

EE 3171 Lecture 16

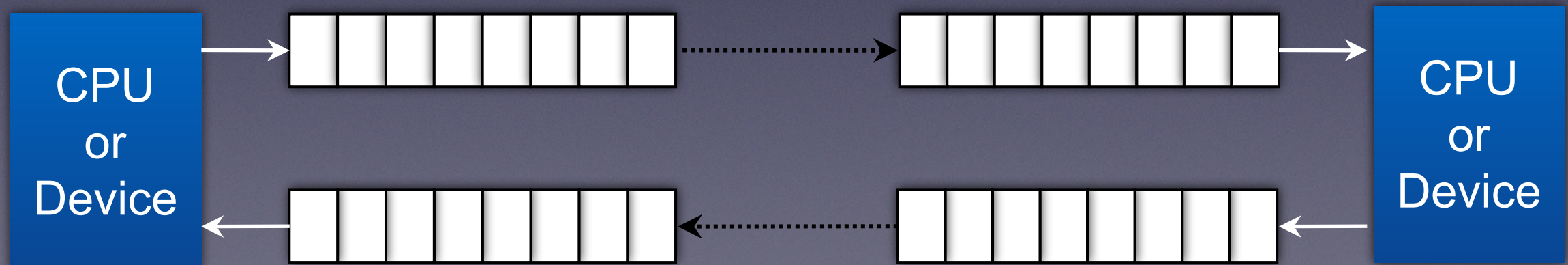
Serial Communications Part 2:
The Serial Peripheral Interface

The Topics...

- Serial Peripheral Interface (SPI)
 - Generic architecture & operation
 - We won't worry so much about the general serial protocol, since we've covered that.
 - Just a bit about the Tiva's implementation

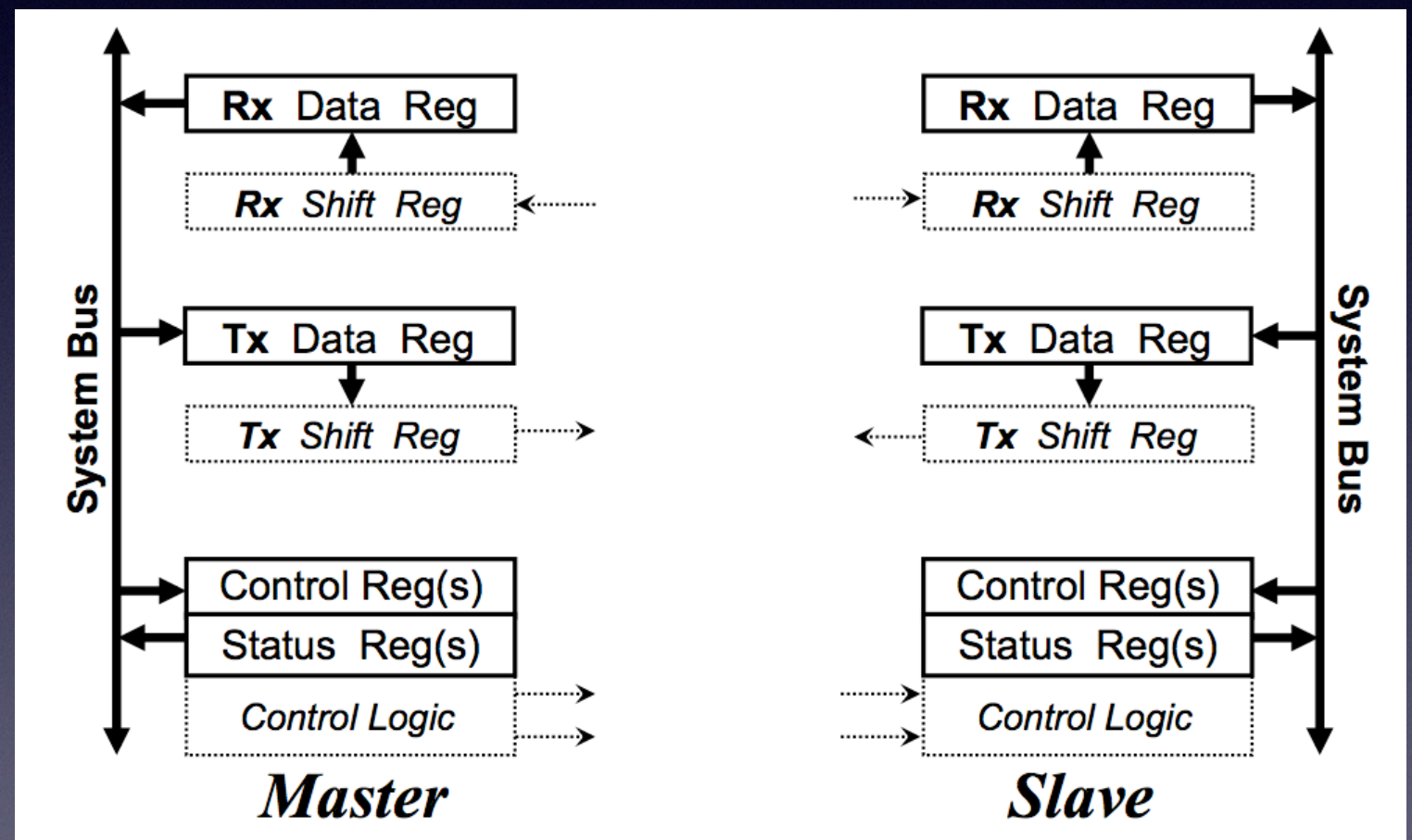
SPI - Introduction

- SPI ≡ Serial Peripheral Interface
 - Industrial Standard for microcontroller I/O
- Very Simple Protocol:
 - Full Duplex
 - Synchronous
 - Master-Slave Protocol
- Conceptual Model
 - Serial data loop between master & slave
 - Implemented with shift registers



SPI - Generic Implementation

- Solid Objects are visible to processor
- Thick Lines are parallel (up-to bus width)



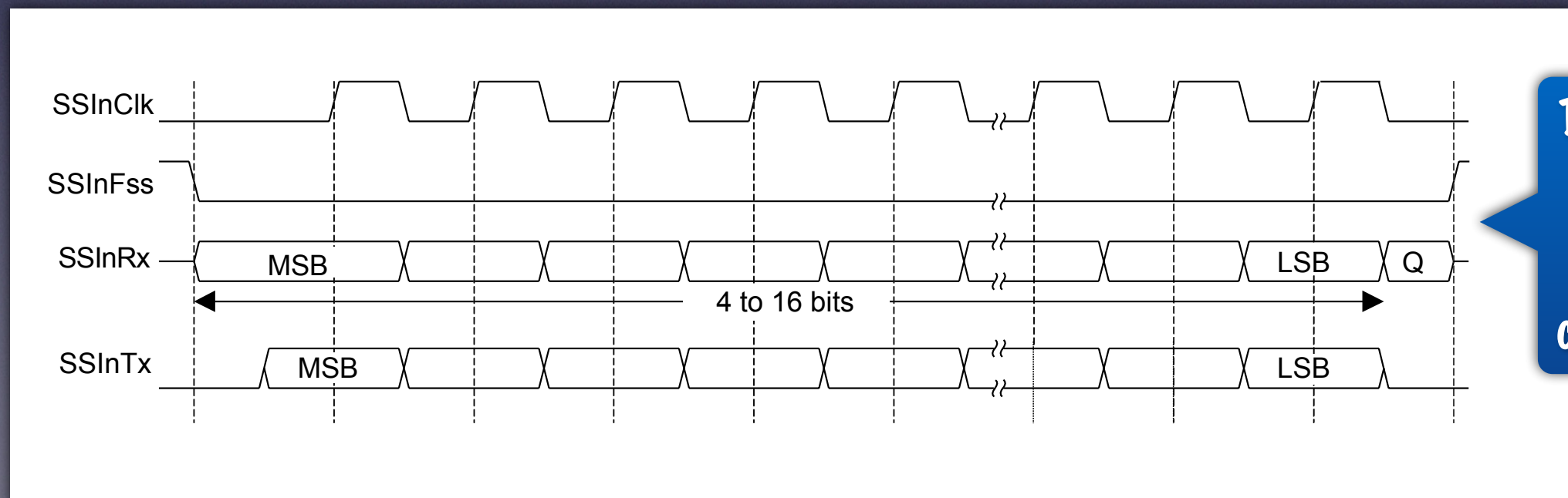
It turns out that the Tiva has a much deeper FIFO than just 1 word.

SPI Control Signals

- MISO = Master-In-Slave-Out
 - Master Rx shift reg input pin
 - Slave Tx shift reg output pin
- MOSI = Master-Out-Slave-In
 - Master Tx shift reg output pin
 - Slave Rx shift reg input pin
- SS = slave-select
 - Master can have multiple SS output pins (SS_0 ... SS_n)
- SCLK = serial clock
 - All Tx shift regs change output value (shift) on one edge
 - All Rx shift regs latch input value (shift) on the other edge

Sample Transfer

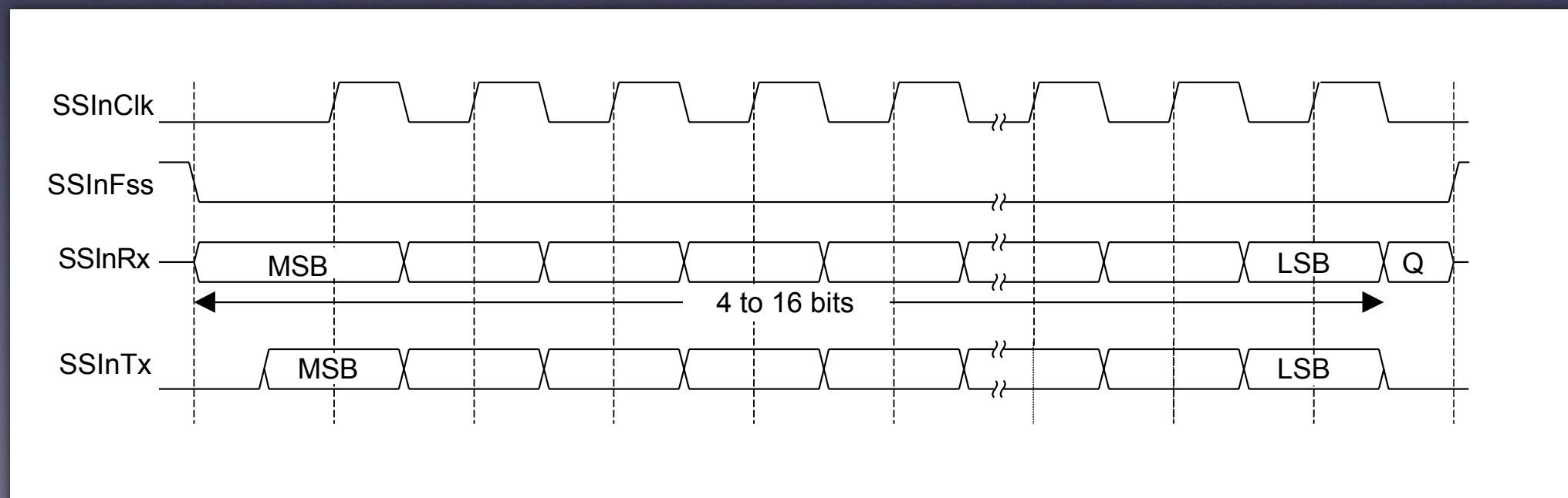
- In SPI, the most significant bit is sent first.
- Each bit is on the interface for one clock cycle.
- The slave select line is asserted first so the target device knows it's supposed to be receiving data.



This diagram
uses TI's
signal
naming
conventions.

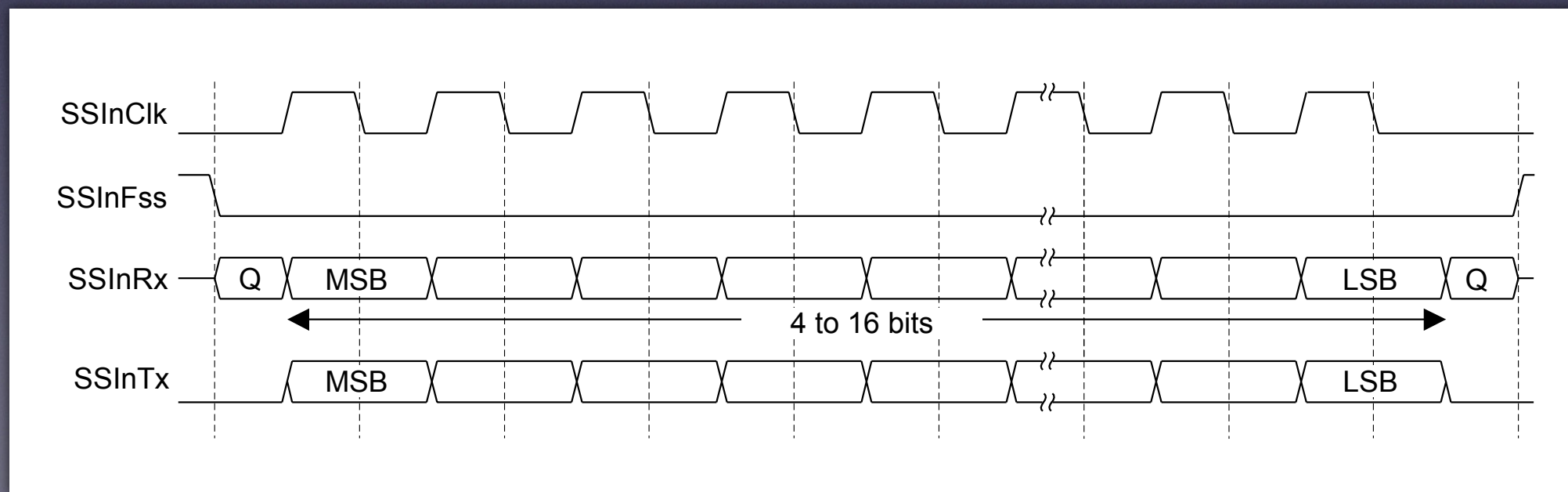
But Wait...

- In SPI, we worry about the clock *phase* and clock *polarity*.
- The image below is working in 0,0 mode. The idle state of the bus is 0 and we sample data on the 0th edge.
 - Or wait 0 edges before sampling data.
- It turns out this ends up being the same as operating in 1,1 mode.
 - The idle state is 1 and we wait 1 edge.
- And therefore, both sample on the *rising edge* of the clock.



But Wait...

- We can also operate in 0,1 and 1,0 mode.
- The image below is 0,1 mode. The clock idles at the 0 value and we wait 1 edge to sample data.
- Therefore, we sample data on the falling edge of the clock.
- There is no standard here — just read the documentation for the device you're trying to interface with.



Tiva C SSI

- Strictly speaking, the Tiva doesn't have an SPI module.
- Rather, it has a multipurpose SSI (Synchronous Serial Interface). The SSI is capable of interfacing with SPI buses, MICROWIRE buses or TI-proprietary buses.

Tiva C SSI

- Master or slave operation
- Programmable clock bit rate and prescaler
- Separate transmit and receive FIFOs, each 16 bits wide and 8 locations deep
- Programmable data frame size from 4 to 16 bits
- Standard FIFO-based interrupts and End-of-Transmission interrupt
- Efficient transfers using Micro Direct Memory Access Controller (μ DMA)

Tiva SPI Basics

- 1. Enable the SSI module using the **RCGCSSI** register.
- 2. Enable the clock to the appropriate GPIO module via the **RCGCGPIO** register.
- 3. Set the GPIO **AFSEL** bits for the appropriate pins.
- 4. Configure the **PMC_n** fields in the **GPIOCTL** register to assign the SSI signals to the appropriate pins.
- 5. Program the **GPIODEN** register to enable the pin's digital function. In addition, the drive strength, drain select and pull-up/pull-down functions must be configured.
- Now the SSI module is enabled. Time to configure it.

Tiva C Configuration

- 1. Ensure that the **SSE** bit in the **SSICR1** register is clear before making any configuration changes.
- 2. Select whether the SSI is a master or slave:
 - a. For master operations, set the **SSICR1** register to **0x0000.0000**.
 - b. For slave mode (output enabled), set the **SSICR1** register to **0x0000.0004**.
- 3. Configure the SSI clock source by writing to the **SSICC** register.
- 4. Configure the clock prescale divisor by writing the **SSICPSR** register.
- 5. Write the **SSICR0** register with the following configuration:
 - Serial clock rate (**SCR**)
 - Desired clock phase/polarity, if using Freescale SPI mode (**SPH** and **SP0**)
 - The protocol mode: Freescale SPI, TI SSF, MICROWIRE (**FRF**)
 - The data size (**DSS**)
- 6. Optionally, configure the SSI module for μ DMA use.
- 7. Enable the SSI by setting the **SSE** bit in the **SSICR1** register.

Using the SSI

- Basically, you transmit data by storing to the SSI Data Register (**SSIDR**) after checking the SSI Status Register (**SSISR**) to make sure that the transmit FIFO isn't full.
- You receive data in much the same way. Check to see if the receive FIFO isn't empty, then read some data from the **SSIDR**.
- The main flags tell us if:
 - The transmit/receive FIFOs are more than half full.
 - If an *overflow error* has occurred (receive FIFO only).

TivaWare SSI Functions

- The five most important are:
 - `SSIConfigSetExpClk()`
 - `SSIEnable()/SSIDisable()`
 - `SSIDataPut()/SSIDataGet()`
- Of course, there are other functions for enabling interrupts, clearing interrupts and all that good stuff too.

SSIConfigSetExpClk()

- Configures the synchronous serial interface.
- ```
void SSIConfigSetExpClk(uint32_t ui32Base,
 uint32_t ui32SSIClk,
 uint32_t ui32Protocol,
 uint32_t ui32Mode,
 uint32_t ui32BitRate,
 uint32_t ui32DataWidth)
```
- **Parameters:**
  - `ui32Base` specifies the SSI module base address.
  - `ui32SSIClk` is the rate of the clock supplied to the SSI module.
  - `ui32Protocol` specifies the data transfer protocol.
  - `ui32Mode` specifies the mode of operation.
  - `ui32BitRate` specifies the clock rate.
  - `ui32DataWidth` specifies number of bits transferred per frame.



# SSIConfigSetExpClk()

- The `ui32Protocol` parameter defines the data frame format and can be one of the following values:
  - `SSI_FRF_MOTO_MODE_0`,  
`SSI_FRF_MOTO_MODE_1`,  
`SSI_FRF_MOTO_MODE_2`,  
`SSI_FRF_MOTO_MODE_3`,  
`SSI_FRF_TI`, or  
`SSI_FRF_NMW`.

- The Motorola frame formats encode the following polarity and phase configurations:

| Polarity | Phase | Mode                             |
|----------|-------|----------------------------------|
| 0        | 0     | <code>SSI_FRF_MOTO_MODE_0</code> |
| 0        | 1     | <code>SSI_FRF_MOTO_MODE_1</code> |
| 1        | 0     | <code>SSI_FRF_MOTO_MODE_2</code> |
| 1        | 1     | <code>SSI_FRF_MOTO_MODE_3</code> |

- The `ui32Mode` parameter defines the operating mode of the SSI module and can be one of the following values: `SSI_MODE_MASTER`, `SSI_MODE_SLAVE`, or `SSI_MODE_SLAVE_OD`.



# SSIConfigSetExpClk()

- The **ui32BitRate** parameter defines the bit rate for the SSI. This bit rate must satisfy the following clock ratio criteria:
  - $F_{SSI} \geq 2 * \text{bit rate}$  (master mode); this speed cannot exceed 25 MHz.
  - $F_{SSI} \geq 12 * \text{bit rate}$  or  $6 * \text{bit rate}$  (slave modes), depending on the capability of the specific microcontroller
  - where  $F_{SSI}$  is the frequency of the clock supplied to the SSI module.
- The **ui32DataWidth** parameter defines the width of the data transfers and can be a value between 4 and 16 (inclusive).
- The peripheral clock is the same as the processor clock. This value is returned by **SysCtlClockGet()**, or it can be explicitly hard coded if it is constant and known.



# SSIEnable()

- Enables the synchronous serial interface.
- `void SSIEnable(uint32_t ui32Base)`
- Parameters:
  - `ui32Base` specifies the SSI module base address.
- This function enables operation of the synchronous serial interface. The synchronous serial interface must be configured before it is enabled.



# SSIDisable()

- Disables the synchronous serial interface.
- `void SSIDisable(uint32_t ui32Base)`
- Parameters:
  - `ui32Base` specifies the SSI module base address.
- This function disables operation of the synchronous serial interface.



# SSIDataGet()

- Gets a data element from the SSI receive FIFO.
- `void SSIDataGet(uint32_t ui32Base, uint32_t* pui32Data)`
- Parameters:
  - `ui32Base` specifies the SSI module base address.
  - `pui32Data` is a *pointer* to a storage location for data that was received over the SSI interface.
- This function gets received data from the receive FIFO of the specified SSI module and places that data into the location specified by the `pui32Data` parameter. If there is no data available, this function waits until data is received before returning.
- If you don't want to wait, there is a `SSIDataGetNonBlocking()`.



# SSIDataPut()

- Puts a data element into the SSI transmit FIFO.
- `void SSIDataPut(uint32_t ui32Base, uint32_t ui32Data)`
- Parameters:
  - `ui32Base` specifies the SSI module base address.
  - `ui32Data` is the data to be transmitted over the SSI interface.
- This function places the supplied data into the transmit FIFO of the specified SSI module. If there is no space available in the transmit FIFO, this function waits until there is space available before returning.
- If you don't want to wait, there is an `SSIDataPutNonBlocking()`.



# Simple Example

```
• char *pcChars = "SSI Master send data.";
 int32_t i32Idx;

 //
 // Configure the SSI.
 //
 SSIConfigSetExpClk(SSI_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE0,
 SSI_MODE_MASTER, 2000000, 8);

 //
 // Enable the SSI module.
 //
 SSIEnable(SSI_BASE);

 //
 // Send some data.
 //
 i32Idx = 0;
 while(pcChars[i32Idx])
 {
 SSIDataPut(SSI_BASE, pcChars[i32Idx]);
 i32Idx++;
 }
```



# Typical Master CPU Ops

- Issue SS to desired slave – by setting a bit in a Control Reg
- Store a “word” into the Transmit Data Register
  - Immediately downloads to Transmit Shift Register
  - Can then load next word into Transmit Data Register
- Start the SCLK – by setting a bit in a Control Register (may also happen automatically)
  - all shift regs shift one bit per cycle
  - slave is automatically sync’ed into master’s SCLK
- When one word is done – as indicated in status register
  - Read the word out of the Receive Data Register
  - When a complete word is received it automatically uploaded from Receive Shift Register to Receive Data Register
- Repeat until all words are done



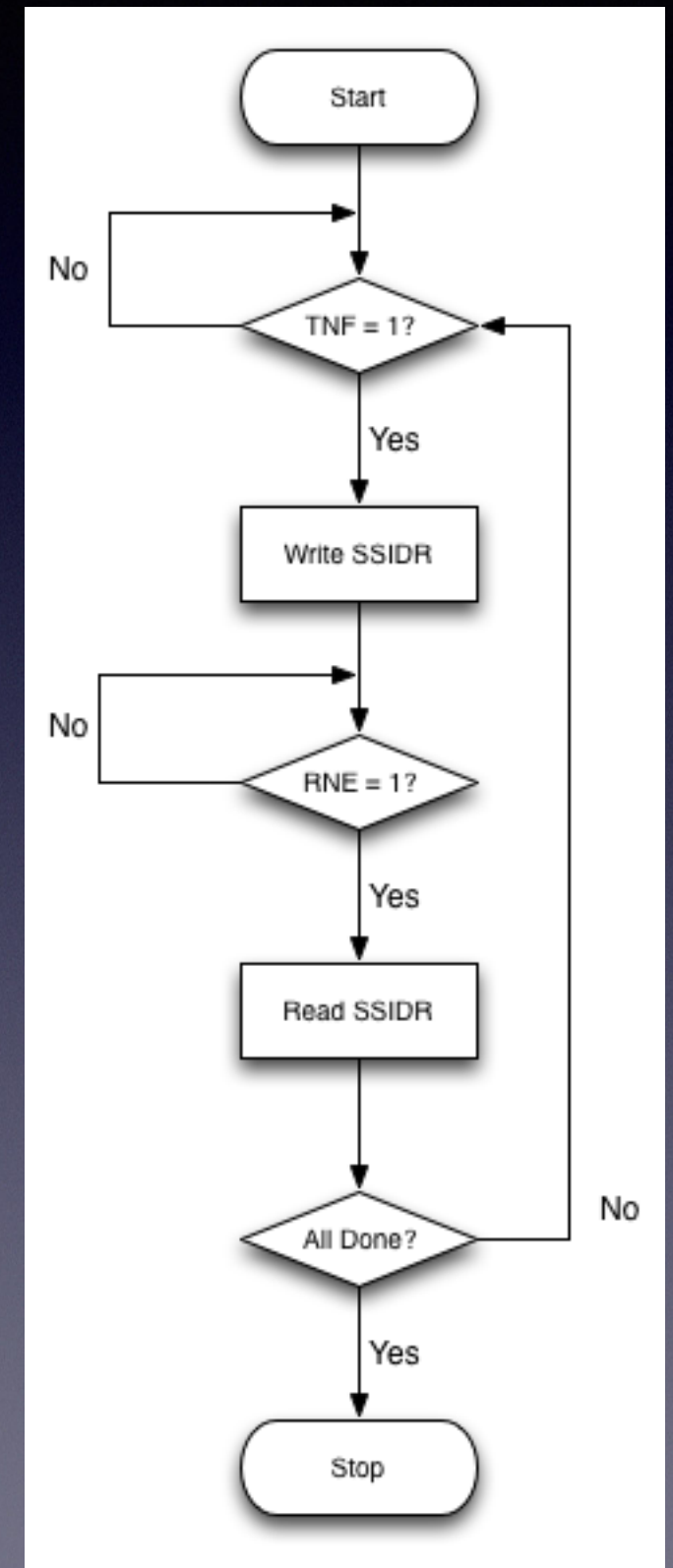
# SPI Service Loop Efficiency

- Remember, SPIs can be fast
  - Theoretically, all the way up to **SCLK** = bus clock
  - Usually bounded more by slave frequency limit, rather than protocol
- Timing issues:
  - CPU needs to read **SSIDR**
    - After word arrives (**RNE** = 1)
    - Before FIFO fills up (else **RORRIS** = 1)
  - CPU can write to **SSIDR**
    - After word loaded into shift reg (**TNF** = 1)
- CPU has to be fast enough, but not too fast
  - Let's look at a polled implementation



# Simple Full Duplex I/O Operation

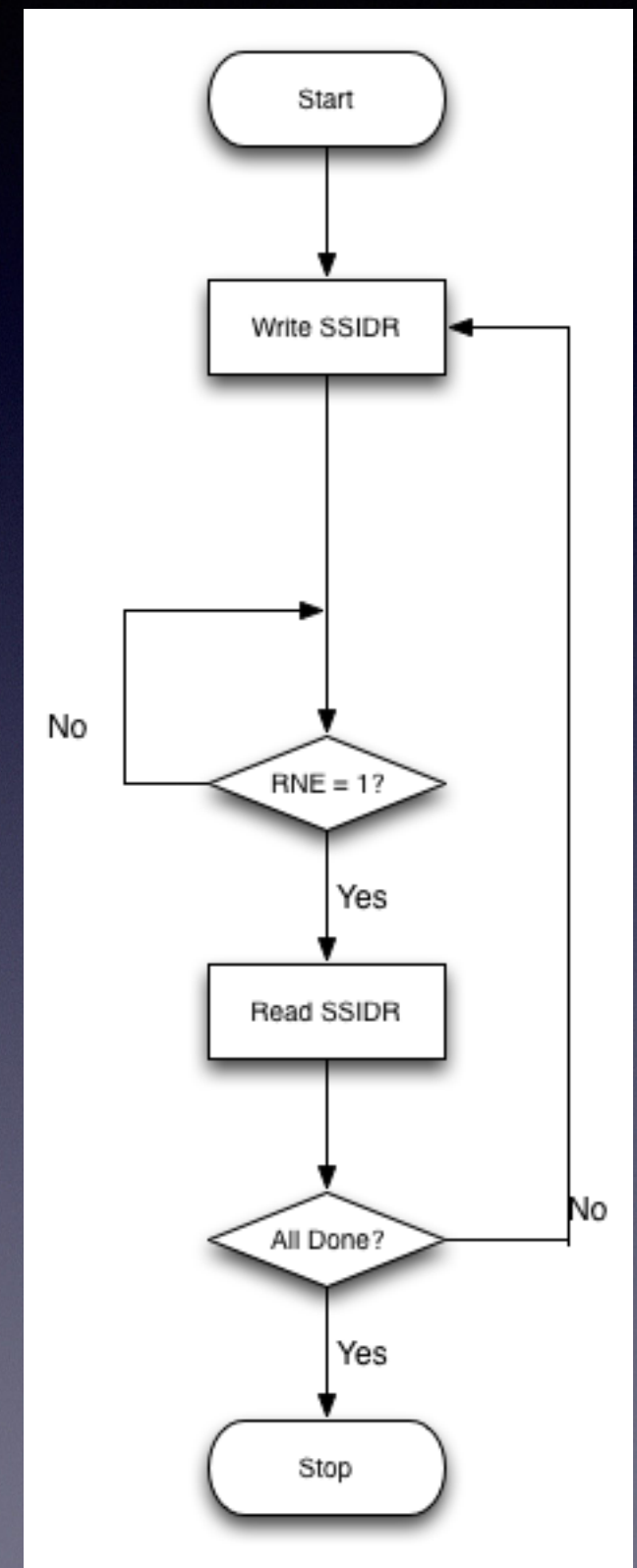
- Theoretically need to check:
  - TNF
  - RNE
- But:
  - We have a deep FIFO (shouldn't even think about it until it's at least half full)
  - RNEB set after TNF would go to a 0 (assuming we aren't constantly writing) and ... basically at the same time(  $\pm$  a clock cycle)!
- Do not need to check both flags (and if you're using TivaWare, don't check the flags at all).





# Better Full Duplex I/O Operation

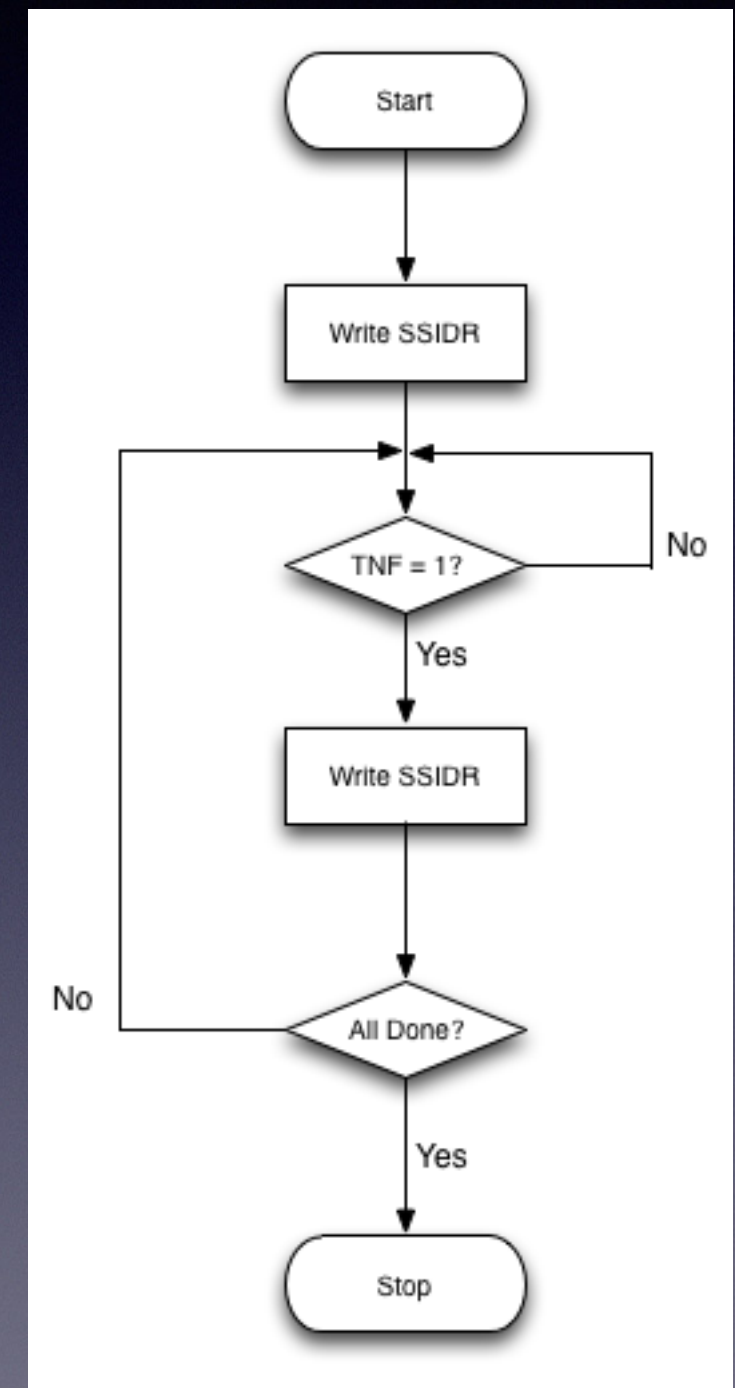
- Assume you can just write a word into SSIDR.
- Wait for a response, then go do it all again.
- Saves a lot of time and hardware instructions.
  - At least one load, a comparison and a branch!





# Simplex Output-Only Operation

- Write only – No Read
- No need to:
  - Check RNE
  - Read SSIDR register
- Will cause an ROR
  - Because SSIDR data is never read
- But who cares?
  - We're not saving the input data anyway
  - Just don't enable ROR IRQ bit





# Summary Slide

- SPI ≡ Serial Peripheral Interface
  - Very Simple Protocol:
    - Full Duplex
    - Synchronous
    - Master-Slave Protocol
  - Signals:
    - MISO
    - MOSI
    - SCLK
    - SS
  - Circular data flow
    - each transmit causes a receive