Chapters 2 & 3

- A review of hardware essentials
  - Most of you have seen this material in other classes
  - Still worth a careful read: may give you new insight
- We'll touch on a few topics of interest

Chapters 2, 3: bits and pieces

- Consider challenge of choosing appropriate memory for program and data in embedded applications

Technology	Read Speed	Write Speed	Write Times
ROM (masked)	Fast	N/A	0
PROM	Fast	N/A	1
EPROM	Fast	N/A	Many
EEROM	Slow	Slow	Millions
Flash	Fast	Slow	~100,000+
RAM	Very fast	Very fast	Infinite

Anticipated memory technologies

Gleaned from the web…

- DVRAM (Deja-Vue RAM)
  - The CPU thinks it has the data before it actually does
- PVRAM (Presque-Vue RAM)
  - The CPU only has to pretend to access RAM to get the data
- ODRAM (Oracle at Delphi RAM)
  - Returns data the CPU plans to access next (first data access has to be a NOP)
- HRAM (Hearsay RAM)
  - CPU talks to other CPUs and uses what they all think the data is, instead of accessing the data (which may be different)
- 711RAM (Seven-Eleven RAM)
  - Always available, but may be held up during the night shift
- ARAM (Audio RAM)
  - Like video RAM, but describes the image verbally instead

Chapters 2, 3: bits and pieces

- Self-test on selected topics:
  - What are *tri-state* devices, and what are challenges if they are controlled by software? (pp. 23-26)
  - What are the characteristics of *flash memory* that make it so popular, and what are its limitations? (pp. 36-37)
  - Is there a difference between a *microprocessor* and a *microcontroller*? (p. 46)
  - How does *memory-mapped I/O* differ from having a separate I/O address space? (p. 51)
  - What are *wait states*, what problem do they address, and how are they inserted? (pp. 55-56)

Chapters 2, 3: bits and pieces

- Other topics of interest:
  - DMA: direct memory access
    - Circuitry that can move data between I/O devices and memory without software assistance; reduces CPU overhead for I/O.
  - Interrupts
    - Hardware signal telling the processor that particular event has occurred.
    - Processor can ignore: under software control.
  - Watchdog timer
    - Resets processor when it expires; shouldn't happen in normal operation.
    - Software resets counter regularly, called *petting the watchdog*.
    - Important: why reset processor and not assert interrupt?
  - Role of caches, pipelining, virtual memory in embedded CPUs?

"A last word about hardware"

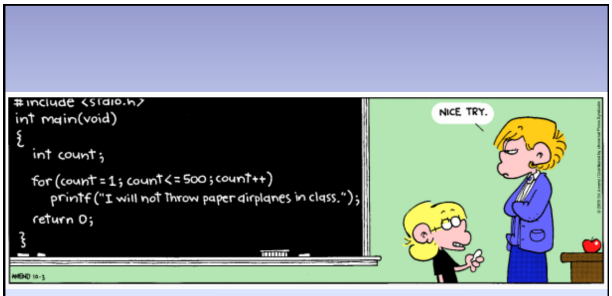
- Every copy of the hardware costs money.
  - In high volume, eliminating a 25 cent part can be a big deal.
- Every part
  - takes up space, and space is at a premium.
  - requires power, adding to battery load or increasing size and cost of power supply.
  - generates heat; eventually you need a fan, or larger fan.
- In general, faster components cost more, use more power, and generate more heat.
  - Hence, clever software often better way to make product fast.

Using C

> Real-time programmers must be masters of their compilers. That is, at all times you must know what assembly language code will be output for a given high-order language statement.
>
> Phillip A. Laplante

> The limits of my language mean the limits of my world.
>
> Ludwig Wittgenstein

- C is widely used in embedded systems, but can be tricky.
  - Important to understand thoroughly the constructs you use.

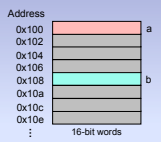


C

- Hopefully you are not seeing it here for the first time
  - The most common language in embedded systems
- Recommended text on C
  - Kernighan & Ritchie
- We'll briefly discuss some pointer issues
  - Critical to understand in your RTOS code

Pointer usage

```
/* source code */
int a;
int *b;
```



Expression	Type	
`a`	int	
`&a`	int *	(‘`&`’ is read “address of”)
`b`	int *	
`*b`	int	(‘`*`’ is read “contents of address”)
`*(&a)`	int	
`*a`	?	What do these do?
`&b`	?	

Pointers as parameters

```
int x;

void f (int a)
{
   a = 2;
}

main()
{
   x = 4;
   f(x);
   printf("%d", x);
}
```

Output is 4

What is output?

```
int x;

void f (int *a)
{
   *a = 2;
}

main()
{
   x = 4;
   f(&x);
   printf("%d", x);
}
```

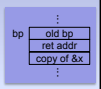
Output is 2

Pointers as parameters

```
int x;

void f (int a)
{
   a = 2;
}

main()
{
   x = 4;
   f(x);
   printf("%d", x);
}
```

Output is 4

```
int x;

void f (int *a)
{
   *a = 2;
}

main()
{
   x = 4;
   f(&x);
   printf("%d", x);
}
```



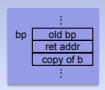
Output is 2

Pointers and parameters

- Pointers useful in writing flexible, clean code.
  - Essential in your code this semester
  - RTOS API includes functions with pointers as arguments, return values
- C parameter passing is *always* "by value"!
  - **Copy** of argument placed in arg build area on stack
    - If arg is struct: copy of entire struct is pushed onto stack
    - If arg is array: address of array is pushed onto stack
  - Within function, parameter is same as local variable
    - Write will not change actual parameter
  - Can simulate "pass by reference" by using pointer to variable
    - Still "by value", but value used is that of pointer

Pass by reference?

```
void f(int *a) {
   *a = 2;
}

main() {
   int x,*b;
   x = 4;
   b = &x;
   f(b);
   /* is b changed? */
   ...
}
```



Using pointers

```
int x;

void f (int *a)
{
   *a = 2;
}

main()
{
   x = 4;
   f(&x);
   printf("%d", x);
}
```

```
; Generated by c86 (BYU-NASM) 5.1 (beta) from ptrex1.i
                CPU     8086
                ALIGN   2
                jmp     main            ; Jump to program start
                ALIGN   2
f:
                jmp     L_ptrex1_1
L_ptrex1_2:
                mov     si, word [bp+4]
                mov     word [si], 2    dereferencing pointer
                mov     sp, bp
                pop     bp
                ret
L_ptrex1_1:
                push    bp
                mov     bp, sp
                jmp     L_ptrex1_2
L_ptrex1_4:
                db      "%d",0xA,0
                ALIGN   2
main:
                jmp     L_ptrex1_5
L_ptrex1_6:
                mov     word [x], 4
                mov     ax, x
                push    ax              passing parameter
                call    f
                add     sp, 2
                push    word [x]
                mov     ax, L_ptrex1_4
                push    ax
                call    printf
                add     sp, 4
                mov     sp, bp
                pop     bp
                ret
L_ptrex1_5:
                push    bp
                mov     bp, sp
                jmp     L_ptrex1_6
                ALIGN   2
x:
                times   2 db 0
```


Observations

- Accessing data via pointer can be quite efficient.
  - In some cases, addressing modes on x86 can use pointers with no extra instructions.
  - However, instructions that access memory are more complex, likely to require more cycles. (This is not reflected in our tools.)
- Thinking about what happens at assembly level can help you keep things straight in your C code.

Array and pointer address arithmetic

Array Expression	Same Expression Using Pointer Arithmetic
`x[a]`	`*(x+a)`
`x[0]`	`*x`
`&x[b]`	`x+b`
`&x[0]`	`x`
`x[a][b]`	`*(*(x+a)+b)`

Assuming multilevel (not nested) array.

Equivalent forms may compile with different levels of efficiency

What do these code examples do?

```
int x;
int y[4] = {2,3,5,7};
main()
{
   x = y[0];
   x++;
}
```

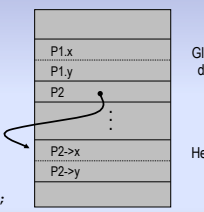
Final values:
x = 3;
y[] = 2,3,5,7;

```
int *x;
int y[4] = {2,3,5,7};
main()
{
   x = y+2;
   *(x++) = (*x)++;
}
```

Final values:
*x = 7;
y[] = 2,3,6,7;

Pointers and structs

```
struct point
{
   int x;
   int y;
};
struct point P1, *P2;
main()
{
   P1.x = 5;
   P1.y = -7;
   P2 = (struct point *)
       malloc(sizeof(struct point));
   P2->x = -31;
   P2->y = 16;
```



`struct *` example

- What happens when you compile this code and run it?

```
struct point
{
   int x;
   int y;
};
struct point p1,*p2;
main()
{
   p1.x = 5;
   p1.y = -7;
   p2->x = -31;
   p2->y = 17;
    ...
}
```

On Linux systems: "Segmentation fault"! Why?

On our system: ?

Void pointers

- void objects have "nonexistent" value, may not be used in any way
  - void used to specify no function return value, empty parameter list, etc.
- Any pointer can be cast to void * and back again without loss of information.
- Operations on void pointers:
  - Assignment is allowed
  - Comparison is allowed
  - Dereferencing is NOT allowed!
- Why use void pointers?
  - Proper type for a *generic* pointer, e.g. return value of `malloc()`
  - Must explicitly cast to another pointer type to dereference
    - Easier to make mistakes using, say, `char *` as generic pointer type

Void pointers: example

```
int a;
long b;

struct thing
{
   int count;
   void *p;
   struct thing *next;
}  xyz, qrs;

xyz.p = (void *)(&a);
qrs.p = (void *)(&b);

xyz.next = &qrs;
```

pointer to anything

for making a linked list

These will have to be cast before they can be dereferenced.

C tips

- Keep your code clean, easy to understand.
- Document your code:
  - Later you and partner will wonder what it does!
- Format/indent your code consistently
  - Good editors will do this automatically
- Use .h files appropriately
  - No code or variable declarations! Nothing that allocates space in memory.
  - Only #defines, typedefs, function prototypes, etc.
- Study assembly code to see what C turns into
  - Be aware of efficiency of generated code

Chapter 4: Interrupt basics

- Interrupt: a mechanism used to transfer attention to something of potential importance.
- Interrupts for humans:
  - Doorbell, phone, oven timer, campus class bells
  - Do you sometimes ignore these interrupts?
- Common scenario in real-time systems:
  - Processor is running job A
  - Something happens that must be dealt with
  - Processor must put A "on hold", handle event

What alternative exists?

- What if designer doesn't want complexity of interrupts?
- Can use polling instead
  - CPU tests for events at regular intervals.
  - For simple embedded applications, polling often good enough
- Disadvantages
  - Hard to balance computation/responsiveness with multiple tasks
  - Regular tests are inefficient use of CPU – if anything else to do

Advantages of interrupts

- Consistent delay in responding to events
  - (Almost) independent of complexity of executing task.
- Task can sleep until its trigger event occurs.
  - Essential in priority-based multitasking system.
  - Most important task can finish for now, give up processor, resume execution when conditions demand.
- These advantages come with a price:
  - Interrupt-based code more complex than code doing polling

What can cause an interrupt?

- Anything that system designer wires to interrupt pins.

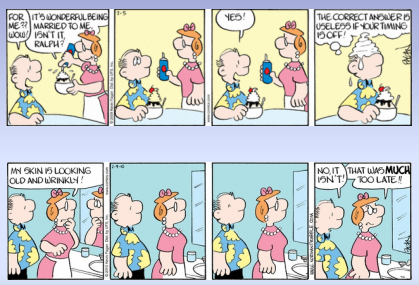
  Example events:
  - UART receives new char
  - Disk controller has data block requested earlier
  - Sensor reports change in data value
  - User presses a button or key
  - Timer expires
  - Power failure
  - Fault or error detected in system, either
    - External to CPU (hardware specific)
    - Internal to CPU (exceptions)

Interrupts: key to responsiveness

- Essentially all processors support interrupts
- Our focus this semester:
  - Systems with challenging response-time constraints
  - Getting the right answer is important, but it must be delivered in time to use it

Definition I: "A *real-time* system is one whose logical correctness is based on both the correctness of the outputs and their timeliness."

- Are there systems in which timeliness is <u>not</u> important?
  - Implication: timely response is *critical*
  - Consequences are far worse than (just) an unhappy user



Definition II

" A *real-time* system is a system that must satisfy explicit response-time characteristics or risk severe consequences, including failure."

- If a **deadline** is not met, system failure may result
  - Plane crashes, nuclear plant goes critical, etc.
- Many embedded systems can be classified as **real-time** based on this definition
  - Depends on criticality of deadlines

Classification of real time systems

Key question: how critical are the deadlines?

increasingly critical

- ***Hard*** real time:
  - Missing a single deadline may cause system failure
- ***Firm*** real time:
  - Occasional deadline can be missed
- ***Soft*** real time:
  - Missing a deadline causes performance degradation

ECEn 425 focus

- We're assuming target is a hard real time system
  - Implication: deadlines must be met!
  - Our interest: what software constructs are used to create these kinds of systems?
- Motivation:
  - These are the most difficult real-time systems to design and build.
  - If we can create these systems, we can build those with less strict requirements.

Interrupts

- Critical in systems with multiple tasks, hard deadlines
- Interrupt mechanism is a nifty collaboration between hardware and software.
  - Both hardware and software play crucial roles.
  - Understanding details is essential part of computer system literacy.
- Let's start here:
  - **What does processor do when interrupt is asserted?**

Interrupts: hardware side

Simple model:

- Interrupt is asserted
- HW saves critical information about current task
- HW sets IP (PC) to start of corresponding code: *interrupt service routine* (ISR).

Questions:

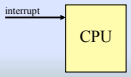
- Is delay possible from assertion to HW response?
  - Finish current instruction
  - Interrupt may be masked or disabled
- What information, where saved?
  - Return address (at least)
  - On stack, in register, in fixed memory location, etc.
- Which is correct ISR, and what is its starting address?
  - SW must provide this info to HW

Interrupts: hardware issues

- Delay in responding
  - Finish current instruction: how long can this take?
  - How long might interrupt be masked or all interrupts disabled?
- Saving state
  - ISR is similar to *hardware induced function call*
  - Similar actions: save return address, jump to new location
  - Key difference: interrupt can occur at any point, so all registers must be saved
- Finding correct interrupt code to run
  - Implementation specific: usually a table of ISR starting addresses, indexed by interrupt number or level (interrupt vector table)

Identifying source of interrupt

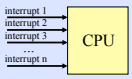
- You are awakened at 2:00 AM by a noise, but you're not sure what it was. What do you do?
  - Was it window banging in wind, an intruder, a child falling out of bed, a car accident, a smoke-detector chirping, or something else?
- How does CPU know what interrupted it?
  - Start with simplest hardware model: single interrupt line/pin



- Single ISR entry point for all interrupts.
- Software must run down checklist: what "woke me up"?
- All interrupts disabled while ISR runs.
- Single location sufficient to store return address.

Multiple interrupt lines

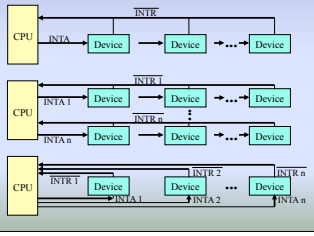
- More complex hardware can be more efficient
  - Useful to have multiple interrupt lines
  - For interrupt *i*, hardware gets entry *i* from table of *n* ISR addresses
    - Software responsible for initializing this *interrupt vector table*
    - In best case, event-specific ISR can begin to run directly



- Questions:
  - What if two lines are asserted at same time?
  - What if second interrupt occurs while handler for first is running?
  - What is required to handle nested interrupts?

Attaching devices

- Consider these alternatives:



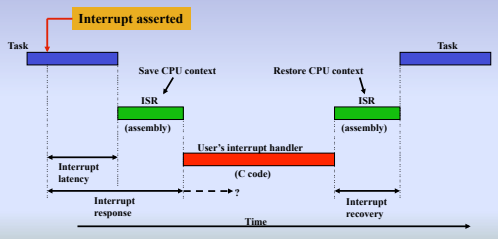
Our interrupt model

- Eight hardware interrupts (IRQs)
  - Priority handled by external chip (PIC)
  - Single interrupt line to processor
  - Each connected to separate device
- Interrupts enabled/disabled by interrupt flag (IF)
  - Special instructions: **sti** sets IF (enabled), **cli** clears IF (disabled)
- PIC has 3 8-bit registers (one bit per IRQ) to manage interrupts
  - IMR (Interrupt Mask Register): selectively enables/disables
  - IRR (Interrupt Request Register): shows asserted interrupts
  - ISR (In-Service Register): shows interrupts currently being serviced
  - In simulator: IMR, IRR, ISR displayed with other registers; can change, monitor changes on, etc.

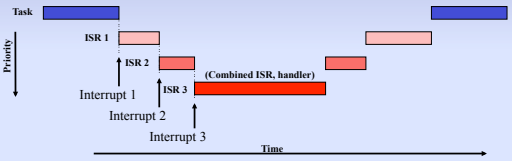
Our interrupt model

- Event sequence handled by hardware (before ISR runs):
  - Device asserts interrupt, PIC signals CPU on single interrupt line
  - CPU acknowledges interrupt response to PIC
  - PIC gives CPU vector # + 8 (e.g., IRQ 4 indicated by value of 12)
  - CPU uses that number as offset to interrupt vector table (stored at address 0:0)
    - Each entry is 4 bytes, gives starting address of ISR
  - CPU pushes flags, CS, and IP on stack (total of three 16-bit words)
  - CPU clears IF, disabling interrupts
  - CPU sets CS and IP to address of ISR (from table), fetches first ISR instruction
- Software responsibilities:
  - ISR must save rest of interrupted state, re-enable interrupts, and call interrupt handler (C function that responds to interrupting event).
  - After handler returns, ISR must restore interrupted context, notify PIC of end of ISR, and execute **iret** instruction which restores IP, CS, and flags.
  - Software must also ensure that interrupt vector table is initialized (at boot).

Interrupt handling illustrated



Interrupt nesting illustrated



Interrupt nesting

- Ensures timely response to most important events.
  - Worst case response time for highest priority interrupt = longest code section with interrupts disabled.
  - Worst case response time for interrupts at other priority levels includes service time for higher priority interrupts.
- What is required to make nesting work?
  - Full context must be saved at each level, including return address.
  - Higher priority interrupts must be enabled, other interrupts disabled.
    - PIC handles details of masking interrupts with lower priority
    - ISR must enable interrupts, since hardware disables them before ISR runs
- A bit trickier to write ISRs, handlers that can be interrupted
  - Interrupt nesting required in your code this semester

Summary: ISR structure

- When 8086 detects an IRQ it
  - suspends execution of current task,
  - saves the return address (including segment) and flags, and
  - jumps to an interrupt service routine.
- In turn, the ISR
  - saves remaining context and does some housekeeping,
  - does what needs to be done to respond to the interrupt (usually by calling a C function), then
  - restores saved context and returns to the code that was interrupted.

Suppose ISR does all the work

- What are its actions if no separate handler is called?
  - Save registers it will use on the stack
  - Respond to the interrupt
  - Restore saved registers
  - Return

Suppose ISR calls a C function

- What are actions of ISR if a handler is called?
  - Save registers that will be used by interrupt code
  - Normal function call to interrupt handler that responds to event
  - Restore saved registers
  - Return
- What registers must ISR save?
  - At minimum: all registers used by handler, but that might change!
  - Safest approach: save them all
  - Future complication: may run other code on return from ISR

CPU context

- The entire processor state must be restored exactly as it was before the interrupt occurred, or bizarre, difficult-to-find errors will be created.
  - Why might behavior be inconsistent, hard to reproduce, and difficult to track down?
- Context typically includes:
  - Return address, normal registers, condition codes, special registers (e.g., mask registers)
- For our labs:
  - ISR must save all CPU registers except SP, SS, CS, IP, and flags
  - PIC handles IMR, ISR, IRR: not part of ISR-saved context

Lab 3 overview

- A lengthy task is busy computing and printing prime numbers. (This code is given to you.)
- Three interrupts and associated actions are defined:
  1. **Reset** must stop program execution
     - Caused by pressing control-R (ctr-R) on the keyboard
  2. **Tick** prints the message "Tick *n*", where *n* is the number of timer ticks processed so far
     - Ticks generated automatically by system at regular intervals
     - Can be generated manually by ctr-T on the keyboard
  3. **Keypress** prints the message "Keypress (x) ignored" for any key other than ctr-R, ctr-T (or ctr-C, ctr-Z!)
     - Key *x* is found in global variable called **KeyBuffer**, defined in clib.s

Lab 3 assignment

- In assembly code, write ISR for each of the 3 interrupts.
  - Modify clib.s with ISR starting addresses
- In C, write interrupt handler for each of the 3 interrupts.
- In general, each ISR will
  - save state,
  - call the appropriate handler (a C function), and
  - restore state and return.
- Remember general philosophy:
  - Do as much as you can do in C.
  - Use assembly only when you can't do it in C.

Lab 3 interrupt timing

- Single task code, three different ISRs
  - Nesting must be supported

Lab 3: nested interrupts

- Lab 3 code: how can we verify that interrupt nesting works?
  - Interrupt handling much faster than our reaction time
- Our solution (for this lab only):
  - Special actions required for "delay" key ('d'):
    - Handler spins in loop, incrementing local variable 5000 times
    - Length ensures that timer tick will occur during delay
  - In other words, this forces a nested interrupt to occur
    - Study output, confirm correct interleaving of ISR actions

Lab 3: sample output

task output →
normal tick →
normal keypress →
nested interrupts

```
TICK 22
 2467 2473 2477 2503
TICK 23
 2521 2531 2539
TICK 24
 2543 2549 2551
 2557 2579 2591 2593 2609 2617 2621 2633 2647
KEYPRESS (8) IGNORED
 2657
 2659 2663 2671 2677 2683 2687 2689
KEYPRESS (k) IGNORED
 2693 2699 2707
TICK 25
 2711 2713 2719 2729 2731 2741 2749 2753
KEYPRESS (j) IGNORED
 2767 2777
 2789 2791 2797 2801 2803 2809 2819 2833 2837 2843
 2851 2857 2861
TICK 26
 2879 2887 2897 2903 2909 2917 2927
 2939 2953 2957
DELAY KEY PRESSED
TICK 27
TICK 28
DELAY COMPLETE
 2963 2969 2971
TICK 29
```


Lab 3 output

- Prime numbers are printed by prime number generator (*background* task code) as they are computed.
- Interleaved with that output are messages from your interrupt routines.
  - For clarity, put message on line by itself (as on previous slide).
- After interrupts, control must be passed back to prime number generator, which resumes its endless computation.
- Requirements (TA will stress-test!):
  - Code (task + ISRs) must not crash or hang, regardless of frequency of keypresses
  - Code must work even with a tick every 500 instructions
    - Dramatic increase from normal frequency of tick per 10,000 instructions

Future labs

- All future labs will make use of ISRs: understand them!
- Lab 3 version is a little simpler than ISRs in later labs
  - Lab 3 ISRs and the task do not share data or communicate.
  - There is just one task.
- Much of complexity of real-time code comes from:
  - Problems with data shared by tasks and/or ISRs
  - Communication, synchronization between tasks, ISRs
  - Requirements of multiple tasks
- This lab is a good starting point, but the complexity ramps up quickly…

Section 4.2: Common questions

- How is the ISR found for an interrupt?
- Can the CPU be interrupted in the middle of an instruction?
- If two or more interrupts happen at the same time, what does the hardware do?
- Can interrupts be nested?
- What happens if an interrupt is asserted and interrupts are disabled or that particular interrupt is masked?
- What happens if you forget to re-enable interrupts?
- What if you enable interrupts when already enabled, or disable when already disabled?

Common questions cont.

- What is state of IF and IMR when simulator starts up?
  - Does this reflect real hardware?
- Can I write ISRs in C?
- How, where does an ISR save context?
- Must all registers be saved?
- How are contexts saved with nested ISRs? How will they be restored?
- What happens if you forget to save a register?
- How can you disable and enable interrupts from C?
- What's the purpose of a *nonmaskable* interrupt?