I/O

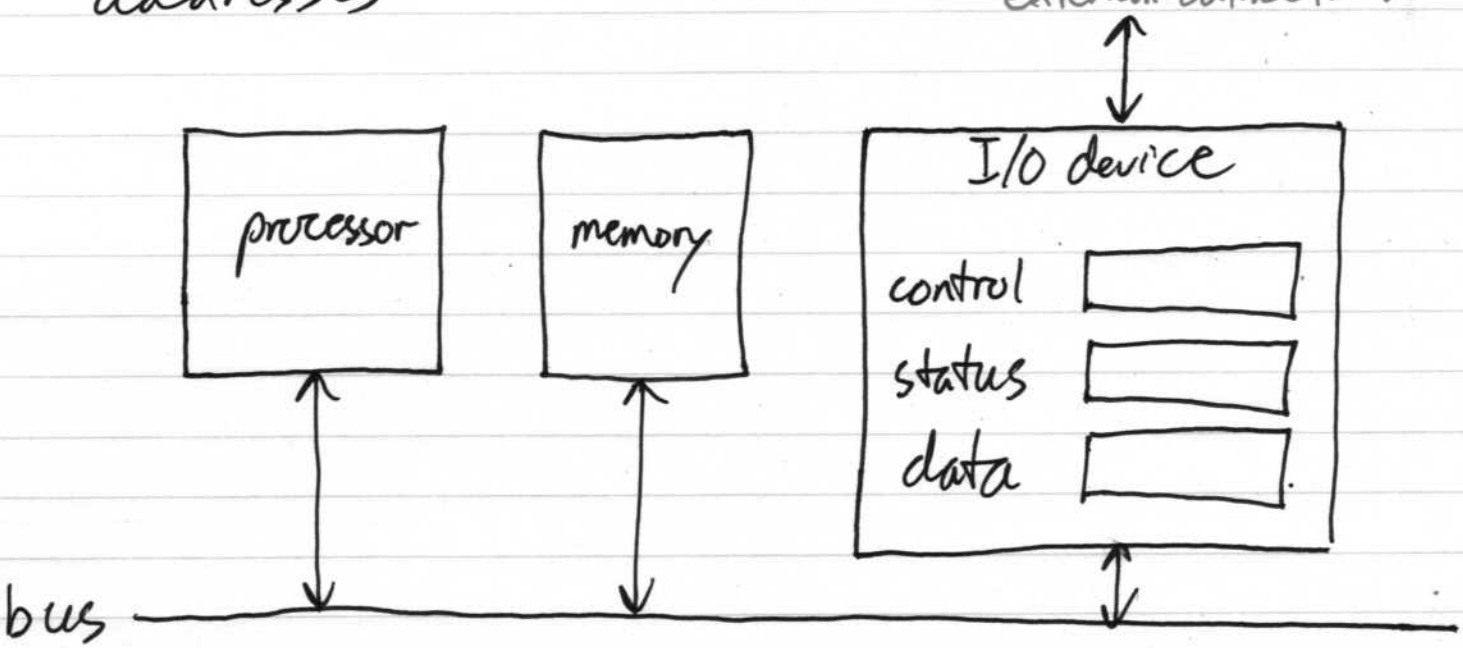
1) Memory-Mapped I/O

- the addresses on the bus are shared between memory and peripheral devices

- examples from the LPC1768 memory map:

Address Range	Device
0x0000 0000 – 0x0007 ffff	Flash Memory
0x1000 0000 – 0x1000 7fff	SRAM Memory
0x1fff 0000 – 0x1fff ffff	ROM Memory
0x2009 c000 – 0x2009 ffff	GPIO (general-purpose I/O)
0x4000 c000 – 0x4000 c030	UART 0 (serial port)
0x5000 0000 – 0x5000 0ff4	Ethernet

- peripheral devices have registers mapped to specific addresses



e.g. UART - universal asynchronous receiver transmitter (serial port)

- data register: Receive Buffer Register (RBR) 0x4000 c000
- control register: Interrupt Enable Register (IER) 0x4000 c004
- status register: Line Status Register (LSR) 0x4000 c014

e.g. code to control UART 0 interrupts

in "uart.h"

```
#define IER_RBR  0x01   // 0000 0001₂
#define IER_THRE 0x02   // 0000 0010₂
#define IER_RLS  0x04   // 0000 0100₂
```

- these are "bit masks" that give names to bits in the Interrupt Enable Register

in "uart.c"

bit-wise OR

before IER

0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0

set bits 0 and 2

```
LPC_UART->IER |= IER_RBR;    // 0000 0001
LPC_UART->IER |= IER_RLS;    // 0000 0101
```

clears bit 0

```
LPC_UART->IER &= ~IER_RBR;   // 0000 0100
```

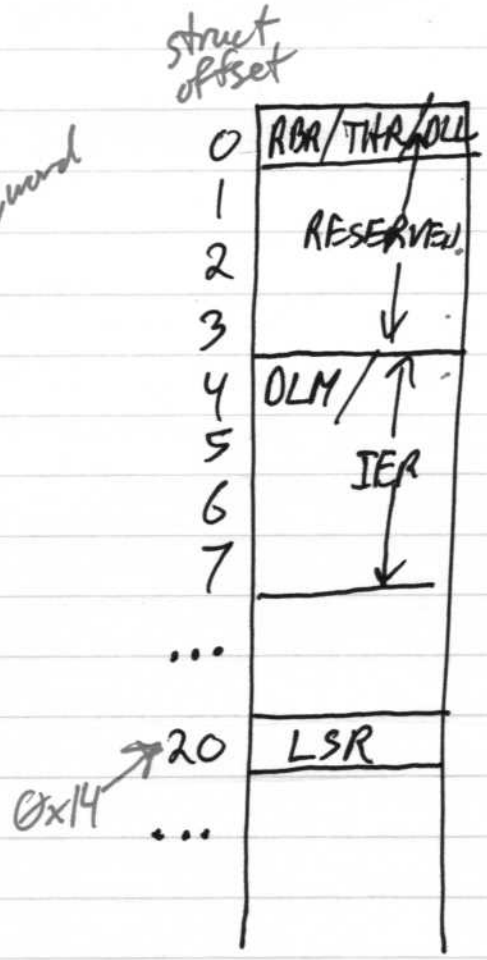
- setting or clearing these bits controls when the UART0 interrupts the processor

- how does LPC_UART->IER address (reference) the IER register at 0x4000 c004
  - this struct is defined in <LPC17xx.h>:

```
typedef struct {
  union {
  __I  uint8_t  RBR;
  __O  uint8_t  THR;
  __IO uint8_t  DLL;
       uint32_t RESERVED0;
  };
  union {
  __IO uint8_t  DLM;
  __IO uint32_t IER;
  };
  ...
  __I  uint8_t  LSR;
  ...
} LPC_UART_TypeDef;
```

a union overlays its members in the same location

not a keyword



- pointer LPC_UART_TypeDef *LPC_UART is assigned to address 0x4000 c000

∴ LPC_UART->IER references 0x4000 c004

to do this manually:

```
LPC_UART = (LPC_UART_TypeDef*) 0x4000c000;
```

Cortex-M3

-in <core_cm3.h>

(input) #define __I volatile const

- volatile: type modifier that tells the compiler not to optimize out reads or writes to this variable
- const: means that the program cannot assign to this variable (it does not mean that it is unchanging)

(output) #define __O volatile

#define __IO volatile

! the volatile keyword is needed whenever accessing memory-mapped peripherals

2) Polling I/O

- the program repeatedly check's an I/O device's status register until it sees that the device is ready to send or receive data

- in "uart.c"

```
uint8_t UARTReceiveChar(...) {
...
while(!(LPC_UART->LSR & 0x01));
return (LPC_UART->RBR);
}
```

loop body

read Line Status Register

volatile

read from data register

7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	?

while be 1 if data is received

- this code repeatedly reads the UART's status register until bit 0 is a '1'; then it reads a byte of data and returns it

- polling can waste processor time and block other useful execution

eg. the above polling loop in ARM assembly:

```
                                  ↙ address of LSR
1 cycle  LOOP   LDRB  R1, [R0]
1 cycle         TST   R1, #0x01
2 cycles        BEQ   LOOP
4 cycles
```

- the Cortex-M3 clock rate = 96 MHz
- the "polling rate" = $96\times10^6 \frac{\text{cycles}}{\text{s}} / 4 \frac{\text{cycles}}{\text{loop}} = 24\times10^6 \frac{\text{loops}}{\text{s}}$
- let's assume that the UART is receiving keyboard data
  - a fast typist $\approx 70 \frac{\text{words}}{\text{min.}}$
  - average English word length is $5.1 \frac{\text{char}}{\text{word}}$
  - the "data rate" $= \frac{70 \text{ word/min}}{60 \text{ s/min}} \times 5.1 \frac{\text{char}}{\text{word}} = 5.95 \frac{\text{char}}{\text{s}}$
- how many times is the status reg. polled on average per character received?

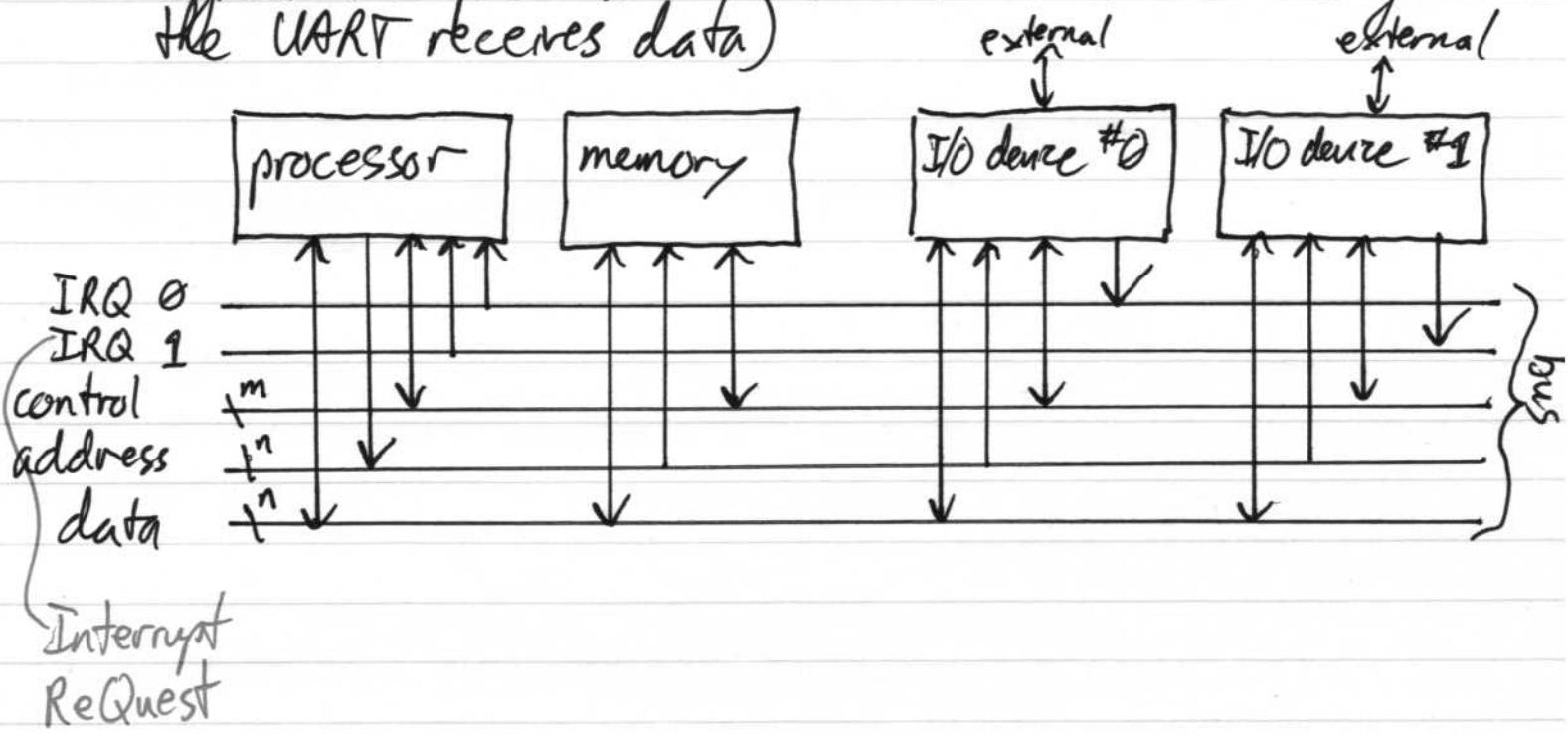
$$\frac{\text{polls}}{\text{char}} = \frac{\text{polling rate}}{\text{data rate}} = \frac{24\times10^6 \frac{\text{loops}}{\text{s}}}{5.95 \text{ char/s}} = 4.03\times10^6 \frac{\text{loops}}{\text{char}}$$

- a simple strategy to get more work done:

```
while (true) {
    do work...
    check UART status
}
```

3) Interrupt-Driven I/O

- it may be better to let the peripheral device signal the processor when it needs service (e.g. when the UART receives data)



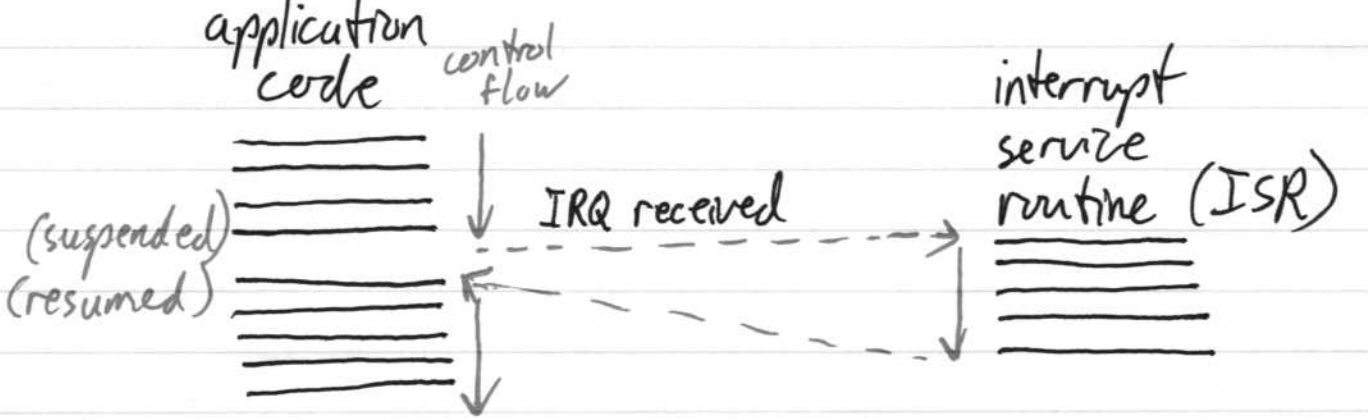
- three exception types

① Interrupt - request for service from a peripheral device

② Fault - indicates a problem executing an instruction e.g. divide by zero, bad memory address

③ Trap - special instructions that request OS services e.g. SVC in ARM

- in all cases an exception causes a change in control flow similar to a subroutine call except that the hardware ~~is~~ invokes the subroutine, not the software



- the application code is unaware that it was interrupted

- terminology:
  - in general, exceptions cause "exception handlers" to execute
  - "ISR" is another name for exception handlers for Interrupts
  - but ARM calls ISRs "IRQ Handler"

3.1) Exception Handling Mechanism [8.2.3, 8.6]

- every exception source, including IRQs, is assigned a "vector" (a number)
- the Vector Table lists the ISR addresses for each vector

- from "startup_LPC17xx.s":

	vector	address of	
faults and trap instr. (internal)	0	(holds initial stack pointer)	
	1	Reset_Handler	runs on startup or reset
	2	NMI_Handler	non-maskable interrupt
	...		
	11	SVC_Handler	service calls to OS
	...		
interrupts (external)	16	WDT_IRQHandler	watchdog timer
	17	Timer0_IRQHandler	
	...		
	21	UART0_IRQHandler	
	...		

- watchdog timers are used in autonomous systems
  - the application must periodically reset the timer ("kick the dog")
  - if it doesn't and the timer reaches the terminal count it resets the system
  - it is used to recover from system failure

- when an IRQ is received, the processor hardware:

takes 12 cycles on Cortex-M3

① looks up the handler address in the vector table

② saves the current execution context on the stack (stack pointer, general-purpose registers, program counter, status register)

③ invokes the exception handler (ISR)

- when the ISR exits, the hardware:

10 cycles

① restores the execution context from the stack

② resumes the application code

3.2) Multiple Interrupts [8.3]

- each exception source (IRQ) is assigned a priority
- IRQs of higher priority can interrupt ISRs of lower-priority IRQs ("nested interrupts")

- ISRs of lower-priority IRQs wait for the ISRs of higher-priority IRQs to finish and then run before the application code is resumed

3.3) Interrupt Service Routines (ISRs)

- normally, dealing with the condition needing service clears the interrupt request (IRQ)
  e.g. reading received data from the UART
  e.g. resetting the timer
- if the IRQ is not cleared, the ISR will be re-invoked when it exits
  - sometimes it is necessary to disable IRQs on a peripheral device (by writing to its control register)

! rules for ISRs

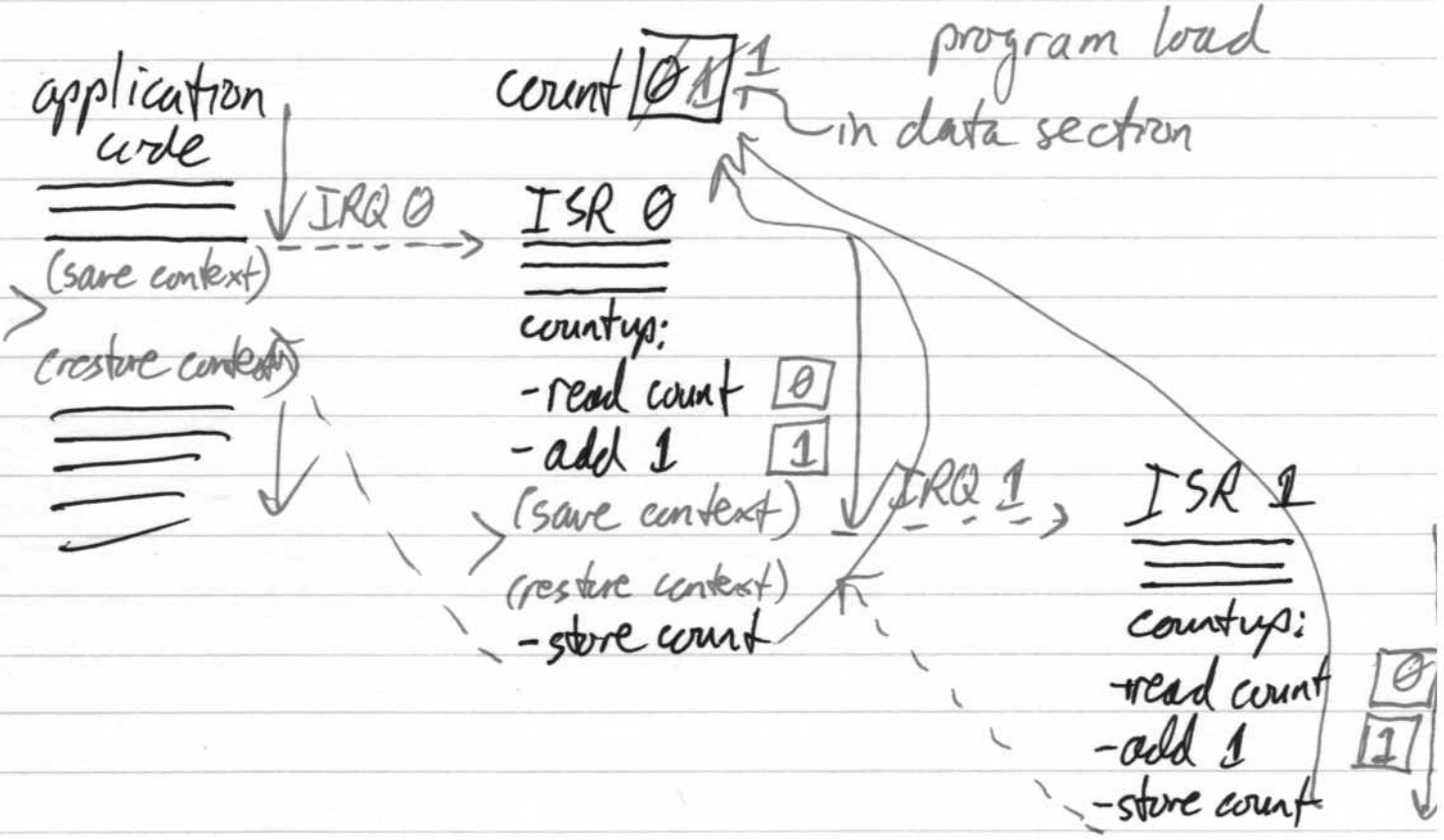
1. keep it short
2. use in-memory buffers to pass data to/from the application
3. if an ISR calls a function, the function must be "re-entrant"

- re-entrant functions can be executed while another instance is executing i.e. can be called concurrently
  - they cannot use global or static variables

not re-entrant

```
e.g. uint32_t countUp() {
        static uint32_t count;
        count = count + 1;
        return count;
     }
```

only one copy that is shared by all invocations of the function
- like a global variable but with local scope
- initialized to 0 on program load



- to use a non-reentrant function in an ISR, you need to disable interrupts

use sparingly

e.g.
```
__disable_irq(); {
        countup();
} __enable_irq();
```

__disable_irq() and __enable_irq() are part of the Cortex Microcontroller Software Interface Standard (CMSIS)

- printf() and malloc() are not re-entrant but as of C11 standard they are "thread-safe" (by adding mutex locks)