Week 9. Interrupts & Serial Communication

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi
INI & ECE
Carnegie Mellon University

CarnegieMellon

Electrical & Computer
ENGINEERING

Overview

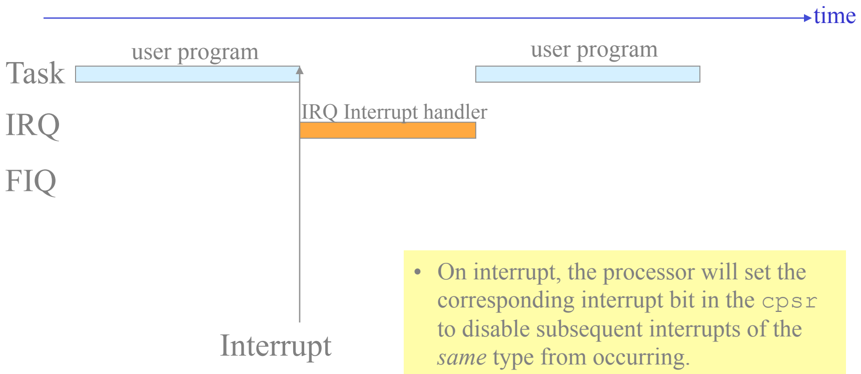
- Interrupts
  - IRQs vs. FIQs
  - Interrupt Controller
- Interrupt latency
  - Nested vs. Non-nested interrupt controllers
- Concurrency issues with Interrupt handlers
- Serial Communications
  - Asynchronous protocols
  - Synchronous protocols
  - RS-232 data interface
  - Parity bits
  - Serial port and bit transmission

Polling vs. Interrupt-Driven I/O

- Polling requires code to loop until device is ready
  - Consumes *lots* of CPU cycles
  - Can provide quick response (guaranteed delay)
- Interrupts don't require code to loop until the device is ready
  - Device interrupts processor when it needs attention
  - Code can go off and do other things
  - Interrupts can happen at any time
    - Requires careful coding to make sure other programs (or your own) don't get messed up
- What do you think real-time embedded systems use?

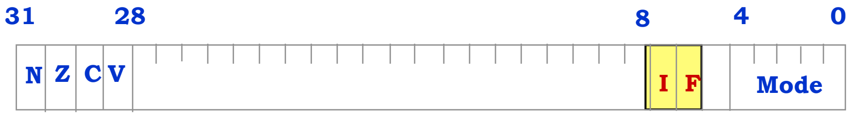
Onto IRQs & FIQs: Interrupt Handlers

- When an interrupt occurs, the hardware will jump to an interrupt handler or interrupt service routine (ISR)



- On interrupt, the processor will set the corresponding interrupt bit in the `cpsr` to disable subsequent interrupts of the *same* type from occurring.
- However, interrupts of a *higher priority* can still occur.

`cpsr` & `spsr` for IRQs and FIQs

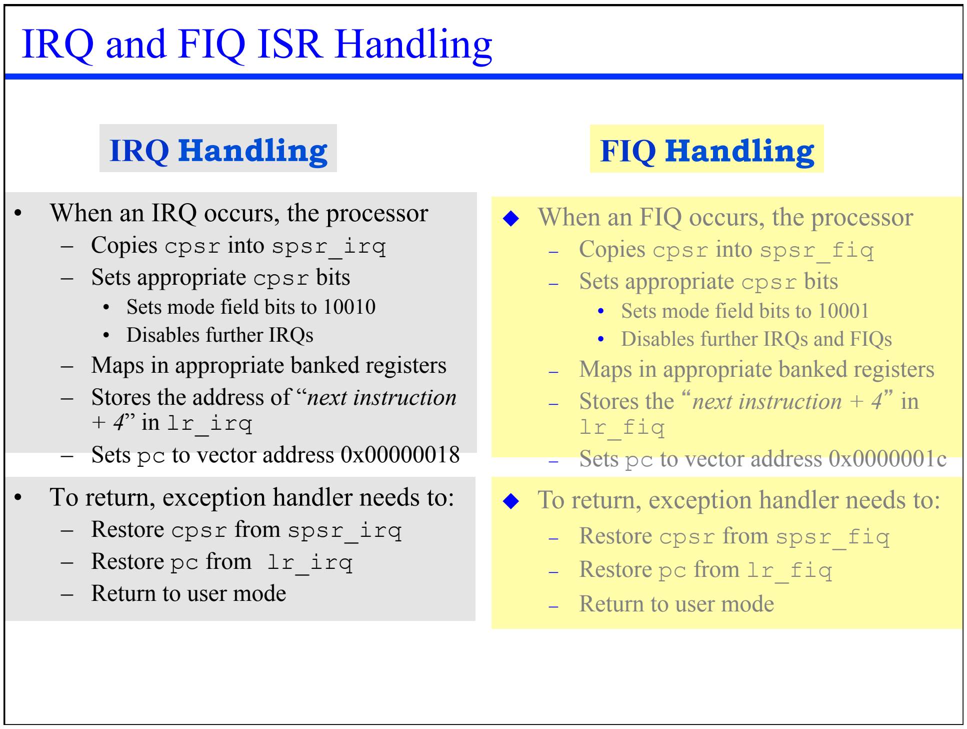


- Interrupt Disable bits
  - I = 1, *disables* the IRQ
  - F = 1, *disables* the FIQ
- Mode bits
  - Processor mode differs

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10111	Abort
11011	Undef
11111	System

Exception Priorities

Exceptions	Priority	I bit (1⇨IRQ Disabled)	F bit (1⇨FIQ Disabled)
Reset	1 (highest)	1	1
Data Abort	2	1	
Fast Interrupt Request (FIQ)	3	1	1
Interrupt Request (IRQ)	4	1	
Prefetch Abort	5	1	
Software Interrupt	6	1	
Undefined Instruction	6 (lowest)	1	



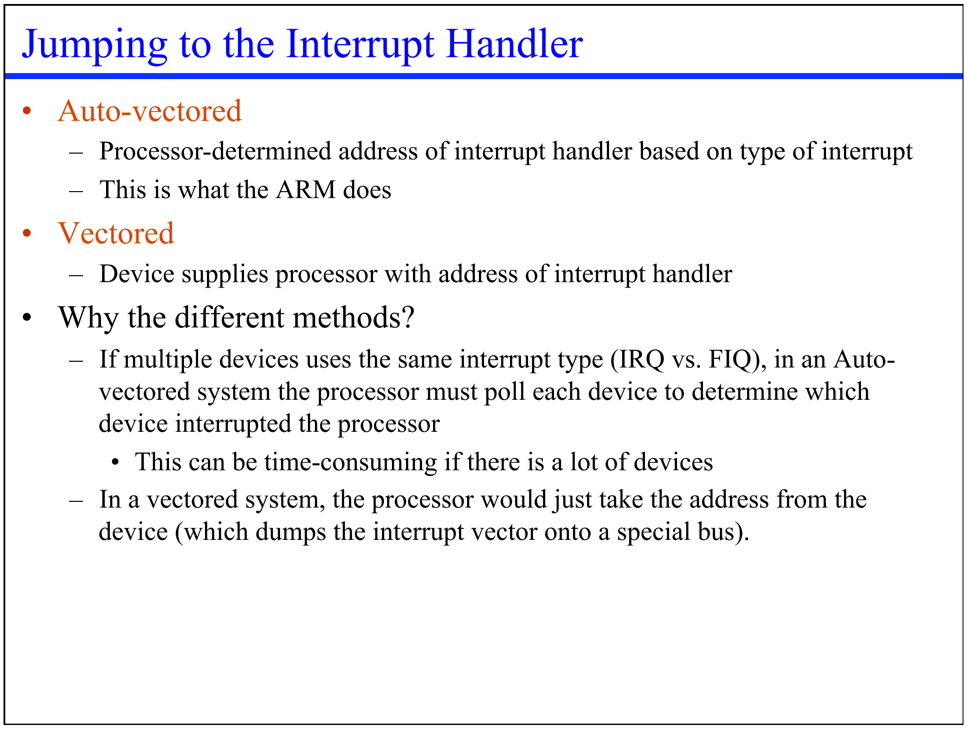
IRQ and FIQ ISR Handling

IRQ Handling

- When an IRQ occurs, the processor
  - Copies `cpsr` into `spsr_irq`
  - Sets appropriate `cpsr` bits
    - Sets mode field bits to 10010
    - Disables further IRQs
  - Maps in appropriate banked registers
  - Stores the address of "*next instruction + 4*" in `lr_irq`
  - Sets `pc` to vector address 0x00000018
- To return, exception handler needs to:
  - Restore `cpsr` from `spsr_irq`
  - Restore `pc` from `lr_irq`
  - Return to user mode

FIQ Handling

- ◆ When an FIQ occurs, the processor
  - Copies `cpsr` into `spsr_fiq`
  - Sets appropriate `cpsr` bits
    - Sets mode field bits to 10001
    - Disables further IRQs and FIQs
  - Maps in appropriate banked registers
  - Stores the "*next instruction + 4*" in `lr_fiq`
  - Sets `pc` to vector address 0x0000001c
- ◆ To return, exception handler needs to:
  - Restore `cpsr` from `spsr_fiq`
  - Restore `pc` from `lr_fiq`
  - Return to user mode



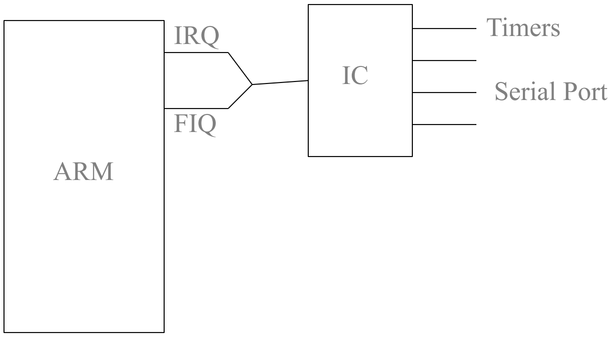
Jumping to the Interrupt Handler

- Auto-vectored
  - Processor-determined address of interrupt handler based on type of interrupt
  - This is what the ARM does
- Vectored
  - Device supplies processor with address of interrupt handler
- Why the different methods?
  - If multiple devices uses the same interrupt type (IRQ vs. FIQ), in an Auto-vectored system the processor must poll each device to determine which device interrupted the processor
    - This can be time-consuming if there is a lot of devices
  - In a vectored system, the processor would just take the address from the device (which dumps the interrupt vector onto a special bus).

Recall: Controlling the Timer on Gumstix

- The following memory-mapped registers can be used to control the OS timer from software
  - **OSCR0**: The OS Timer Count register
    - is a 32-bit count-up counter that increments on the rising edge of the clock (3.25 MHz clock)
  - **OSMRx**: The OS Timer Match register(#) is a set of four registers
    - Each register is a 32-bit read-write register that holds a timer counter target and are matched against OSCR0
  - **OSSR:** The OS Timer Status register
    - holds bits indicating that a match had occurred between the OSCR and the corresponding OSMR
  - **OIER:** The OS Timer Interrupt Enable register has four non-reserved bits

Interrupt Controller



Gumstix Interrupt Controller

- Interrupt Controller Memory-mapped registers–
  - There are 6 memory-mapped registers which can be used to determine the peripheral devices that have raised an interrupt , mask interrupts and configure which sources will cause and IRQ or an FIQ

- Interrupt Controller Level Register 0x40D00008

```
bit 29   if set indicates OSMR3=OSCR generates FIQ otherwise IRQ
bit 28   if set indicates OSMR2=OSCR generates FIQ otherwise IRQ
bit 27   if set indicates OSMR1=OSCR generates FIQ otherwise IRQ
bit 26   if set indicates OSMR0=OSCR generates FIQ otherwise IRQ
```

- Interrupt Controller Mask Register

```
Memory mapped address                   0x40D00004
  bit 29      if set=0 masks OSMR3=OSCR interrupts
  bit 28      if set=0 masks OSMR2=OSCR interrupts
  bit 27      if set=0 masks OSMR1=OSCR  interrupts
  bit 26      if set=0 masks OSMR0=OSCR  interrupts
```

Registers on Gumstix Interrupt Controller

- Interrupt Controller IRQ Pending Register 0x40D00000

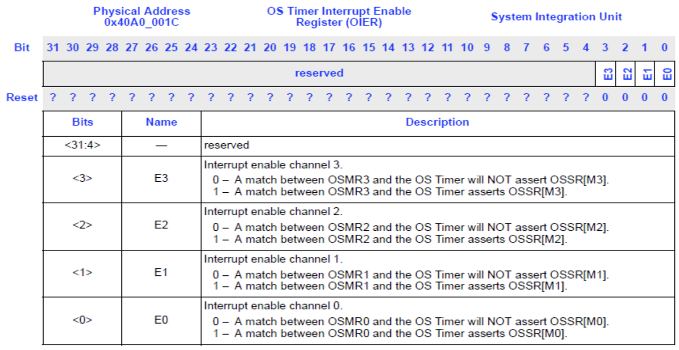
```
bit 29   if set indicates OSMR3=OSCR is not masked and caused IRQ
bit 28   if set indicates OSMR2=OSCR is not masked and caused IRQ
bit 27   if set indicates OSMR1=OSCR is not masked and caused IRQ
bit 26   if set indicates OSMR0=OSCR is not masked and caused IRQ
```

- Interrupt Controller FIQ Pending Register

```
Memory mapped address                   0x40D0000C
bit 29      if set masks OSMR3=OSCR is not masked and caused FIQ
bit 28      if set masks OSMR2=OSCR is not masked and caused FIQ
bit 27      if set masks OSMR1=OSCR is not masked and caused FIQ
bit 26      if set masks OSMR0=OSCR is not masked and caused FIQ
```

Masking/Disabling Interrupts on ARM-Based Embedded Systems

- Three methods of masking/disabling interrupts
  - Disable Interrupts: set I/F bits in CPSR
  - Mask an Interrupt: unset the bits in Interrupt Controller Mask register
  - Don’t use interrupts: configure the peripheral device to not generate interrupts
    - E.g. – Timer interrupt can be disabled through OIER



Physical Address 0x40A0_001C

OS Timer Interrupt Enable Register (OIER)

System Integration Unit

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved																												E3	E2	E1	E0
Reset	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	0	0	0	0

Bits	Name	Description
<31:4>	—	reserved
<3>	E3	Interrupt enable channel 3. 0 – A match between OSMR3 and the OS Timer will NOT assert OSSR[M3]. 1 – A match between OSMR3 and the OS Timer asserts OSSR[M3].
<2>	E2	Interrupt enable channel 2. 0 – A match between OSMR2 and the OS Timer will NOT assert OSSR[M2]. 1 – A match between OSMR2 and the OS Timer asserts OSSR[M2].
<1>	E1	Interrupt enable channel 1. 0 – A match between OSMR1 and the OS Timer will NOT assert OSSR[M1]. 1 – A match between OSMR1 and the OS Timer asserts OSSR[M1].
<0>	E0	Interrupt enable channel 0. 0 – A match between OSMR0 and the OS Timer will NOT assert OSSR[M0]. 1 – A match between OSMR0 and the OS Timer asserts OSSR[M0].

Interrupt Latency

- Interrupt latency
  - Interval of time from an external interrupt-request signal being raised to the first fetch of an instruction of a specific ISR
- Affected by the number of simultaneous interrupt sources to handle
- Contributors to interrupt latency
  - Time taken to recognize the interrupt
  - Time taken by the CPU to complete its current instruction (depends on whether the CPU is executing a multi-cycle or a single-cycle instruction)
  - Time taken by the CPU to perform a context switch (switching in banked registers, saving program counter, etc.)
  - Time taken to fetch the interrupt vector
  - Time taken to start the ISR executing
- For real-time systems, you need to compute worst-case interrupt latency

Minimizing Interrupt Latency

- Nested interrupt handler
  - Allows further interrupts to occur when a current interrupt is being serviced
  - Re-enable interrupts at a safe point in the current ISR
  - Ultimately, nested interrupts roll back to the original ISR
- Prioritization
  - Ignore interrupts of the same/lower priority than the current interrupt
  - Higher-priority tasks can interrupt the current ISR
  - Higher-priority interrupts have a lower average interrupt latency

Types of Interrupt Handlers

- Non-nested interrupt handler (simplest possible)
  - Services individual interrupts sequentially, one interrupt at a time
- Nested interrupt handler
  - Handles multiple interrupts without priority assignment
- Prioritized (re-entrant) interrupt handler
  - Handles multiple interrupts that can be prioritized

Non-Nested Interrupt Handler

- Does not handle any further interrupts until the current interrupt is serviced and control returns to the interrupted task

- Not suitable for embedded systems where interrupts have varying priorities and where interrupt latency matters
  - However, relatively easy to implement and debug

- Inside the ISR (after the processor has disabled interrupts, copied `cpsr` into `spsr_mode`, set the etc.)
  - Save context – subset of the current processor mode's nonbanked registers
  - Not necessary to save the `spsr_mode` – why?
  - ISR identifies the external interrupt source – how?
  - Service the interrupt source and reset the interrupt
  - Restore context
  - Restore `cpsr` and `pc`

Nested Interrupt Handler

- Allows for another interrupt to occur within the currently executing handler
  - By re-enabling interrupts at a *safe point* before ISR finishes servicing the current interrupt

- Care needs to be taken in the implementation
  - Protect context saving/restoration from interruption
  - Check stack
  - Increases code complexity, but improves interrupt latency

- Does not distinguish between high and low priority interrupts
  - Time taken to service an interrupt can be high for high-priority interrupts

Prioritized (Re-entrant) Interrupt Handler

- Allows for higher-priority interrupts to occur within the currently executing handler
  - By re-enabling higher-priority interrupts within the handler
  - By disabling all interrupts of lower priority within the handler
- Same care needs to be taken in the implementation
  - Protect context saving/restoration from interruption, check stack overflow
- Does distinguish between high and low priority interrupts
  - Interrupt latency can be better for high-priority interrupts

Interrupts and Stacks

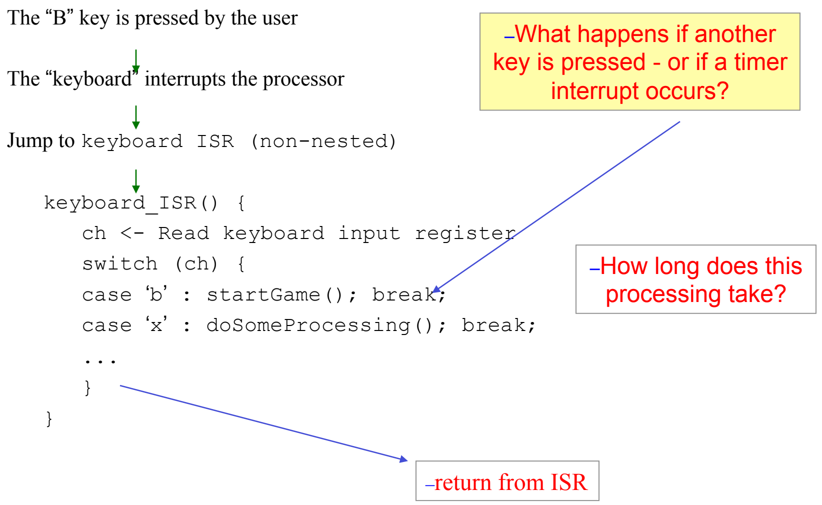
- Stacks are important in interrupt handling
  - Especially in handling nested interrupts
  - Who sets up the IRQ and FIQ stacks and when?
- Stack size depends on the type of ISR
  - Nested ISRs require more memory space
  - Stack grows in size with the number of nested interrupts
- Good stack design avoids stack overflow (where stack extends beyond its allocated memory) – two common methods
  - Memory protection
  - Call stack-check function at the start of each routine
- Important in embedded systems to know the stack size ahead of time (as a part of the designing the application) – why?

Resource Sharing Across Interrupts

- Interrupts can occur asynchronously
- Access to shared resources and global variables must be handled in a way that does not corrupt the program
- Normally done by masking interrupts before accessing shared data and unmasking interrupts (if needed) afterwards
  - Clearly, when interrupt-masking occurs, interrupt latency will be higher
- Up next – start with a simple keyboard ISR and then understand
  - What can happen when the ISR takes a while to execute
  - How do we improve its interrupt latency
  - What can go wrong

Starting With a Simple Example

- Keyboard command processing



The “B” key is pressed by the user

The “keyboard” interrupts the processor

Jump to keyboard ISR (non-nested)

```
   keyboard_ISR() {
      ch <- Read keyboard input register
      switch (ch) {
      case ‘b’ : startGame(); break;
      case ‘x’ : doSomeProcessing(); break;
      ...
      }
   }
```

–What happens if another key is pressed - or if a timer interrupt occurs?

–How long does this processing take?

–return from ISR

Improving Interrupt Latency

- Add a buffer (in software or hardware) for input characters.
  - This decouples the time for processing from the time between keystrokes, and provides a computable upper bound on the time required to service a keyboard interrupt
  - Commands stored in the `input_buffer` can be processed in the user/ application code

Stores the input and then quickly returns to the "main program" (process)

A key is pressed by the user

The "keyboard" interrupts the processor

Jump to `keyboard ISR`

```
keyboard_ISR() {
  *input_buffer++ = ch;
  ...
}
```

return from ISR

Application Code

```
…
…
while (!quit){
   if (*input_buffer){
      processCommand(*input_buffer);
      removeCommand(input_buffer);
   }
}
…
```

What Can Go Wrong? Buffer Processing

- Must be careful when modifying the buffer with interrupts turned on

```
keyboard_ISR(){
    ch <- Read ACIA input register
   *input_buffer++ = ch;
}
```

return from ISR

application code

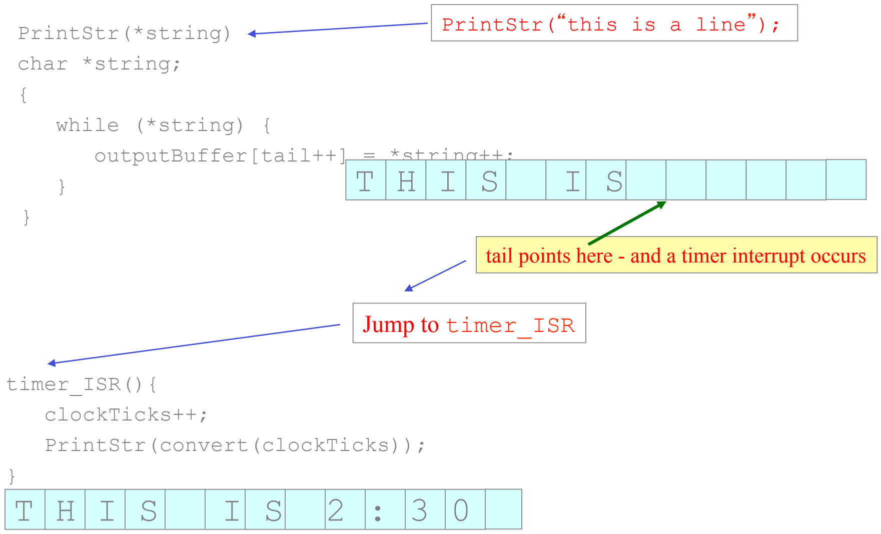
```
…
while (!quit){
   if (*input_buffer){
      processCommand(*input_buffer);
      removeCommand(input_buffer);
   }
}
…
```

What happens if another command is entered as you remove one from the inputBuffer?

Another Concurrency Problem

- An application (another game!) uses the serial port to print characters on the terminal emulator (Hyper Terminal)
  - The application calls a function `PrintStr` to print characters to the terminal
  - In the function `PrintStr`, the characters to be printed are copied into an output buffer (use of output buffer to reduce interrupt latency)
- In the serial port ISR
  - See if there is any data to be printed (whether there are new characters in the output buffer)
  - Copy data from the output buffer to the transmit holding register of the UART
- The (game) display also needs to print the current time on the terminal – a timer is used (in interrupt mode) to keep track of time
- In the timer ISR
  - Compute current time
  - Call `PrintStr` to print current time on the terminal emulator

Another Example: Buffer for Printing Chars to Screen



```
PrintStr(*string)
char *string;
{
   while (*string) {
      outputBuffer[tail++] = *string++;
   }
}
```

`PrintStr("this is a line");`

T H I S   I S

tail points here - and a timer interrupt occurs

Jump to `timer_ISR`

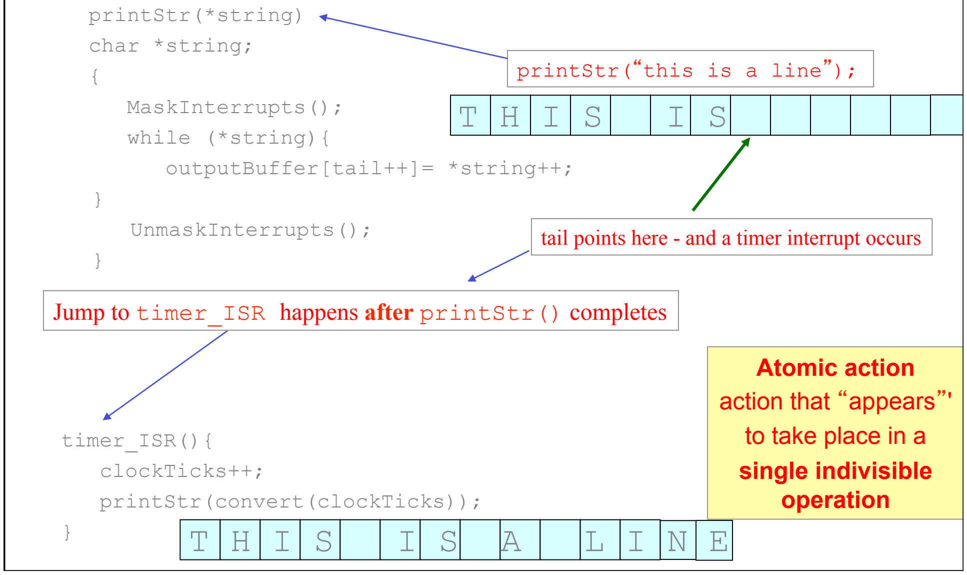
```
timer_ISR(){
   clockTicks++;
   PrintStr(convert(clockTicks));
}
```

T H I S   I S 2 : 3 0

Critical Sections of Code

- Pieces of code that must appear as an **atomic action**

```
printStr(*string)
char *string;
{
   MaskInterrupts();
   while (*string){
      outputBuffer[tail++]= *string++;
   }
    UnmaskInterrupts();
}
```



printStr(“this is a line”);

tail points here - and a timer interrupt occurs

Jump to timer_ISR happens **after** printStr() completes

```
timer_ISR(){
   clockTicks++;
   printStr(convert(clockTicks));
}
```

Atomic action
action that “appears”'
to take place in a
single indivisible operation

Shared-Data Problems

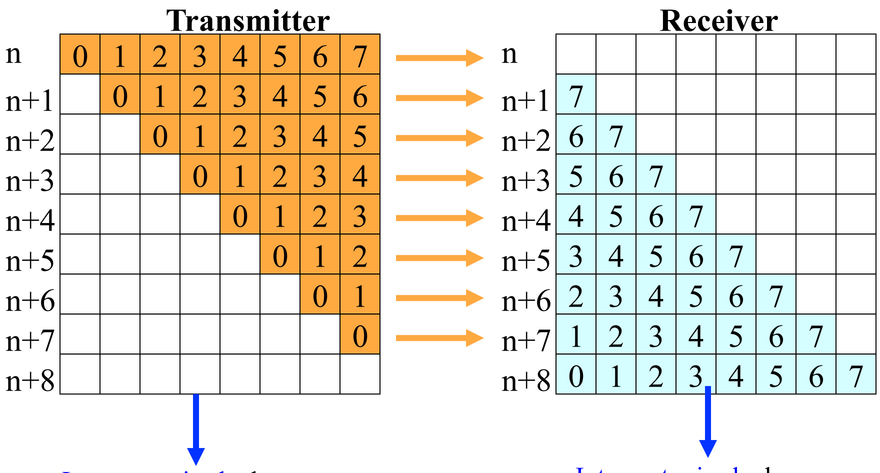
- Previous examples show what can go wrong when data is shared between ISRs and application tasks
- Very hard to find, and debug such concurrency problems (if they exist)
  - Problem may not happen every time the code runs
    - In the previous example, you may not have noticed the problem if the timer interrupt did not occur in the PrintStr function
- Lessons learnt
  - Keep the ISRs short
  - Analyze your code carefully, if any data is shared between ISRs and application code

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

Simplistic View of Serial Port Operation



Transmitter

n	0	1	2	3	4	5	6	7
n+1		0	1	2	3	4	5	6
n+2			0	1	2	3	4	5
n+3				0	1	2	3	4
n+4					0	1	2	3
n+5						0	1	2
n+6							0	1
n+7								0
n+8								

Receiver

n								
n+1	7							
n+2	6	7						
n+3	5	6	7					
n+4	4	5	6	7				
n+5	3	4	5	6	7			
n+6	2	3	4	5	6	7		
n+7	1	2	3	4	5	6	7	
n+8	0	1	2	3	4	5	6	7

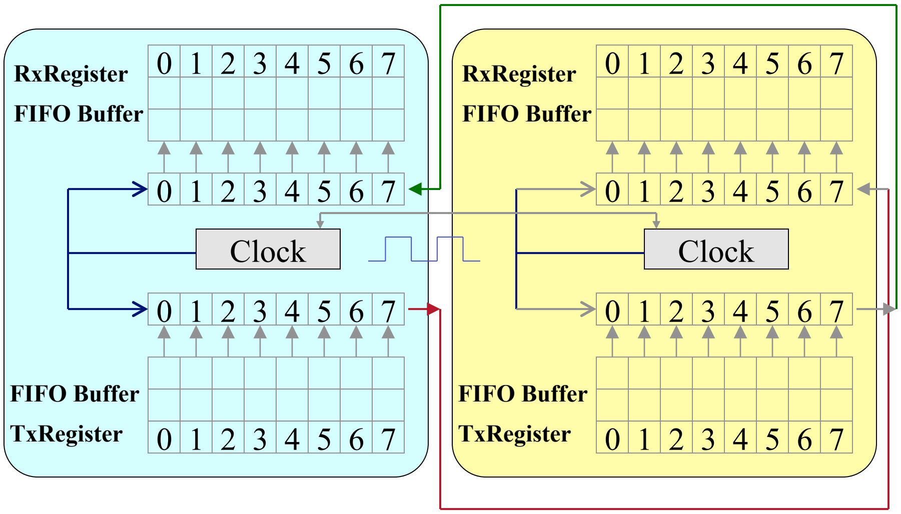
Interrupt raised when
Transmitter (Tx) is empty
⇨ Byte has been transmitted
and next byte ready for loading

Interrupt raised when
Receiver (Rx) is full
⇨ Byte has been received
and is ready for reading

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



RxRegister 0 1 2 3 4 5 6 7

FIFO Buffer

0 1 2 3 4 5 6 7

Clock

0 1 2 3 4 5 6 7

FIFO Buffer

TxRegister 0 1 2 3 4 5 6 7

RxRegister 0 1 2 3 4 5 6 7

FIFO Buffer

0 1 2 3 4 5 6 7

Clock

0 1 2 3 4 5 6 7

FIFO Buffer

TxRegister 0 1 2 3 4 5 6 7

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

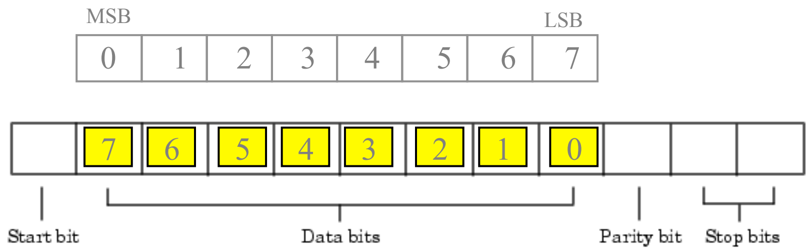
Types of Serial Communications

- 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

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 plus the parity bit produce an even number of 1s
- Odd parity
  - The data bits plus the parity bit 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

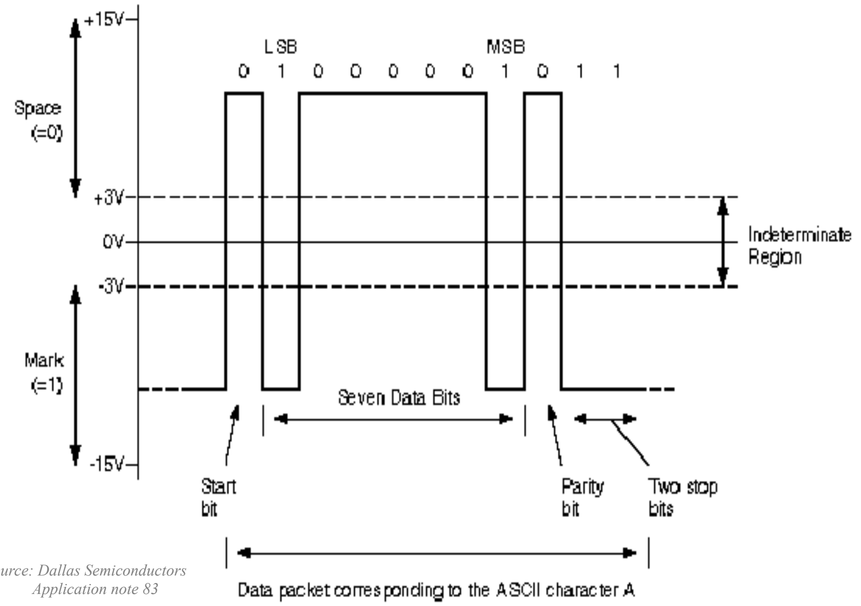
Data Modulation

- 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, 1-1/2, or 2 closing stop bits
  - Bits are usually encoded using ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange)

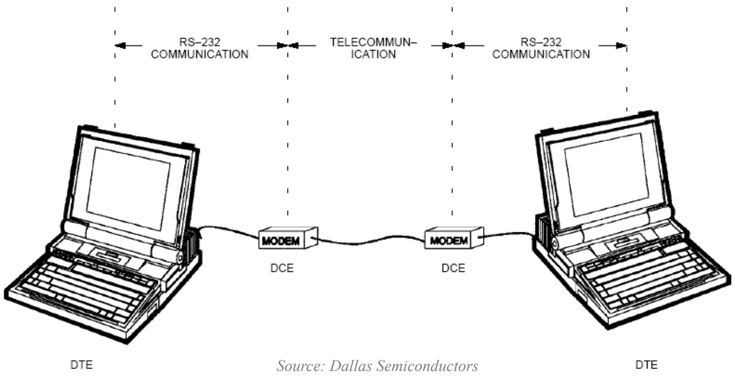
RS-232 Signal Levels



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



Source: Dallas Semiconductors Application note 83

Serial Port Connector

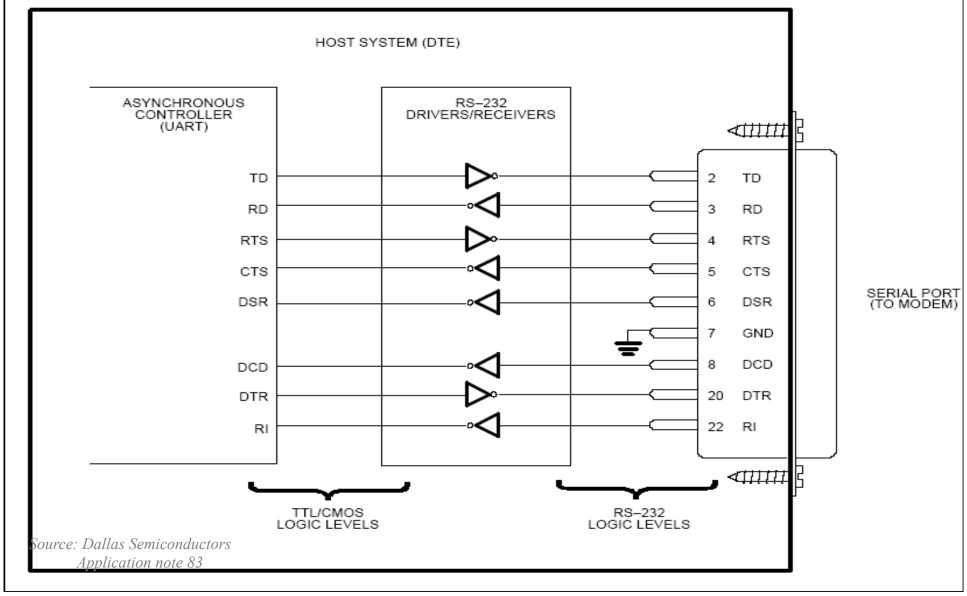
- 9-pin (DB-9) or 25-pin (DB-25) connector

- 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?

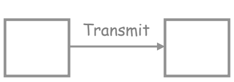
RS-232 Pin Connections

TYPICAL RS–232 MODEM APPLICATION Figure 3



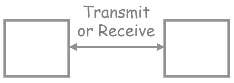
Source: Dallas Semiconductors Application note 83

Serial Data Communication Modes



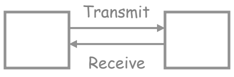
Simplex Mode

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.

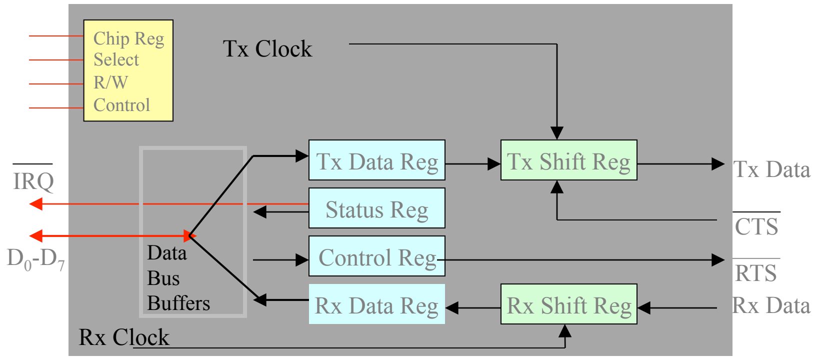


Full-duplex Mode

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



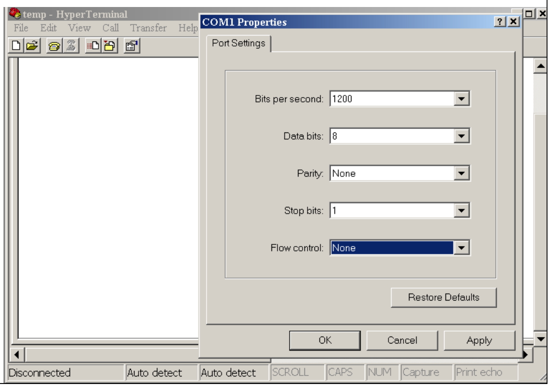
Flow Control

- 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

- 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



temp - HyperTerminal

COM1 Properties

Port Settings

Bits per second: 1200

Data bits: 8

Parity: None

Stop bits: 1

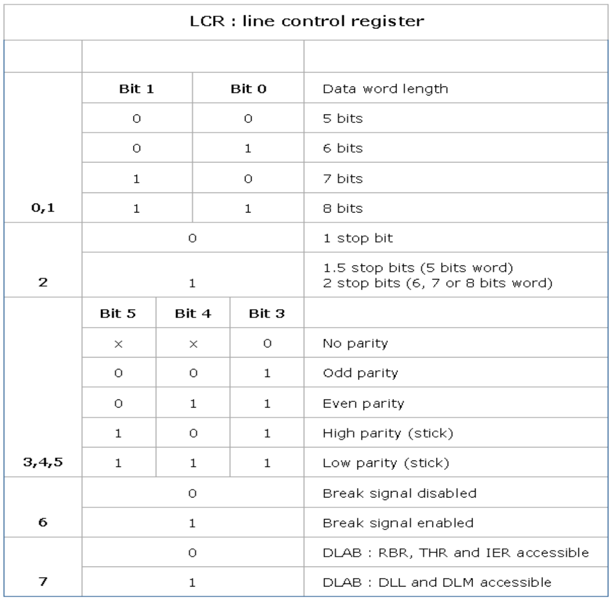
Flow control: None

Disconnected | Auto detect | Auto detect

Some Register Definitions for the UART

```
#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 – Line Control Register



LCR : line control register				
0,1	Bit 1	Bit 0		Data word length
	0	0		5 bits
	0	1		6 bits
	1	0		7 bits
	1	1		8 bits
2	0			1 stop bit
	1			1.5 stop bits (5 bits word) 2 stop bits (6, 7 or 8 bits word)
3,4,5	Bit 5	Bit 4	Bit 3	
	x	x	0	No parity
	0	0	1	Odd parity
	0	1	1	Even parity
	1	0	1	High parity (stick)
	1	1	1	Low parity (stick)
6	0			Break signal disabled
	1			Break signal enabled
7	0			DLAB : RBR, THR and IER accessible
	1			DLAB : DLL and DLM accessible

Initializing/Finalizing Serial Port Driver

- Inside `init_serial()`
  - Set the baud rate to be 1200
    - Set the LSB and MSB of the clock frequency divider register
  - Set the data bits in the UART Line Control Registers to be 8
  - Set the parity bits in the UART Line Control Registers to be 0
  - Set the stop bits in the UART Line Control Registers to be 1
  - Disable the use of FIFO buffers in the UART FIFO Control registers
  - Disable loop-back mode in Modem Control Register
  - Disable interrupts in the Interrupt Enable Register
  - Print a “library initialized” message
- Inside `end_serial()`
  - Put the serial port back in loop-back mode

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

Summary of Lecture

- Serial Communications
  - Asynchronous protocols
  - Synchronous protocols
  - RS-232 data interface
  - Parity bits
  - Serial port and bit transmission