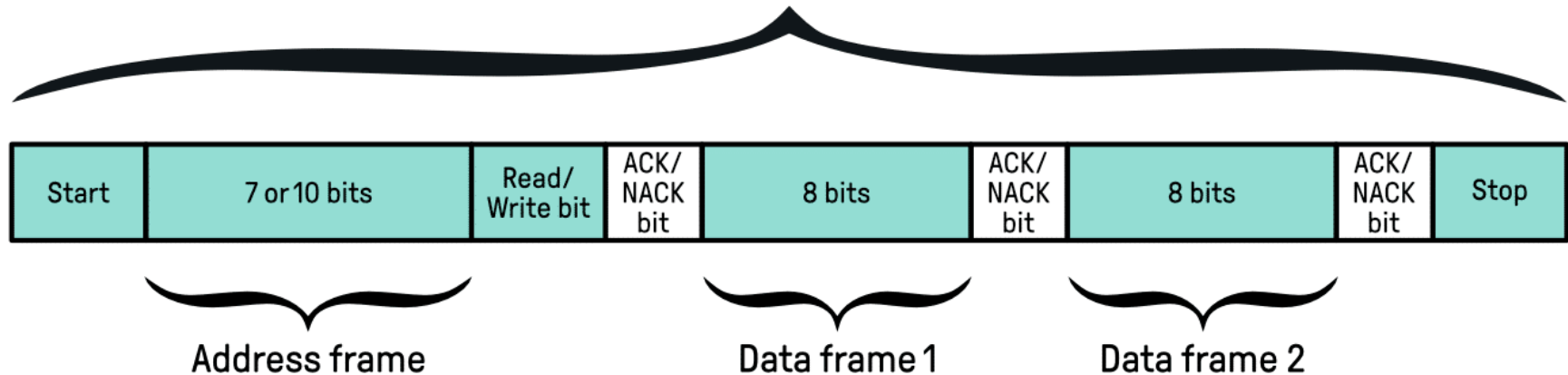
- I²C is a **synchronous** serial communication with only a serial data (SDA) and a serial clock (SCL) lines.
- I²C is a **master-slave** protocol.
 - The **master** initiates all communications (usually **MSB first**);
 - The **slave** devices transmit only when the master allows them to (by **device addressing**).
 - Each message includes **device addressing information**, and only the addressed device will respond to a message.



I²C Message Format

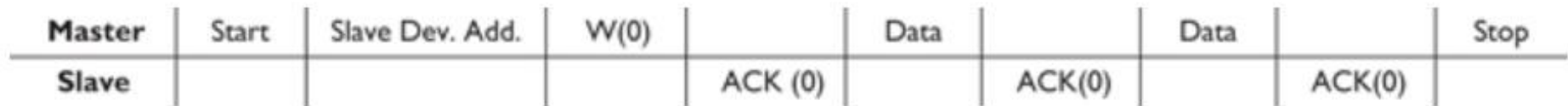


Message



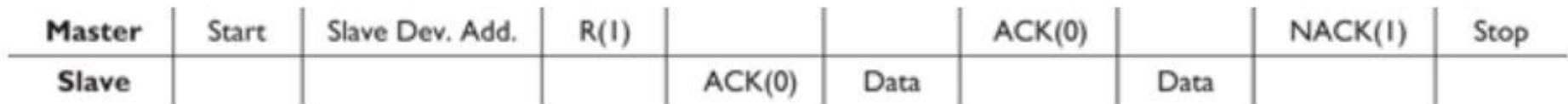
- The **start condition** indicates the start of a message.
- The **address frame** identifies the target of the communication.
- The **read/write bit** indicates whether the following data is to be read/written.
- The **acknowledgment bit** has two uses: ① to indicate if the addressed slave is present, and ② whether more data will be read.
- **One or more data frames** (bytes). (*Note: For some I²C slave devices, the first data byte will be interpreted as a register address.*)
- A **stop condition** indicates the end of a message.
- *An optional **repeated start condition** is used in some types of messages.

Ex1: Master writing two bytes



- The master first sends the **start condition**, the **slave device address**, and a **write command** (zero).
 - If the addressed slave is present, it will assert the **ACK** bit.
 - If the **ACK** bit is not asserted, then the master will terminate the message with a stop condition.
- The master sends the **first byte of data**.
- The slave sends an **ACK** to indicate it has been received.
- The master sends the **second byte of data**.
- The slave sends an **ACK** to indicate it has been received.
- The master sends the **stop condition**, indicating to all slaves that the message has completed.

Ex2: Master reading two bytes

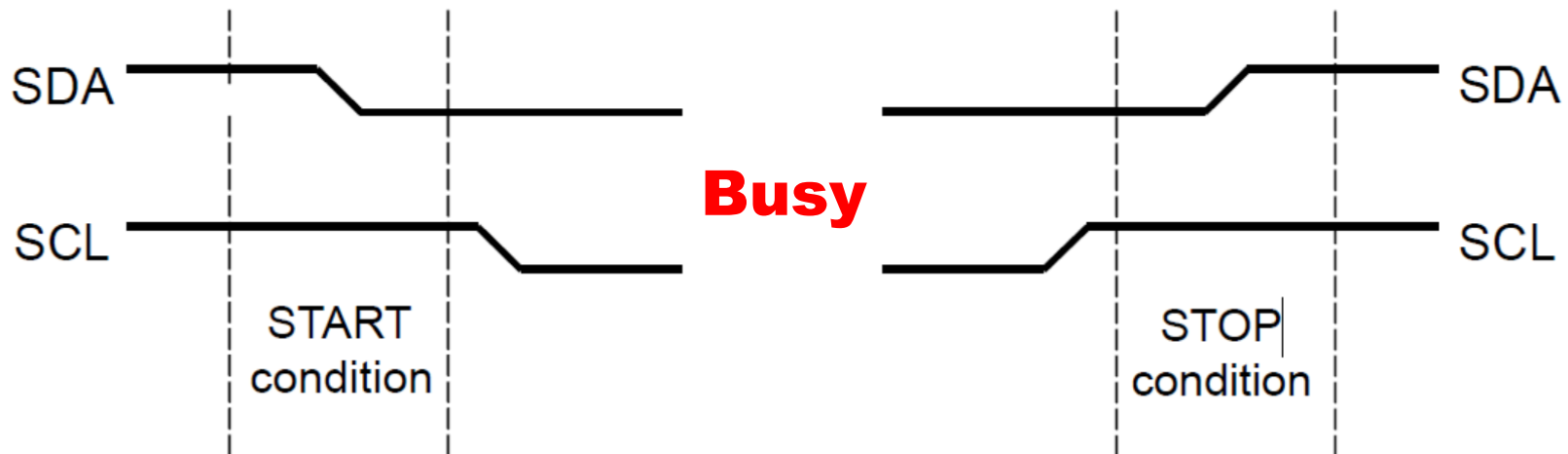


- The master first sends the **start condition**, the **slave device address**, and a **read command** (one).
 - If the addressed slave is present, it will assert the **ACK** bit.
 - If the **ACK** bit is not asserted, then the master will terminate the message with a stop condition.
- The master clocks the **first byte of data** out of the slave.
- The master sends an **ACK** to indicate it will read more data.
- The master clocks the **second byte of data** out of the slave.
- The master sends a **NACK** (one) to indicate that it does not want to read any more data in this message.
- The master sends the **stop condition**, indicating to all slaves that the message has completed.

I²C Start and Stop Conditions



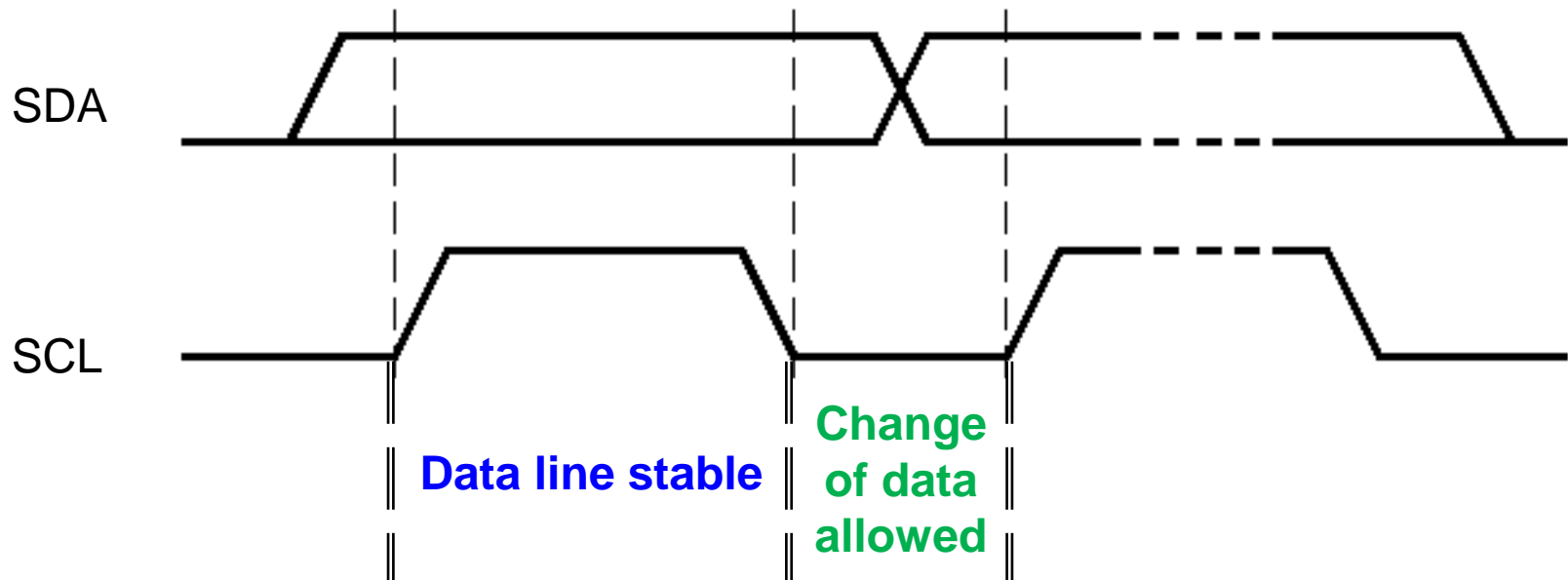
- **START Condition:** a High-to-Low transition on the SDA line while the SCL is High.
- **STOP Condition:** a Stop-to-Low transition on the SDA line while the SCL is High.
- The bus is considered **busy** after a START condition and free after a STOP condition.



I²C Data Validity



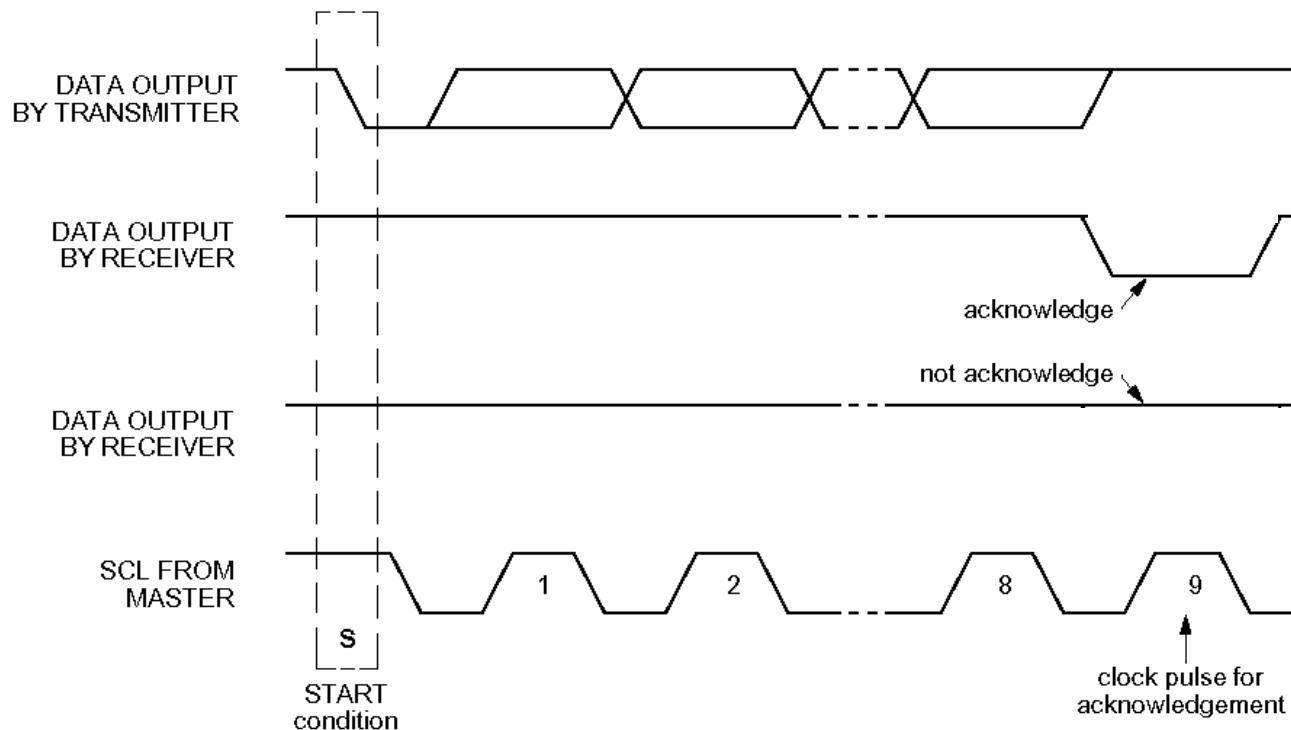
- The data on the **SDA** line **must be stable** during the high period of the clock.
- The data line can only **change when SCL is Low**.



I²C (Negative) Acknowledgement



- The **transmitter** (which can be the master or slave) **releases the SDA line**.
- The **receiver** **pulls down** the **SDA line** (or **leaves high**) to **acknowledge** (or **not acknowledge**) the transaction.
 - The acknowledgement must follow the **data validity**.



Transmitter **releases SDA line**.

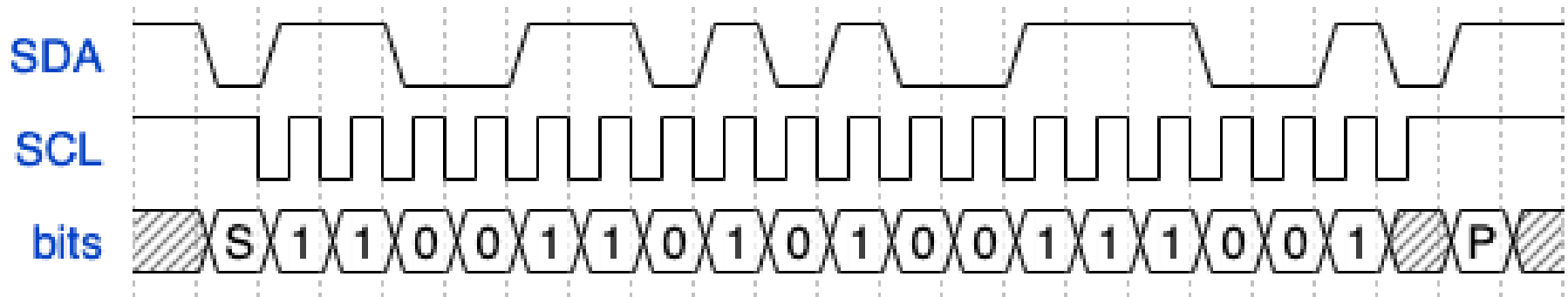
Receiver pulls down SDA to **ack**.

Receiver leaves SDA high to **not ack**.

Class Exercise 9.2

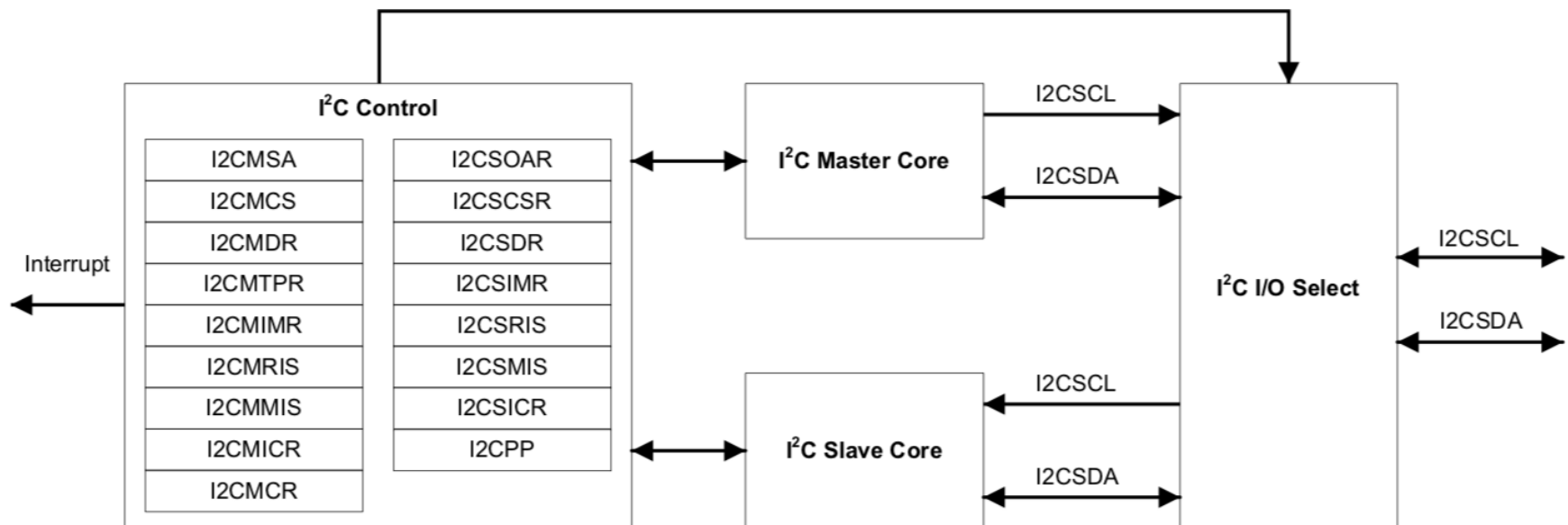


- Suppose the address is of 7 bits, and the addressed slave is present with acknowledgement.
- Given the waveforms of **SDA** and **SCL** lines of an **I²C master**, determine what does it intend to do?





- Tiva™ has I²C modules with the following features:
 - Devices on the I²C bus can be designated as either a **master** or a **slave**;
 - Four I²C modes: master/slave X transmit/receive;
 - Four transmission speeds: 100/400 Kbps; 1/3.33 Mbps;
 - Etc.





16 Inter-Integrated Circuit (I2C)

Introduction	323
API Functions	324
Programming Example	351

16.1 Introduction

The Inter-Integrated Circuit (I2C) API provides a set of functions for using the Tiva I2C master and slave modules. Functions are provided to initialize the I2C modules, to send and receive data, obtain status, and to manage interrupts for the I2C modules.

The I2C master and slave modules provide the ability to communicate to other IC devices over an I2C bus. The I2C bus is specified to support devices that can both transmit and receive (write and read) data. Also, devices on the I2C bus can be designated as either a master or a slave. The Tiva I2C modules support both sending and receiving data as either a master or a slave, and also support the simultaneous operation as both a master and a slave. Finally, the Tiva I2C modules can operate at the following speeds: Standard (100 kbps), Fast (400 kbps), Fast plus (1 Mbps) and High Speed (3.33 Mbps).

Both the master and slave I2C modules can generate interrupts. The I2C master module generates interrupts when a transmit or receive operation is completed (or aborted due to an error); and on some devices when a clock low timeout has occurred. The I2C slave module generates interrupts when data has been sent or requested by a master; and on some devices, when a START or STOP condition is present.

16.1.1 Master Operations

When using this API to drive the I2C master module, the user must first initialize the I2C master module with a call to `I2CMasterInitExpClk()`. That function sets the bus speed and enables the master module.

The user may transmit or receive data after the successful initialization of the I2C master module. Data is transferred by first setting the slave address using `I2CMasterSlaveAddrSet()`. That function is also used to define whether the transfer is a send (a write to the slave from the master) or a receive (a read from the slave by the master). Then, if connected to an I2C bus that has multiple masters, the Tiva I2C master must first call `I2CMasterBusBusy()` before attempting to initiate the desired transaction. After determining that the bus is not busy, if trying to send data, the user must call the `I2CMasterDataPut()` function. The transaction can then be initiated on the bus by calling the `I2CMasterControl()` function with any of the following commands:

- `I2C_MASTER_CMD_SINGLE_SEND`
- `I2C_MASTER_CMD_SINGLE_RECEIVE`
- `I2C_MASTER_CMD_BURST_SEND_START`
- `I2C_MASTER_CMD_BURST_RECEIVE_START`

Any of those commands results in the master arbitrating for the bus, driving the start sequence onto the bus, and sending the slave address and direction bit across the bus. The remainder of the transaction can then be driven using either a polling or interrupt-driven method.

For the single send and receive cases, the polling method involves looping on the return from `I2CMasterBusy()`. Once that function indicates that the I2C master is no longer busy, the bus transaction has been completed and can be checked for errors using `I2CMasterErr()`. If there are no errors, then the data has been sent or is ready to be read using `I2CMasterDataGet()`. For the burst send and receive cases, the polling method also involves calling the `I2CMasterControl()` function for each byte transmitted or received (using either the `I2C_MASTER_CMD_BURST_SEND_CONT` or `I2C_MASTER_CMD_BURST_RECEIVE_CONT` commands), and for the last byte sent or received (using either the `I2C_MASTER_CMD_BURST_SEND_FINISH` or `I2C_MASTER_CMD_BURST_RECEIVE_FINISH` commands). If any error is detected during the burst transfer, the `I2CMasterControl()` function should be called using the appropriate stop command (`I2C_MASTER_CMD_BURST_SEND_ERROR_STOP` or `I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP`).

For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C master interrupt; the interrupt occurs when the master is no longer busy.

16.1.2 Slave Operations

When using this API to drive the I2C slave module, the user must first initialize the I2C slave module with a call to `I2CSlaveInit()`. This function enables the I2C slave module and initializes the slave's own address. After the initialization is complete, the user may poll the slave status using `I2CSlaveStatus()` to determine if a master requested a send or receive operation. Depending on the type of operation requested, the user can call `I2CSlaveDataPut()` or `I2CSlaveDataGet()` to complete the transaction. Alternatively, the I2C slave can handle transactions using an interrupt handler registered with `I2CIntRegister()`, and by enabling the I2C slave interrupt.

This driver is contained in `driverlib/i2c.c`, with `driverlib/i2c.h` containing the API declarations for use by applications.

16.2 API Functions

Functions

- `uint32_t I2C_FIFODataGet (uint32_t ui32Base)`
- `uint32_t I2C_FIFODataGetNonBlocking (uint32_t ui32Base, uint8_t *pui8Data)`
- `void I2C_FIFODataPut (uint32_t ui32Base, uint8_t ui8Data)`
- `uint32_t I2C_FIFODataPutNonBlocking (uint32_t ui32Base, uint8_t ui8Data)`
- `uint32_t I2C_FIFOStatus (uint32_t ui32Base)`
- `void I2CIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))`
- `void I2CIntUnregister (uint32_t ui32Base)`
- `void I2CLoopbackEnable (uint32_t ui32Base)`
- `uint32_t I2CMasterBurstCountGet (uint32_t ui32Base)`
- `void I2CMasterBurstLengthSet (uint32_t ui32Base, uint8_t ui8Length)`
- `bool I2CMasterBusBusy (uint32_t ui32Base)`
- `bool I2CMasterBusy (uint32_t ui32Base)`
- `void I2CMasterControl (uint32_t ui32Base, uint32_t ui32Cmd)`
- `uint32_t I2CMasterDataGet (uint32_t ui32Base)`

Demo Code: mpu_i2c_demo.c (1/2)



```
double accelX = 0, accelY = 0, accelZ = 0;
double gyroX = 0, gyroY = 0, gyroZ = 0;
static uint8_t MPU6050_Buf_14_uint8[14];

#define MPU6050_ADDR 0x68
#define DATA_REG_ADDR 0x3B
#define PWR_MGMT_1 0x6B
#define CONFIG_ADDR 0x1B

int main(void) {
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);
    Initialization();

    while (1) {
        // Read raw data from MPU6050 and store in Buffer
        I2C_Read_bytes(MPU6050_ADDR, DATA_REG_ADDR, 14, MPU6050_Buf_14_uint8);

        // Extract data from Buffer and store in the following variables
        accelX = (MPU6050_Buf_14_uint8[0] << 8) | MPU6050_Buf_14_uint8[1];
        accelY = (MPU6050_Buf_14_uint8[2] << 8) | MPU6050_Buf_14_uint8[3];
        accelZ = (MPU6050_Buf_14_uint8[4] << 8) | MPU6050_Buf_14_uint8[5];
        gyroX = (MPU6050_Buf_14_uint8[8] << 8) | MPU6050_Buf_14_uint8[9];
        gyroY = (MPU6050_Buf_14_uint8[10] << 8) | MPU6050_Buf_14_uint8[11];
        gyroZ = (MPU6050_Buf_14_uint8[12] << 8) | MPU6050_Buf_14_uint8[13];

        // Adjust the delay as necessary
        SysCtlDelay(SysCtlClockGet()/3000) ;
    }
}
```



Demo Code: mpu_i2c_demo.c (2/2)



```
void I2C_Read_bytes(uint8_t dev_addr, uint8_t reg_addr, uint8_t num, uint8_t *data)
{
    uint8_t count = 0; // Initialize the counter
    // Set the I2C slave address for writing (false indicates a write operation)
    I2CMasterSlaveAddrSet(I2C0_BASE, dev_addr, false);
    // Load the register address into the data register
    I2CMasterDataPut(I2C0_BASE, reg_addr);
    // Send a start signal to initiate a read operation (7 indicates a read request)
    I2CMasterControl(I2C0_BASE, 7);
    // Wait until the I2C bus is not busy
    while(I2CMasterBusy(I2C0_BASE));
    // Set the I2C slave address for reading (true indicates a read operation)
    I2CMasterSlaveAddrSet(I2C0_BASE, dev_addr, true);
    // Send a start signal to prepare for reading data (11 indicates a repeated start)
    I2CMasterControl(I2C0_BASE, 11);
    // Wait until the I2C bus is not busy
    while(I2CMasterBusy(I2C0_BASE));
    // Read the first byte of data
    data[0] = I2CMasterDataGet(I2C0_BASE);
    // Loop to read the middle bytes
    for (count = 1; count < num - 1; count++) {
        // Control to read the next byte (9 indicates read and acknowledge)
        I2CMasterControl(I2C0_BASE, 9);
        // Wait until the I2C bus is not busy
        while(I2CMasterBusy(I2C0_BASE));
        // Store the received byte in the data array
        data[count] = I2CMasterDataGet(I2C0_BASE);
    }
    // Send a stop signal to end the transmission (5 indicates send stop)
    I2CMasterControl(I2C0_BASE, 5);
    // Wait until the I2C bus is not busy
    while(I2CMasterBusy(I2C0_BASE));
    // Read the last byte of data without acknowledging (no more data expected)
    data[count] = I2CMasterDataGet(I2C0_BASE);
}
```



- **Serial Peripheral Interface (SPI)**
 - Logic Signals & Data Transmission
 - Clocking: Polarity and Phase
 - SPI on Tiva™ & in TivaWare™ Library
- **Inter-Integrated Circuit Bus (I²C)**
 - Message Format
 - Start and Stop Conditions, Data Validity, and Acknowledgement
 - I²C on on Tiva™ & in TivaWare™ Library
- **Bluetooth (Notes)**
 - Serial Port Profile (SPP)

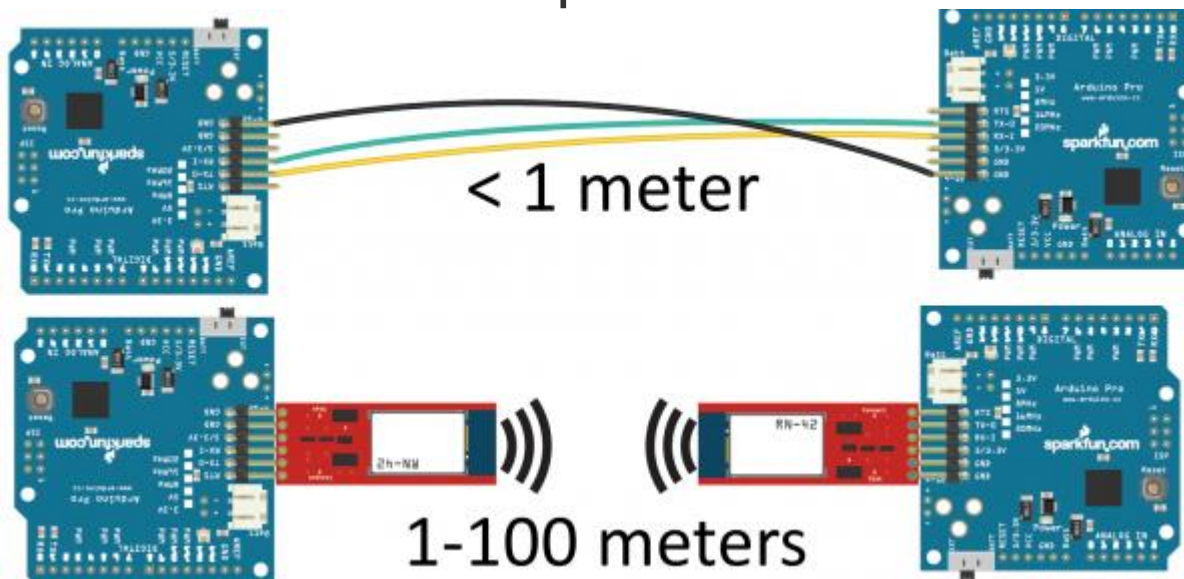
-  **Bluetooth**[®] is a **short-range wireless** technology standard that is used for exchanging data between fixed and mobile devices over short distances.
- To use it, a device must be compatible with the subset of **Bluetooth profiles**:

Profile Name	Features	Applications
A/V Remote Control Profile (AVRCP)	Remote control of A/V equipment	A/V equipment, in-vehicle equipment, personal computers, mobile phones, and headphones
Advanced Audio Distribution Profile (A2DP)	Streaming music	Headphones, A/V equipment, in-vehicle equipment, personal computers, and mobile phones
Basic Imaging Profile (BIP)	Transmission of basic image data	Printers, digital cameras, personal computers, and mobile phones
Basic Printing Profile (BPP)	Connection between a device (without a printing function) and a printer	Mobile phones, PDAs, and printers
Dial-up Networking Profile (DUN)	Dial-up internet connection via a mobile phone	Mobile phones, personal computers, and PDAs
File Transfer Profile (FTP)	Data transfer between personal computers	Personal computers and mobile phones
...
Serial Port Profile (SPP)	Using a Bluetooth device as a virtual serial port	Mobile phones and personal computers

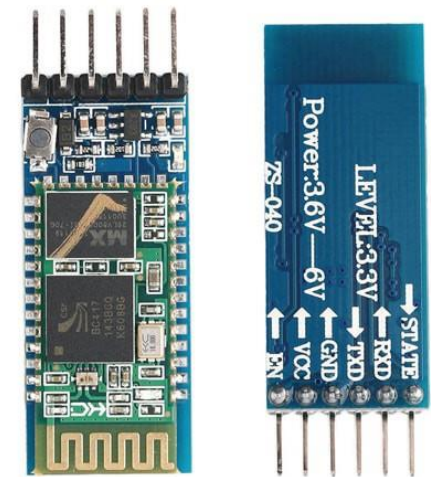
Serial Port Profile (SPP)



- **SSP** is typically used for replacing a serial communication interface (like **UART**) with Bluetooth.
 - With **SPP**, each connected device can send/receive data just as if there were **RX/TX** lines connected between them.
 - **HC-05** module is an easy to use Bluetooth SPP (Serial Port Protocol) module, designed for transparent wireless serial connection setup.



HC-05



- **Serial Peripheral Interface (SPI)**
 - Logic Signals & Data Transmission
 - Clocking: Polarity and Phase
 - SPI on Tiva™ & in TivaWare™ Library
- **Inter-Integrated Circuit Bus (I²C)**
 - Message Format
 - Start and Stop Conditions, Data Validity, and Acknowledgement
 - I²C on on Tiva™ & in TivaWare™ Library
- **Bluetooth (Notes)**
 - Serial Port Profile (SPP)

UART vs. SPI vs. I2C



Protocol	UART	SPI	I2C
Complexity	Simple	Complex as device increases	Easy to chain multiple devices
Speed	Slowest	Fastest	Faster than UART
Number of devices	Up to 2 devices	Many, but gets complex	Up to 127, but gets complex
Number of wires	2	4	2
Duplex	Full Duplex	Full Duplex	Half Duplex
No. of masters and slaves	Single to Single	1 master, multiple slaves	Multiple slaves and masters