CprE 288 – Introduction to Embedded Systems
Exam 1 Review
Overview of Today’s Lecture

• Announcements
• Exam 1 Review

http://class.ece.iastate.edu/cpre288
Announcements

• Exam 1: Thursday 2/25, in class
  – Open book, open notes, and calculator allowed
  – 75 minutes
  – No electronic devices, except calculator
Exam Format

• Focus
  – C programming, questions specific to the AVR ATmega128 processor
  – Memory Mapped I/O
  – Memory Layout
  – Interrupts

• Open notes, no electronic devices except calculators

• Covers material covered in lectures
  – No questions about UARTs
  – No questions about ADC

• 60 points total
  – 15% of your course grade
EXAM TOPICS
Suggested preparation steps:

- Review the following slides. You should have a deep understanding of the content that appears on them.
- If not, go back and review the lecture material on the given topic.
- Run through the questions at the end of this PowerPoint.

This set of slides are not comprehensive.

- Review all lecture slides.
- Review homework questions, try to re-do those questions.

http://class.ece.iastate.edu/cpre288
Keywords

- char
- short
- int
- long
- float
- double
- enum
- struct
- union
- break
- case
- continue
- default
- do
- else
- for
- goto
- if
- return
- switch
- while
- auto
- const
- extern
- register
- signed
- static
- unsigned
- volatile
- sizeof
- typedef
- void
• Could you write the following statements by hand?
  – Loops (for, while, do)
    • Write a **for loop** to sum elements of an array or count characters in a string
    • Do you know the syntax of a **do while loop, for loop, and while loop**?
  – typedef
    • Could you write a typedef definition
    • Do you know what it means when you see a variable type like **uint8_t**?
  – Switch statements
    • Do you know where the semi-colon and colons go in a switch/case statement?
    • Do you understand how the control flow can fall through a case?
  – Control flow
    • Do you understand the keywords **break** and **continue** and their use?
Know your Operators

• Do you know the difference between these two sets of operators?

```plaintext
&
|
~
^  
```
char a = 20, b = 10, r = 5;

// math operations
r = a + b;
r = a - b;
r = a * b;
r = a \ b;
r = a % b;

// bitwise operators
r = a & 3;
r = a | b;
r = a ^ 0xFF;
r = ~a;
r = a >> 3;
r = b << r;

// conditional
r = (r) ? a : b;

// boolean
r = a || b;
r = a && b;
r = !a;
r = a < 20;
r = b <= 15;
r = b > 10;
r = a >= b;

// post and prefix
a++;
++a;
b--;
--b;

// assignments
r = a = b = 42;
r += a;
r -= b;
Know how to use Operator Precedence

- Can you use this table?

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>++ -- ()</td>
<td>Suffix/postfix increment and decrement</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>2</td>
<td>++ -- ()</td>
<td>Prefix increment and decrement</td>
<td>Right-to-left</td>
</tr>
<tr>
<td></td>
<td>+ - ! ~</td>
<td>Unary plus and minus, Logical NOT and bitwise NOT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(type)</td>
<td>Type cast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>* &amp; sizeof</td>
<td>Indirection (dereference), Address-of, Size-of</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>* / %</td>
<td>Multiplication, division, and modulus (remainder)</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>4</td>
<td>+ -</td>
<td>Addition and subtraction</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&lt;&lt;= &gt;&gt;=</td>
<td>Bitwise left shift and right shift</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>For relational operators &lt; and ≤ respectively, &gt; and ≥ respectively</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>== !=</td>
<td>For relational = and ≠ respectively</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>&amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>^</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>&amp;&amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>?:</td>
<td>Ternary conditional</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>14</td>
<td>= += -= *= /= &amp;= ^=</td>
<td>=</td>
<td>Assignment by sum and difference, Assignment by product, quotient, and remainder, Assignment by bitwise left shift and right shift, Assignment by bitwise AND, XOR, and OR</td>
</tr>
<tr>
<td>15</td>
<td>,</td>
<td>Comma</td>
<td></td>
</tr>
</tbody>
</table>

http://class.ece.iastate.edu/cpre288
void main() {
    char x = 5, y = 10;
    char z;
    char array1[10];
    char array2[] = {1, 2, 3};
    char array3[5] = {1, 2, 3};
    char *str = “Hello!”;

    int i = 7;
    int *ptr = &i;
    int **pp = &ptr;
    char *p;
}
Know your Declarations

• Do these declarations make sense?

```c
struct House {
    unsigned long value;
    unsigned char baths;
    unsigned char bedrooms;
    unsigned char stories;
    unsigned long footage;
};

void main() {
    struct House my_home;
    struct House *bob_home = &my_home;

    my_home.baths = 1;
    my_home.value = 115000;
    bob_home->baths = 3;
    bob_home->value = 230000;
}
```
Data Structures

• Array access
• Pointers
  – Dereference
  – Address operator
• Access members of structs and unions
• Bit-field defined struct
• Know the difference between a struct and union
Pointers

• What are pointers
• Relationship between
  • pointers
  • array names
• Pointer arithmetic
• Students: Be sure to review class examples and homework problems related to pointers
C-strings

• Review the concept of C-strings
• Relationship between
  – C-strings
  – arrays
• Understand the importance of the NULL byte of a C-string
Bitwise operations

- Setting bits to 1
- Clearing bits to 0
- Testing for bits set to 1
- Testing for bits cleared to 0
- Generic systematic checking example

```c
if( (x & MASK_ALL1s) == MASK_ALL1 &&
    (~x & MASK_ALL0s) == MASK_ALL0s &&
    (x & MASK_ANY1s) &&
    (~x & MASK_ANY0s) )
```

http://class.ece.iastate.edu/cpre288
Variable Scope

- Global variables
- Local variables
- Local static variables
- Global static variables
- volatile variables
Memory Mapped I/O

• Concept of Memory Mapped I/O
  – Device registers can be accessed from a program as if accessing memory (i.e. devices’ registers are “mapped” to memory addresses)
  – Allow sending commands and checking status of a devices by just reading and writing to memory mapped locations.
• Polling device status vs. device initiating an interrupt.
Memory Layout

• Stack
  – What type of information is placed in the stack?
  – Typically starts at the top of memory and grows downward

• Function’s/Procedure’s **Stack-Frame**: The information placed on the stack by a function when it is called (e.g. local variables, input parameters, return address)

• Heap
  – Stores dynamically allocated memory (i.e. allocated using `malloc`).
  – Typically starts at a low address and grows upward

• Static Data segment: Region of memory where Global variables and static local variables are stored.
Interrupts

• What is an interrupt?
  – An event that occurs from outside your program (e.g. Keyboard requesting service, Timer that times out, UART receiving a byte of information)

• What is an Interrupt Service Routine (ISR)?
  – Code associated with a given interrupt (i.e. a given event), that executes when that interrupt occurs.

• What actions are taken when an interrupt occurs?
  – Normal program execution is paused
  – ISR execution begins
  – Normal program execution resumes after ISR completes
There will be 1 or 2 problems where you will be asked to write the body of a function to implement a defined computation.
Show your work

• Please show intermediate steps. This will help us give partial credit

• Write down assumptions. In general Profs/TAs will not answer questions during the test.
  – They will just tell you to write down your assumptions
  – The exception will be if a major typo is found by a student.
A. How many bytes are each of the following types (on the ATmega128)?
   • char, short, int, long, float, double

B. What range of values can be stored in an unsigned char?

C. What range of values can be stored in a signed char?

D. What is the value stored in \( x \) after this code runs?
   ```c
   int x, y, z;
   x = y = z = 10;
   ```
<table>
<thead>
<tr>
<th>Name</th>
<th>Number of Bytes</th>
<th><code>sizeof()</code></th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td></td>
<td>-128 to 127</td>
</tr>
<tr>
<td>signed char</td>
<td>1</td>
<td></td>
<td>-128 to 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td></td>
<td>0 to 255</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td></td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
<td></td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>int (on ATmega 128)</td>
<td>2</td>
<td></td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>(pointer on ATmega 128)</td>
<td>2</td>
<td></td>
<td>Address Space</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td></td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>signed long</td>
<td>4</td>
<td></td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4</td>
<td></td>
<td>0 to 4294967295</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td></td>
<td>-4294967295 to 4294967295</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td></td>
<td>±1.175e-38 to ±3.402e38</td>
</tr>
<tr>
<td>double (on ATmega 128)</td>
<td>4</td>
<td></td>
<td>±1.175e-38 to ±3.402e38</td>
</tr>
</tbody>
</table>
A. Analyze the following code:

```c
char r = 0, s = 1, t = 2;
char *p1 = &s;
char *p2 = &t;
char **pp3 = &p1;

*p1 = 10;
**pp3 = 15;
*p2 = 30;
*pp3 = &r;
**pp3 = 5;
*p1 = 25;
```
A. Analyze the following code:

```c
char r = 0, s = 1, t = 2;
char *p1 = &s;
char *p2 = &t;
char **pp3 = &p1;

*p1 = 10; // s = 10
**pp3 = 15; // s = 15
*p2 = 30; // t = 30
*pp3 = &r; // p1 = &r
**pp3 = 5; // r = 5
*p1 = 25; // r = 25
```

<table>
<thead>
<tr>
<th>r</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>25</td>
<td>15</td>
<td>30</td>
</tr>
</tbody>
</table>
Question 3a

• When is the condition of the following \textbf{if} statement true?

\begin{verbatim}
if (((x = 3) || (x & 1)) {
   // do something
}
\end{verbatim}
• When is the condition of the following `if` statement true?

```java
if ((x = 3) || (x & 1)) {
  // do something
}
```

• **The statement is always true.** Know the difference between the assignment operator (`=`) and the equality operator (`==`).
  – The value on the left `(x = 3)` is always true, as the value of an assignment is the value that was assigned. This allows programmers to have compound assignments.
Question 3b

• When is the condition of the following if statement true?

```java
if ((x == 3) || (x & 1)) {
    // do something
}
```
• When is the condition of the following `if` statement true?

```java
if ((x == 3) || (x & 1)) {
    // do something
}
```

• The statement is true if `x` is either equal to 3 or bit 0 is set.
Question 4a

• When is the condition of the following if statement true?

```java
if (x & 0x08 == 0x08) {
    // do something
}
```
Question 4a (answer)

• When is the condition of the following if statement true?

```java
if (x & 0x08 == 0x08) {
    // do something
}
```

• The statement is true if bit 0 of x is 1. Operator precedence evaluates the == operator before the bitwise AND (&). Assumes TRUE (i.e. 0x08 == 0x08) evaluates to 1
• When is the condition of the following if statement true?

```java
if ((x & 0x08) == 0x08) {
    // do something
}
```
• **When is the condition of the following** \textbf{if} **statement true?**

\begin{verbatim}
if ((x & 0x08) == 0x08) {
    // do something
}
\end{verbatim}

• **The statement is true if bit3 of x is set.**
  - \( x = 0b00001000; \)  condition is \textbf{TRUE}
  - \( x = 0b01001110; \)  condition is \textbf{TRUE}
  - \( x = 0b00101001; \)  condition is \textbf{TRUE}
  - \( x = 0b00000000; \)  condition is \textbf{FALSE}
  - \( x = 0b11100000; \)  condition is \textbf{FALSE}
What are the values of c1, c2, c3, and c4 after the following code executes?

```c
char myarray[3] = {1, 2, 3};
char *ptr = myarray;

char c1 = *ptr++;
char c2 = *ptr;
char c3 = myarray[0];
char c4 = myarray[1];
```
char myarray[3] = {1, 2, 3};
char *ptr = myarray;

char c1 = *ptr++;
char c2 = *ptr;
char c3 = myarray[0];
char c4 = myarray[1];

• Postfix increment has higher association precedence than dereference operator. But the increment does not occur until the next line

• c1 is 1
• c2 is 2
• c3 is 1
• c4 is 2
Question 6b

• What are the values of c1, c2, c3, and c4 after the following code executes?

```c
char myarray[3] = {1, 2, 3};
char *ptr = myarray;

char c1 = (*ptr)++;
char c2 = *ptr;
char c3 = myarray[0];
char c4 = myarray[1];
```
char myarray[3] = {1, 2, 3};
char *ptr = myarray;

char c1 = (*ptr)++;  // c1 is 1
char c2 = *ptr;  // c2 is 2
char c3 = myarray[0];  // c3 is 2
char c4 = myarray[1];  // c4 is 2