Lecture 13: Serial Communication

eUSCI

SCI (UART)

SPI

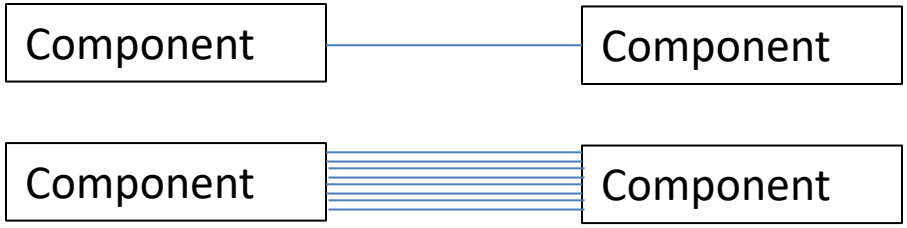
I2C

Chapter 24 in the TRM

The slides with this format come from the book "Embedded Systems: Introduction to the MSP432 Microcontroller", ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2015

eUSCI

- Enhanced Universal Serial Communication Interface
  - Supports multiple serial communication modes
- Parallel versus Serial Communication
  - Serial incorporates shifting over bits over time.



- Simplex, Half-Duplex, Full Duplex
- Synchronous versus Asynchronous

eUSCI

- Enhanced Universal Serial Communication Interface
  - Supports multiple serial communication modes

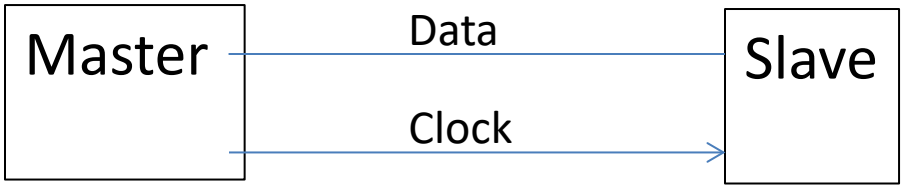
Recall: LCD

1. SPI – Synchronous Peripheral Interface
   - Synchronous, 50 Mbps (higher bandwidth applications, audio/video)
2. UART – Universal Asynchronous Receiver-Transmitter
   - 0.3 Kbps to 1 Mbps (low bandwidth like sensors)
3. I2C – Inter-Integrated Circuit

   – low-speed (speed grades of 100kbps, 400 kbps, 1 mbps) , short distance intra-board, half-duplex, synchronous

Clocking

- Recall: SPI is Synchronous – A common clock is shared between two controllers, to shift the bits
  - Master is the one that generates the clock



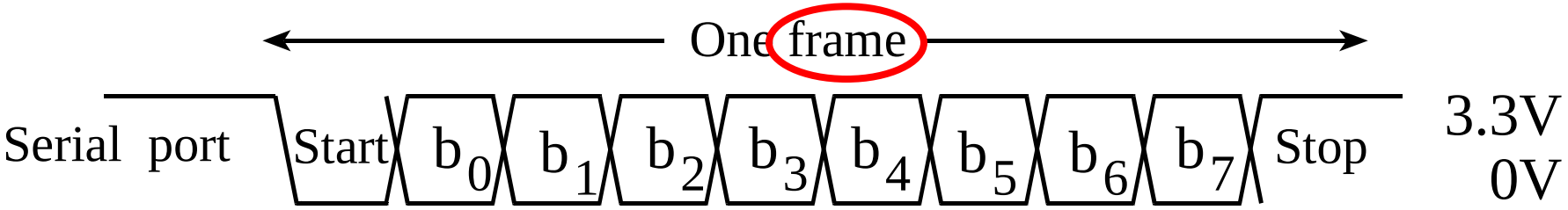
- UARTs are Asynchronous – Each device has its own clock which must be set to a common frequency
  - Allows devices to be physically remote (via a cable)

Interactive Simulation of UARTs

- Interactive Webpage by Valvano – Scroll down to Interactive Tool 11.4: Serial Communication of a Simple Two-Byte Message
- URL : http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C11_Interactives.htm

Universal Asynchronous Receiver/Transmitter (UART)

- UART (Serial Port) Interface



  - Send/receive a *frame* of (5-8) data bits with a single (start) bit prefix and a 1 or 2 (stop) bit suffix and optional parity bit (not shown)
  - Baud rate is total number of bits per unit time
    - Baudrate = 1 / bit-time
  - Bandwidth is data per unit time
    - Bandwidth = (data-bits / frame-bits) * baudrate

Parity

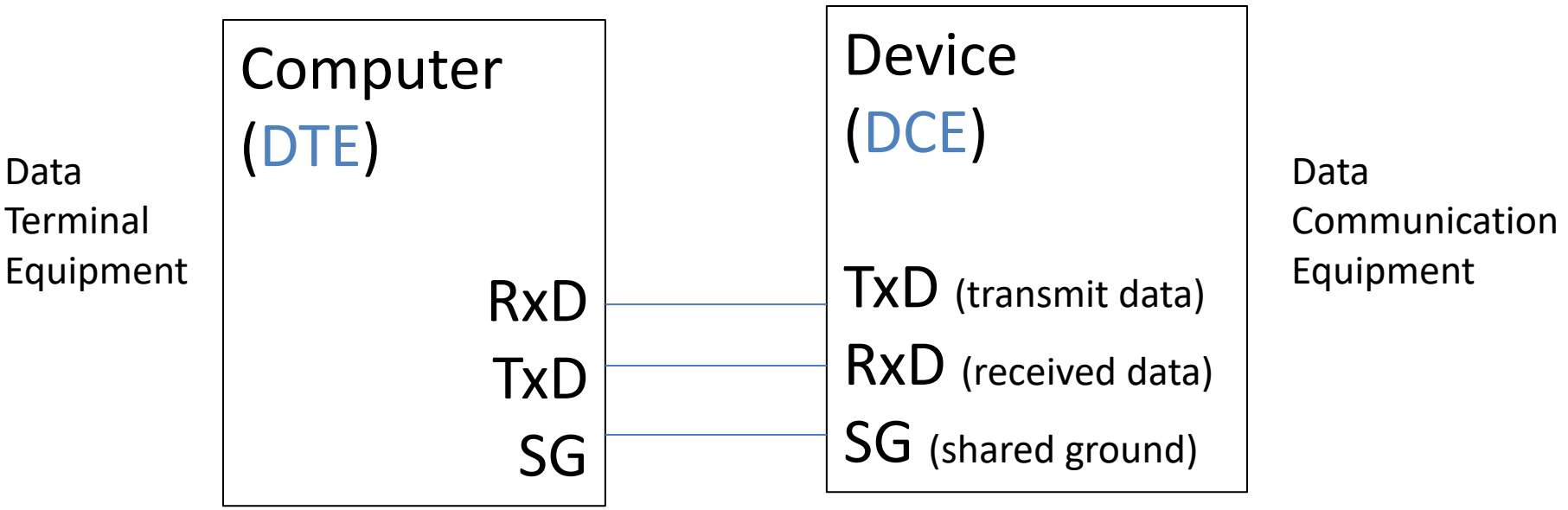
- Rudimentary error detection
- Even parity : data plus parity bit has an even number of 1’s
- Odd parity: data plus parity bit has an odd number of 1’s

Exercise

- Assuming 8-bit data, 1 start bit, 1 stop bit and no parity, what is the bandwidth in bytes/sec if the baud rate is 1000 bits/second?

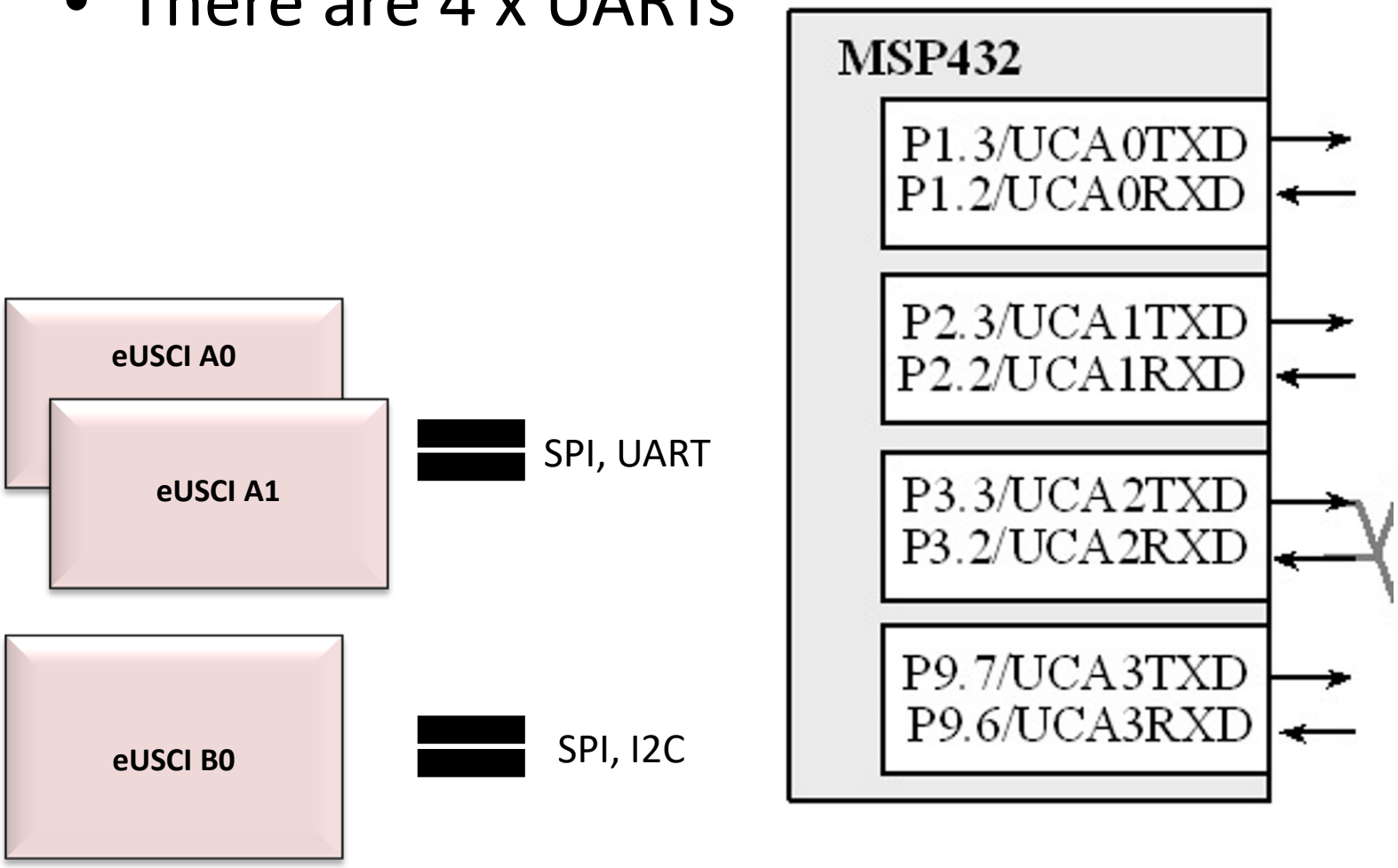
RS232 and EIA-574 Protocol Standards

- RS232 has 25 pins! EIA-574 has 9 pins!
- In simple configurations, 3 are used for full-duplex communication
  - Each component has its own clock but each must be configured to be run at the same speed



On the MSP432

- There are 4 x UARTs



MSP432 UART I/O Pins

Pin	PxSEL1=0, PxSEL0=0	PxSEL1=0, PxSEL0=1
P1.2	*Port*	*UCA0RXD/UCA0SOMI*
P1.3	*Port*	*UCA0TXD/UCA0SIMO*
P2.2	Port	UCA1RXD/UCA1SOMI
P2.3	Port	UCA1TXD/UCA1SIMO
P3.2	Port	UCA2RXD/UCA2SOMI
P3.3	Port	UCA2TXD/UCA2SIMO
P9.6	Port	UCA3RXD/UCA3SOMI
P9.7	Port	UCA3TXD/UCA3SIMO

Table 4.1. SEL1 and SEL0 bits on the MSP432 specify alternate functions. *P1.2* and *P1.3* are hardwired to the serial port.

Roadmap

- As usual, we will learn the hardware in two stages so that we can get going quickly in the lab, yet still understand the hardware at the end

1. Valvano's UART drivers
2. UART theory

CAUTIONs

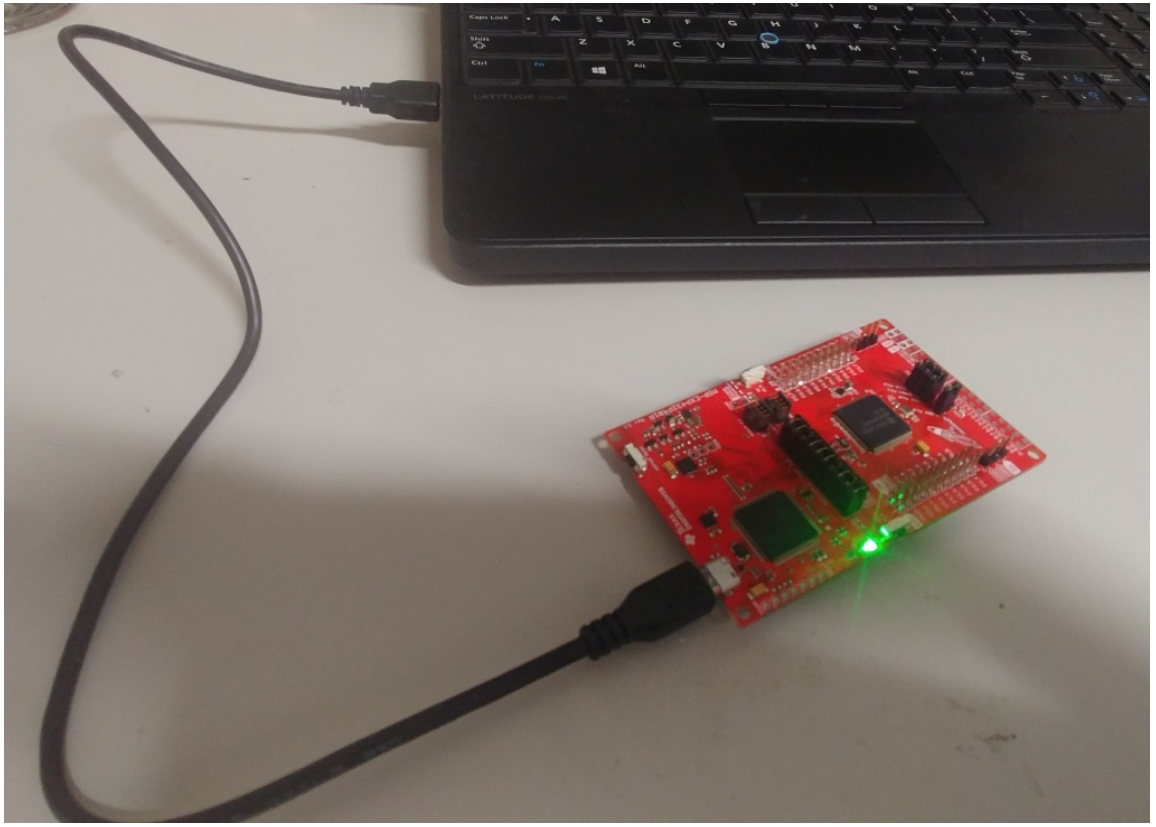
1. This is notoriously unreliable. Sometimes it just won't work. Go away. Disconnect. Come back and try again.
2. To use the demo code, you need to re-use my exact project (not just the .c file but also the project file). I don't know why … yet.

Our USB “Back channel”

- From the MSP432p401r User’s Guide:
- The XDS110-ET provides a "backchannel" UART-over-USB connection with the host, which can be very useful during debugging and for easy communication with a PC.
- The backchannel UART allows communication with the USB host that is not part of the target application's main functionality. This is very useful during development, and also provides a communication channel to the PC host side. This can be used to create GUIs and other programs on the PC that communicate with the LaunchPad development kit.

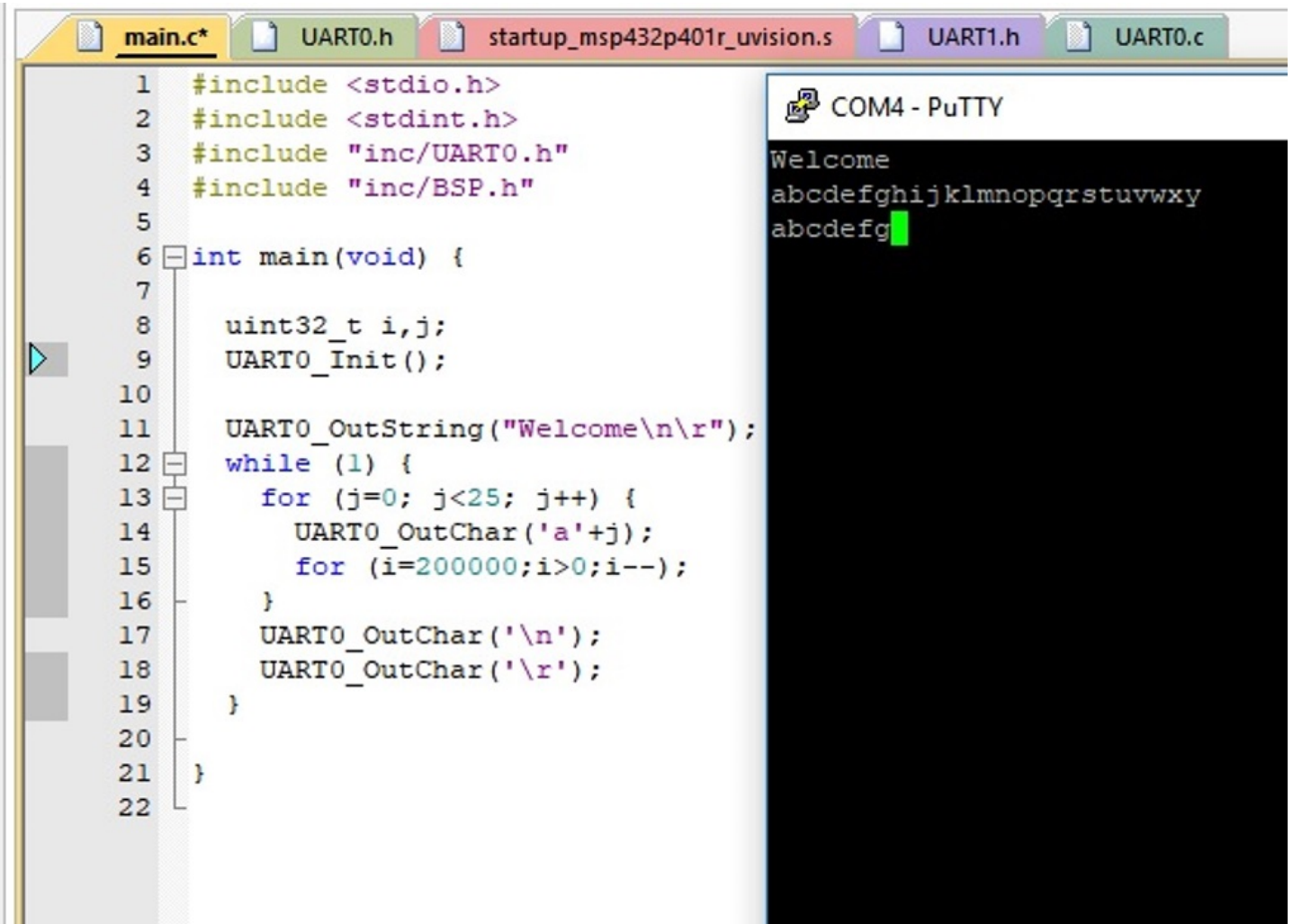
Backchannel UART

- eUSCI_A0 is routed through the USB cable, to become a virtual COM port on your host machine



Our System

- Two windows: Keil and **Putty**

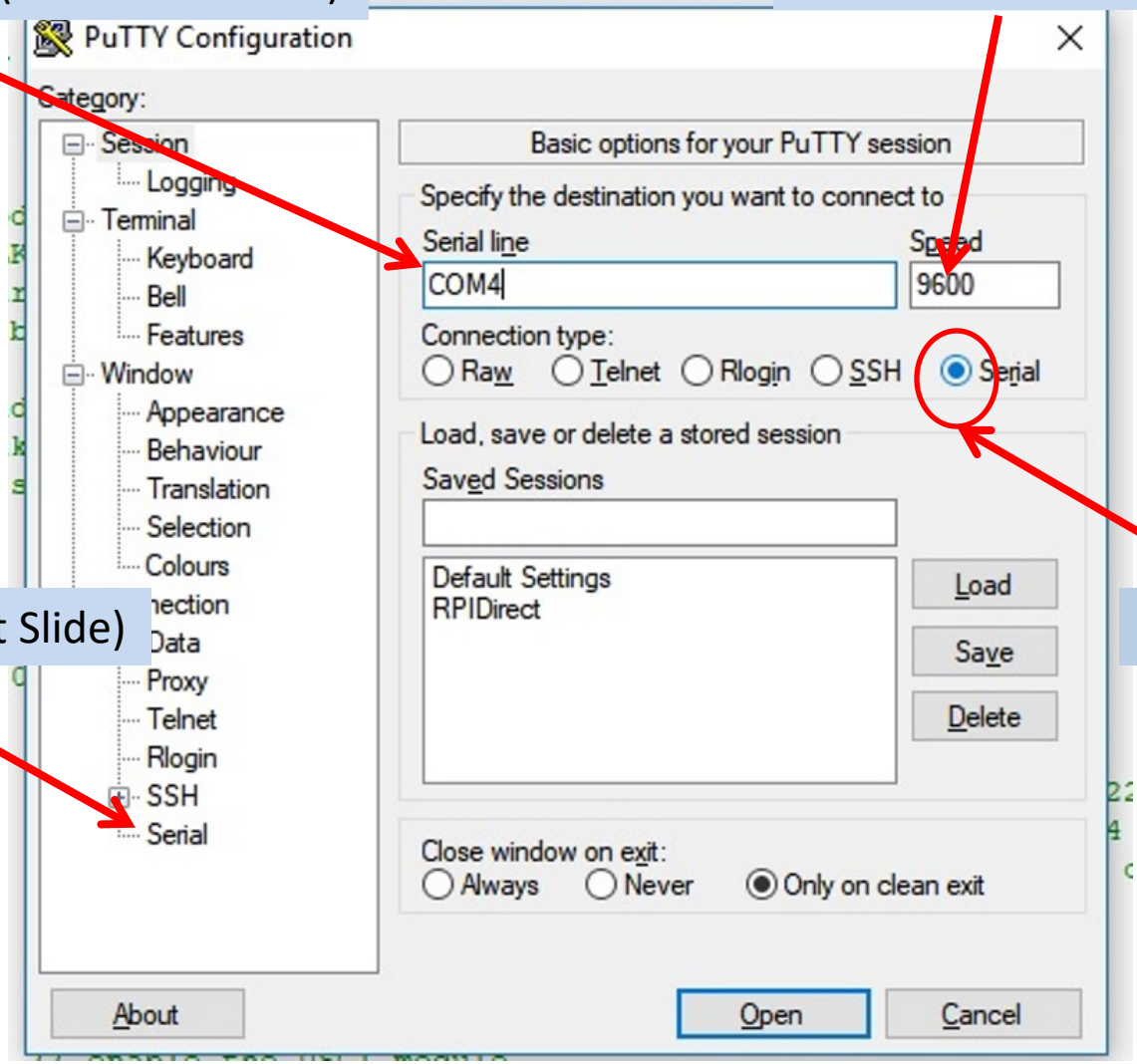


Putty

- Must select the SERIAL protocol (not default)

2. Enter your COM (How? Next slide)

3. Change Speed to 115200

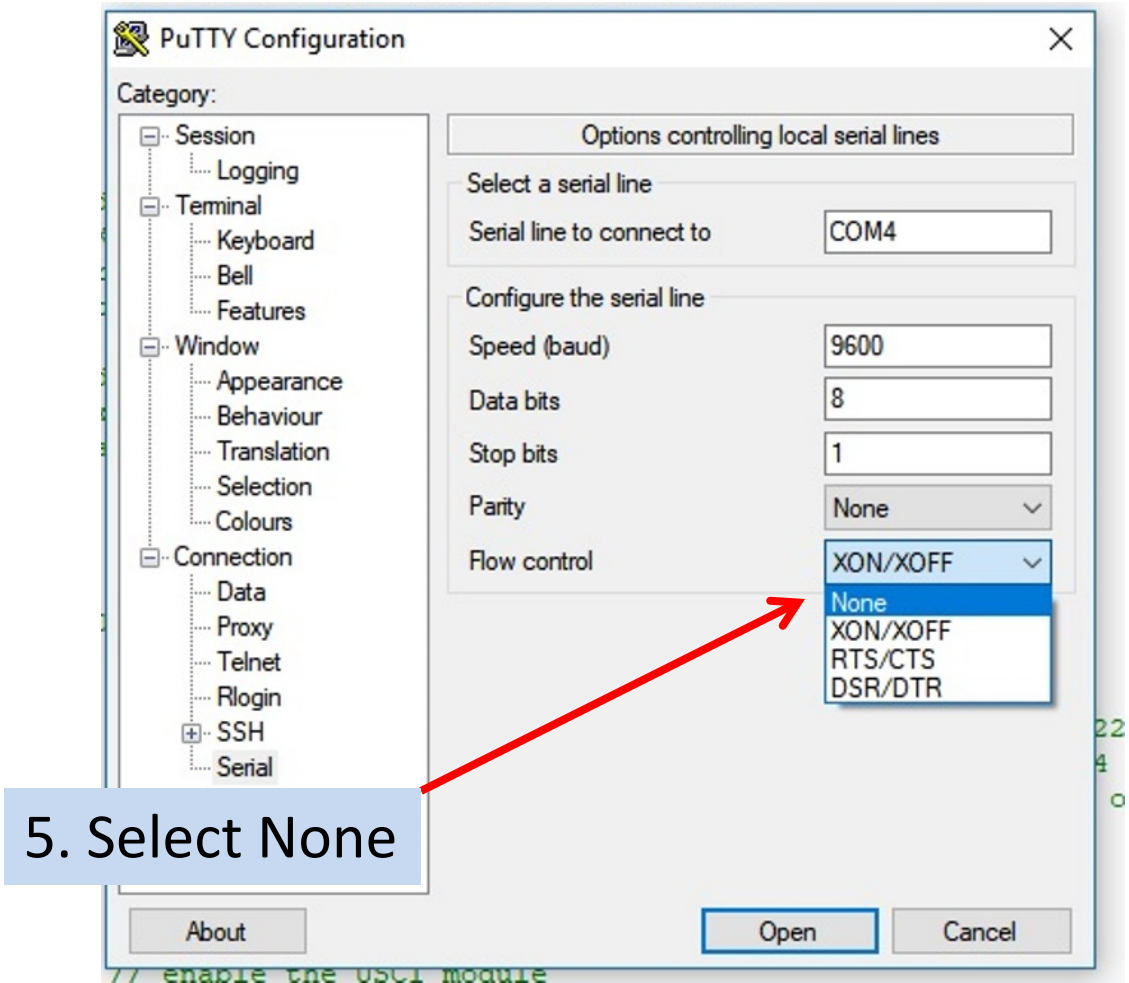


4. Open up Serial (Next Slide)

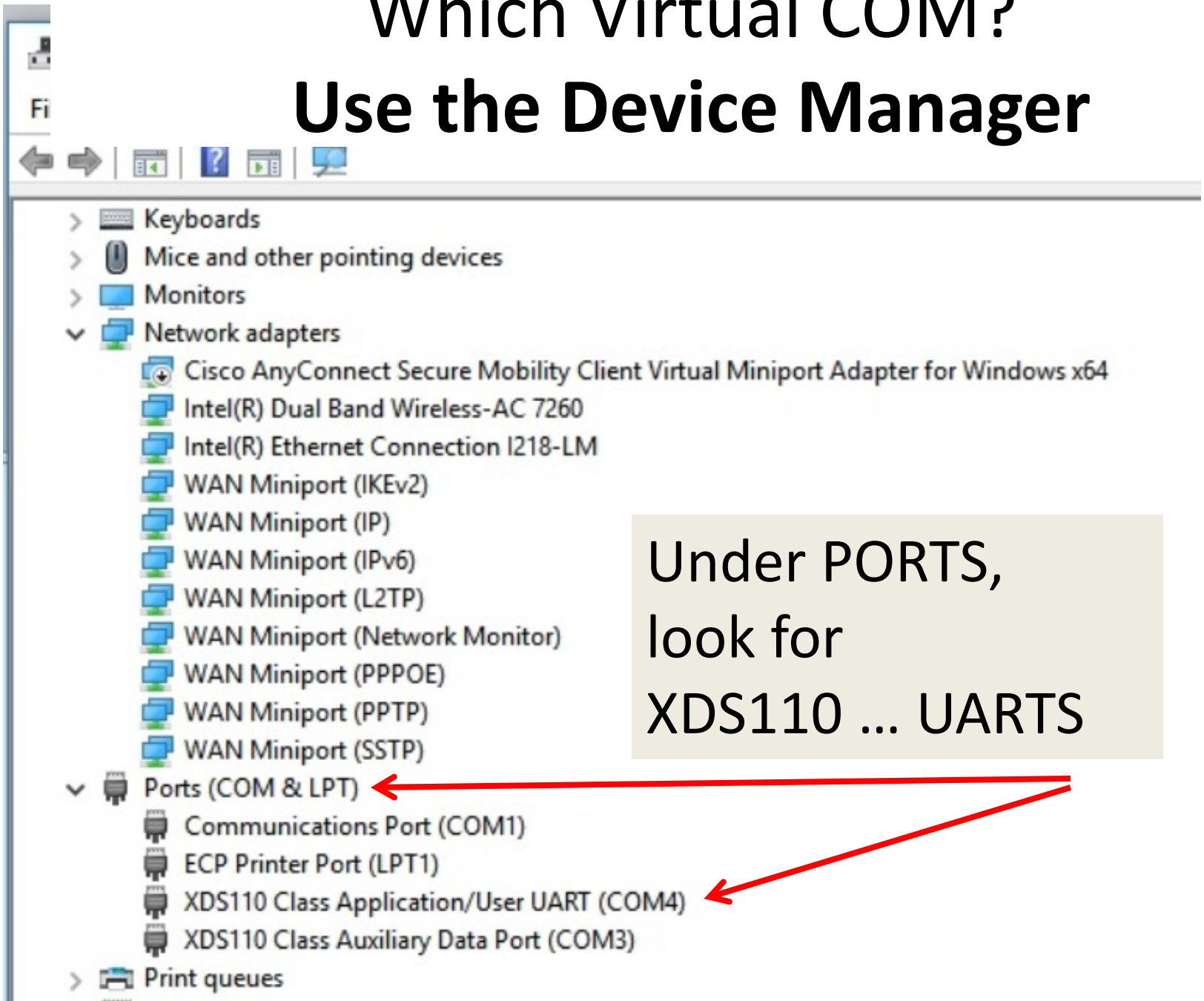
1. Select Serial

Putty - Setup

- Check the additional parameters



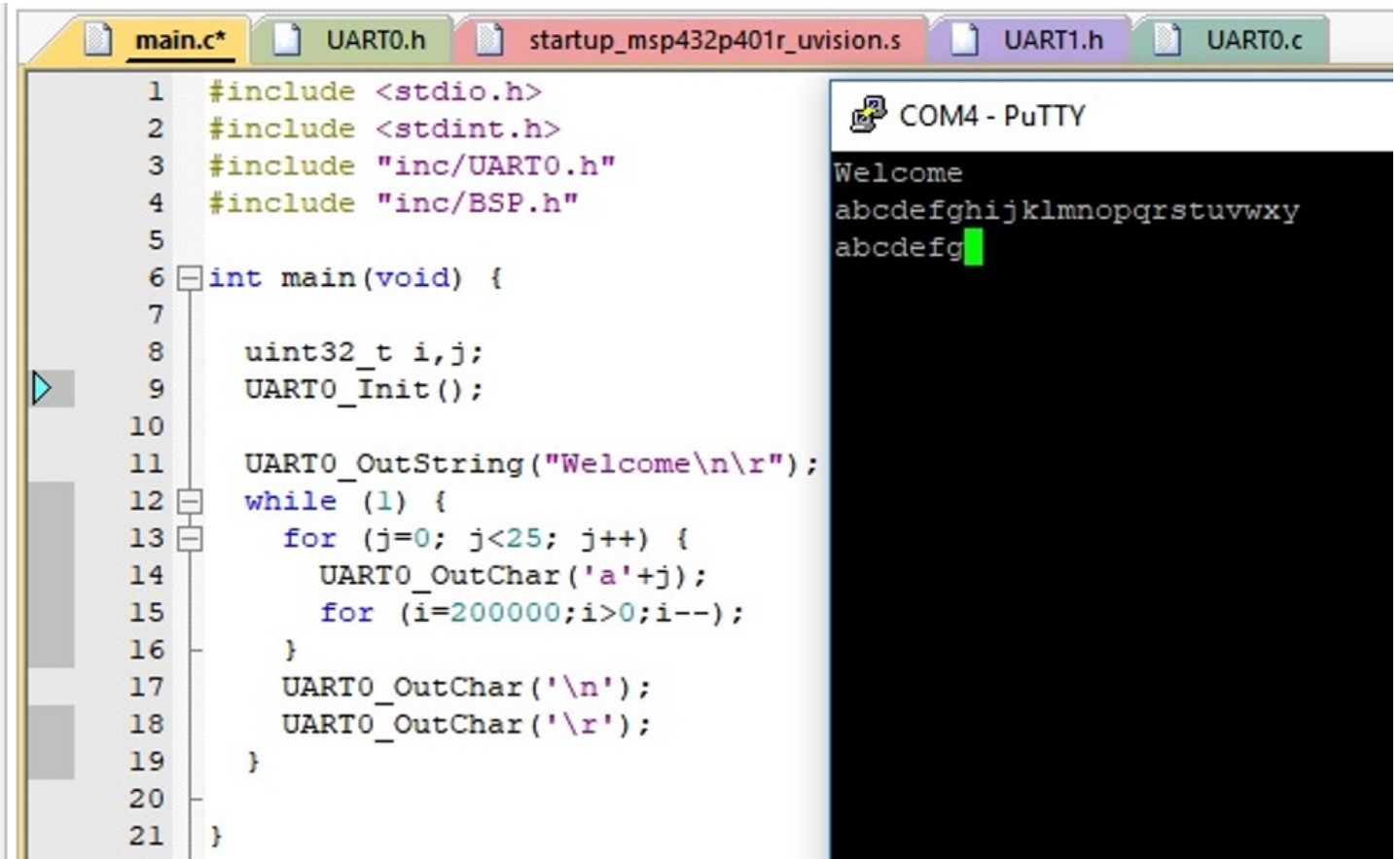
Which Virtual COM?
Use the Device Manager



Under PORTS,
look for
XDS110 … UARTS

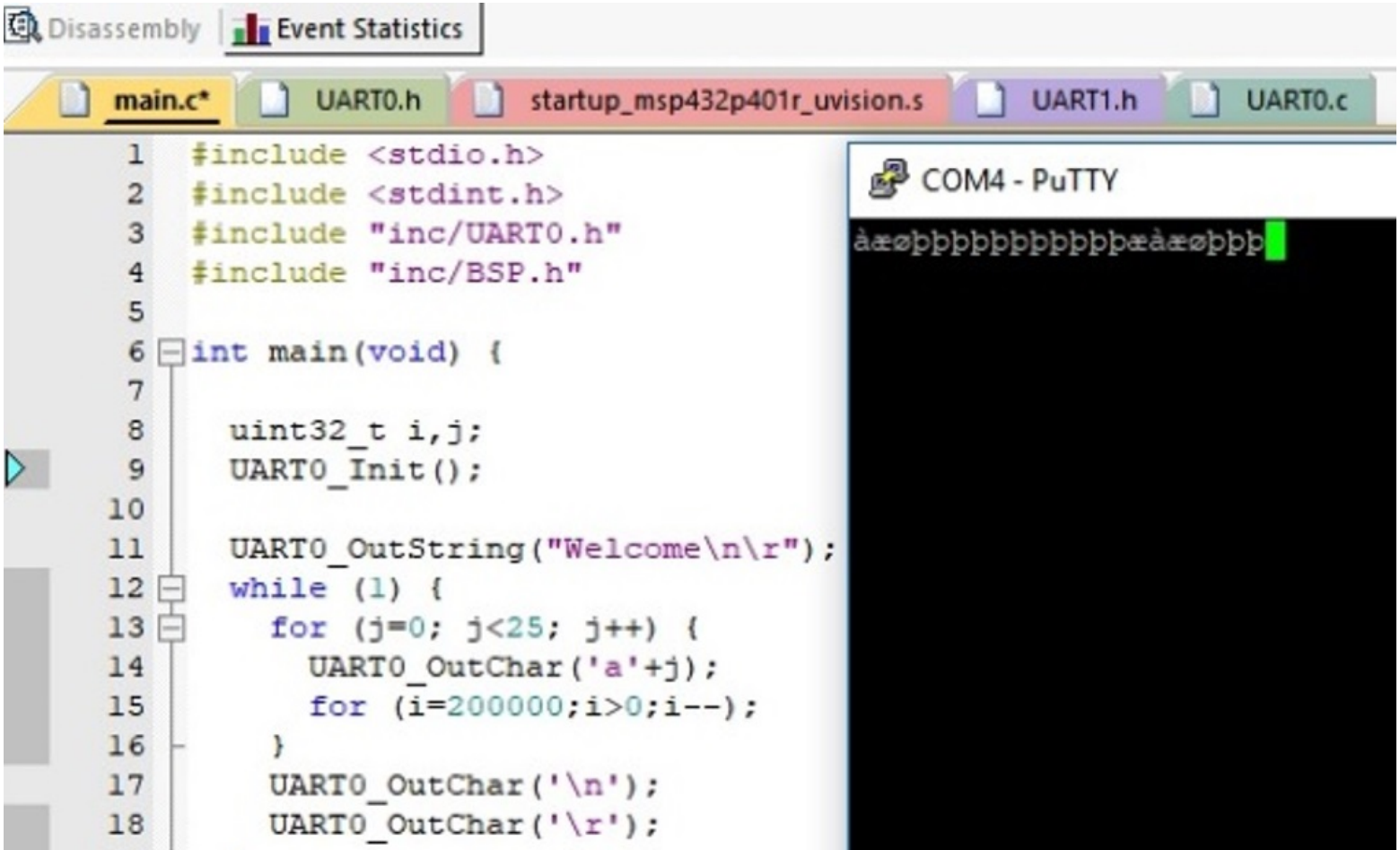
Correct Baud Rate

- Run your program as normal with Putty running alongside in another window



Incorrect Baud Rate

- You're close!



Two Different Demo Programs

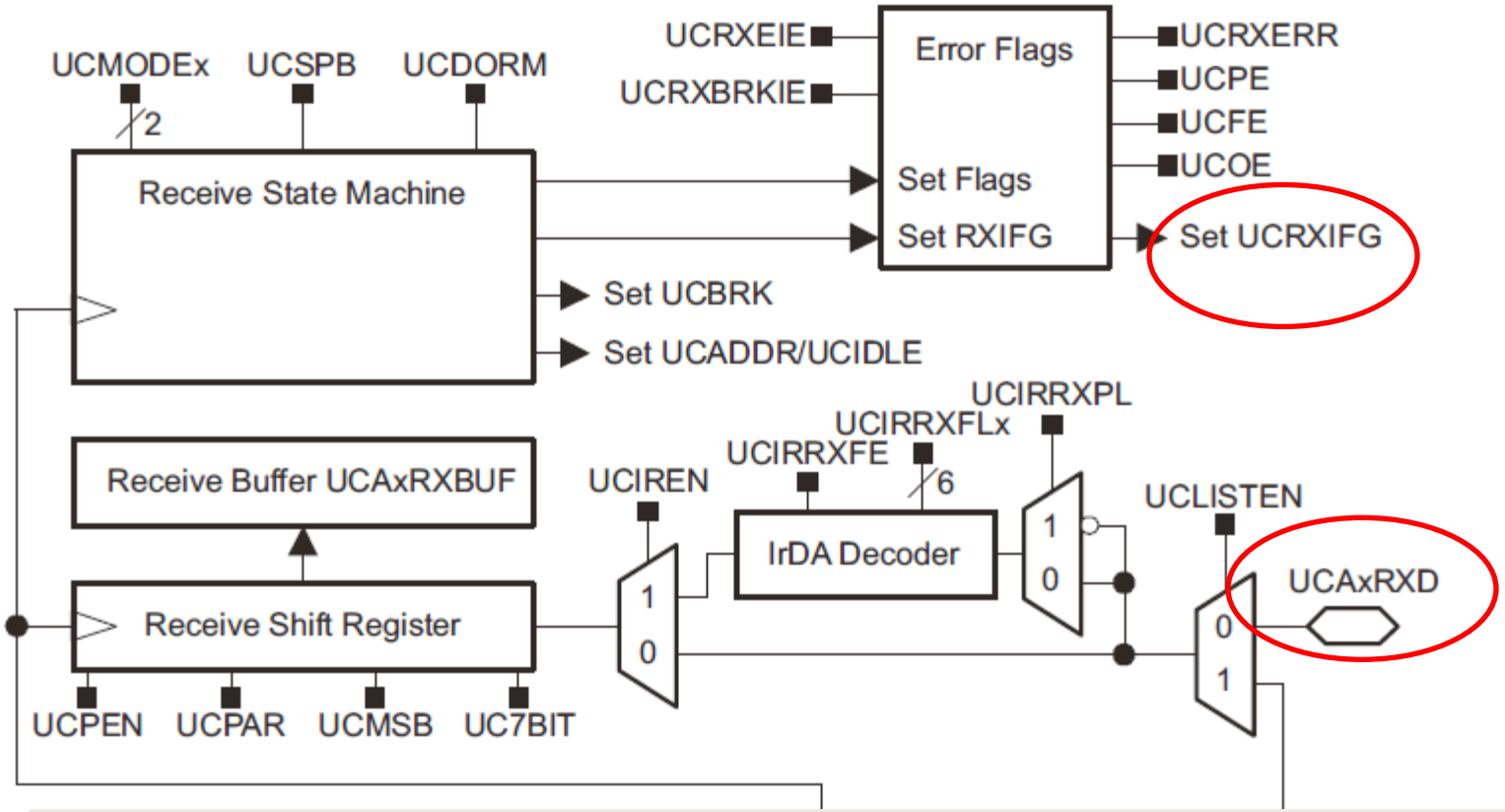
1. Demo-UART-usingOutChar
   - Uses UART0.c and UART1.c
     - UART0.c is polled and uses A0
     - UART1.c is interrupt-driven and uses A2
   - Does not use printf. Instead using OutChar()
   - Found under Keil_v5/MSP432ValvanoWare/inc
2. Demo-UART-print and scan
   - Uses UART.c
   - Uses printf and scanf
   - Found under c:\Documents\ValvanoWare as printf and scanf
   - Requires COMPLETE copy of the PROJECT file because building from a new project does not work. Don't know why … yet

Roadmap

- As usual, we will learn the hardware in two stages so that we can get going quickly in the lab, yet still understand the hardware at the end

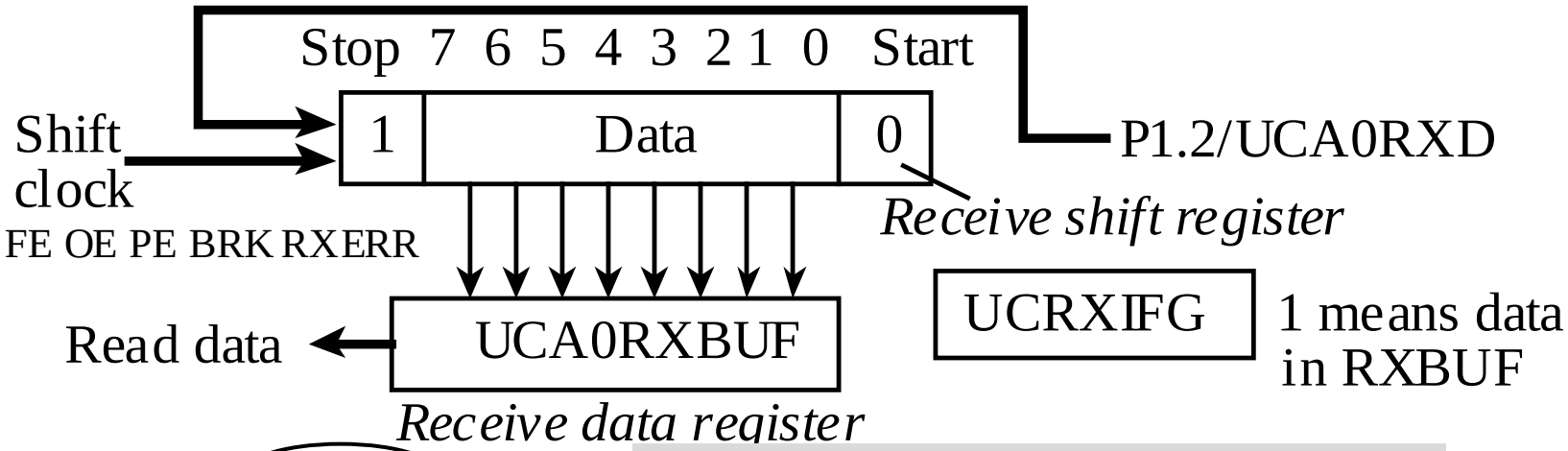
1. Valvano's UART drivers
2. UART theory

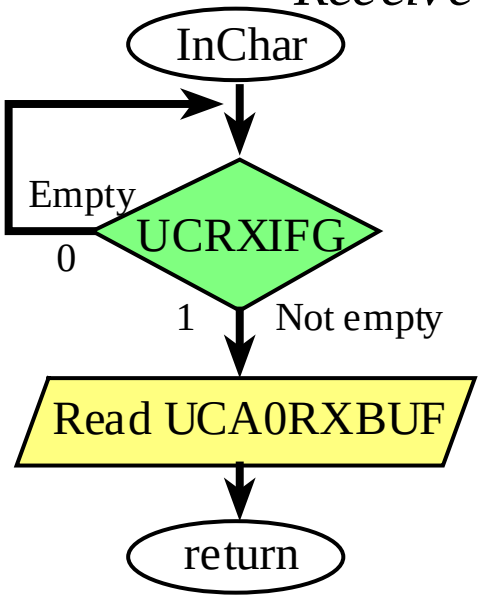
Figure 24-1 TRM – Receive Portion



- Shift Register is not accessible by programmer
- UCRXIFG is 1 when Buffer is Full (A full frame has been received)
- The Buffer is READ-only
- Bits shifted in from LSB to MSB
- Many errors are possible upon receive largely due to timing mismatches in the Tx:Rx clocks: overrun (when data is coming too quickly), parity, framing (stop bit is incorrect)

UART - Receiver

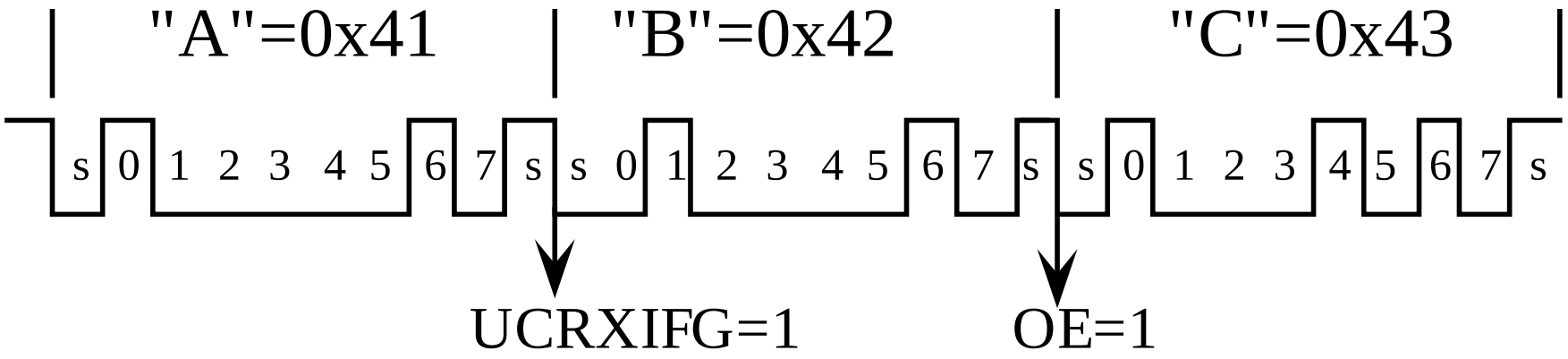




- Logic for receiving is typical: interrupt/flag occurs when the receiving is complete and data is ready.
- Software must read the character from the BUFFER before the next frame is completely shifted in
  - There is some limited concurrency offered : Can be reading previous byte while next byte is being shifted in.

Bill Bard, Andreas Gerstlauer, Jon Valvano, Ramesh Yerraballi

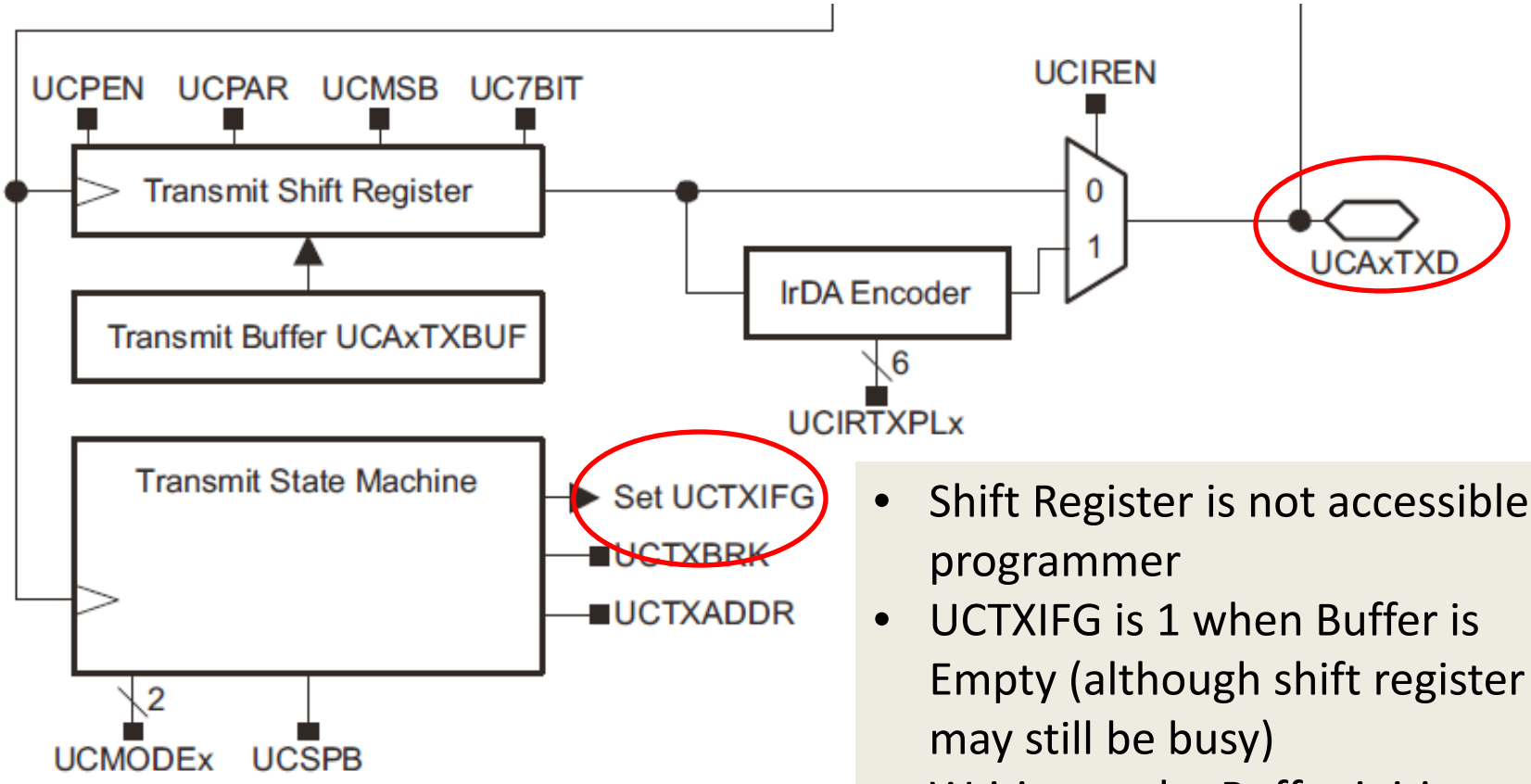
UART – Overrun Error



3 frames transmitted and none read => overrun error

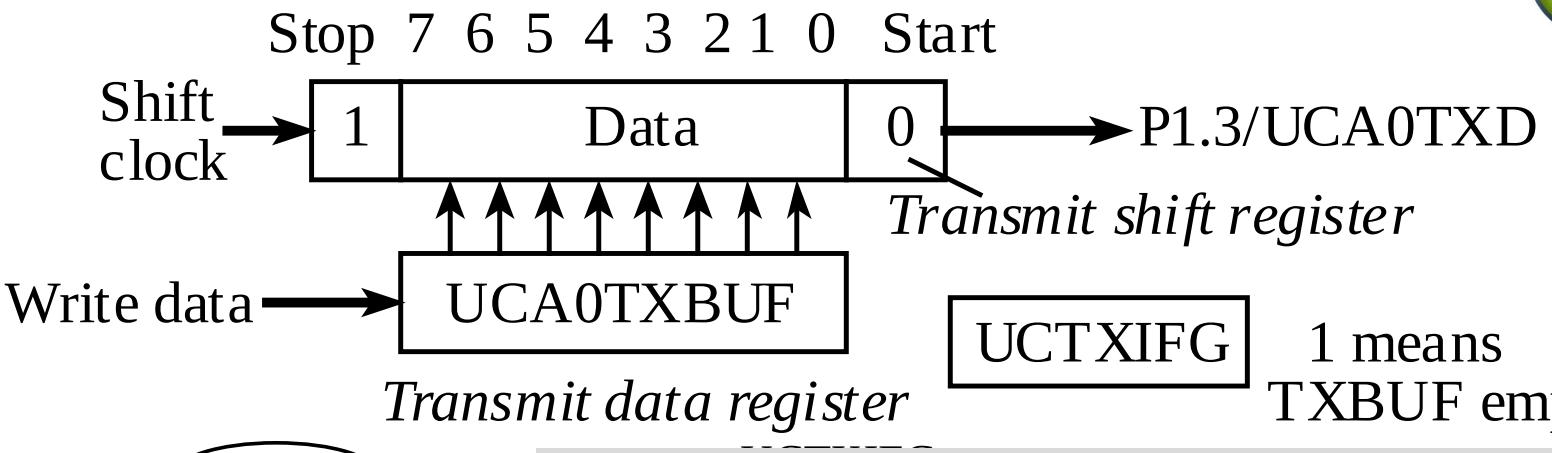
Bill Bard, Andreas Gerstlauer, Jon Valvano, Ramesh Yerraballi

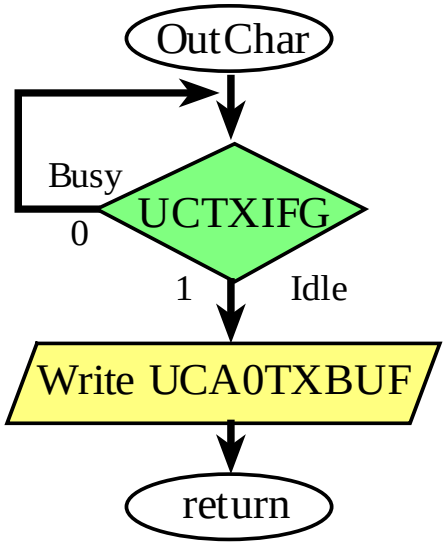
Figure 24-1 TRM – Transmit Portion



- Shift Register is not accessible by programmer
- UCTXIFG is 1 when Buffer is Empty (although shift register may still be busy)
- Writing to the Buffer initiates a new transmission (when done previous byte)
- Bits shifted out from LSB to MSB

UART - Transmitter





- Logic for transmission is inverted (respective to the receive-logic)
- Software waits until hardware is not busy to initiate a character-transmission (frame)
  - BUFFER is copied into Shift Register
- Software can then proceed to other tasks (while transmission occurs)
  - Does not need to wait until complete
  - Unless, next task is to write another frame
- Next frame can be written to BUFFER but if there is still data in the shift register, it will wait until the first is finished before starting

UART is full-duplex

- The Transmit module (shift register and buffer) and the Receive module (shift register and buffer) are completely separate
- Simultaneous transmission and reception.
  - Makes for some interesting programming!
- But first, we need to configure the UART

MSP432 UART – Registers

	15	14	13	12	11	10	9	8	
0x40001000	PEN	PAR	MSB	7BIT	SPB	MODEx		SYNC	UCAxCTLW0
	7	6	5	4	3	2	1	0	
	SSELx		RXEIE	BRKIE	DORM	TXADDR	TXBRK	SWRST	UCAxCTLW0

	15 – 0	
0x40001006	UCBRx	UCAxBRW

	15 – 8	7 – 4	3 – 1	0	
0x40001008	BRSx	BRFx		UCOS16	UCAxMCTLW

	7	6	5	4	3	2	1	0	
0x4000100A	LISTEN	FE	OE	PE	BRK	RXERR	IDLE	BUSY	UCAxSTATW

	15 – 8	7 – 0	
0x4000100C		RXBUFx	UCAxRXBUF

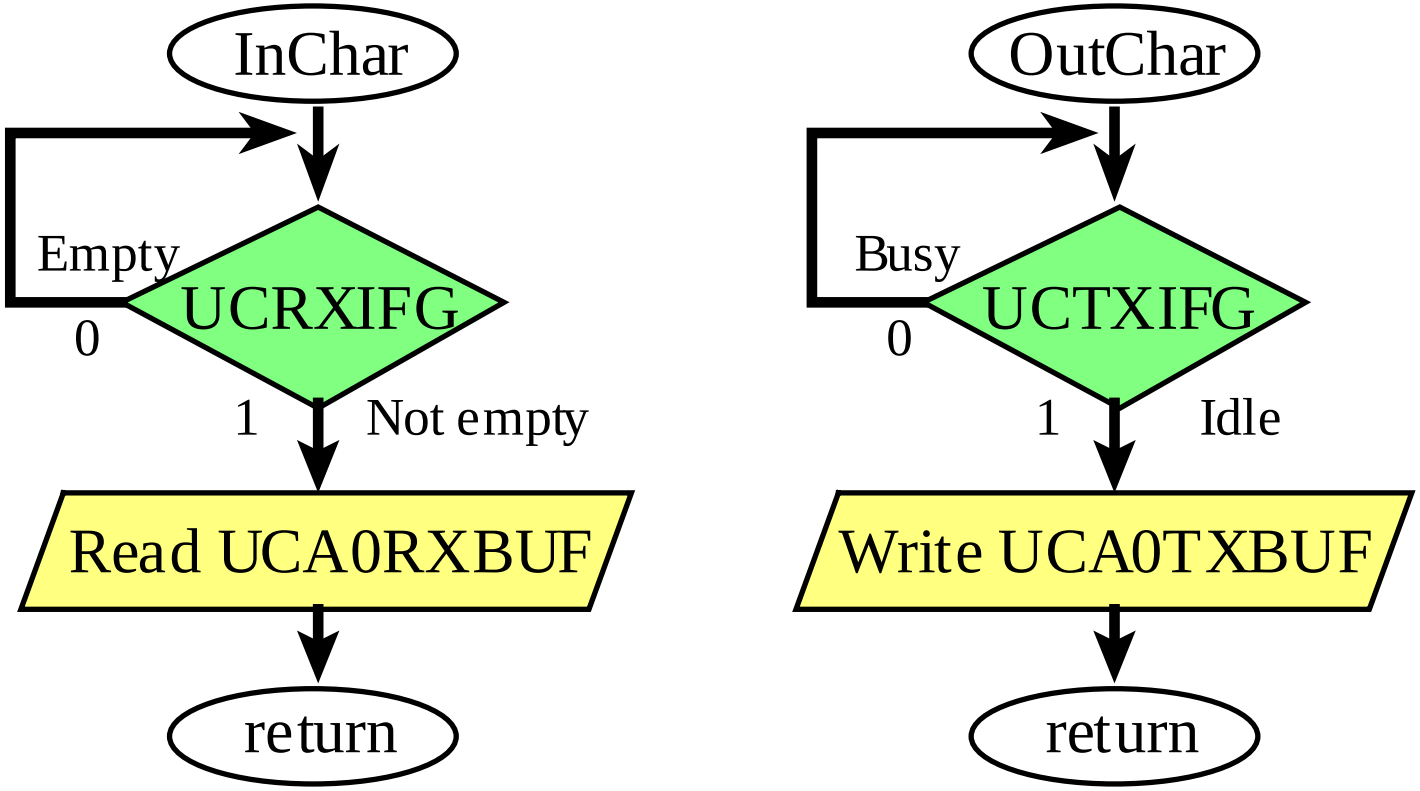
	15 – 8	7 – 0	
0x4000100E		TXBUFx	UCAxTXBUF

	15 – 4	3	2	1	0	
0x4000101A		TXCPTIE	STTIE	TXIE	RXIE	UCAxIE

	15 – 4	3	2	1	0	
0x4000101C		TXCPTIFG	STTIFG	TXIFG	RXIFG	UCAxIFG

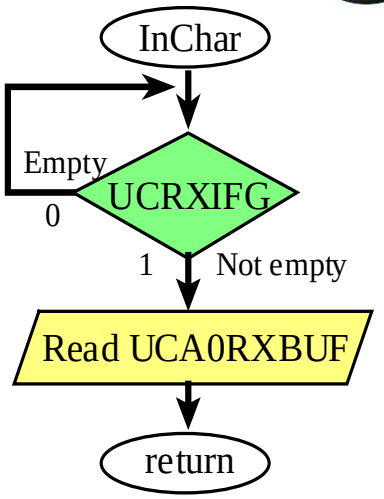
UART Synchronization

- Busy-wait operation

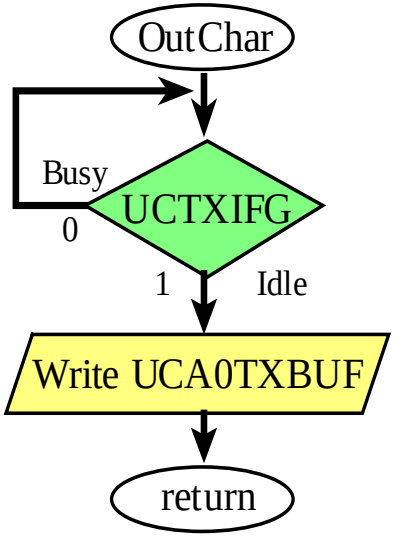


UART Busy-Wait Send/Recv

```
// Wait for new input,
// then return ASCII code
char UART_InChar(void){
  while((UCA0IFG&0x01) == 0);
  return((char)(UCA0RXBUF));
}
```




```
// Wait for buffer to be not full,
// then output
void UART_OutChar(char data){
  while((UCA0IFG&0x02) == 0);
  UCA0TXBUF = data;
}
```



Buffers

- Before we continue to consider interrupt-driven software for the UART, let's introduce the use of buffers common to UARTs
- Why? Typically, we don't send one character, we send a string of characters
- At the simplest, **a buffer is an array**

Simple UART Receiver

- For simplicity, consider only the Receiver of our UART (TX later)
- Describe the behaviour of this program (perhaps using a CPU utilization diagram). What's the flaw?

```
main()
{
  ; Init Hardware
  ; Install ISR
  ; Enable local interrupts
  ; Enable global interrupts
  for (;;)   {
    print data;
  }
```

```
char data;
```

Global
Shared
Data

```
EUSCIA0_IRQHandler

 data = UCA0RXBUF;
```

Simple UART Receiver

- Synchronization between the foreground and background thread is provided by a special value (a flag) of the global variable

```
main()
{
  ; Init Hardware
  ; Install ISR
  ; Enable local interrupts
  ; Enable global interrupts
  for (;;)   {
    while (data== $FF) {}
    print data;
    data= $FF
  }
```

```
char data=0xFF;
```

Global
Shared
Data

```
EUSCIA0_IRQHandler

 data = UCA0RXBUF;
```

- Is this any different than polling? What is the advantage of the interrupt-driven strategy?

Buffered UART Receiver

- Synchronization between the foreground and background thread is loosened by a buffer
- In this first pass: Ignore the wraparound of the buffer

```
main()
{
  ; Init Hardware
  ; Install ISR
  ; Enable local interrupts
  ; Enable global interrupts
  for (;;)   {
    while (head == tail) {}
    print data[tail++];
  }
```

```
char data[10];
int head=0;
int tail=0;
```

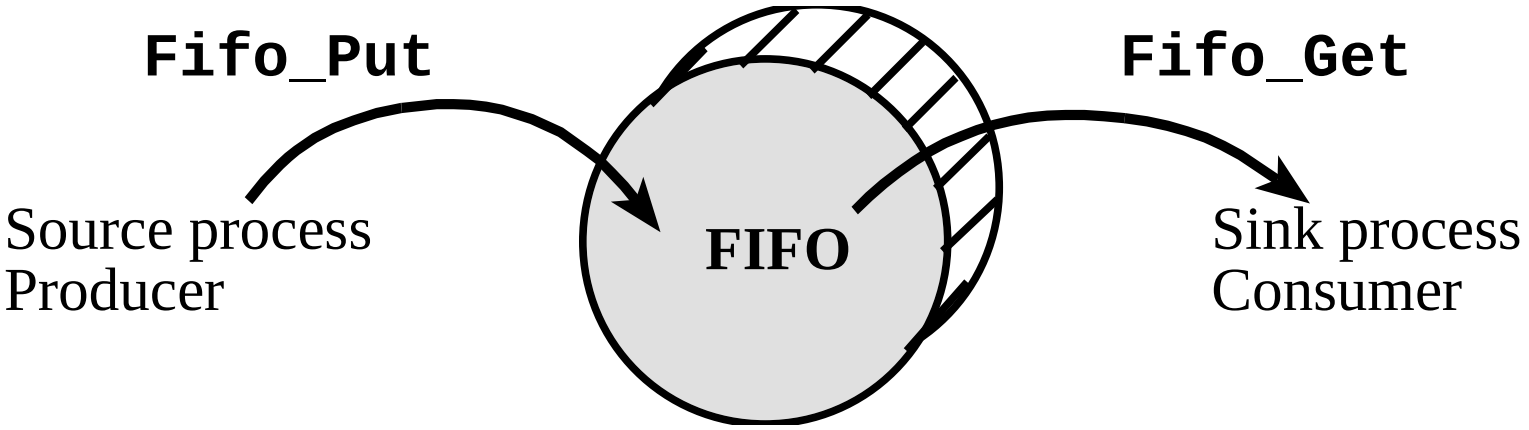
Global *Shared* Data

```
EUSCIA0_IRQHandler

 data[head++] =
        UCA0RXBUF;
```

- Now, main() must read the characters before the entire buffer is full (not after each character reception)
- The degree of de-coupling is relative to the size of the buffer

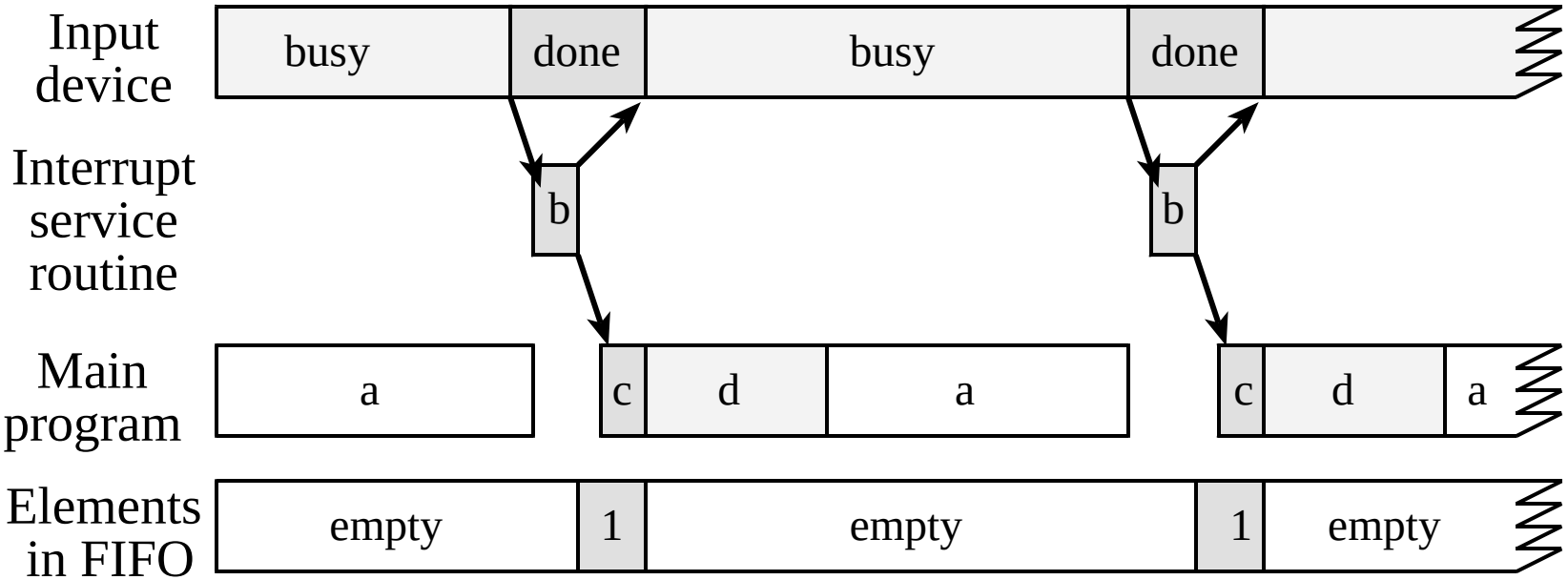
First-In/First-Out (FIFO) Queues



- Order preserving
- Producer(s) put (on tail end)
- Consumer(s) get (from head end)
- Buffer decouples producer & consumer
  - Even out temporary mismatch in rates

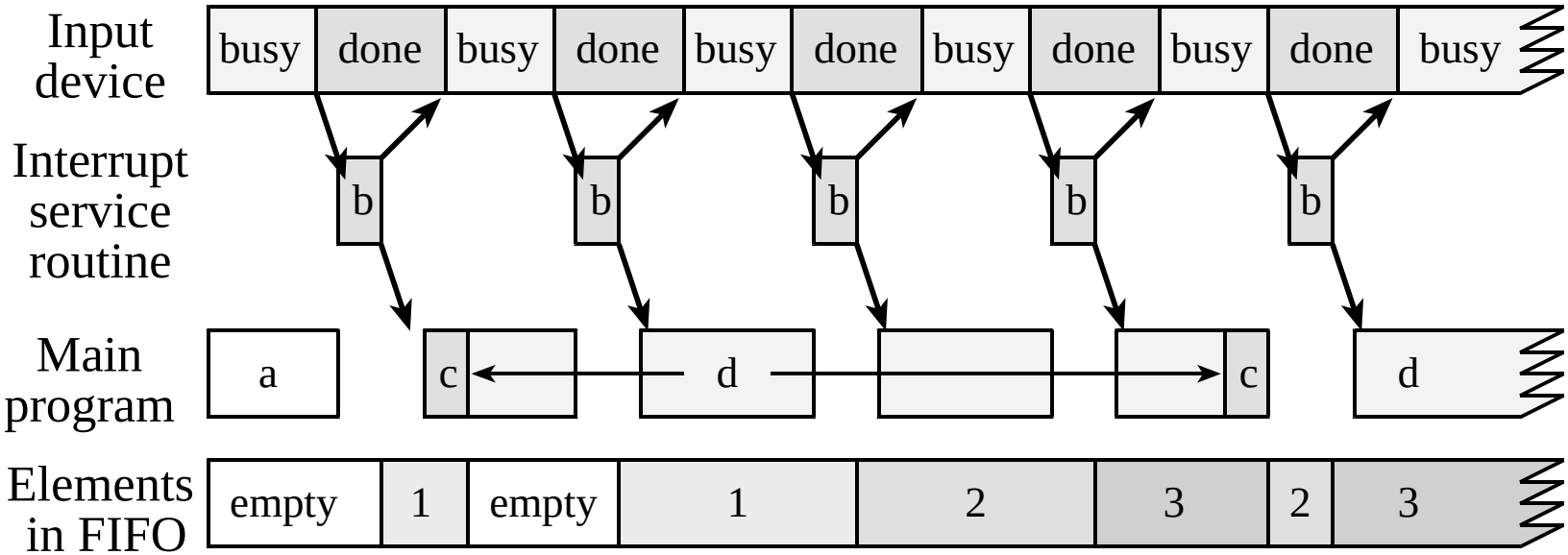
FIFO Operation

- I/O bound input interface



FIFO Operation

- High bandwidth input burst



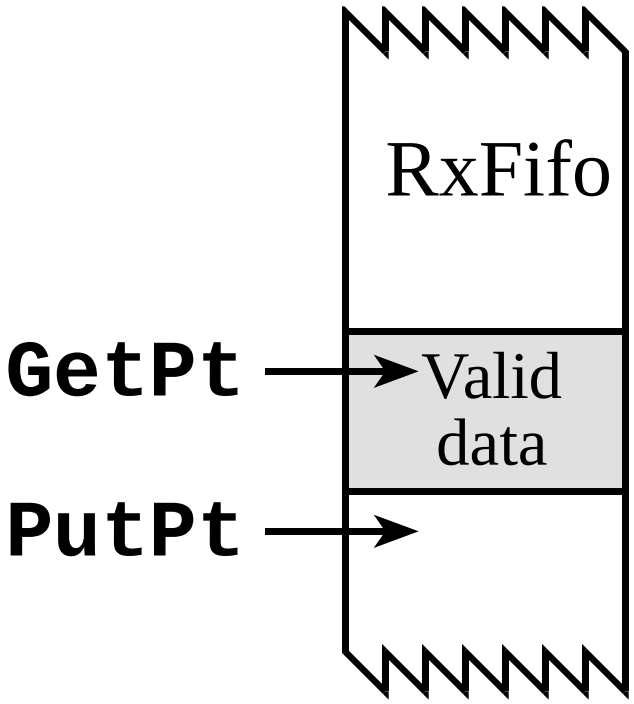
FIFO Queue Implementation

- How is memory allocated?
  - FIFO implies that we write new data at the ***head*** of the queue and we read data from the ***tail*** of the queue
  - What problem does this cause?
- To address that problem the queue is operated in a circular manner
  - An array of locations is processed so that the FIRST element of array appears to follow the LAST element of the array

FIFO Queue Implementation

- Two parameters are needed to specify the FIFO buffer (array)
  - FIRST (usually index 0)
    - points to the first location of the buffer
  - LIMIT (usually index SIZE)
    - points to ***last location+1*** of the buffer
    - ***since LIMIT is outside the buffer, data must not be written there***
  - FIRST and LIMIT are constants
    - they are NOT modified by either the source or sink processing routines

FIFO Queue Implementation

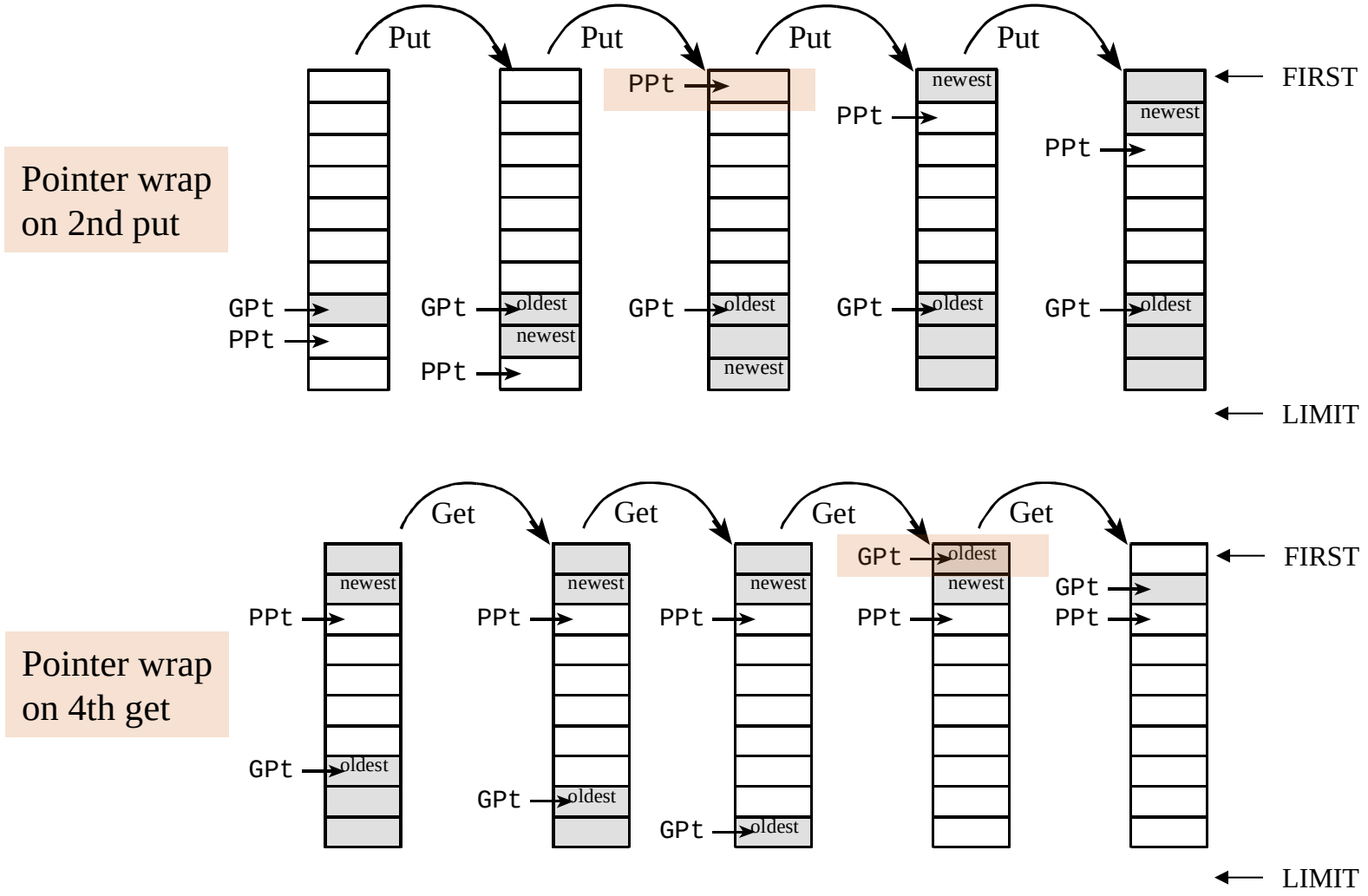


- PutPt: Points to the location where the next element to be added goes
- GetPt: Points to the location of the oldest valid element, hence the element to be removed first

FIFO Full/Empty Conditions

- FIFO Parameter Relations
  - Buffer is EMPTY
    - PutPt equals GetPt
  - Buffer is FULL
    - PutPt + 1 equals GetPt
      - note that there is no data stored at PutPt
      - as a result, if N locations are allocated for a buffer, only N-1 data elements will fill the buffer

FIFO Wrapping



FIFO Queue

- FIFO Implementations
  - FIFO_Put
    - stores a single value on the FIFO queue
      - operates with interrupts disabled
      - updates PutPt
        - ✓ detects buffer full condition
      - handles transition from LIMIT-1 to FIRST
  - FIFO_Get
    - reads a single value from the FIFO queue
      - operates with interrupts disabled
      - updates GetPt
        - ✓ detects buffer empty condition
      - handles transition from LIMIT-1 to FIRST

FIFO in C


```
#define FIFO_SIZE 10
int32_t static *PutPt;
int32_t static *GetPt;
int32_t static Fifo[FIFO_SIZE];

void Fifo_Init(void){
  PutPt = GetPt = &Fifo[0];
}
```

static means private to this file

FIFO Routines in C


```
int Fifo_Put(int32_t data)
{
int32_t *tempPt;
  tempPt = PutPt+1;    // see if there is room
  if(tempPt==&Fifo[FIFO_SIZE]){
    tempPt = &Fifo[0];
  }
  if(tempPt == GetPt){
    return(0);         // full!
  }
  else{
    *(PutPt) = data;   // save
    PutPt = tempPt;    // OK
    return(1);
  }
}
```

Actually plus 4 bytes

FIFO Routines in C

```
int Fifo_Get(int32_t *datapt){

  if(PutPt == GetPt){
    return(0);   // Empty
  }
  else{
    *datapt = *(GetPt++);
    if(GetPt==&Fifo[FIFO_SIZE]){
       GetPt = &Fifo[0];
    }
    return(1);
  }
}
```

Actually plus 4 bytes

FIFO Full Errors

Average producer rate exceeds the average consumer rate

- Sample ADC every 50 ms
- Average time to process the sample is 51 ms
- Solution: decrease producer rate or increase consumer rate
  - Lower sampling rate
  - Faster computer
  - More efficient compiler
  - Rewrite time-critical code in assembly
  - More computers (distributed processing)

Producer rate temporarily exceeds the consumer rate

- Sample ADC every 50 ms
- Every 100th sample it takes 1 sec to process
- Solution: increase FIFO queue size

Back to UART TX Interrupts

- Let’s repeat the discussion for TX interrupts

Buffered UART Transmitter

- The main thread is now the producer and the ISR is the consumer
- Again, for simplicity, let's ignore wrap-around for now, so that we focus on the interrupt-issues

```
main()
{
  ; Init Hardware
  ; Install ISR
  ; Enable local interrupts
  ; Enable global interrupts
  for (;;)   {
     // Do some task
     data[head++] = someValue;
  }
```

```
char data[10];
int head=0;
int tail=0;
```

Global *Shared* Data

```
EUSCIA0_IRQHandler

 UCA0TXBUF =
        data[tail++]
```

- Conundrum! When does the TX interrupt occur? When will the ISR run?

TXIFG (and TX Interrupts)

- Writing to the UCAxTXBUF clears UCTXIFG (in the UCAxIFG register)
- UCTXIFG – is set when UCAxTXBUF is empty

  0 – No interrupt pending

  1 – Interrupt Pending
- In other words, an interrupt is pending while the UCAxTXBUF is empty; writing to the UCAxTXBUF removes the interrupt

Buffered UART Transmitter

- The main thread is now the producer and the ISR is the consumer
- Again, for simplicity, let's ignore wrap-around for now, so that we focus on the interrupt-issues

```
char data[10];
int head=0;
int tail=0;
```

Global *Shared* Data

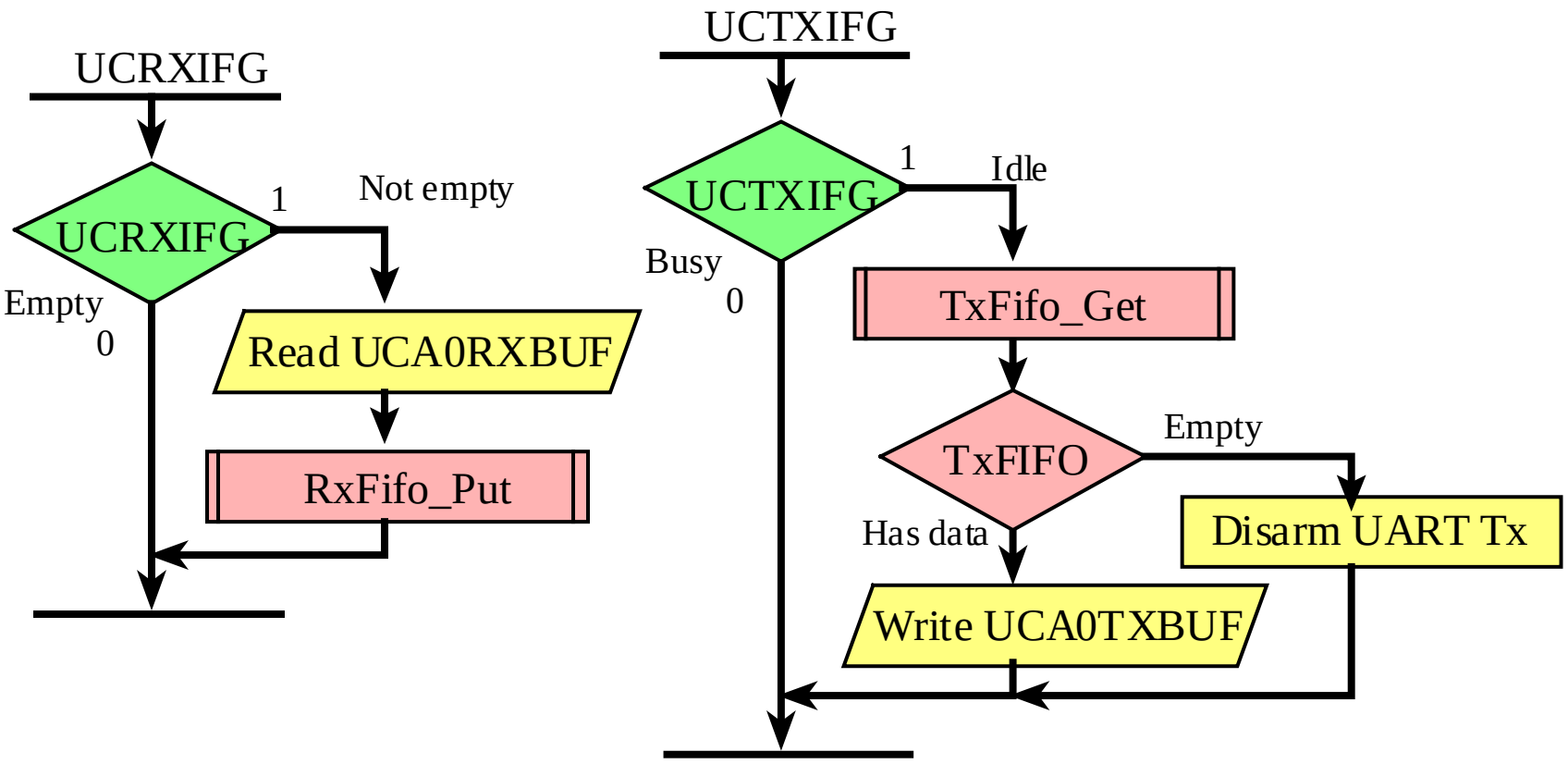
```
main()
{
  ; Init Hardware
  ; Install ISR
  ; Disable TX interrupt
  ; Enable global interrupts
  for (;;)   {
     // Do some task that
     // produces a value
     data[head++] = someValue;
     Enable TX interrupt
  }
```

```
EUSCIA0_IRQHandler

 UCA0TXBUF =
        data[tail++]
 if data[] is now
  empty (no more left)
   Disable TX interrupt
```

- ISR(Consumer) disables TX interrupts when there is no more data to be sent
- Producer re-enables TX interrupts whenever adding data to the queue

UART Interrupts



Full Duplex UART Interrupts

```
// interrupt 16 occurs on either:
// UCTXIFG TX data register is empty
// UCRXIFG RX data register is full
// vector at 0x00000080 in startup_msp432.s
void EUSCIA0_IRQHandler(void){ char data;
  if(UCA0IFG&0x02){           // TX data register empty
    if(TxFifo_Get(&data) == FIFOFAIL){
      UCA0IE = 0x0001;        // disable interrupts on transmit empty
    }else{
      UCA0TXBUF = data;       // send data, acknowledge interrupt
    }
  }
  if(UCA0IFG&0x01){           // RX data register full
    RxFifo_Put((char)UCA0RXBUF);// clears UCRXIFG
  }
}
```

Thought Exercise: What if we used else-if?

UART Setup


```
// Assumes a 3MHz bus clock, creates 115200 baud rate
void UART_Init(void){
  RxFifo_Init();       // initialize FIFOs
  TxFifo_Init();
  UCA0CTLW0 = 0x0001;  // hold in reset mode
  UCA0CTLW0 = 0x00C1;  // UART,SMCLK, 8bit, no parity,
  UCA0BRW = 26;        // = 3000000/115200 = 26.0417
  UCA0MCTLW = 0x0000;  // first and second, UCOS16=0
  UCA0IE = 0x0001;     // enable interrupts on receive
// disable interrupts on transmit, start, complete
  P1SEL0 |= 0x0C;
  P1SEL1 &= ~0x0C;   // P1.3 and P1.2 as primary module
  NVIC_IPR4 = (NVIC_IPR4&0xFFFFFF00)|0x00000040; // 2
  NVIC_ISER0 = 0x00010000; // enable inte 16 in NVIC
  UCA0CTLW0 &= ~0x0001;    // enable the USCI module
}
```

Race Conditions

- A **race condition** or **race hazard** is the behavior of an electronics, software, or other system where the output is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when events do not happen in the order the programmer intended. [Wikipedia]
- In software, races happen in critical sections

Where is the Critical Section
What is the race condition?

```
main()
{
  ; Init Hardware
  ; Install ISR
  ; Enable local interrupts
  ; Enable global interrupts
  for (;;)   {
    while (data== $FF) {}
    print data;
    data= $FF
  }
```

```
char data=0xFF;
```

Global
Shared
Data

```
EUSCIA0_IRQHandler

 data = UCA0RXBUF;
```

- Is this any different than polling? What is the advantage of the interrupt-driven strategy?

Where is the Critical Section

- Synchronization between the foreground and background thread is loosened by a buffer
- In this first pass: Ignore the wraparound of the buffer

```
main()
{
  ; Init Hardware
  ; Install ISR
  ; Enable local interrupts
  ; Enable global interrupts
  for (;;)   {
    while (head == tail) {}
    print data[tail++];
  }
```

```
char data[10];
int head=0;
int tail=0;
```

Global *Shared* Data

```
EUSCIA0_IRQHandler

 data[head++] =
        UCA0RXBUF;
```

- Now, main() must read the characters before the entire buffer is full (not after each character reception)
- The degree of de-coupling is relative to the size of the buffer