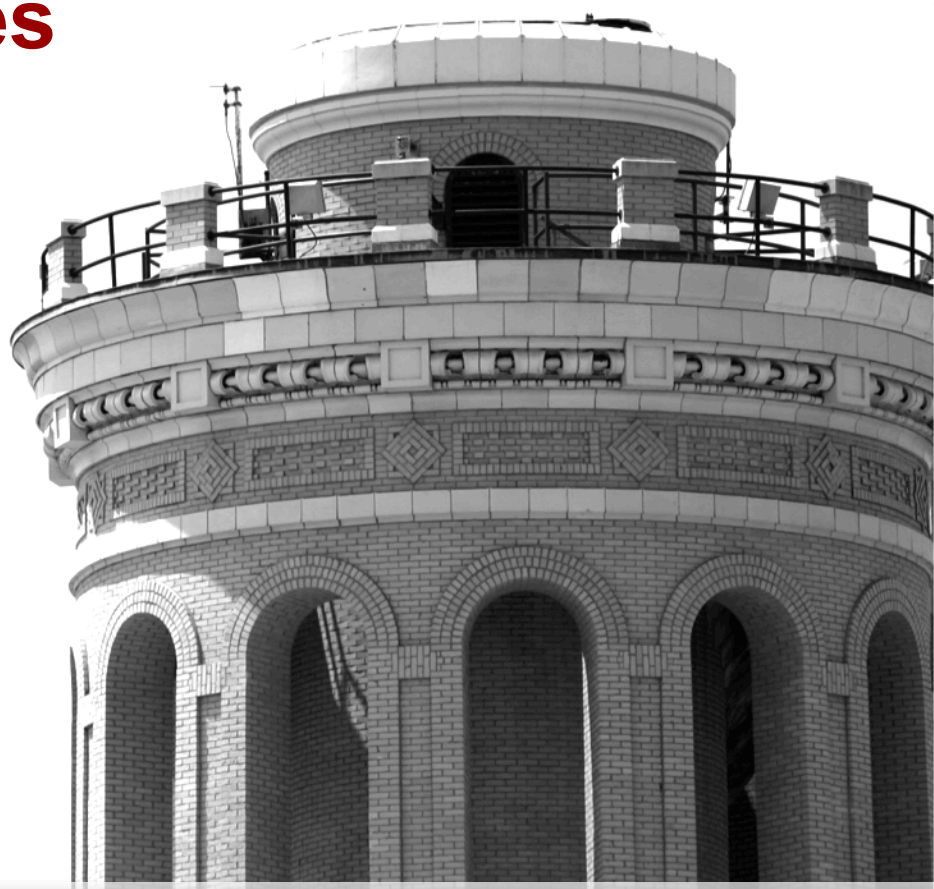
18-349: Introduction to Embedded Real-Time Systems

Lecture 5: Serial Buses

Anthony Rowe
Electrical and Computer Engineering
Carnegie Mellon University



Electrical & Computer ENGINEERING

Carnegie Mellon University

Last Lecture

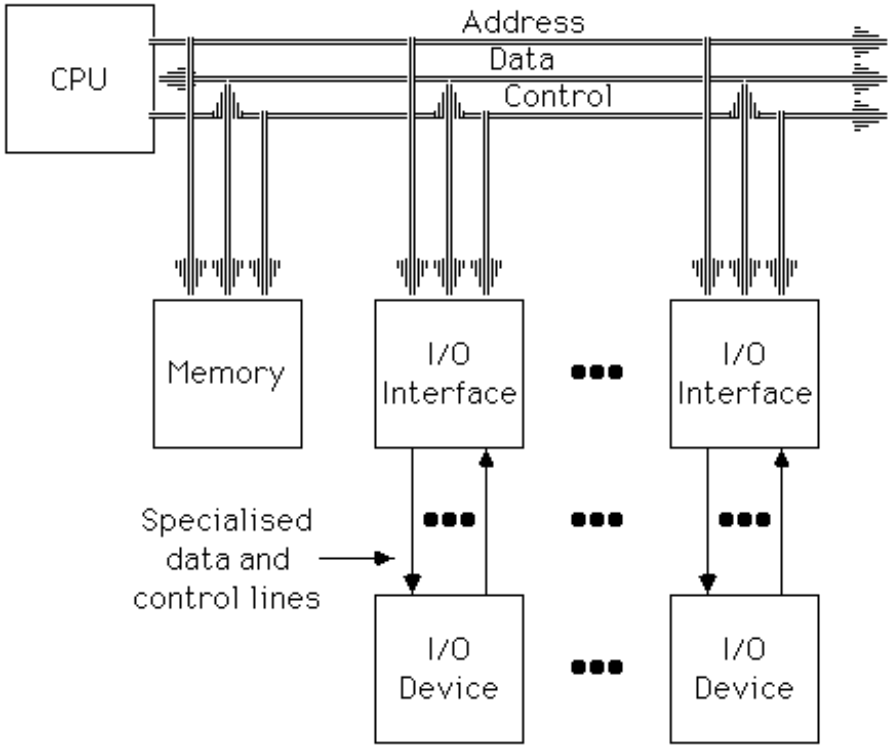
- ARM ASM Part 2
  - Addressing Modes
  - Batch load
  - Stack

- Memory Mapped Input Output (MMIO)

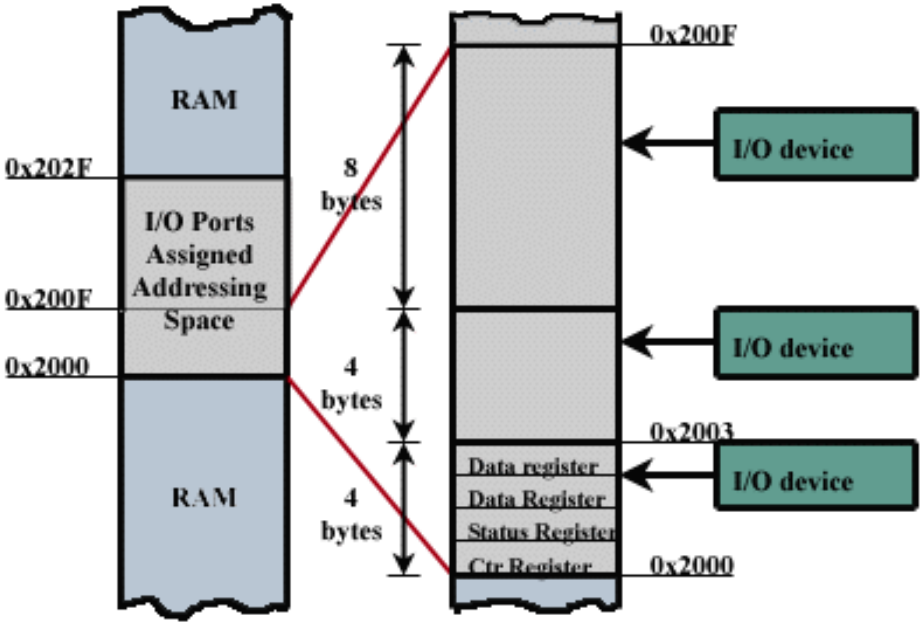
Lecture Overview

- Memory Mapped I/O (review)
  - Volatile
- Serial Communication
  - Asynchronous protocols
  - Synchronous protocols
  - RS-232 data interface
  - Parity bits
  - Serial and bit transmissions
  - SPI
  - I2C

Memory Mapped I/O



Physical Layout



Programmer's View

Writing Code to Access the Devices

- Portability issues – hard-coding the address may pose problems in moving to a new board where the address of the register is different

```
LDR   R0,=0x20200000
MOV   R1,#0x0C
STRB  R1,[R0]
```

- **Should** use EQU assembler directive: Equates a symbolic name (e.g., BASE) to a numeric value

```
BASE  EQU 0x20200000
LDR   R0, =BASE
```

- *Can* also access devices using C programs
  - C pointers can be used to write to a specific memory location

```
unsigned char *ptr;
ptr  = (unsigned char *) 0x20200000;
*ptr = (unsigned char) 0x0C;
```

I/O Register Basics

- I/O Registers are NOT like normal memory
  - Device events can change their values (e.g., status registers)
  - Reading a register can change its value (e.g., error condition reset)
    - For example, can't expect to get same value if read twice
  - Some are readonly (e.g., receive registers)
  - Some are writeonly (e.g., transmit registers)
  - Sometimes multiple I/O registers are mapped to same address
    - Selection of one based on other info (e.g., read vs. write or extra control bits)

- Cache must be disabled for memorymapped addresses – why?
- When polling I/O registers, should tell compiler that value can change on its own and therefore should not be stored in a register
  - `volatile int *ptr;` (or `int volatile *ptr;`)

Making the case for `volatile`

- Have you experienced any of the following in your C/C++ embedded code?
  - Code that works fine-until you turn optimization on
  - Code that works fine-as long as interrupts are disabled
  - Flaky hardware drivers
  - Tasks that work fine in isolation-yet crash when another task is enabled

- `volatile` is a qualifier that is applied to a variable when it is declared
- It tells the compiler that the value of the variable may change at any time---most importantly, even with no action being taken by the code that the compiler finds nearby

Syntax of `volatile`

- volatile variable

```
volatile int foo;
int volatile foo;
```

- pointer to a volatile variable

```
volatile int *foo;
int volatile *foo;
```

- volatile pointer to a non-volatile variable (very rare)

```
int * volatile foo;
```

- volatile pointer to a volatile variable (if you're crazy)

```
int volatile * volatile foo;
```

- If you apply `volatile` to a struct or union, the entire contents of the struct/union are volatile
  - If you don't want this behavior, you can apply the volatile qualifier to the individual members of the struct/union.

The Use of volatile (1)

- A variable should be declared volatile if its value could change unexpectedly
  - Memory-mapped I/O registers
  - Global variables that can be modified by an interrupt service routine
  - Global variables within multi-threaded applications

- Example: Let's poll an 8-bit I/O status register at 0x1234 until it is non-zero

```
unsigned int *ptr = (unsigned int *) 0x1234;
// wait for I/O register to become non-zero
while (*ptr == 0);
// do something else
```

What's wrong with this code? How would you fix it?

The Use of volatile (2)

- Example: Write an interrupt-service routine for a serial-port to test each character to see if it represents an EOL character. If it is, we will set a flag to be TRUE.

```
int eol_rcvd = FALSE;
void main() {    ...    while (!eol_rcvd)    {        // Wait    }    ... }

interrupt void rx_isr(void) {    ...    if (EOL == rx_char)    {
```

How might an optimizer handle this code? How would you fix it?

Thoughts on volatile

- What does the keyword volatile accomplish?
  - Tells the compiler not to perform certain optimizations
  - Tells the compiler not to use the cached version of the variable
  - Indicates that that variable can change asynchronously
- Some compilers allow you to declare everything as volatile
  - Don't! It's a substitute for good thinking
  - Can lead to less efficient code
- Don't blame the optimizer and don't turn it off
- If you are given a piece of code whose behavior is unpredictable
  - Look for declarations of volatile variables
  - Look for where you should declare a variable as volatile

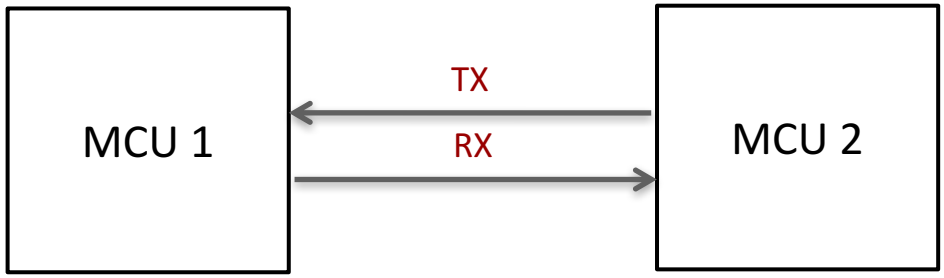
Why Serial Communication?

- Serial communication is a **pin-efficient** way of sending and receiving bits of data
  - Sends and receives data one bit at a time over one wire
  - While it takes eight times as long to transfer each byte of data this way (as compared to parallel communication), only a few wires are required
  - Typically one to send, one to receive (for full-duplex), and a common signal ground wire

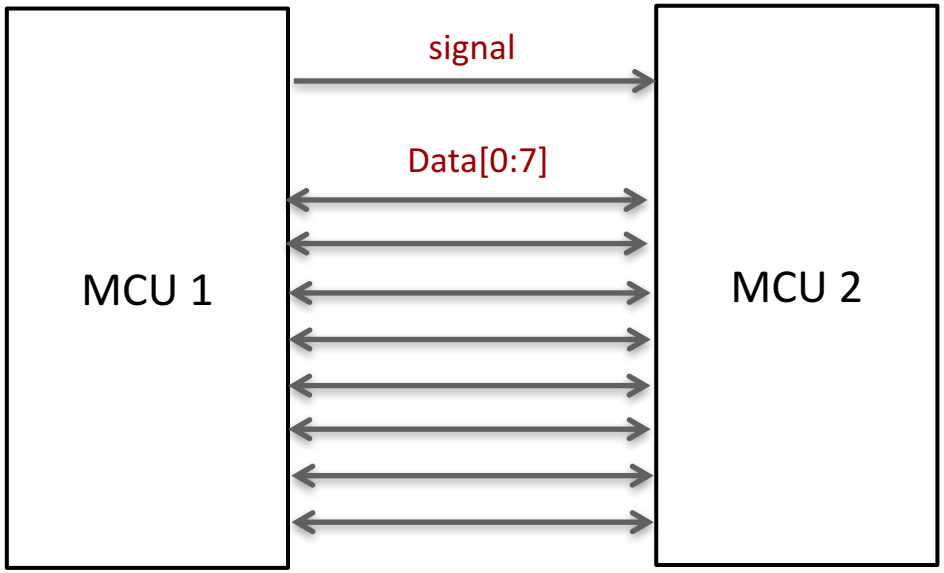
- **Simplistic** way to visualize serial port
  - Two 8-bit shift registers connected together
  - Output of one shift register (transmitter) connected to the input of the other shift register (receiver)
  - Common clock so that as a bit exits the transmitting shift register, the bit enters the receiving shift register
  - Data rate depends on clock frequency

Serial vs. Parallel

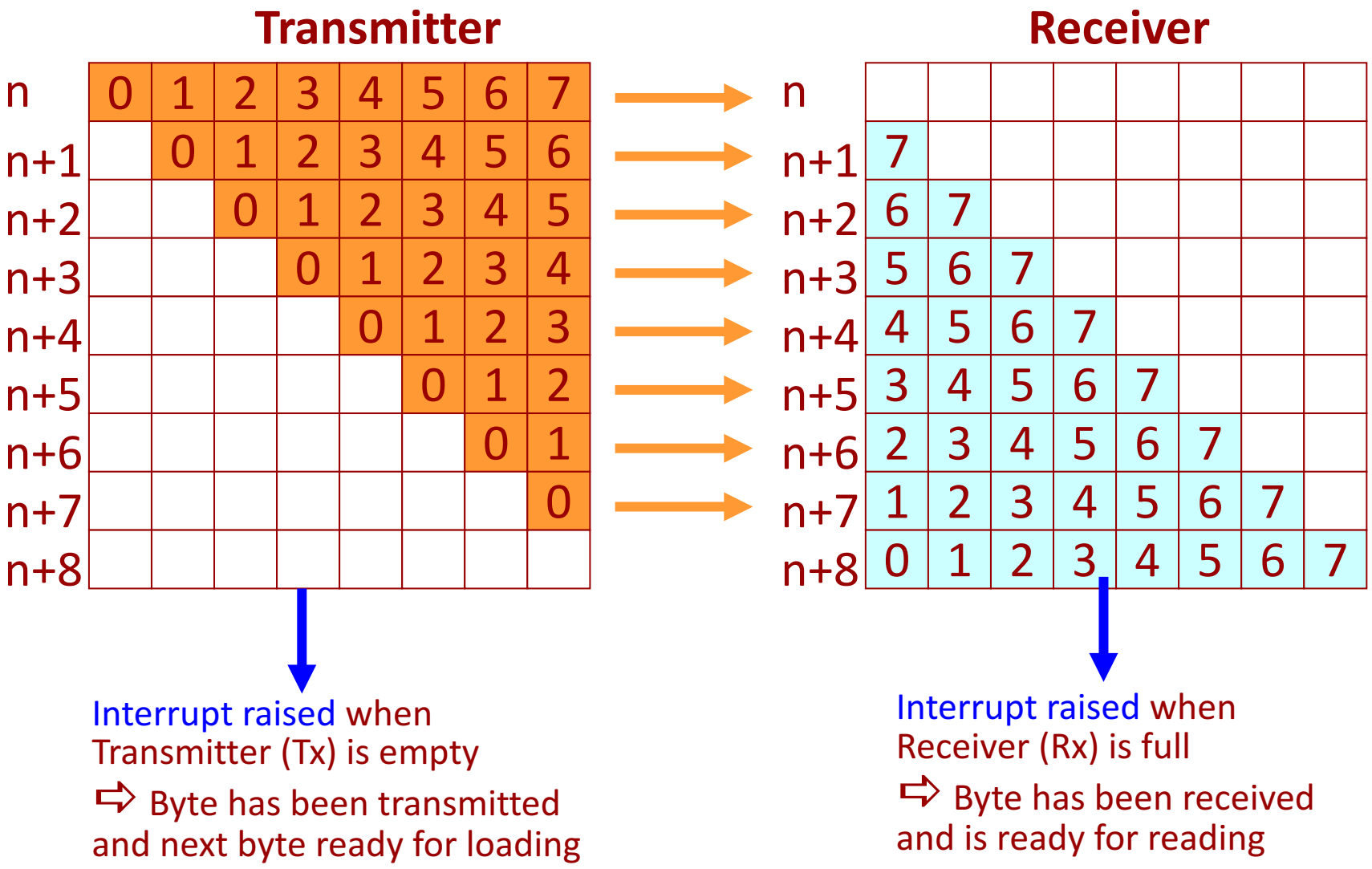
Serial



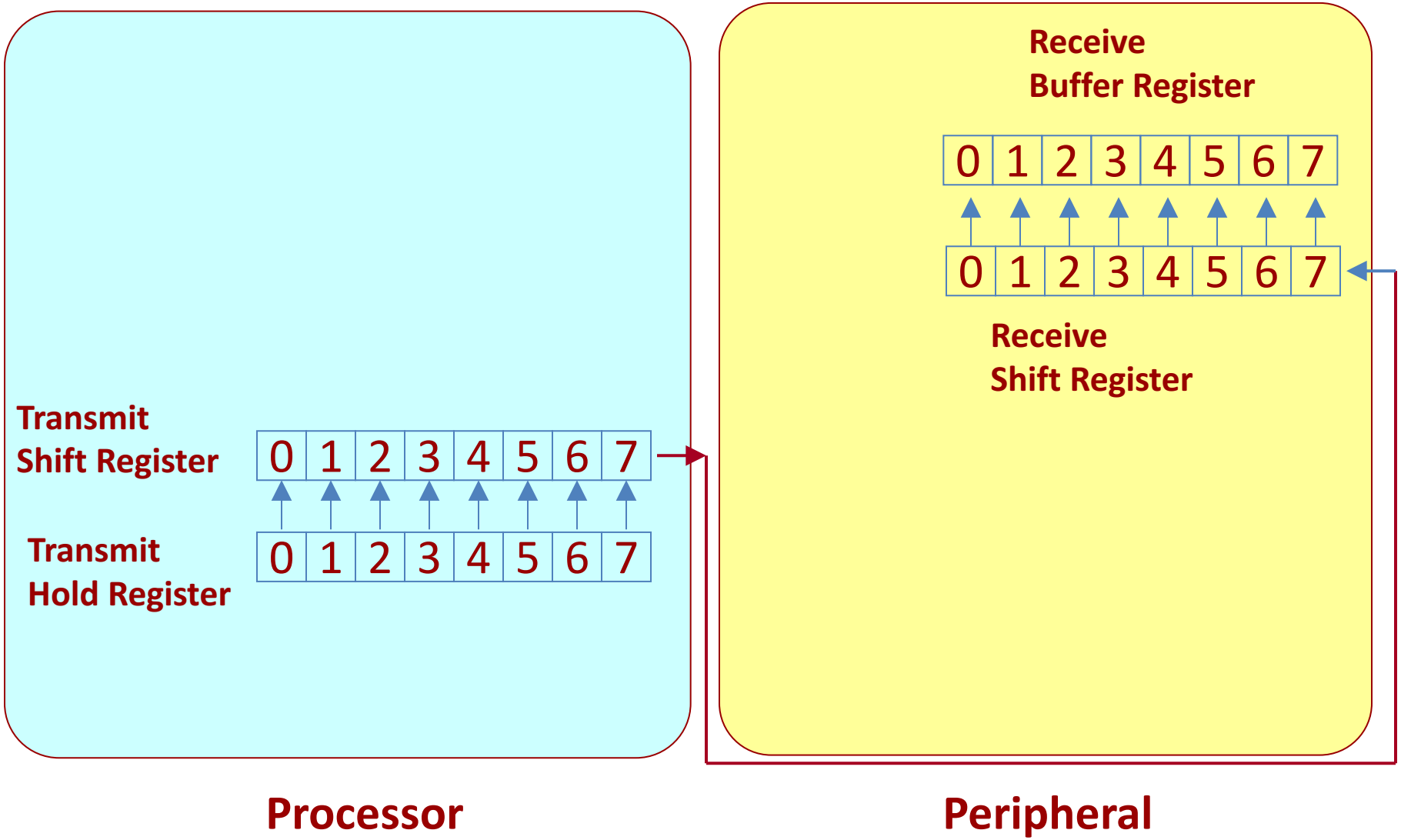
Parallel



Simplistic View of Serial Port Operation



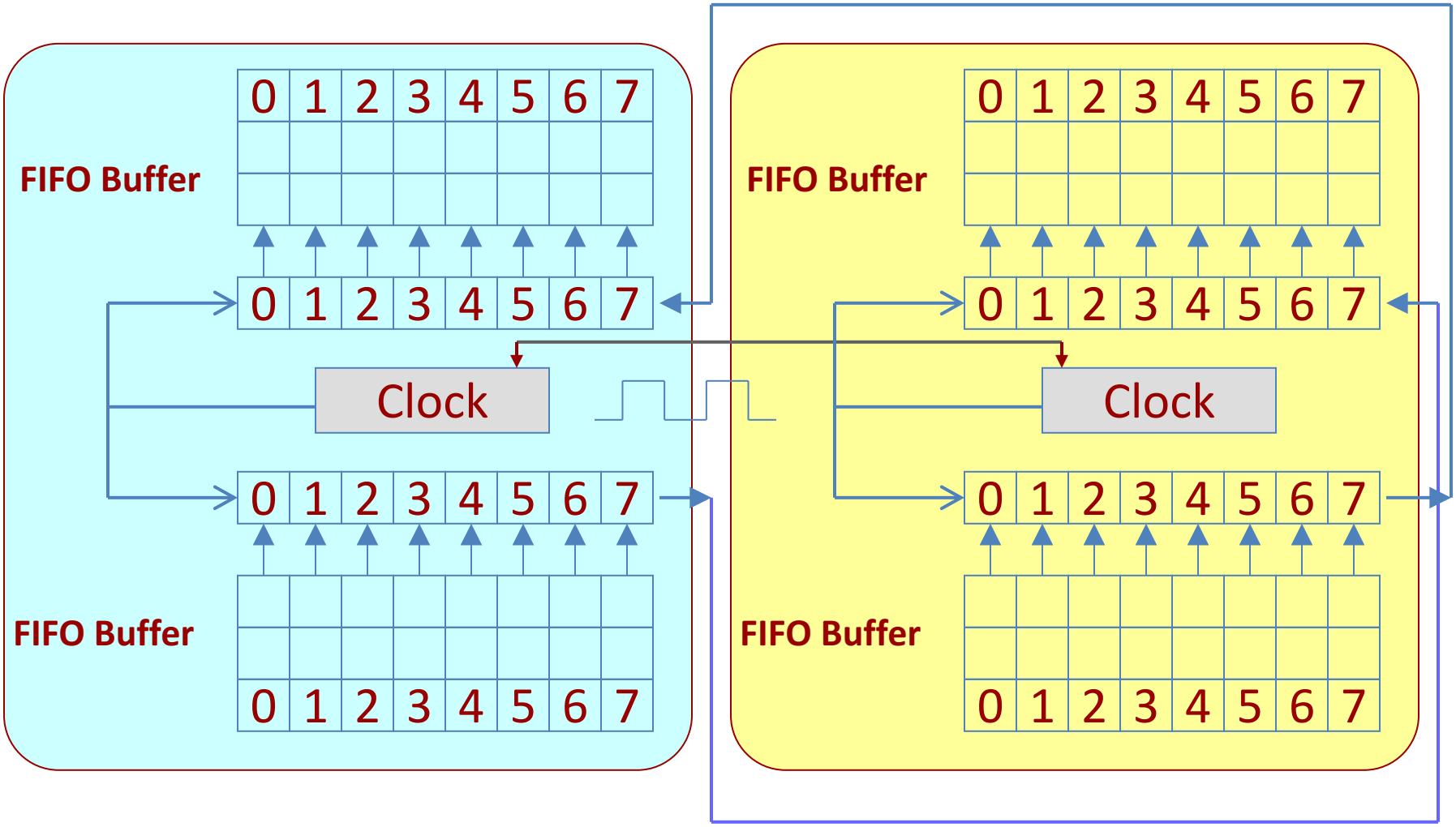
Simple Serial Port



Protecting Against Data Loss

- How can data be lost?
  - If the transmitter starts to send the next byte before the receiver has had a chance to process/read the current byte
  - If the next byte is loaded at the transmitter end before the current byte has been completely transmitted
- Most serial ports use FIFO buffers so that data is not lost
  - Buffering of received bytes at receiver end for later processing
  - Buffering of loaded bytes at transmitter end for later transmission
  - Shift registers free to transmit and receive data without worrying about data loss
- Why does the size of the FIFO buffers matter?

Serial Port



Processor **Peripheral**

What is RS-232?

- So far, we've talked about clocks being synchronized and using the clock as a reference for data transmission
  - Fine for short distances (e.g., within chips on the same board)
- When data is transmitted over longer distances (off-chip), voltage levels can be affected by cable capacitance
  - A logic "1" might appear as an indeterminate voltage at the receiver
  - Wrong data might be accepted when clock edges become skewed
- Enter RS232: Recommended Standard number 232
  - Serial ports for longer distances, typically, between PC and peripheral
  - Data transmitted asynchronously, i.e., no reference clock
  - Data provides its own reference clock

Types of Serial Communications

- Synchronous communication
  - Data transmitted as a steady stream at regular intervals
  - All transmitted bits are synchronized to a common clock signal
  - The two devices initially synchronize themselves to each other, and then continually send characters to stay synchronized
  - Faster data transfer rates than asynchronous methods, because it does not require additional bits to mark the beginning and end of each data byte
- Asynchronous communication
  - Data transmitted intermittently at irregular intervals
  - Each device uses its own internal clock resulting in bytes that are transferred at arbitrary times
  - Instead of using time as a way to synchronize the bits, the data format is used
  - Data transmission is synchronized using the start bit of the word, while one or more stop bits indicate the end of the word
    - Asynchronous communications slightly slower than synchronous

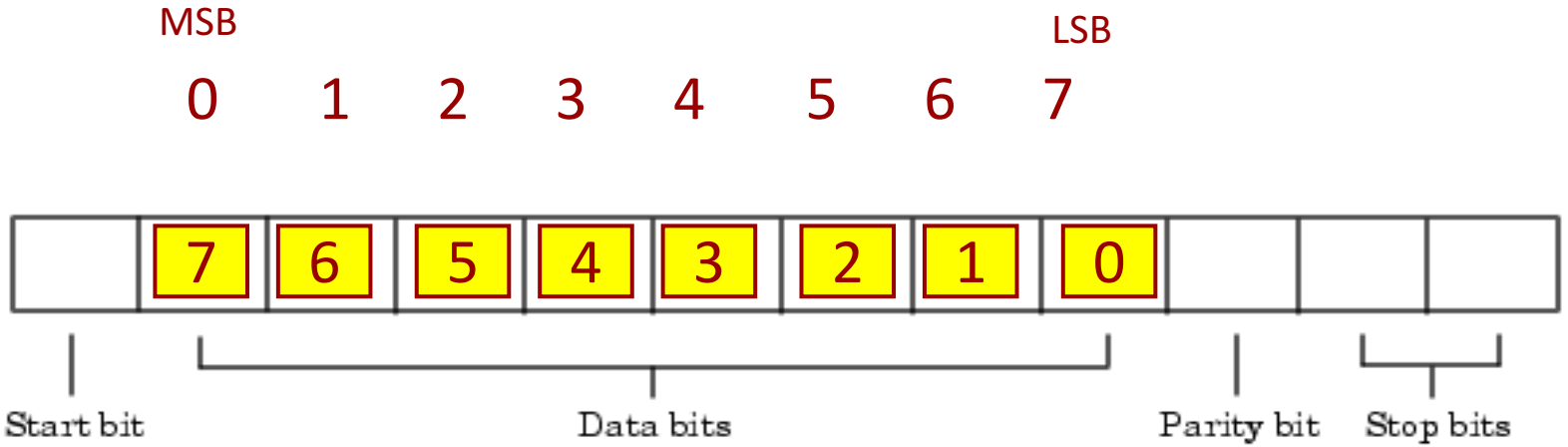
Sync vs. Async

- Synchronous communications
  - Requires common clock (SPI)
  - Whoever controls the clock controls communication speed

- Asynchronous communications
  - Has no clock (UART)
  - Speed must be agreed upon beforehand (the baud-rate configuration accomplishes that)

RS232 – Bits and Serial Bytes

- Serial ports on IBM-style PCs support asynchronous communication only
- A "serial byte" usually consists of
  - *Characters:* 5-8 data bits
  - *Framing bits:* 1 start bit, 1 parity bit (optional), 1-2 stop bits
  - When serial data is stored on your computer, framing bits are removed, and this looks like a real 8-bit byte
- Specified as number of data bits - parity type - number of stop bits
  - 8-N-1 a eight data bits, no parity bit, and one stop bit
  - 7-E-2 a seven data bits, even parity, and two stop bits



Parity Bits

- Simple error checking for the transmitted data
- Even parity
  - The data bits produce an even number of 1s
- Odd parity
  - The data bits produce an odd number of 1s

- Parity checking process
  1. The transmitting device sets the parity bit to 0 or to 1 depending on the data bit values and the type of parity checking selected.
  2. The receiving device checks if the parity bit is consistent with the transmitted data; depending on the result, error/success is returned

- Disadvantage
  - Parity checking can detect only **an odd number of bit-flip errors**
  - Multiple-bit errors can appear as valid data

Parity Example

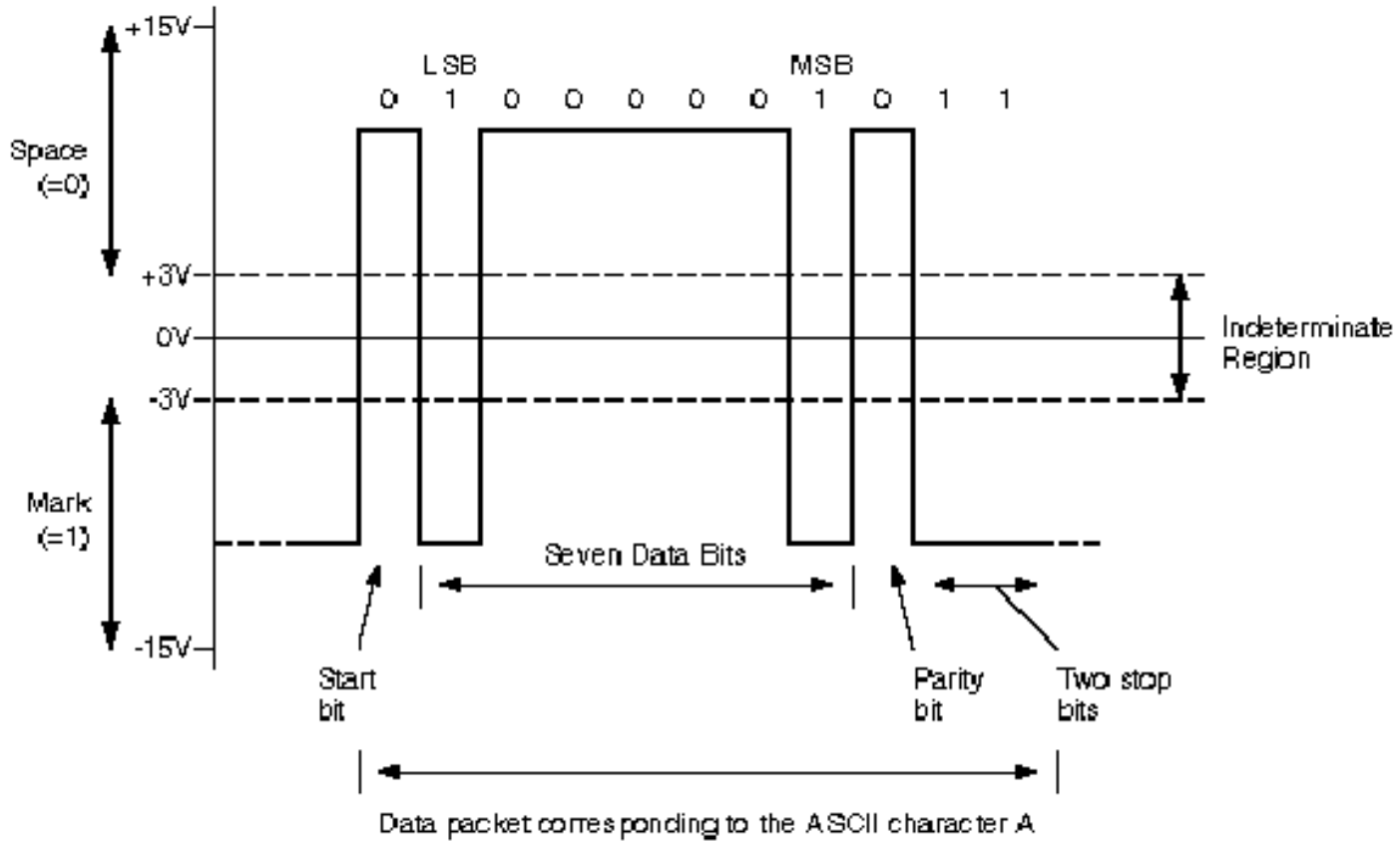
7 bits of data	(count of 1-bits)	8 bits including parity	
		even	odd
0000000	0	00000000	00000001
1010001	3	10100011	10100010
1101001	4	11010010	11010011
1111111	7	11111111	11111110

Value Typically ***Including*** Parity Bit

Data Modulation

- When sending data over serial lines, logic signals are converted into a form the physical media (wires) can support
- RS232C uses bipolar pulses
  - Any signal greater than +3 volts is considered a space (0)
  - Any signal less than 3 volts is considered a mark (1)
- Conventions
  - Idle line is assumed to be in high (1) state
  - Each character begins with a zero (0) bit, followed by 5-8 data bits and then 1, 11/2, or 2 closing stop bits
  - Bits are usually encoded using ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange)

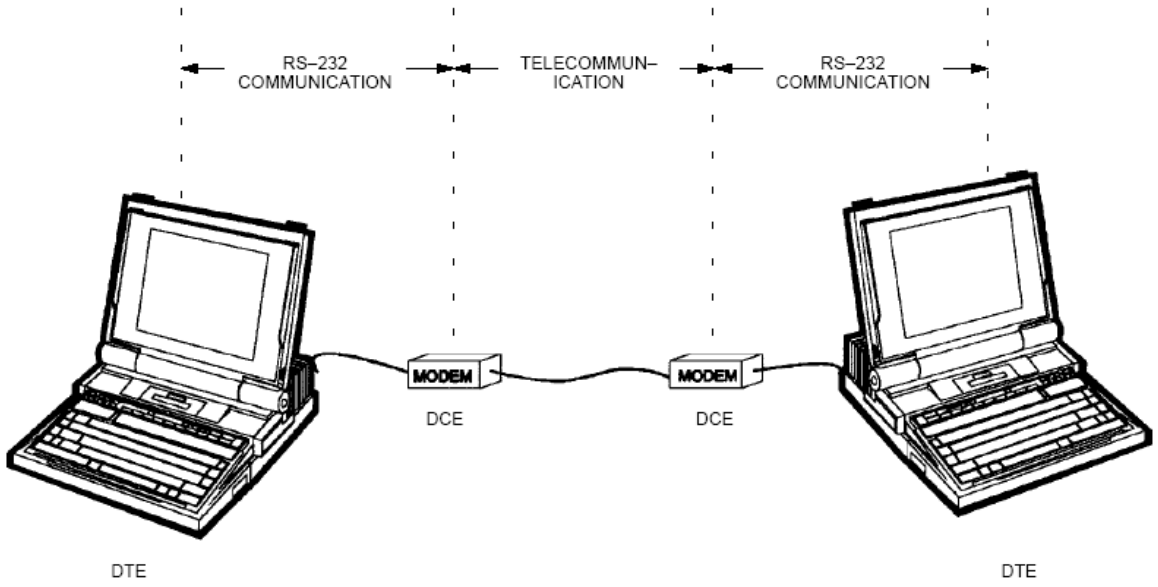
RS-232 Signal Levels



Source: Dallas Semiconductors Application note 83

Terminology

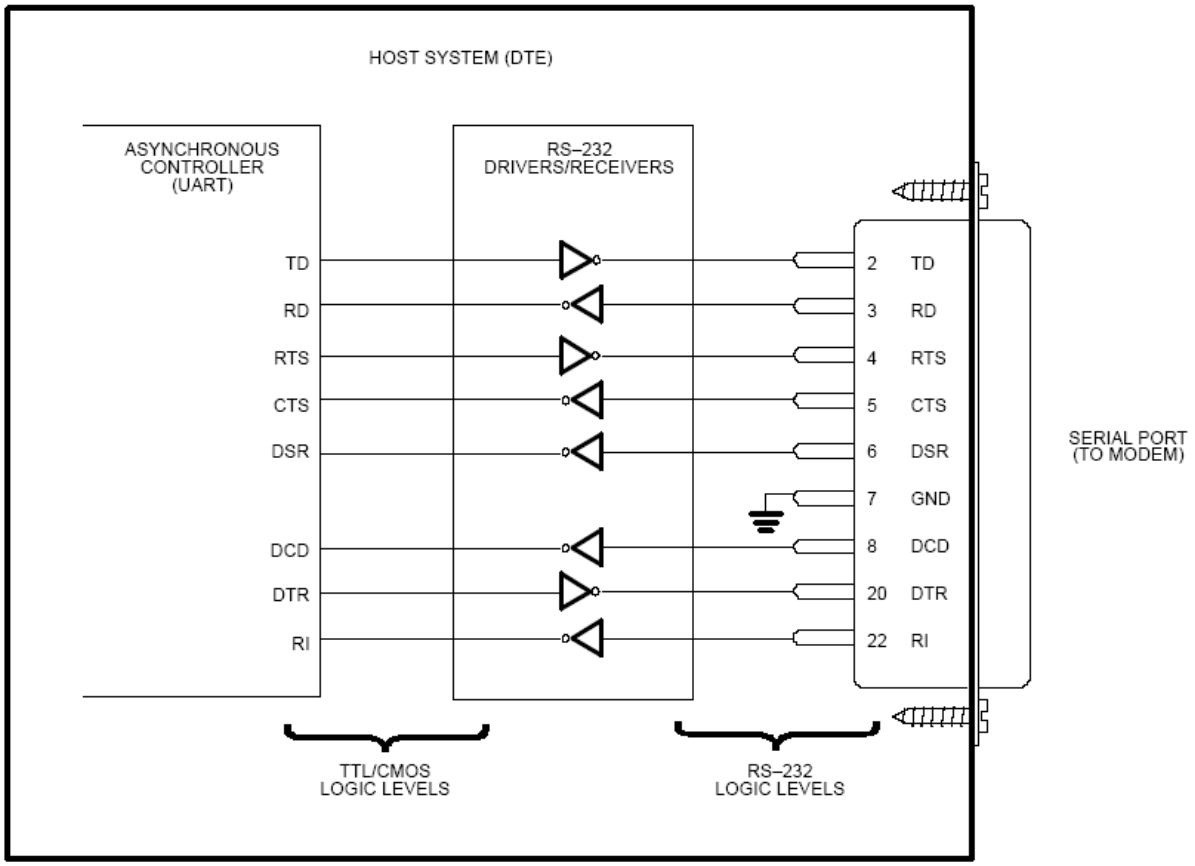
- DTE: Data terminal equipment, e.g., PC
- DCE: Data communication equipment, e.g., modem, remote device
- Baud Rate
  - Maximum number of times per second that a line changes state
  - Not always the same as bits per second



Serial Port Connector

- 9-pin (DB-9) or 25-pin (DB-25) connector
- Inside a 9-pin connector
  - **Carrier Detect** - Determines if the DCE is connected to a working phone line
  - **Receive Data** - Computer receives information sent from the DCE
  - **Transmit Data** - Computer sends information to the DCE
  - **Data Terminal Ready** - Computer tells the DCE that it is ready to talk
  - **Signal Ground** - Pin is grounded
  - **Data Set Ready** - DCE tells the computer that it is ready to talk
  - **Request To Send** - Computer asks the DCE if it can send information
  - **Clear To Send** - DCE tells the computer that it can send information
  - **Ring Indicator** – Asserted when a connected modem has detected an incoming call
- What's a null modem cable?

RS-232 Pin Connections



Source: Dallas Semiconductors Application note 83

Handshaking

- Some RS232 connections using handshaking lines between DCE and DTE
  - RTS (ReadyToSend)
    - Sent by the DTE to signal the DCE it is Ready To Send
  - CTS (ClearToSend)
    - Sent by the DCE to signal the DTE that it is Ready to Receive
  - DTR (DataTerminalReady)
    - Sent to DTE to inform the DCE that it is ready to connect
  - DSR (DataSetRead)
    - Sent to DCE to inform the DTE that it is ready to connect
- Handshaking lines can make it difficult to set up the serial communications, but seamless after set-up.
- Also, software handshaking (XON/XOFF)

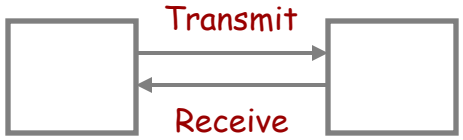
Serial Data Communication Modes

Simplex Mode

Transmission is possible only in one direction.

Half-duplex Mode

Data is transmitted in one direction at a time but the direction can be changed.

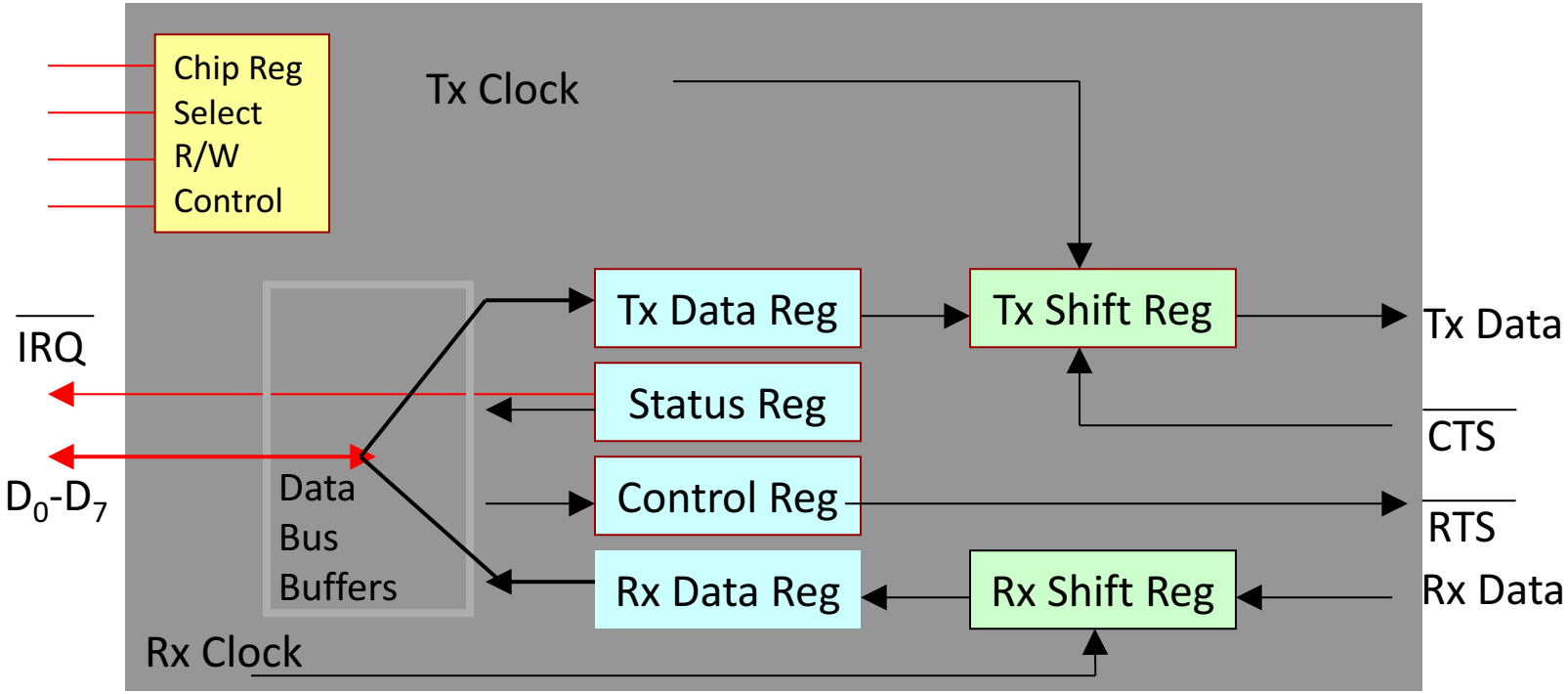


Full-duplex Mode

Data may be transmitted simultaneously in both directions.

Interfacing Serial Data to Microprocessor

- Processor has parallel buses for data need to convert serial data to parallel (and vice versa)
- Standard way is with UART
- UART Universal asynchronous receiver and transmitter



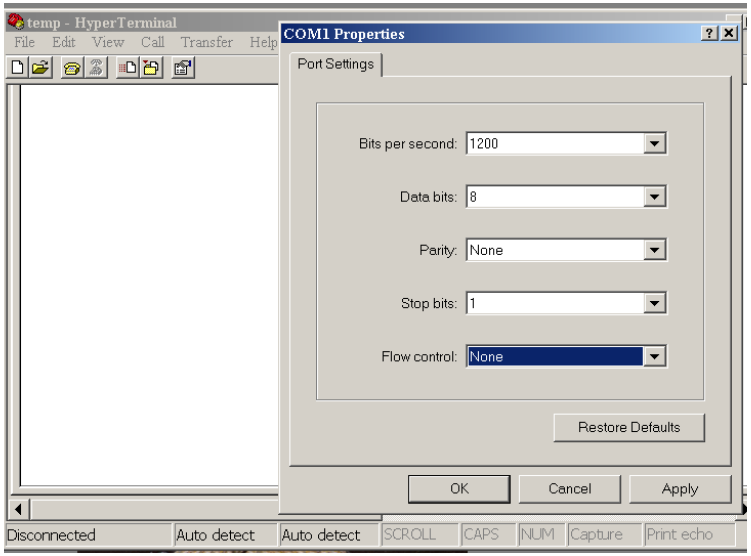
Flow Control

- Necessary to prevent terminal from sending more data than the peripheral can consume (and vice-versa)
  - Higher data rates can result in missing characters (data-overrun errors)
- Hardware handshaking
  - Hardware in UART detects a potential overrun and asserts a handshake line to prevent the other side from transmitting
  - When receiving side can take more data, it releases the handshake line
- Software flow-control
  - Special characters XON and XOFF
  - XOFF stops a data transfer (control-S or ASCII code 13)
  - XON restarts the data transfer (control-Q or ASCII code 11)
- Assumption is made that the flow-control becomes effective before data loss happens

HyperTerminal / Minicom

- A (hyper) terminal program is an application that will enable a PC to communicate directly with a serial port
  - Can be used to display data received at the PC's serial port

- Can be used to configure the serial port
  - Baud rate
  - Number of data bits
  - Number of parity bits
  - Number of stop bits
  - Flow control



UART and MMIO (Example)


```
#define UART2_BASE          0x20100000
#define UART2_LS_DIV        (UART2_BASE + 0x00)
#define UART2_MS_DIV        (UART2_BASE + 0x01)
#define UART2_TX_REG        (UART2_BASE + 0x00)
#define UART2_RX_REG        (UART2_BASE + 0x00)
#define UART2_INT_ID        (UART2_BASE + 0x01)
#define UART2_INT_EN_REG    (UART2_BASE + 0x01)
#define UART2_FIFO_CNTRL    (UART2_BASE + 0x02)
#define UART2_LINE_CNTRL    (UART2_BASE + 0x03)
#define UART2_MODM_CNTRL    (UART2_BASE + 0x04)
#define UART2_LINE_STAT     (UART2_BASE + 0x05)
#define UART2_MODM_STAT     (UART2_BASE + 0x06)
#define UART2_SCRATCH       (UART2_BASE + SCRATCH_OFFSET)
```

Example – RPI LCR Register

AUX_MU_LCR_REG Register (0x7E21 504C)

SYNOPSIS The AUX_MU_LCR_REG register controls the line data format and gives access to the baudrate register

Bit(s)	Field Name	Description	Type	Reset
31:8		Reserved, write zero, read as don't care		
7	DLAB access	If set the first to Mini UART register give access the the Baudrate register. During operation this bit must be cleared.	R/W	0
6	Break	If set high the UART1_TX line is pulled low continuously. If held for at least 12 bits times that will indicate a break condition.	R/W	0
5:1		Reserved, write zero, read as don't care *Some of these bits have functions in a 16550 compatible UART but are ignored here*		0
0	data size	If clear the UART works in 7-bit mode If set the UART works in 8-bit mode	R/W	0

Example – RPI status register

AUX_MU_STAT_REG Register (0x7E21 5064)

SYNOPSIS The AUX_MU_STAT_REG provides a lot of useful information about the internal status of the mini UART not found on a normal 16550 UART.

Bit(s)	Field Name	Description	Type	Reset
31:28		Reserved, write zero, read as don't care		
27:24	Transmit FIFO fill level	These bits shows how many symbols are stored in the transmit FIFO The value is in the range 0-8	R	0
23:20		Reserved, write zero, read as don't care		
19:16	Receive FIFO fill level	These bits shows how many symbols are stored in the receive FIFO The value is in the range 0-8	R	0
15:10		Reserved, write zero, read as don't care		
9	Transmitter done	This bit is set if the transmitter is idle and the transmit FIFO is empty. It is a logic AND of bits 2 and 8	R	1
8	Transmit FIFO is empty	If this bit is set the transmitter FIFO is empty. Thus it can accept 8 symbols.	R	1
7	CTS line	This bit shows the status of the UART1_CTS line.	R	0
6	RTS status	This bit shows the status of the UART1_RTS line.	R	0
5	Transmit FIFO is full	This is the inverse of bit 1	R	0
4	Receiver overrun	This bit is set if there was a receiver overrun. That is: one or more characters arrived whilst the receive FIFO was full. The newly arrived characters have been discarded. This bit is cleared each time the AUX_MU_LSR_REG register is read.	R	0
3	Transmitter is idle	If this bit is set the transmitter is idle. If this bit is clear the transmitter is idle.	R	1
2	Receiver is idle	If this bit is set the receiver is idle. If this bit is clear the receiver is busy. This bit can change unless the receiver is disabled	R	1
1	Space available	If this bit is set the mini UART transmitter FIFO can accept at least one more symbol. If this bit is clear the mini UART transmitter FIFO is full	R	0

Serial vs. Parallel

- Serial ports
  - Universal Asynchronous Receiver/Transmitter (UART): controller
  - Takes the computer bus' parallel data and serializes it
  - Transfer rate of 115 Kbps
  - Example usage: Modems
- Parallel ports
  - Sends/receives the 8 bits in parallel over 8 different wires
  - 50-100 KBps (standard), upto 2 MBps (enhanced)
  - Example usage: Printers, Zip drives

Other Serial Buses

RS-232

Point-to-point +/-12V

I2C

Two wire chip interconnect, multi-drop

I2S

Audio format similar to SPI

RS-485

Multi-drop RS-232

SPI

Four wire only chip interconnect, multi-drop

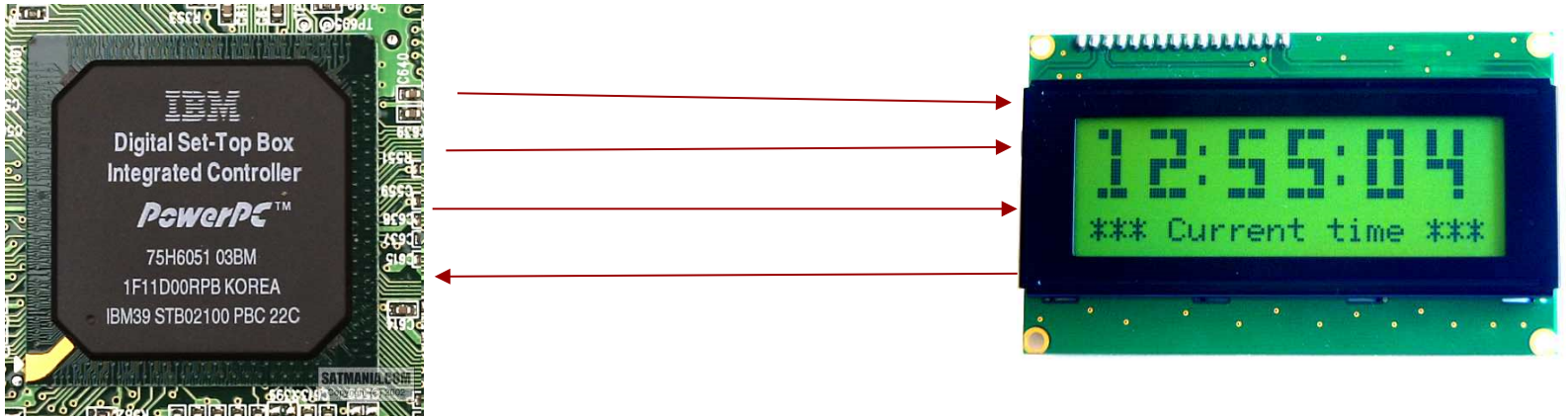
Many more…

Serial Peripheral Interconnect (SPI)

- Another kind of serial protocol in embedded systems (proposed by Motorola)
- Four-wire protocol
  - SCLK — Serial Clock
  - MOSI/SIMO — Master Output, Slave Input
  - MISO/SOMI — Master Input, Slave Output
  - SS — Slave Select
- Single master device and with one or more slave devices
- Higher throughput than I2C and can do "stream transfers"
- No arbitration required
- But
  - Requires more pins
  - Has no hardware flow control
  - No slave acknowledgment (master could be talking to thin air and not even know it)

What is SPI?

- Serial Bus protocol
- Fast, Easy to use, Simple
- Everyone supports it



SPI Basics

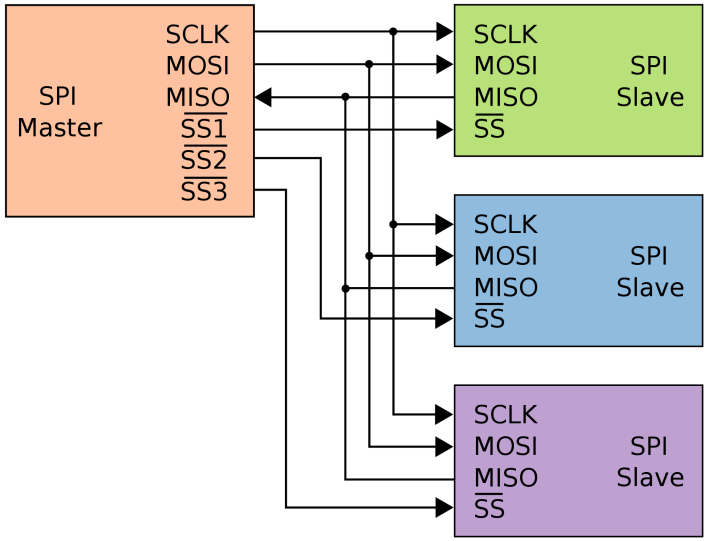
- A communication protocol using 4 wires
  - Also known as a 4 wire bus
- Used to communicate across small distances
- Multiple Slaves, Single Master
- Synchronized

SPI Capabilities

- Always Full Duplex
  - Communicating in two directions at the same time
  - Transmission need not be meaningful
- Multiple Mbps transmission speed
- Transfers data in 4 to 16 bit characters
- Multiple slaves
  - Daisy-chaining possible

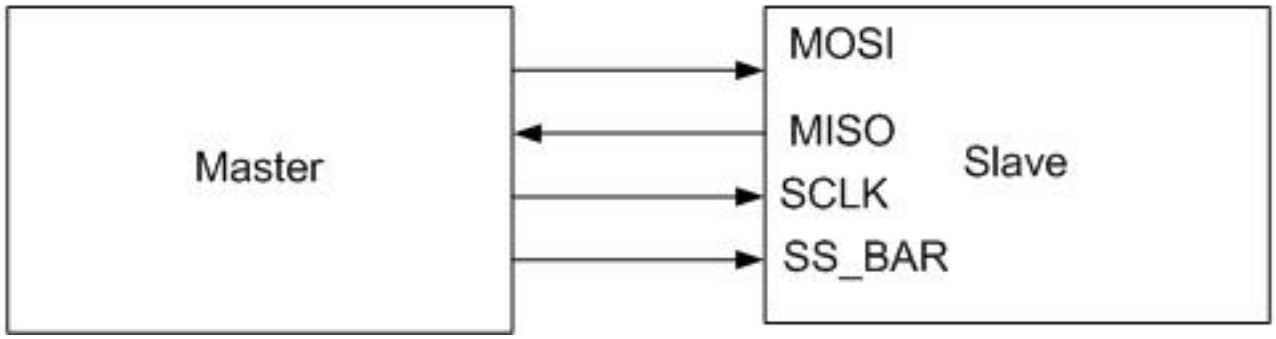
SPI Protocol

- Wires:
  - Master Out Slave In (MOSI)
  - Master In Slave Out (MISO)
  - System Clock (SCLK)
  - Slave Select 1…N
- Master Set Slave Select low
- Master Generates Clock
- Shift registers shift in and out data

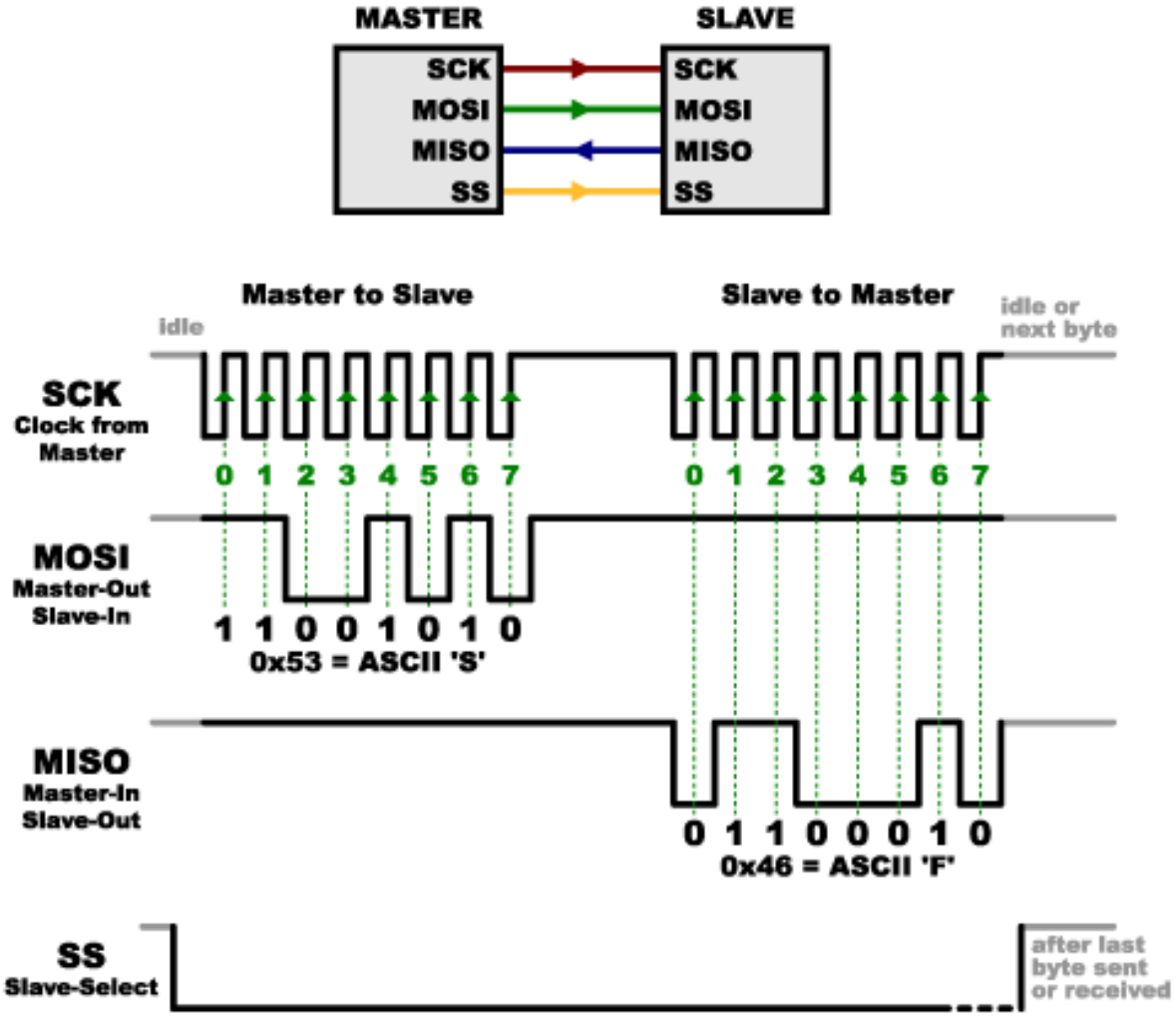


SPI Wires in Detail

- MOSI – Carries data out of Master to Slave
- MISO – Carries data from Slave to Master
  - Both signals happen for every transmission
- SS_BAR – Unique line to select a slave
- SCLK – Master produced clock to synchronize data transfer



SPI Communication



SPI Pros and Cons

- Pros:
  - Fast and easy
    - Fast for point-to-point connections
    - Easily allows streaming/Constant data inflow
    - No addressing/Simple to implement
  - Everyone supports it

- Cons:
  - SS makes multiple slaves very complicated
  - No acknowledgement ability
  - No inherent arbitration
  - No flow control

I2C Background

- I2C is also written as I^2C (pronounced "eye-squared-see" or "eye-two-see")
  - Stands for Inter-Integrated Circuit (IIC)
- Two-wire party-line bus for "inside the box" communication
- Intended for short-range communication between ICs on a circuit board or across boards in an embedded system
- I2C devices commonly used in industrial applications
  - EEPROMs, thermal sensors, real-time clocks, RF tuners, video decoders/encoders
- Philips Semiconductors is the primary champion of I2C
  - Specification publicly available at http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf
  - Originally developed for communication between devices inside a TV set in the mid-1980s

I2C Purpose

- Designed by Philips ~20 years ago
- Original purpose was to allow easy communication between components which resided on the same circuit board
- Combines hardware and software protocols to provide a bus interface that can connect many peripheral devices
- I2C is now not only used on single boards, but also to connect components which are linked via cable
- All I2C-compatible devices have an on-chip interface that allows them to communicate directly with each other via the I2C-bus
- Supports easy, ready-to-use interfacing of various boards and digital circuits (even if they are independently designed)
- Allows for “plug-and-play” and evolution of ICs into a larger system

I2C Characteristics

- Only two bus lines are required
- Each device connected to the bus is software-addressable by a unique address and
  - Simple master/slave relationships
- True multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer
- Serial, 8-bit oriented, bidirectional data transfers
  - Up to 100 kbit/s in the standard mode
  - Up to 400 kbit/s in the fast mode
  - High-speed (3.4 Mbps), I2C version2.0
- On-chip filtering rejects spikes on the bus data line to preserve data integrity

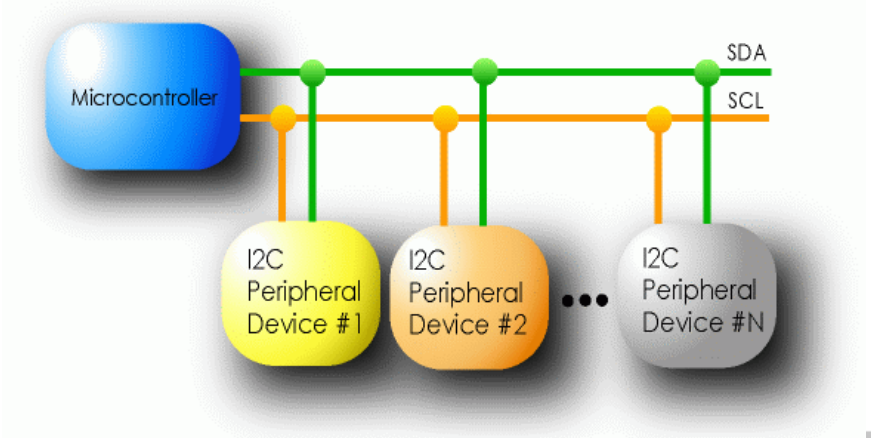
I2C Design Criteria

- First of all, this is a serial bus
  - Targeting 8-bit microcontroller applications
  - Serial vs. parallel – anyone remember pros and cons?

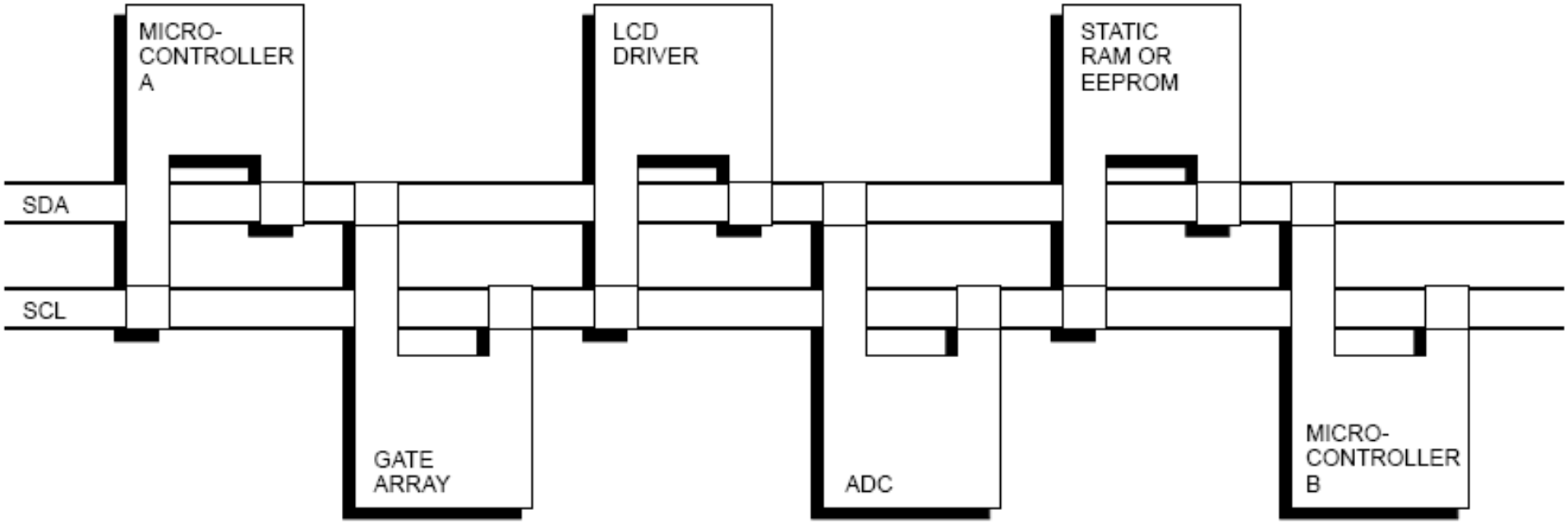
- Criteria for design of I2C
  - Need to avoid confusion between connected devices
  - Fast devices must be able to communicate with slow ones
  - Protocol must not be dependent on the devices that it connects
  - Need to have a mechanism to decide who controls the bus and when
  - If different devices with different clock speeds are connected, the bus clock speed must be defined

I2C Details

- Two lines: Serial data line (SDA) & serial clock line (SCL)
- Each I2C device recognized by a unique address
- Each I2C device can be either a transmitter or receiver
- I2C devices can be masters or slaves for a data transfer
  - Master (usually a microcontroller): Initiates a data transfer on the bus, generates the clock signals to permit that transfer, and terminates the transfer
  - Slave: Any device addressed by the master at that time
  - Roles/relationships are not permanent



I2C-Connected System



Example I2C-connected system with two microcontrollers

(Source: I2C Specification, Philips)

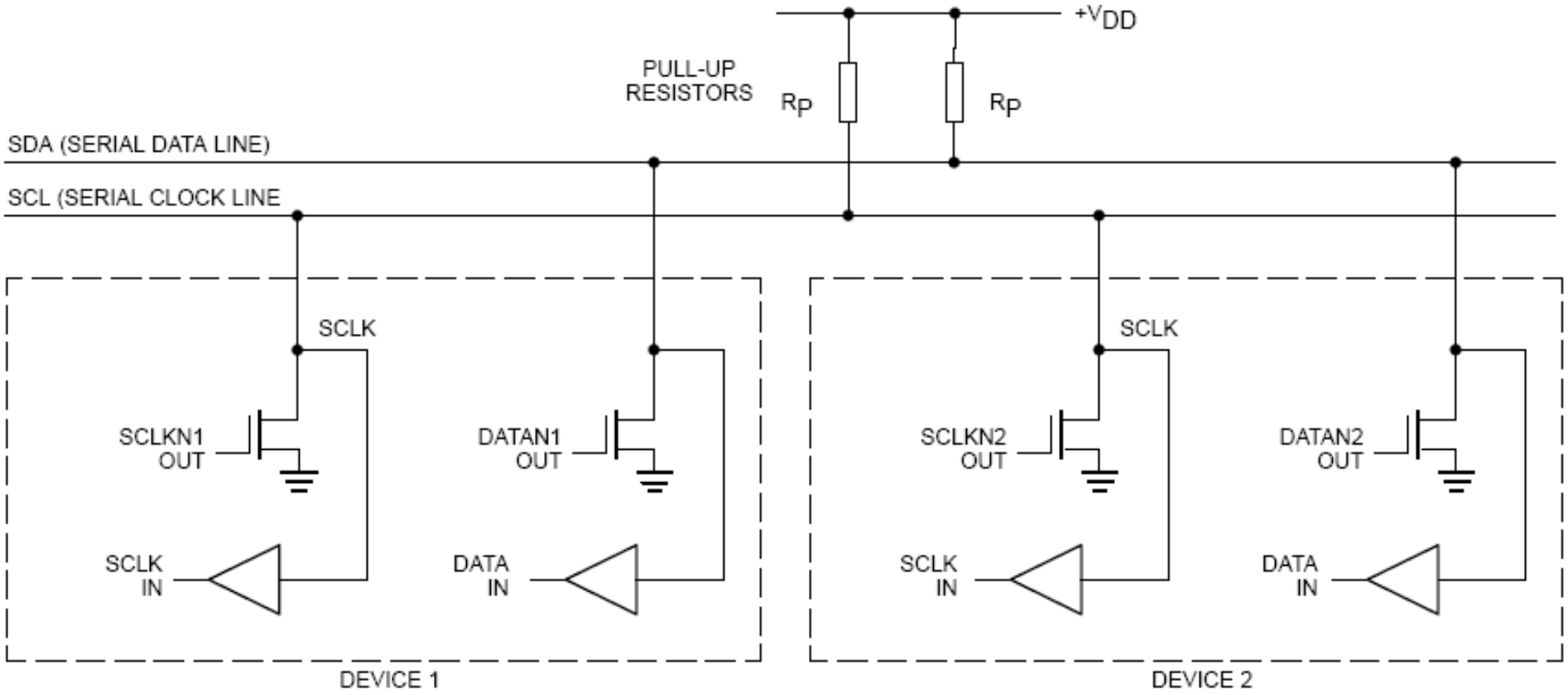
Master-Slave Relationships

- Masters can operate as master-transmitters or master-receivers
- Suppose microcontroller A wants to send information to microcontroller B
  - A (master) addresses B (slave)
  - A (master-transmitter), sends data to B (slave-receiver)
  - A terminates the transfer.
- If microcontroller A wants to receive information from microcontroller B
  - A (master) addresses microcontroller B (slave)
  - A (master-receiver) receives data from B (slave-transmitter)
  - A terminates the transfer
- In both cases, the master (microcontroller A) generates the timing and terminates the transfer

Multi-Master Capability

- Clearly, more than one microcontroller can be connected to the bus
  - What if both microcontrollers want to control the bus at the same time?
- Multi-master I2C capability supports this without corrupting the message
- Wired-AND connection of all I2C interfaces to the bus for arbitration
- If two or more masters try to put information onto the bus, the first to produce a ‘one’ when the other produces a ‘zero’ loses
- Clock signals during arbitration are a synchronized combination of the clocks generated by the masters using the wired-AND connection to the SCL line
- Generation of clock signals on the bus
  - Each master generates its own clock signals when transferring data on the bus
  - A master’s bus clock signals can be altered when stretched by a slow-slave device holding down the clock line, or by another master during arbitration

Connecting I2C Devices to the Bus



Addressing

- First byte of transfer contains the slave address and the data direction
  - Address is 7 bits long, followed by the direction bit
  - Like all data bytes, address is transferred with the most significant bit first
- 7-bit address space allows for 128 unique I2C device addresses
  - 16 addresses are reserved for special purposes
  - Leaves only 112 addresses with this 7-bit address scheme
- New 10-bit address scheme has been introduced
- "General call" broadcast – to address every device on the bus
- What is the maximum number of devices in I2C limited by?

Clock Stretching

- Form of flow control
- An addressed slave device may hold the clock line low after receiving (or sending) a bit, indicating that it is not yet ready to process more data
- Master that is communicating with the slave will attempt to raise the clock to transfer the next bit, but
  - If the slave is clock stretching, the clock line will still be low
- Mechanism allows receivers that cannot keep up with a transmitter to control the flow of incoming data