EE472 Lecture Notes Pack

Blake Hannaford, James Peckol, Shwetak Patel
Department of Electrical Engineering
The University of Washington

April 3, 2009

Contents

Embedded Microcomputer Systems

Embedded: Special purpose computing devices. Computers that are *embedded* or hidden inside something else. Examples: Xerox machine controller, Automotive Anti-lock braking system, Airliner autopilot, MP3 Player, DVD player, TiVo Box, CelPhone.

Microcomputer: A computer you can buy for $10 - $1000. Almost all of the same software and hardware issues we will study have been around since the 1960's. The only thing different is the cost of the hardware. All of the following had *essentially* the exact same *software environment* to what we are studying in this class:

Era	Technology	Cost
1960's	Mainframe	$1M
1970's	Minicomputer	$100k
1980's	PC's	$5k
1990's	Cellphone	$500
2000's	Microcontroller	$50

Systems: From "The Random House Dictionary"

> **1.** a group or combination of things or parts forming a complex or unified whole.
> ...
> **5.** a set of body organs or related tissues concerned with the same function.
> ...
> **7.** Often: an organized set of computer programs.

Chapter 2

C Program Structure

C Programs in Multiple Files

A C program is typically written in multiple files.

Simpler editing

Helps modularity and multiple authors.

File: main.c

```
main()
  {
   ... code  ...
  }

fcn1()
  {
  ... code ...
  }
```

File: functions.c

```
fnc2()
  {
   ... code  ...
  }

fcn3()
  {
  ... code ...
  }

... etc ...
```

Execution starts with first line of `main()` .

All functions (except `main()`) require a **function prototype** in any file in which they will be called or defined.

All functions are visible from any file (but prototype must be supplied in the file).

2.1 Include Files (.h files)

Text can be read in to your C-Source file just prior to compilation.

File: A.c

```
#include X.h
main()
  {
   ... code  ...
  }
```

File: X.h

```
/*
 hello, this is just a comment
*/
```

output:

```
/*
 hello, this is just a comment
*/
main()
  {
   ... code  ...
  }
```

Reasons:

- `#include` text which should be **the same** in multiple ".c" files.
- keep and edit just 1 copy.
- less typing and ensures consistency.

2.2 Other C pre-processor functions

2.2.1 `#define`

like the search-replace function in your word processor.

ex:

`#define TRUE 1`

c pre-processor will replace all occurances of `TRUE` with `1`.

2.2.2 `#ifdef / #endif`

C pre-processor can turn on-off blocks of code before compilation.

ex:

```
#define LINUX 1
//#define WINDOWS 1

...

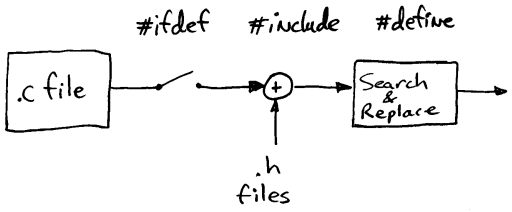
#ifdef WINDOWS
```

```
     ... windows-specific code ...
#endif

#ifdef LINUX
     ... linux-specific code ...
#endif
```

2.2.3 C pre-processor overview

(draw your own graphic)



2.3 Building a C program

For each ".c" file:

1. Run C pre-processor on the ".c" file
2. Compile output into "object" file.
   - Incomplete machine language program
   - ".o" file for each ".c" file

Then: "Link" all ".o" files, and library files, to create "executable" binary file.

2.4 Variable Scope

- Variable declared inside a function is *local*.
- Variable declared outside a function is visible anywhere inside that ".c" file.

- Variable declared outside a function can also be visible in other ".c" files — if `extern` is used.
- Variables declared with prefix `extern` must be declared outside a function in *another* ".c" file.

2.5 Function Scope

- Functions cannot be used without a function prototype.
- Each ".c" file must have a function prototype for each function which is used in that ".c" file.
- `#include` can help manage your function prototypes.

Chapter 3

Pointers

3.1 Memory Addresses and Pointers

Consider the following code:

```
\\ variable declarations
int j=3;
int *pa;
int a;
int b = 0x2F;

\\ executable code
pa = &j;
a = *pa;
*pa = b;
```

3.1.1 Memory Map

Assume compiler assigns memory up from `0x3000`. Also assume that an integer is 32 bits (4 bytes). Show each byte of memory as it would be assigned by the compiler.

Name	Addr				
	3000				

3.1.2 Incrementing Pointers

C Pointers advance in increments of the thing they point to.

```
int myInt;
float myFloat;

int* intPtr = &myInt;
float* floatPtr = &myFloat;
```

```
int szint = sizeof(myInt);
int szflt = sizeof(myFloat);

// Assume intPtr has the value 0x3000
//  and floatPtr == 0x4000
//  and szint = 4, szflt = 8 (bytes)

printf("intPtr   = %d\n", (int)intPtr);
printf("intPtr+1 = %d\n", (int)intPtr+1);
printf("fltPtr   = %d\n", (int)fltPtr);
printf("fltPtr++ = %d\n", (int)fltPtr++ );
```

3.1.3 Pointer Dereferencing

Consider

```
int *pa;  // (assume (int)pa == 0x3000)
*pa = 0x000B;
*(pa++) = 0x0010;
*(pa+1) = 0x00C0;
```

What does memory look like?
What are the values of

```
*pa + 1 == ??
(int)pa == ??
```

Name	Addr				
	3000				

3.1.4 Example: Pointers and Arrays

(Lewis, Section 3.6)
Consider the original IBM-PC 80 col x 40 rows, display buffer

This is the graphics hardware that generates the clunky screen that you use to configure your `BIOS`.

	addr	col 1		col 2		col 3	
row 1	B8000	ch	cl	ch	cl	...	
row 2	B80A0	ch	cl	...			
...		ch	...				
		...					

where ch is a `char` display character, and cl is a `char` which encodes the color of that character position.

Lewis Approach

```
#define DISP_BUFFER 0xB8000
int row, col;
char disp_char;
char *p;
//       Let's display "A" in row 4, col 20
disp_char = 'A';
row = 4;        // for example ...
col = 20;       //  "    "

p = (char *) (DISP_BUFFER + 2 *(80*row + col))
*p = disp_char;
```

3.1.5 Exercise in Class

Complete this example: write code to set the byte which controls the **color** of the character `A` we just displayed.

Assume the color byte should be assigned the value `COLOR_BYTE`.

The color byte has 2 4-bit fields which set the background color (16 possible colors) and the character color (16 possible colors).

3.1.6 Arrays

An array is an ordered set of memory elements. For example, `int j[3]` sets up memory for three integers known as `j[0], j[1], j[2]`.

Graphically, if

```
int a[5]=0;  // array of 5 ints
             // each int is 2 bytes
```

Element name	a[0]	a[1]	a[2]	a[3]	a[4]
Address	3000	3002	3004	3006	3008

3.1.7 Initializing Arrays

When you declare an array such as

```
int a[5];
```

Enough storage is allocated for 5 integers (typically 5×32 bits or 40 bytes). But you should not count on any initial value.

To initialize the array use one of two methods:

```
int q[3] = {0,1,2};

or

int i, q[3];

...

for (i=0;i<3;i++) q[i] = 0;
```

3.1.8 Arrays and Pointers

Arrays are implemented inside `C` just like pointers.

Example:

```
int a[10], *p;
p = &a[0];
a[3]  <----->  *(p+3)
are exactly the same.

AND

a <------->  &a[0]
are exactly the same.
```

You can use the array name (`a`) without the subscript (`[1]`) and it is identical to a pointer to the first element. Also, any pointer can be subscripted whether it was declared as an array or not.

3.1.9 Array/Pointer Example

To find the character at each row/col, we had
`p = (char *) (DISP_BUFFER + 2 *(80*row + col)`
Can we express this as an array reference?

Yes

```
p = (char *) (DISP_BUFFER + 2 *(80*row + col));
p = 'A';
```
is just the same as

```
char *disp_buff = DISP_BUFFER;

disp_buff[2 *(80*row + col)] = 'A';
```

Even more succinctly:

```
typedef char      CELL[2];  // a CELL is 2 chars
typedef CELL      ROW[80];  // a ROW is 80 CELLs

ROW *disp_buff = (ROW *)DISP_BUFFER;

disp_buff[row][col][0] = 'A';
disp_buff[row][col][1] = COLOR_BYTE ;
```

3.1.10 Exercise in Class

Modify the scheme in Lewis, sec 3.6, so that we can address display characters and display colors each in a separate row/column array.

3.2 Generic (`void`) Pointers

Sometimes we want a pointer which is not locked to a specific type. It can potentially point to anything.

```
void *name ;   // declare a generic pointer called name
```

- `name` can point to anything in the computer.
- `name` cannot be dereferenced with *
- Must instead assign value of `void` pointer to a pointer of the type you want.

Examples

```
void* myGenericPtr;
int t, *ip, myvalue = 3;
myGenericPtr = &myvalue;  // OK
t = *myGenericPtr;  // NO!!

//*****************

ip = nyGenericPtr;
t = *ip;   // OK!!
```

3.2.1 Null Pointer

If a pointer has the value `NULL`, it points to nothing. `NULL` is a predefined constant in `<stddef.h>` or use
`#define NULL 0`
`NULL` is illegal to dereference.
`NULL` can be tested for:

```
int i,*ip = NULL;

[...]

if(ip == NULL) {
    // I haven't defined ip yet
    }
    else {  // OK, now I can use it!
    i = *ip;}
```

3.2.2 Function Pointers

C can have pointers to functions.

```
type (* functionpointer)(arg list)
```

Examples

```
int(*IntFuncPtr)();
\\   IntFuncPtr is a pointer to a function with
\\    no arguments which returns an int

double(*doubleFuncPtr)(int arg1, char arg2)
\\   doubleFuncPtr is a pointer to a function with
\\     an int and a char arg which returns a double.
```

Assignment to function pointers:

```
int (* IFP)(int x) = NULL;    // empty function pointer
int realfunction( int x);     // an actual function

IFP = &realfunction;
IFP = realfunction;    // can skip &
```

Dereferencing function pointers:

```
(*IFP)(5)    // call function realfunction with arg 5

IFP(5)       // can also use function pointer just like
             // original function name
```

3.3 Pointer Arithmetic

A powerful feature of pointers is the ability to compute with them like integers. However only some operations are allowed with pointers.

Allowed:

- Add a scalar to a pointer
- Subtract pointers

Not Allowed:

- Add two pointers

- Multiply or Divide Pointers
- Multiply by a scalar
- Divide by a scalar

Example: Find the midpoint in an array.

```
#define SIZE 100
int  length, buffer[SIZE];
int *ptr1, *ptr2, *ptr3;

ptr1 = buffer;          // points to start of array
ptr2 = ptr1 + 100;      // points to end of array
length = ptr2 - ptr1;   // length = 100
ptr3 = ptr1 + length/2  // ptr3 points to mid-point of array
```

3.3.1 Pointer comparisons

`==, !=` Determine if two pointers are equal or not.
`<, <=, >=, >` Which pointer points to a higher address in memory? Which way will subtraction come out?

Chapter 4

Constants and Defines in C

`C` Constants, Defines and Integers [1]

4.1 Review of C Constants

We will often need to specify the content of memory very precisely.

4.1.1 HEX and OCTAL review

Entering binary numbers can wear out the 1 and 0 keys on your computer! HEX is a good shorthand for binary since each HEX character maps to a specific bit pattern.

bits	HEX	Octal
0000	0	0
0001	1	1
0010	2	2
...	...	...
0111	7	7
1000	8	10
1001	9	11
1010	A	12
1011	B	13
...	...	...
1111	F	17

Hex Place value:
To convert `0xD2FB` to decimal:

D	2	F	B
2^{12}	2^{8}	2^{4}	2^{0}
4096	256	16	1

Octal can also be used and is easier to memorize because there are only 8 symbols (0-7).

[1] B. Kernigan and D. Ritchie, "The C Programming Language,," 2nd Edition, Prentice Hall, 1988.
Daniel W. Lewis, Fundamentals of Embedded Software, *Where C and Assembly Meet*, Prentice Hall, 2002.

However, to fill a byte of memory requires 2.5 octal characters since each character only encodes 3 bits. For example,

`HEX 9A` = Binary `1011010` = Octal `132`

- Ordinary numbers are interpreted by the C compiler as decimal values. Example: `25 = 1101`
- Hex values are indicated to the C compiler by the prefix `0x` for example: `0x2F`
- Octal values are indicated by a leading zero: `028` is a syntax error (why?).

4.1.2 Constant Syntax in C

It is very important to use symbolic constants (`#define`'s) for *all* your numeric constants. Here are the main reasons:

- Code is more readable by humans.
- Code is much easier to change.

Example and Exercise

```
// Constants
#define EXAMPLE_CONST_D  1234 ;  /* DECIMAL value  */
#define EXAMPLE_CONST_H 0x4D2 ;  /* HEX value  */
#define EXAMPLE_CONST_O 02322  ; /* OCTAL value */

int x = EXAMPLE_CONST_D;
int y = EXAMPLE_CONST_H;
int z = EXAMPLE_CONST_O;

// test yourself here: (which of these print?)
if (x == y) printf ("Hello there ... \n");
else {};

if (y == 1234) printf ("What's up doc?? \n");
else {};

if (z == 2322)  printf ("The Rain in Spain  ... \n");
else {};

if (z = 2322)  printf ("Falls mainly on the plains.\n");
else {};
```

Example 2

Suppose you are going to control a CD-ROM drive and you want to open and close the drive door. The bit to open and close the drawer is the 4th bit in a register located at address `0x2DA`. Let's say that to set that bit we use the function

`IO_Register_Set([addr],[value])`

where `[addr]` is the bus address of the I/O register you want to manipulate and `[value]` is the bit pattern you want to put there.

It is "technically" correct to use the following code:

```
\\ open CD-ROM door
IO_Register_Set(0x2DA, 0x10) ;
```

However, if we come back to that code later (1 week or 10 years) we will not know what that statement does (or if the comment is correct) without a lot of research. Instead we use defines for much better code as follows:

```
\\ (this part is in the preamble before "main ()"
\\  or in a special ".h" include file
#define  CD_ROM_CONTROL_REG  0x2DA
#define  CD_ROM_DOOR_OPEN    0x010

\\ .... skip to inside the code

\\ open CD-ROM door
IO_Register_Set(CD_ROM_CONTROL_REG, CD_ROM_DOOR_OPEN) ;
```

Now, isn't that better?!! It is a requirement of EE472 that your code be written this way. We will take off points for any numberical constants in the main body of code.

4.2 Bitwise vs. Logical Operators

The familiar Boolean operations **AND**, **OR** and **NOT** are provided *two ways* in C.

4.2.1 Logical Operators

These work on *arithmetic* types or pointers.

True	a non zero value.
False	zero.
&&	**AND**
`A && B`	True if *both* `A` and `B` are non-zero.
`||`	**OR**
`A || B`	True if *either* `A` and `B` are non-zero.
`!`	**NOT**

4.2.2 Bitwise Operators

A very important C feature for embedded microcomputer systems. These operators are only defined for integer-like variables i.e. `char, short, int, and long` signed or unsigned.

&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	left shift
>>	right shift
~	ones complement

4.2.3 Examples:

```
/* Use of bitwise logical operators  */

#define  Bit_Zero   0x01
#define  Bit_One    0x02
#define  Bit_Two    0x04
#define  Bit_Three  0x08
#define  Bit_Four   0x10
#define  Bit_Five   0x20
#define  Bit_Six    0x40

// etc etc etc ..


int x = 0x0B;           // x  = [... 0 0 0 0 1 0 1 1]

int y = 011;            // y  = [... 0 0 0 0 1 0 0 1]

int z = x << 2;         // z  = [... 0 0 1 0 1 1 0 0]

int z1 = y & Bit_Three; // z1 = [... 0 0 0 0 1 0 0 0]

int z2 = z | Bit_Four;  // z2 = [... 0 0 1 1 1 1 0 0]
```

4.3 In-class Exercise:

```
Convert 0x1A to decimal: __________


Convert 0xA2 to decimal:  __________


Convert 020 to Hex:  ______________________


Convert 0x20 to Octal: ______________________


/* Give the binary value of the least significant
8 bits of the following values:   */

int aa = 0xC5;     // aa = [                    ]

int ab = 017 + 1;  // ab = [                    ]

int a = x | y;     // a  = [                    ]

int b = Bit_Two | Bit_Five | Bit_Seven;
                   // b  = [                    ]

int c = ~((z | Bit_Zero) << 1);
                   // c  = [                    ]

int d = !(z1 | Bit_Two);
                   // d  = [                    ]
```

4.4 Binary Integers

4.4.1 Unsigned Integer

Typically just a 16 bit binary number. Straightforward, but cannot represent negative numbers.

If `x` is of type `unsigned int`, then

$$0 \leq \texttt{x} \leq 65535$$

C declaration of unsigned integers:

```
unsigned char a,b;            // 8 bit unsigned
unsigned int  x,y,z;          // 16 bit unsigned
unsigned long int p,q,r;      // 32 bit unsigned
```

Exercise

Write a C function using bitwise logical operators etc. to print the binary value of a 16 bit word.

4.4.2 2's Complement Integers

Most common way to represent integers which can have both positive and negative values. Most significant bit (MSB) is the "sign bit" 0=positive, 1=negative. however, if MSB is 1, other bits are subject to "2's complement".

To convert a number to or from two's complement,

1. complement all bits
2. add 1

Taking the 2's complement of the entire n bits is equivalent to negation.

For an n bit number, x, the range of all possible integers in 2's complement is

$$-2^{n-1} \leq x \leq (2^{n-1} - 1)$$

(see Lewis Figure 2-4, page 22 for a useful chart).

Exercise

Write a c code fragment which negates a signed (2's complement) integer using bitwise logical operators and addition. Include declarations for your variables. (do not just multiply by -1)

Chapter 5

Tasks and Schedulers

5.1 Tasks, Processes, and Threads

5.1.1 Processor Context

The CPU of a computer has several state variables. Once they are specified we know the exact state of the CPU. These include:

- Program Counter (location in memory of next instruction)
- Value of each register
- Processor Status Flags
- Stack Pointer

5.1.2 Tasks or Processes

A *Task* or *Process* is a unit of code and data which is described by one processor context when it is running.

Examples: *The Tasks of Labs 2-4*

A task is usually implemented in `C` by writing a function.

Types of Tasks:

Periodic Tasks

- Found in Hard-Real time applications
- Examples: 1) Control: Every 10 ms., Read sensors → compute control → output command 2) Multimedia: Every 22.727μsec, Get music sample → compute DSP filter → output sample to DAC.
- Characterized by three attributes:
  1. P, Period: the regular time interval between runs of this task.
  2. C, Computing Resources: How much CPU time does it require each time. Obviously $C <= P$. (C may not be the same each time.

     Different code branches in the task may take different amounts of time. Use $C = C_{max}$.

3. D, Deadline: How quickly must the task me completed after it is started each time tick. $C < D < P$

- Typical Code Structure:

```
name() {
   compute for C seconds;
   return();
   }
```

Intermittent Tasks

- Found in all types of applications
- Examples: 1) Send an email every night at 4:00 AM
  2) Save all data when power is going down
  3) Send a message to plant operator when tank runs low.
  4) Calibrate a sensor on startup.
- Characterized by two attributes:
  1. C, Computing Resources: How much CPU time does it require each time.
  2. D, Deadline: How quickly must the task be completed after it is started. (whenever that happens to be).
- Typical Code Structure:

```
name() {
   compute for C seconds;
   if(! done) return();
   else halt_me();
   }
```

Background Tasks

- A soft real time or non real time task.
- Lower Priority
- Will be accomplished only as CPU time is available when no hard real time tasks are ready.
- Characterized by:
  - C, Computing Resources: How much CPU time does it require each time between scheduler accesses.
- Typical Code Structure:

```
name() {
   compute for C seconds;
   if(! done) return();
   else halt_me();
   }
```

Complex Tasks

- Found in all types of applications
- Examples: 1) Microsoft Word
  2) Apache Web Server.
- Characteristics:
  1. Continuous need for CPU time.
  2. Frequent requests for I/O which free up the CPU.
  3. Waits for user input which free up CPU.
- Typical Code Structure:

```
name() {
   while(1) {
      compute for C seconds;
      request_IO();  // also starts scheduler
      }
   }
```

I/O activity frees up the CPU because the task must wait, usually a significant amount of time, for the I/O activity on a physical device. It cannot compute between the time it requests the I/O and the time it gets the data or writes the data so now the CPU is free for other tasks.

5.1.3 Task States

Tasks are in one of four states:

1. `Running`
2. `Ready` to Run (but not running)
3. `Waiting` (for something *other* than the CPU.)
4. `Inactive`

Only one task can be **Running** at a time (unless we are using a "multicore" CPU). A task which is waiting for the CPU is **Ready**. When a task has requested I/O or put itself to sleep, it is `Waiting`. An `Inactive` task is waiting to be allowed into the schedule. It is like Microsoft Word when you are NOT running it.

We'll see later how a task can get into or out of the `Inactive` state.

5.2 Threads: Smaller Units of Computation

We have talked about:

- Programs
- Tasks
- Subroutines/Functions[1]
- Processes

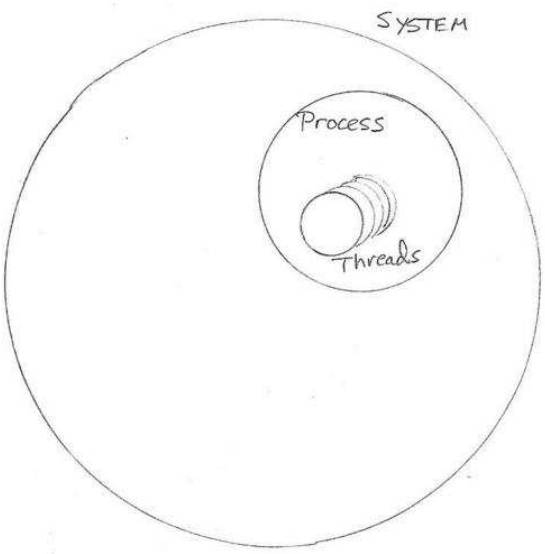
[1]When I use a slash (/) in a list like this I mean that the two terms are equivalent

- Threads

Confused yet??

With any complex computing system we naturally break it up into *units* .
For example in the EE472 lab exercise: "tasks"
Each unit has

- Code/instructions
- Data
- Context/State
- Resources (memory, I/O devices, semaphores)



A *Program or Process* is a unit of computation with all of the above attributes.

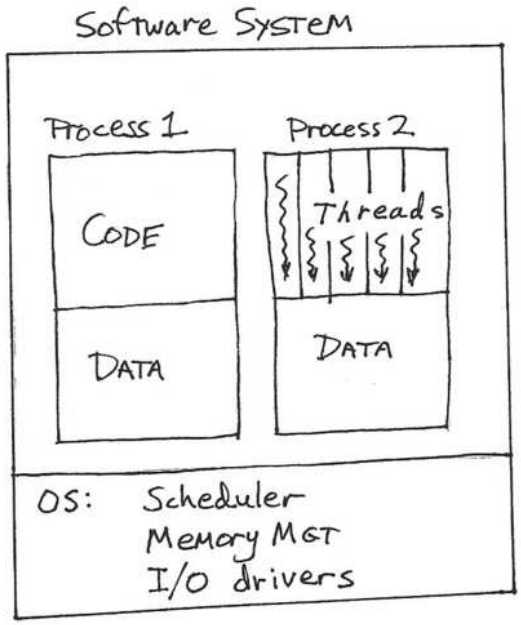
A *Subroutine or Function* is a unit of comptuation with code, data, and context, but no managed resources.

I am using the term "managed resources" to mean resources which can be assigned and deassigned by the operating system such as memory or I/O devices. Of course a function within a program has its own memory, but if it requests memory from the OS (for example with `malloc()`, the OS assigns it to the program, not to the function within that program.

A *Thread* is a unit of computation with code and context, but no private data. Threads may even share code with each other.

Threads are owned by a program/process.

Threads are like pieces within a program which can be scheduled by the OS independently.



A complete software system with two processes.
Process 1 is a "normal" single-threaded process. Process 2 is a multithreaded process.
Scheduler manages 6 units: Process 1 and the 5 threads of process 2.

5.2.1 Thread Context Switching

Since threads have a smaller context than programs, context switching is faster.

- Only save/restore CPU state
- No need to change memory setup

  (with programs in systems bigger than the 16-bit 80188, this can be elaborate).

Why use threads?
An example.
Web server.
Web server pseudocode:

```
while(1)  {
    OS_read_network();      // get incoming request
    compute;                // figure out what html page to get
    OS_Disk_I/O();          // read disk file for www page
    OS_write_network();     // send page to user
}
```

Alternative 1: Run above as a single process.
Problem: When process is waiting for network disk is idle and vice versa.
Alternative 2: Set up several threads of same code:
Advantage: Scheduler can start a thread which is done with disk I/O while another thread is waiting on network and vice versa.
CPU is used more efficiently and system delivers more pages per second with same hardware performance.
Potential Problem: more time spent context switching.

5.2.2 Types of threads.

There are lots of possibilities for size of *units*.

The more context a thread contains, the "heavier" it is.

Heavier threads take more time for context switching.

OS designer determines what kind of threads to support — application programmer gets only one option.

5.3 RTOS Scheduling Concepts

RTOS = Real Time Operating System

5.3.1 Time

Time is measured by a piece of hardware which records actual clock time: a *real time clock*: We use the terms

"Time Quanta" "Time Slice" "Ticks"

Let's call the period of time ticks: T.

Typically $T < P_{min}$. Sometimes $T << P_{min}$.

Where P_{min} is the shortest period of all tasks in the system. In other words, the operating system measures time in units, T, which are smaller than the shortest period of any task.

5.3.2 Soft vs. Hard Real Time

Real Time: A software system with specific speed or response time requirements.

Soft Real Time: If the deadlines are not met, performance is considered low.

Hard Real Time: A computer system in which at least one task must meet deadlines in time. If the deadlines are not met, the system has failed.

Super Hard Real Time: Mostly Periodic Tasks: Task periods = the OS time tick, task compute times (C) and deadlines (D) are very short. $P = D = T, C << P$.

5.3.3 Scheduling

At each opportunity, OS asks: If N tasks are **Ready**, which one should I run?

If you can show that scheduler always can achieve deadlines, the system is "deterministically schedulable".

5.3.4 Scheduling Goals

1. CPU Utilization:
$$U_{CPU} = 1 - \frac{idle}{period}$$
Tradional measure for mainframes. 100% is better.
But U_{CPU} = 100% is not safe for real-time systems
Goal: low load - 40%, high load - 90%.
2. Throughput: Work units completed per unit time.
3. Turnaround Time: Total time from task start to completion (waiting + execution + I/O)
4. Waiting Time: Choose scheduler which minimizes waiting time.
5. Response Time ** How quickly system responds to external asynchronus events.

Scheduler Types:

Periodic Schedulers

- Infinite Loop
  - Most primitive of all
  - Also known as Non-preemptive Round Robin.
  - ```
    while(1)
    {
    task1_fcn();
    task2_fcn();
    task3_fcn();
    ...
    }
    ```
  - Each task **must** voluntarily return to scheduler quickly
  - $C << P$, $P = T$.

    ```
    taskN_fcn()
    {
    compute a little bit;
    return();
    }
    ```
- Synchronized Infinite Loop
  - the top of the loop waits on a hardware clock.
  - ```
    while(1)
    {
    wait(CLOCK_PULSE);
    task1_fcn();
    task2_fcn();
    task3_fcn();
    ...
    }
    ```
  - Each task **must** voluntarily return to scheduler quickly, as above.
  - $C << P$, $T = P$.

The schedulers above are appropriate when all tasks are periodic and hard real-time. However, because execution times (C) vary, we have to allow extra time in each loop of the non-preemptive round robin schedule (taken up by the "wait()" function).

5.4 Scheduling Exercises

5.4.1 Overview

In this section we will conduct a series of in-class exercises. We will zoom in on exactly how each scheduling method works with a variety of real-time task types. Our exercises will study the following schedulers (with increasing sophistication):

1. Our basic `while(1)` loop: Non-preemptive Round Robin (RR).

2. The same loop with synchronization timing loop or timer function: Synchronized Non-premptive RR
3. Special hacks to the synchronized non-preemptive RR to enable different task periods.
4. Adding a `sleep(int d)` delay option for tasks: Non-premptive synchronized RR with delays.
5. Making the scheduler dynamic: Non-premptive synchronized RR with `halt_me()` function.
6. Handling I/O requests which take time.
7. Adding the ability to break into running tasks: *Pre-emptive* RR
8. Allowing for programmer-set priorities among the tasks: Pre-emptive *Priority Based*.

Diagramming Scheduler Operation

To study scheduling, print out a "Scheduler Diagramming Worksheet" on page 43. This chart plots time going down the page and CPU processes (tasks) across the page. We can think of the horizontal axis as the memory space of the computer. We will put a shaded box in a square (or part of a square) when a certain task is running on the CPU. For example, when the scheduler is running, we would shade under the "OS" column. We might also have three tasks labeled A, B, C. We would shade their columns when they are running. Remember, *only one column may be shaded in each row.*

Assume that each line represents 333μs. Assume that there is a system clock which generates timer events every 1,000 μs. To indicate these, make a mark at the left edge of every third horizontal line.

Use scheduler worksheet.

Each Line = 333 μsec.

Assume a system interrupt every 1,000 μsec. Mark interrupts at left edge of every third line. Each time scheduler runs it requires 100μsec (approximately 1/3 of a line).

Exercise 1

Scheduler Type: Non-synchronized, non-preemptive Round Robin

Task List and Characteristics:

- The three tasks are labeled A, B, C
- The nature of the three tasks is as follows:

  A Periodic. $P = .001$sec, $C = 200\mu$sec, $D = P$.

  B Periodic. $P = .002$sec, $C = 200\mu$sec, $D = P$.

  C Periodic. $P = .002$sec, $C = 200\mu$sec, $D = P$.

Diagram the tasks A,B,C as described above for the following scheduler policies:

Exercise 2

Scheduler Type: Synchronized, non-preemptive Round Robin

Task List and Characteristics:

- Same as Section 5.4.1
- The three tasks are labeled A, B, C
- The nature of the three tasks is as follows:

  A Periodic. $P = .001$sec, $C = 200\mu$sec, $D = P$.

  B Periodic. $P = .002$sec, $C = 200\mu$sec, $D = P$.

  C Periodic. $P = .002$sec, $C = 200\mu$sec, $D = P$.

Exercise 3

Scheduler Type: Diagram the schedule for the following variation on round-robin:

```
while(1) {
  wait(CLOCK_PULSE);
  taskA();
  taskB();
  taskC();
  wait(CLOCK_PULSE);
  taskA();
  }
```

Task List and Characteristics:

- Same as Section 5.4.1
- The three tasks are labeled A, B, C
- The nature of the three tasks is as follows:

  A Periodic. $P = .001$sec, $C = 200\mu$sec, $D = P$.

  B Periodic. $P = .002$sec, $C = 200\mu$sec, $D = P$.

  C Periodic. $P = .002$sec, $C = 200\mu$sec, $D = P$.

Delay Functions and Scheduling

Tasks frequently need to wait for a time interval.
Reasons:

- To create time-based events (such as Flash an LED for 0.5 sec).
- To wait for something external to happen such as for a device to be ready for more data.
- To free up CPU time for other processes.
- To determine for themselves how often they run:

```
while(1)
  {
  compute;
  sleep(20ms);  // Run every ~20ms.
  }
```

This task sets its own period, P, to 20ms. How accurate?

The worst way to do this is a delay loop such as

```
#define DELAY    22000
for (i=0; i< DELAY ; i++) ;   // just loop to use up time
```

Here we assume that we have figured out that 22000 is the right value to use for the time delay we want by trial-and-error or calculation. The worst bad thing about this method is that it doesn't meet the third reason, it uses all the CPU time itself. (There are other problems as well). We can define the call `sleep(int delay)` which does the following:

The scheduler can help implement delays with a function: `sleep(int delay)`:

- returns to the scheduler
- sets a software counter to `delay`. (This counter will be decremented every so often by an ISR until `counter == 0`.)
- schedules the next task.
- the task which called `OS_TimeDelay()` will be set to `Waiting` (not started by the scheduler)
- when the `delay` is over, set task to `Ready`

Exercise 4

Scheduler Type: Assume a time delay function: `sleep(int d)` is now available to the tasks within a non-preemptive round robin scheduler. Tasks take the form:

```
while(1) {
  compute for a while;
  sleep(25);
  return;
  }
```

Task List and Characteristics:

- The three tasks are labeled A, B, C
- The nature of the three tasks is as follows:

  A P=2.0ms, C= 233μsec, D=1.0ms

  B P=4.0ms, C= 233μsec, D=1.0ms, task pseudocode:

  ```
  taskB(void) {
    compute for 233 mu sec;
    sleep(3);
    return;
    }
  ```

  C P=1.0ms, C=100μsec, D=1.0ms

Dynamic Scheduling

Starting and Stopping Sometimes it is desireable to make a task stop and start. Recall the task states defined in Section 5.1.3. We defined the task state `Inactive`, to describe a stopped task. To implement this, assume that functions are available: `halt_me()` which moves a task from `Running` to `Inactive` states (takes a task out of the scheduler's loop). And `run(task)` which moves a task from `Inactive` to `Ready` (puts a task into the scheduler loop).

Why not allow tasks to start and stop?
Assume two functions:
`halt_me()`. Moves a task from `Running` to `Inactive`
`start(task)` Moves a task from `Inactive` to `Ready`.

Exercise 5: Scheduling with `halt_me()` function

Scheduler Type: Assume a halt function (`halt_me()`) is now available to the tasks within a non-preemptive round robin scheduler. Tasks now might look like

```
taskA(void *p){
  static int i=0;
  compute a while;
  if(i++ > 2) halt_me();  // turn off after 3 loops
  return;
  }
```

Task List and Characteristics:

- The three tasks are labeled A, B, C
- The nature of the three tasks is as follows:

  A Intermittent:

  ```
  while(1) {
  compute for 0.6566 ms;
  after 3 cycles of this halt_me();
  return;
  }
  ```

  B Complex: run for 1.0ms, then call the OS to do I/O which lasts for 2.0 ms. Repeat.

  C Background:

  ```
  while(1) {
  compute for 0.233 ms;
  return();
  }
  ```

Scheduling with I/O requests

I/O Sometimes tasks in embedded systems do I/O by directly manipulating hardware. For example, a task may set some bits on a digital output port using the various digital I/O registers in the microcontroller. These types of operation take very little time so they can be appropriate for periodic hard real time tasks.

On the other hand, more sophisticated devices such as disk drives or network interfaces require complex driver software and importantly take significant time to respond to commands. This type of I/O is typically supported by an embedded operating system which handles all

I/O requests through subroutine calls. Since tasks using these devices will have to wait a long time for them to respond, the scheduler is invoked after I/O requests. If a task is waiting for I/O, the scheduler will try to run another task until the I/O is complete. From a scheduler's point of view, I/O requests are a lot like `sleep(int d)` except `d` is not known in advance (we will specify `d` for schedule diagramming excercises or exam problems).

- Bit level I/O is fast and can be done by periodic hard real time tasks.
- I/O to devices like disk drives and network interfaces is different.
- Complex, slow.
- Requires driver software
- Operating System handles I/O with subroutine calls to driver.
- Example:
  - After task "B" calls for I/O, scheduler runs another task.
  - Task "B" will not be started by scheduler until requested I/O is finished.

Exercise 6. I/O Delays

Scheduler Type: Assume an I/O function: `OS_request_IO(int device)` is now available to the tasks within a non-preemptive round robin scheduler. Tasks now might look like

```
taskA(void *p){
  int device = HARD_DRIVE_0;
  compute a while;
  OS_request_IO(device);
  return;
  }
```

Task List and Characteristics: Diagram the schedule for the following tasks:

A $P = 1.0ms, C = 0.233ms, D = 0.5ms$

B Complex task:

```
while(1){
  compute for 0.233ms;
  I/O Request lasting for 1.333ms;
  }
```

Exercise 6a: Contention

Scheduler Type: Synchronized Non-preemptive RR.

Task List and Characteristics:

A Periodic. $P = 2$msec, $C = 233\mu$sec, $D = P$.

B Periodic. $P = 2$msec, $C = 656\mu$sec, $D = P$.

C Periodic. $P = 2$msec, $C = 233\mu$sec, $D = P$.

Who starts when 3 tasks are ready?

Exercise 6b

Scheduler Type: Synchronized Non-preemptive RR.

Task List and Characteristics:

A: Periodic. $P = 2$msec, $C = 233\mu$sec, $D = 500\mu$sec.

B: Complex:

```
while(1){
  compute for 1.5ms;
  Request I/O lasting 0.667ms;
  }
```

Foreground/Background

- A mix of hard real-time periodic tasks with Soft real time or non real time intermittent and complex background tasks.
- "Foreground" — the hard real time tasks
- "Background" — the soft/non real time tasks
- Example: Factory Automation Controller:
  - Real time control system (periodic, hard real-time, forground)
  - Hourly production report generator (intermittent, soft-real time, background)
  - Inventory Database Update (complex, non-realtime, background)

Exercise 7: Foreground/Background

Scheduler Type: Non-preemptive RR

Task List and Characteristics:

A: Periodic. $P = 2$msec, $C = 233\mu$sec, $D = 500\mu$sec.

```
while(1){
  compute for 0.233ms;
  sleep(1);
  return;
  }
```

B: Complex (background):

```
while(1){
  compute for 0.233ms;
  return;
  }
```

5.5 Pre-emptive Scheduling

So far, we have required that tasks return to the scheduler frequently so that other tasks get a chance to run in the CPU. This has two problems. First, we have to code our tasks somewhat artificially. They have to break their job up into chunks and keep track. Second, we are vulnerable if some task doesn't "play fair".

The solution to this is a scheduler that can break in to a running task using a regular interrupt such as a clock interrupt. There are many complex issues in having this ability but the value is very great. Preemptive schedulers solve this problem. While a task is running, the pre-emptive scheduler can stop it and run something else.

This idea changes how we can write tasks — for the better! We no longer have to divide our work into bite-size chunks which can fit into a small part of the OS tick and then keep returning. Instead, we just pretend that we are the only task needing the CPU:

Pre-emptive Scheduler can

- respond to a clock interrupt during a running task
- save context of the interrupted task
- look at the entire task list
- start another task (or possibly re-start the same task)

```
while(1){
  compute;
  }
```

The scheduler will now take care of breaking our work into small chunks. If we have nothing to do, we can use the `sleep()` function to release time to other tasks that we do not need.

If we have nothing to do for a while: sleep!

```
while(1){
  compute;
  sleep(100);
  }
```

In this case, we do our whole job, and then we wait for 100ms. We can use this creatively to manage time easily. For example, consider a task to generate two 10ms pulses, 26ms apart, followed by 200ms, then repeat the pulses. We could write this:

Example: Generate a 10ms pulse, wait 26ms, generate a 10ms pulse, wait 200ms, repeat.

```
while(1){
  Output(PIN, 1) ;   // set the first pulse
  sleep(10);
  Output(PIN, 0) ;
  sleep(26);
  Output(PIN, 1) ;   // set the second pulse
  sleep(10);
  Output(PIN, 0) ;
  sleep(200);
  }
```

Scheduler Types

Several types of pre-emptive schedulers are described below.

Types of pre-emptive schedulers:

- Pre-emptive Round Robin
  - Regular OS interrupts ("ticks").
  - Scheduler moves to next task each tick.
  - ```
    Task()
    {
    while{1}
       { compute ; }
    }
    ```
- Priority Based
  - Each task assigned priority by programmer
  - Highest priority task always gets the CPU if ready.
  - `(same)`
- Rate Monotonic (RMS)
  - As above but priorities are assigned based on period, P.
  - Short period = high priority.
  - Can prove optimality and schedulability (with assumptions).
  - `(same)`

  RMS Schedulability:

  C_i = execution time of task i.

  P_i = period of task i.

  n = number of tasks.

  if

  $$\sum_{i=0}^{n-1} \frac{C_i}{P_i} \leq n(2^{1/n} - 1)$$

  the system **is** schedulable using RMS.

  Assumption: All tasks pre-emptable 100% of the time.
- Priority Based: How to avoid `starvation`

  > **starvation**: The case in scheduling where one or more tasks get no CPU time but needs it.

  - Fixed *unique* priority, task-specific delay.
  - Must tweak delay values, M_i, to avoid starvation.
  - ```
    Task()
      {
      compute_some();
      OSTimeDelay(Mi);
      }
    ```
- Priority Based Round Robin: when each task must run each OS tick. (because of fast requirements).

  - All tasks delay 1 tick
  - All tasks execute $< \frac{T}{N}$ where T is the OS tick period and N is the number of tasks.
  - ```
    Task()
      {
      compute_a_little();
      OSTimeDelay(1);
      }
    ```

- FreeRTOS Style Priority-Based.
  - 7 priority levels
  - RR within each level.

Preemptive Scheduling Exercises

Exercise 8: Preemptive Priority Based Scheduling

Scheduler Type: Preemptive Priority Based. Priority order: A=2, B=1, C=3. (1= highest)

Task List and Characteristics:

A Intermittent: Run for 2ms and then `halt_me()`.

B Complex: run for 1.0ms, then call the OS to do I/O which lasts for 2.0 ms. Repeat.

C Background: Run forever.

Exercise 9: Preemptive Round Robin Scheduling

Scheduler Type: Preemptive RR (no priorities)

Task List and Characteristics:

A Intermittent: Run for 2ms and then `halt_me()`.

B Complex: run for 1.0ms, then call the OS to do I/O which lasts for 2.0 ms. Repeat.

C Background: Run forever.

Exercise 10: Pre-emptive RR Scheduling

Scheduler Type: Preemptive RR (no priorities)

Task List and Characteristics:

5 tasks. For each one P = 20ms, C=3ms, D=20ms.

OS time tick is 1ms.

How can you set up priorities (each task gets a different priority) and delay values to meet their schedules?
    ```

5.6 Scheduling Summary

Issue	Scheduling Concepts
How often will each task run?	Task types; Round Robin ; delay functions
Which task runs next?	Task State; Round Robin; Priority Schemes
How do tasks give up CPU?	`return()/yield()` ; Request I/O ; interrupts/preemption
What if task doesn't need to run all the time?	`dynamic scheduling ;` `halt_me()/start(task)`
How can we meet deadlines **and** make full use of CPU?	Background Tasks

5.7 RTOS Types

Type	C	D	Scheduling
``Super HRT''	$\ll P$	P	Trivial
``Hard RT''	$(1 \sim 5)T$	$(10 \sim 20)T$	Complex
``Soft RT''	$(10 \sim 20)T$	$\sim 100T$	Complex
Non RT	X	X	Trivial

P =task period; C = Compute Time; D =Deadline; T = Time Quantum Where P, C, D, T are defined in sections 5.1.2 and 5.3.1.

5.8 Scheduler Diagramming

Print out multiple copies of the Scheduler Diagramming Worksheet on the next page. We will use them in class to practice analysis of scheduling situations.

We will use each line in the worksheet to represent 333 μsec. Thus every three lines is one millisecond (our usual operating system time tick). Color in each column when its task should be running.

OS		A	B	C

Chapter 6

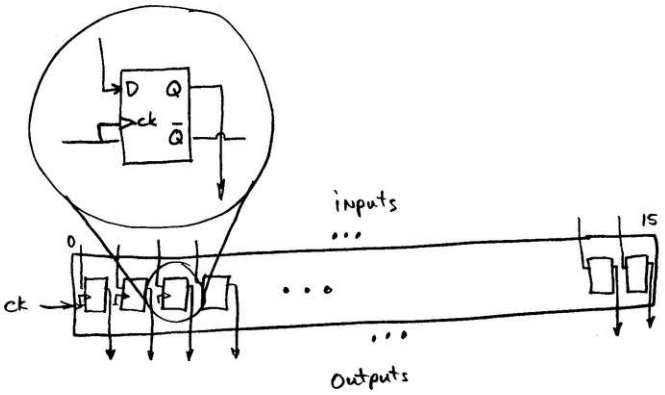
Embedded System Hardware and I/O

Hardware and Basic I/O

6.1 Basic Machine Organization

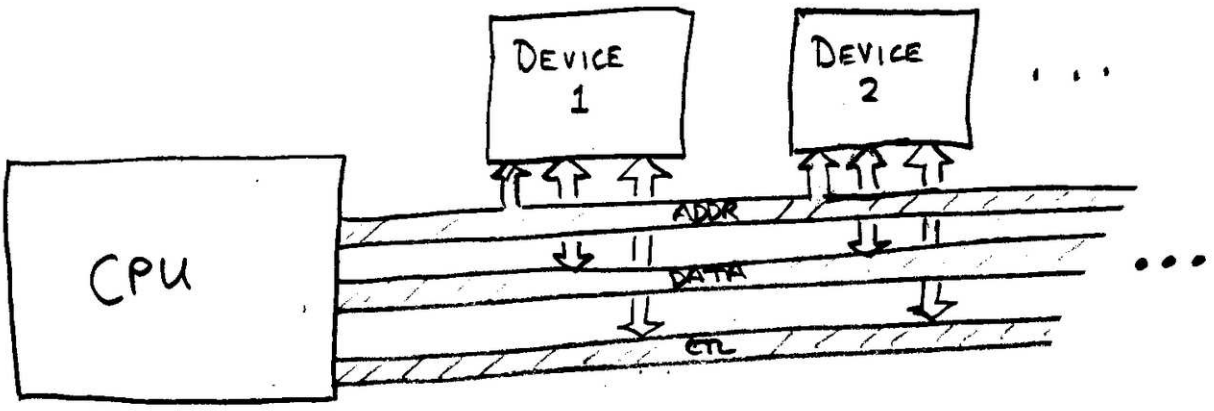
6.1.1 Registers

Registers are an array of D flip-flops which can store a collection of bits.
An n bit register has n inputs, n outputs, and one clock line.



6.1.2 Busses

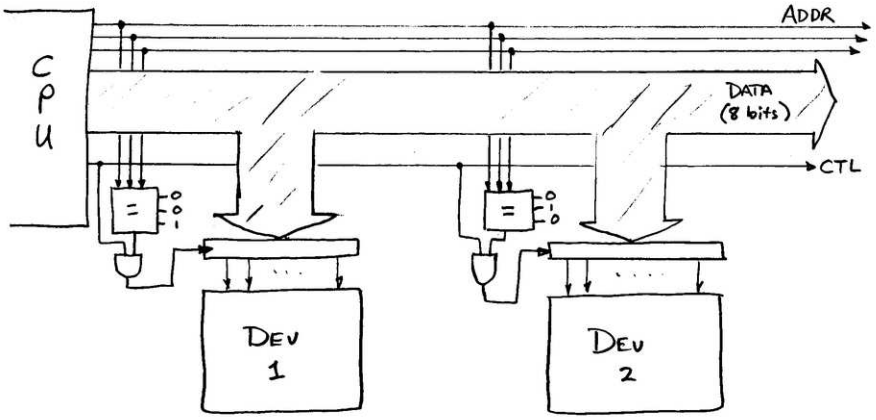
A bus is a parallel bi-directional data path
Multiple devices can send and receive on one bus



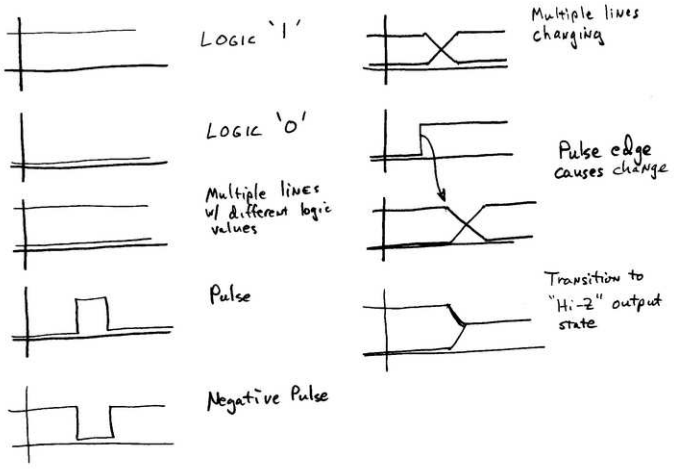
Address: N bits which specify *which location.*
Data: Contents to/from memory or I/O device
Control: 'traffic signals'
More detailed view:
Address Comparator

- A combinatorial logic circuit at each device
- 2 n-bit binary inputs:
  1. ADDR bus (changing)
  2. Device address (fixed)
- Control input
- Output = 1 iff input1 == input2

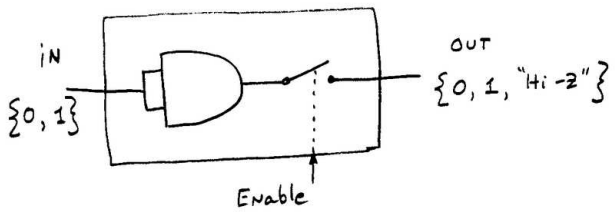
Control bus is gated by ADDR comparator
Device register is strobed by CTL pulse only if addresses match.



Bus Logic Notation:
Below is a simple notation for showing timing relationships of signals on busses.



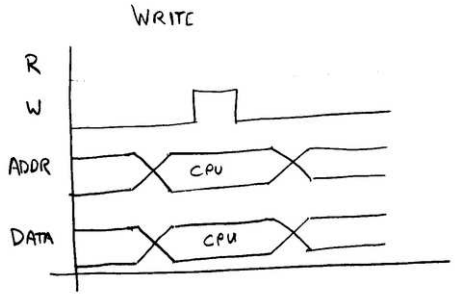
Problem: How can multiple logic outputs be connected to the same wire?
Solution: Tri-state logic.



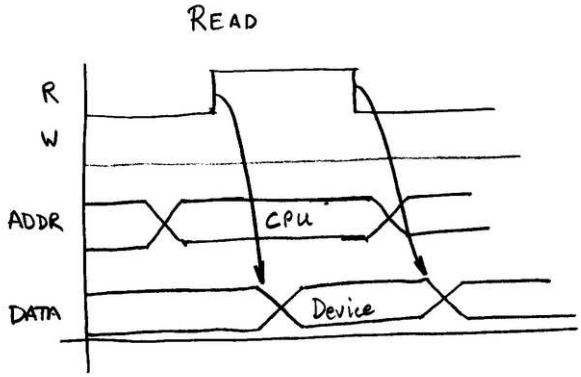
States: Logic 1, Logic 0, switch OFF
CPU and all devices can "talk" on data bus.
Only **ONE** device my have the tri-state switch closed at any time!

6.1.3 Bus Timing

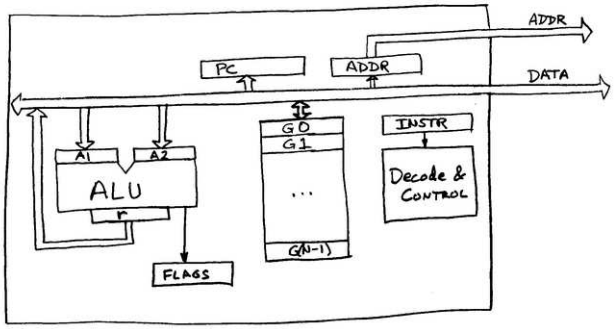
Write Cycle



Read Cycle



6.1.4 Microprocessor: Register View



- ALU: Arithmetic & Logic Unit

- PC: Program Counter. Holds Address of next instruction.
- IR: Instruction Register. Holds current instruction.
- ADDR: Address Register. Holds address of next bus access.
- GP#: General Purpose Registers. Hold intermediate results.
- A1, A2: ALU Arguments. Hold inputs to an ALU operation.
- r: 'Result', holds result of ALU operation.
- FLAGS: a set of bits which tell things like zero/non-zero, negative, etc about the last Result.

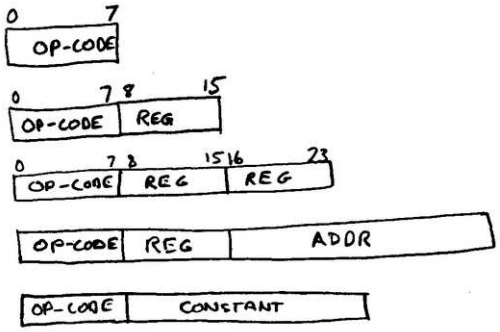
6.1.5 Steps to add two numbers

C statement: `c = a + b ;`

1. Load addr of `a` into ADDR
2. Wait for data from memory
3. Clock data into A1
4. Load addr of `b` into ADDR
5. Wait for data from memory
6. Clock data into A2
7. Send ADD command to ALU
8. Load addr of `c`
9. Transfer RES to data bus
10. Wait for data write to memory.

6.2 Machine Language Instructions

The above example may be controlled by one or more machine instructions. A machine instruction is binary data typically broken up into fields:
The Operation Code (Op-Code)
One, two, or three Operands.
Each instruction is typically between one and eight bytes.
Examples:



6.2.1 Example: Add two numbers

Note: All assembler below is pseudo-code!

C: `c = a + b;`

Assembler:

mem addr	instruction	comment
0xA000	MOV a, A1	move mem location a to ALU Arg 1
0xA002	MOV b, A2	move mem location b to ALU Arg 2
0xA004	ADD	a one-byte instruction
0xA005	MOV r, c	move ALU result to mem location c

There are many more types of operands in most architectures. Examples:

- INDEXED

```
MOV   0xCA04(R2), G1     \\move data from memory location
                         \\ 0xCA04 + contents of R2 to register G1
```

- REGISTER INDIRECT

```
MOV   (R2), G1           \\move data from memory location
                         \\ contents of R2
```

6.3 Basic I/O programming

Intel I/O instructions

`IN R, char` where `char` is an 8-bit constant

`OUT char, R`

In intel architecture, `R` can be `AL, AX, or EAX` registers.

Chapter 7

Interrupts

Process Context and Interrupts

7.1 Process Context

7.1.1 What is context?

A process is sometimes called a task, subroutine or program. Process **context** is all the information that the process needs to keep track of its state.

Registers Temporary storage locations for a few bytes. Typical CPU has 8-64 registers.

Program Counter A special register which points to the next instruction to be executed.

Stack Pointer Another special register which points to the top (bottom) of a stack of data

Processor Flags Hardware bits in the CPU which detect errors or arithmetic results.

Sometimes also called "processor state".
Example: reading a book and the phone rings.

7.1.2 Context Switching

When we call a subroutine (function) or when an interrupt happens, we need to **save** the current context because we are starting a new one.

This happens alot so it must be efficient.

- After function completes, must know where to resume.
- New process can mess up information in old process. (i.e. what if they both use the same register?)
- Saving the context and restoring it is called **context switching**.

Procedure:

1. Control is passed from Proc1 to OSH
2. OSH saves processor state from CPU registers(immediately!)
3. OSH finds stored state of Proc2 and restores it into CPU registers. IR is restored last.
4. CPU jumps to Proc1 code according to IR.

OSH = Operating System or Hardware.

7.1.3 Types of Context Switches

- Subroutine Call. Either to user routine or to OS.
- Interrupt. Hardware initiated context switch.

7.1.4 Calls

- User subroutine (e.g. `sin(`θ`)`).
  Main program context is saved, sin routine is loaded.
- System Calls Examples
  - I/O such as write block of data to disk or apply a value to D/A converter.
  - Process Control: halt, wait (pend), start, kill, or unblock another process.
  - Interprocess Communication: pipes, mailboxes, semaphores.
  - Return from Interrupt.

7.1.5 The Stack

Problem: How to allocate space for context storage. Consider nesting of subroutine calls.
Solution: A stack is a data structure modeled on a stack of plates(!). (spring balanced).

- **Push:** a piece of information on the stack.
- **Pop:** a piece of information off of the stack.
- Always put/take info from current stack top.

Stacks are implemented with a **Stack Pointer**. Enter the proper C code on the blank lines according to class discusion.

- **Push:** `_____ =piece_of_information`
- **Pop:** `piece_of_information = _____`
- **Initialize Stack:** `SP = stack_top`
- Stack grows down.
- Have to allocate space for maximum stack depth.

7.1.6 Subroutine Call

How to call a subroutine (function in C). (the compiler generates instructions to do all this so you don't have to!)

1. Push i argument *values* to stack:
   `foreach i _____ = argument(i);`
2. Generate `JSR` machine instruction.
   (a) Push machine registers onto stack.
   (b) Load subroutine addr into PC
   (c) execute next instruction (i.e. jump to *PC).
3. Subroutine looks for arguments at `*(SP - MACH_REG_SIZE)`

4. When Subr. completes, execute `RET` instruction.
   (a) Pop machine registers from stack.
   (b) Clean up stack from function args: `foreach i SP++;`
   (c) Load old value into PC
   (d) execute next instruction (i.e. jump to *PC).

7.2 Interrupts

A context switch caused by an **external hardware event.**
Types of inputs that can cause an interrupt:

- User pushes a button or moves a mouse.
- Packet arrives on network interface.
- A sensor detects an event.
- Disk drive completes an operation and has a bunch of data ready for CPU.
- A clock can be programmed to set up interupts at regular intervals.
- A hardware timer can count clock pulses and generate an interrupt when it gets to zero.
- A thermostat indicates that a temperature setpoint is reached.

Connections Interupts are logic signals physically wired to the CPU (or interrupt processor peripheral).
There are various schemes for wiring multiple interrupts:

Priority

- 7 lines leading into CPU.
- peripherals can pull one line low for example using open collector "wired OR".
- A small address bus identifies the specific interrupt within a line.
- Lines are ranked by priority e.g. ISR for Line 0 cannot be interrupted by any other interrupt, ISR for Line 6 can be interrupted by ANY other interrupt.

Interrupt Vectors Each interupt has a specific code number which is applied to the processor interrupt address bits (maybe 8bits).
CPU has a specific hard-wired memory address called the **interrupt vector table.** The interrupt vector table is an array of function pointers. The functions they point to are called **interrupt service routines, ISRs**.

Typical Interupt Vector Table

Memory Addr	Contents
1000	isr_0()
1002	isr_1()
1004	isr_2()
1006	isr_3()
...	...
1100	isr_N()

Meaning of ISRs Each hardware device is hooked up to a specific interrupt vector table address. We need to write each ISR to handle the proper device.

7.2.1 ISRs

Servicing an interrupt

Hardware Level The hardware does the following steps when an interupt is detected:

1. Push processor context onto stack.
2. Load appropriate address from IVT into program counter (PC).
3. jump to *PC.

Software Level The ISR is now in control. The ISR must do the following.

1. (There are no arguments)
2. block ("mask") all other interrupts if necessary
3. "service" the device.
4. unmask other interrupts
5. Execute a return from interrupt instruction

Hardware Level Now the hardware takes over again to clean up the interrupt.

1. Pop processor context from the stack.
2. Last item is PC
3. continue whatever was being done when interrupt occured.

7.2.2 ISR Programming

Some tips for successful ISR programming

Problems with ISRs

- ISRs are virtually impossible to debug!
- The debugger uses "software interrupts" to handle tricks like breakpoints and single stepping.
- thus even correct ISRs will break the debugger OR
- the debugger will break your ISRs
- ISRs take over the processor – even pre-emptive schedulers.
- ISRs can easily mess up other processes if they
  - write in wrong global variables
  - take up a lot of CPU time
  - play around with the stack

ISR Solutions

- Keep your ISRs *small, simple, & quick*
- No loops inside ISR's
- No complex logic (no nested `if`'s)
- Don't do unrelated I/O in the ISR (for example, no `printf("ISR message");`).
- Guidelines: Max: 1/2 page of C, 1 page of ASM
- Do not trust / use debugger.
- Do absolute minimum of stuff in ISR, do the rest in a regular task.
- Check and recheck your use of interrupt masking and how you save and restore the context.

7.3 Real Time Control

7.3.1 Basic Task

Map inputs to outputs (through a control law) at *regular* time intervals.

```
\% Initialization
set_up(timer0);
set_up(input,output);

\% endless control loop
while(1)
        {
        wait(timer0);
        x=read(input);
        y=control_function(x);
        output(y,output);
        }
```

7.3.2 Timing

Different ways to ensure regular time sampling.

- While loop + Interrupt
  - "main" program is `while(1);`
  - Timer is set up for regular interrupts every `T` seconds.
  - ISR is I/O and control law:
    ```
    x=read(input);
    y=control_function(x);
    output(y,output);
    return();
    ```
- "spin-lock":
  `wait(timer0)` → `while(timer0 == 0) ;`
- OS Call:
  `wait(event)`
  return control to OS until **event** occurs. **event** is timer0.

Chapter 8

Serial Communication

EE 472: Serial Communication

8.1 Serial Communication

Send bits from point A to point B one at a time.

Info Sources: `http://www.seetron.com/ser_an1.htm`
`http://www.taltech.com/TALtech_web/resources/intro-sc.html`
(on-line 12-Nov-02)

RS232C

An international standard for serial communication (1960s).

8.1.1 Electrical Signaling

- +2.5V — +15V logic 0 "space"
- -2.5V — -15V logic 1 "mark"

High voltage gives some immunity to noise.
Also a hassle since many systems don't otherwise need +15V, -15V supplies.
Requires special driver chips for +- voltages.
Some use

- +5V logic 0
- 0V logic 1

to save power supply costs.

8.1.2 Signaling

Idle line in "mark" state (logic 1 - -15V)

Pos.	Value	Function
0	space	Start Bit
1	X	Data 0 (lsb)
2	X	Data 1
...	X	...
8	X	Data 7 (msb)
9	X	Parity*
10	mark	Stop Bit 1
11	mark	Stop Bit 2*

*** Note:** Parity and Stop Bit 2 are optional, can have 1.5 Stop Bits.

8.1.3 Speed and Timing

Baud is short for Baudot, 19th century French telegrapher who invented first fixed length alphanumeric code.

Baud means number of signal changes on the line per second. Commonly equal to bits per second.

8.1.4 Synchronization

There is no common clock between sender and receiver. How does receiver know when to test levels?
`A1`: Synchronus mode — rarely used. Continuous stream of characters.
`A2`: Asynchronus mode — start bit is known to be logic 0. Falling (rising?) edge triggers a local clock in the receiver.

8.1.5 Multiplexing

Terms:
Simplex One sender and one receiver.
Half Duplex One side can send at a time.
Full Duplex Both sides can send at same time.
Typical serial communication today is *Full Duplex*.

8.1.6 Wiring

Minimum 3 wires:

- TX (transmit)
- RX (receive)
- gnd (V=0 reference)

Maximum 23 wires!

8.1.7 Connections

Must have:
TX → RX
RX ← TX
Cables are easier to make if pins are "straight through". i.e. pin n connects to pin `n`.

8.1.8 Connectors

23 pin (ancient, un-needed)
9 pin (IBM PC standard)
Two connector wiring types
`DTE:` pin 2 = RX, pin 3 = TX "Data Terminal Equipment"
`DCE:` pin 2 = TX, pin 3 = RX "Data Communications Equipment"
Thus, `DCE` can talk to `DTE` with a "straight through" cable.
`DTE, DCE` are ancient names from time sharing days.
Typical today:
PC == `DTE`
modem, printer, etc, == `DCE`

8.1.9 Multi-Drop

Can also have more than 2 systems interconnected by a *serial bus*.
In that case all users transmit and listen to same wire.
Must implement protocol to prevent two or more users from talking simultaneously.
(like half-duplex)

8.1.10 "Null Modem"

What if you want to connect a `DTE` to another `DTE` (such as PC-PC)?
Use a special cable which connects all lines "straight through" *except*
pin 2 → pin 3 *and*
pin 3 → pin 2

8.1.11 Flow Control

Sometimes a slow device cannot handle data as fast as sender (even at same baud rate).
Receiver needs a way to stop and start sender.
Hardware solution: Additional wires:
`RTS` "Request to Send". `DTE` sets this to mark to allow data to be sent to it by `DCE`.
`CTS` "Clear to Send". `DCE` sets this to mark to allow data to be sent to it by `DTE`.
Hardware flow control can be enabled/disabled.
Software solution: Special Control Characters:
`XON` (`ctl Q`) Start Transmitting
`XOFF`(`ctl S`) Stop Transmitting

8.1.12 Flow Control Note

Note: If you use Hardware Flow control, then "null modem" cable must also cross the `RTS` and `DTS` wires (pins 7,8 on 9-pin IBM standard).

8.2 Error Control

8.2.1 Causes

Unfortunately, serial communication is subject to errors. Sources:

- Electrical interference
- Long wire lengths
- Ground potential differences between communicating systems.
- Timing errors between sender and receiver clocks (should be within 1%).

8.2.2 Solutions

Character Parity 9th data bit. Very weak and bandwidth hog.

Checksum Add up a group of bytes. Send sum at end of packet. Compare.

Parity Send parity of some group of bits (rows or colums) in the packet.

CRC Use theory of binary polynomials. Send a code so that combined message is divisible by some known polynomial.

8.2.3 Packets

Grouping bytes into packets gives structure to the serial communication. Packet structure must be same between software both sides.

Typical Packet:

Start	Type	Length	...	Checksum

Start A character which the software can recognize as the start of a packet. Ideally should not appear anywhere else in packet.

Type Kind of packet.

Length value which tells how long the packet will be.

... some number of data bytes (the payload).

Checksum Checksum computed on previous packet.

8.2.4 Packet Error Control

What if a packet is received and the Checksum is wrong?
Normal:

1. Sender sends and *saves* the packet.

2. Receiver checks Checksum and *acknowleges* receipt of packet.
3. Sender discards packet and sends next packet.

Error:

1. Sender sends and *saves* the packet.

2. Receiver detects error and sends *negative acknowlege*.
3. goto step 1.

8.2.5 ACK/NAK

Special short packets:

`ACK` Acknowledge `NAK` Negative Acknowledge

Start	ACK/NAK	Checksum

8.2.6 Protocol Issues

What Happens If:

- Transmission is so unreliable that Checksum fails every time?
- `ACK/NAK` packets are subject to errors?

Do we have to send `ACK/NAK` on receipt of an `ACK/NAK`?

Answers to all these questions must be worked out in a protocol spec.

Chapter 9

TCP/IP and Socket Communciation

Source: Wikipedia

9.1 The OSI Network Protocol Stack

OSI = "Open Systems Interconnect" A **model** of network organization in seven layers of abstraction.

There are two similar modeling systems: OSI and TCP/IP. We will use the slightly more general OSI layer to give the overview and then specialize to TCP/IP.

There is something of a religious war between the two.

The seven OSI layers are:

1. Physical Layer
2. Data Link Layer
3. Network Layer
4. Transport Layer
5. Session Layer
6. Presentation Layer
7. Application Layer

9.1.1 Physical and Application Layers

Easiest to understand are (1) and (7):

- (1) Physical Layer: Cat-5 cable, network jack, ethernet card.
- (7) Application Layer; Firefox, Chat client, BitTorrent etc.

9.1.2 (2) Data Link Layer

Group bits into Frames and move frames of data from one node to another across a single physical link. Detect and correct bit errors in the link.

Examples:

- ethernet
- 802.11 wireless

9.1.3 (3) Network Layer

End-to-end data delivery. String together point-to-point links as necessary to get data from one end to another.

Examples:

- Address Resolution Prototcol (ARP)
- Connectionless Internet Protocol (IP)

9.1.4 (4) Transport Layer

Distribute data from the network layer among various applications on the same node. Assign socket numbers to each stream so that data is multiplexed on to the network. Provide end-to-end error checking and re-transmission (optional). Provide flow control so that links and computers are not overloaded.

Examples:

- UDP (used in address resolution protocol)
- TCP (used for web browsing, email, chat, etc)

9.1.5 (5) Session Layer

Establish semi-permanent dialogs between end-user applications. Not used by all applications. Not part of Tcp/Ip model.

Examples:

- Appletalk
- RPC (Remote Procedure Call)
- SSH/SCP (Secure Shell/Secure Copy) between hosts.

9.1.6 (6) Presentation Layer

Work on the data itself.

Examples:

- Encryption/Decryption
- Compression / Decompression
- Character Set conversion

9.2 Tcp/Ip Internet Model

Four layers.
There is a **rough** mapping to OSI layers:

1. Link Layer (OSI 1,2)
2. Internet Layer (OSI 3)
3. Transport Layer (OSI 4)
4. Application Layer (OSI 5,6,7)

9.2.1 Link Layer

- Physical layer
- Node to node transmission

9.2.2 Internet Layer

- Bridging different (sub)networks
- source computer to destination computer routing

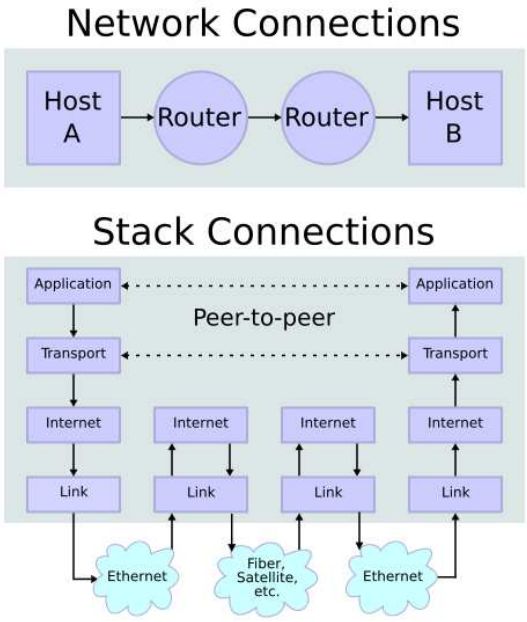
9.2.3 Transport Layer

- End-to-end transfer, identification of ports.
- Application to application routing.
- Error control / retransmission
- Flow control
- TCP connection oriented, reliable packets with re-transmission
- UDP connectionless “datagram” packets without confirmation.

9.2.4 Application Layer

Examples:

- HTTP
- SMTP (sendmail transfer protocol) for email.
- SSH/SCP (Secure Shell/Secure Copy) between hosts.



Example of an Internet connection going through the 4-layer Internetprotocol stack.

9.2.5 Internet Protocol Packet Types

Two major types of packets on internet:

- **TCP** Reliable. IP Layer 3 will retransmit until valid data is received at final destination. Slow
- **UDP** Fast. IP Layer 3 sends and forgets. Packets can be lost or arrive at destination out of sequence.

9.3 Internet Addressing

9.3.1 Address Format

An internet protocol address consists of four integers $0 <= i < 256$. They are written (for humans) separated by periods:

Example: `128.95.42.10` is "ee.washington.edu".

Every node on the internet has an IP address. Each digit of the IP address can be represented by an unsigned 8 bit value. There are three ranges of IP addresses which can be freely used on **private** networks (such as the one we will have in the lab). These are:

Private Address Ranges

`10.0.0.0` → `10.255.255.255`
`172.16.0.0` → `172.31.255.255`
`192.168.0.0` → `192.168.255.255`

9.3.2 How to obtain an address

There are two ways to get an IP address (such as for a new machine for example):

- Get one dynamically (automatically) using a protocol called DHCP. (May change each time you restart).
- Get a static (permanent) one assigned to you by the network administrator.

Servers usually require static IP addresses because users need to know how to connect to them, but clients often get their IP dyanmically assigned through DHCP. This allows networks to recycle addresses when not in use.

9.3.3 Ports

Numbers which identify specific user and server connections within a computer.

0 ← 2000 Reserved for system ports

2001 ← 32767 User ports

Servers bind to a specific port which indicates type of server. Example: Http is port 80.

Clients get a random port number assigned to them by OS. Servers can "`accept()`" multiple connections on their port. Clients are one-to-one with port numbers.

9.4 EE472 Private Lab Network

The EE472 lab will have a separate local ethernet which connects all the Make Controllers with two Linux servers. This net will not be bridged to the Internet or the EE department network.

In the 472 lab, we will assign a static IP to each team for you to program into your Make Controller as follows:

`192.168.0.100` → `192.168.0.120` Make Controllers

`192.168.0.25` and `192.168.0.26` Linux servers.

9.5 Sockets

9.5.1 Berkeley Sockets API

Sockets are an API into the Internet Transport Layer for sending information between applications located on different systems over the network using TCP or UDP.

Originated at U.C. Berkeley in 1980's. Most common API for networking.

Windows version: `winsock`.

Steps to program a connection with sockets:

```
#include <sys/socket.h>
```

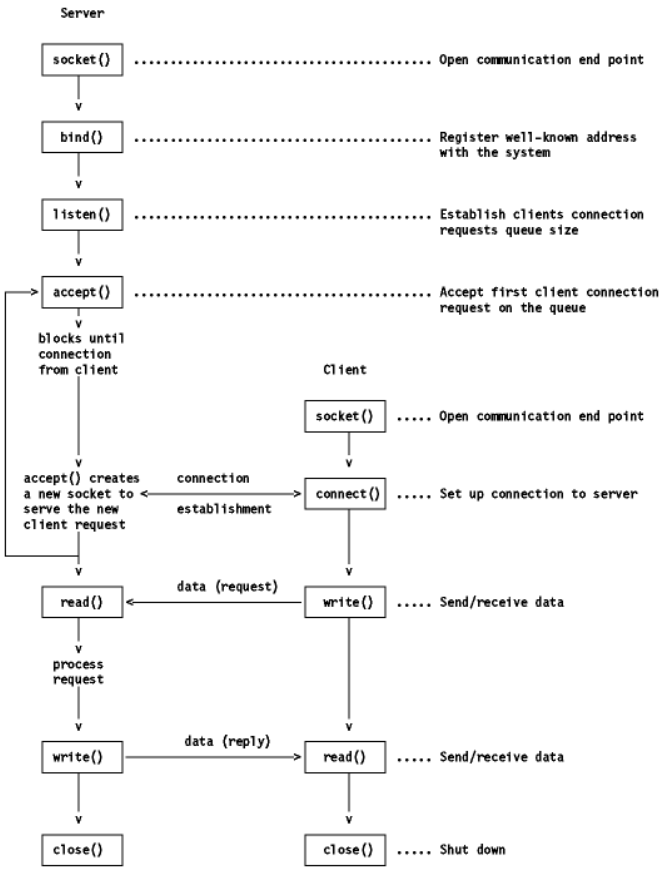
Server:

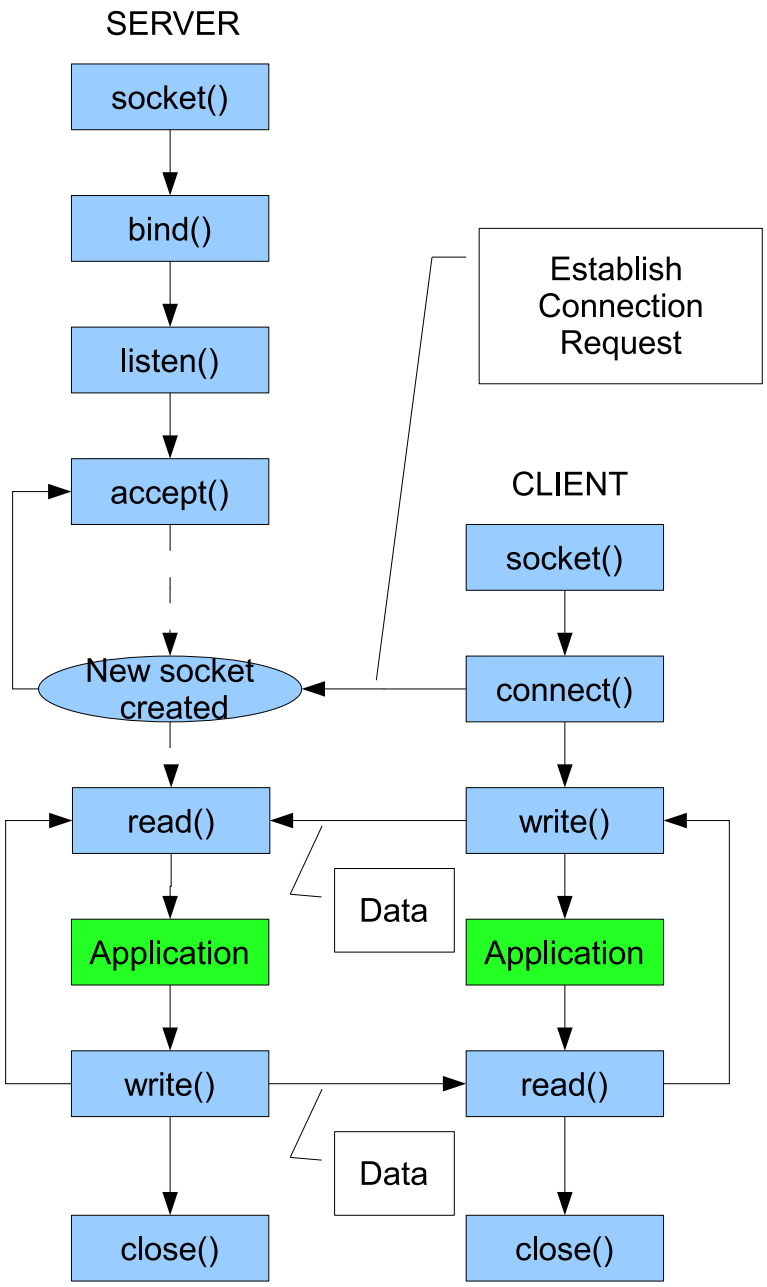
1. Open a "socket" with `socket()`.
2. Server binds the socket to a "port" identified by an integer using `bind()`
3. Server makes the socket "listen" for incoming connections. `listen()`
4. Accept incoming connection with `accept()`
5. Send or receive data to/from remote system use `send()/recv() or write()/read()`
6. Close the socket when done with `close()`

Client:

1. Create a socket with `socket()`

2. Connect to a server. Specify host address and port number with `connect()`
3. Send or receive data to/from remote system using `send()/recv() or write()/read()`
4. Close the socket when done with `close()`





Overview of Socket Client/Server model

9.5.2 Using Sockets in the Make API

The Making Things Core API has a simplified programmers interface to sockets which we will use in the lab. The MakingThings API simplifies the process by
1) combining some of the steps above.
2) using slightly simpler data structures for the function parameters.
The procedures for TCP based servers and clients using the Make Core API are:
Server:

1. Create a socket with `ServerSocket(int port)`.
2. Wait for a connection to your socket with `ServerSocketAccept( void* serverSocket)`.

Code Example:

```
// create a socket and start listening on port 10101
 struct netconn* server = ServerSocket( 10101 );
 // ServerSocketAccept( ) will block until an incoming connection is made
 struct netconn* newConnection = ServerSocketAccept( server );
 // now grab the data from the new connection
```

Client:

1. Create a connected socket to an IP address and port with `Socket(int address, int port)`.
   Use the macro `IP_ADDRESS(a,b,c,d)` to convert a numerical IP address into an address integer.

Code Example:

```
// use the IP_ADDRESS macro to format the address properly
int addr = IP_ADDRESS( 192, 168, 0, 54 );
// then create the socket, connecting on port 10101
struct netconn* socket = Socket( addr, 10101 );
```

Chapter 10

Introduction to Secure Communication

EE 472: Introduction to Secure Communication

References: J. Freer, Computer and Communication Networks, 2nd Edition, IEEE Press, 1996. Chapter 11., Wikipedia.

10.1 XOR Review

At its most basic level, the most common approach to encryption of digital data boils down to the exclusive OR operation, XOR. The truth table is:

Truth Table:

x	y	$x \otimes y$
0	0	0
0	1	1
1	0	1
1	1	0

Two interpretations:

- "odd number of ones"
- "programmable inverter"

We'll use the second: "`a` controls whether or not `b` is inverted."

Basic idea: scramble message by XOR'ing with a random series of bits. Idea as old as WWI, 1917 teletypes (Vernam).

This is the basis of most encryption systems. The "random bits" are called the *key*.

Let the received data be `a` = `b` $\otimes$ `c`. `c` is the key and `b` is the message. When we receive `a` we need to know if each bit of `c` was 1 or 0 in order to know the bits of `b`.

In `C` the exclusive OR operation ($\otimes$ above) is represented with $\wedge$.

One bit messages are not very interesting. The following code could use an 8 bit key on an 8-bit message:

```
char a,b,c;

b = 0x33;         // our message
c = 0xA5;        // the key

a = b ^ c ;      // a is b XOR c
```

What is `c`?

One way to view this is to realize that for each bit of `c` that is one, the corresponding bit of `b` is changed. If a bit of `c` is zero, that bit in `b` is not changed. If we think of `b` as the data and `c` as a key, then this is one way that data can be encrypted. If an attacker gets the encrypted data, `a`, s/he has no way to recover the data without knowing `c`. `c=0` is NOT a good key. (Why not?). How can we encrypt many bytes instead of just one? One way would be to use a single key as follows:

```
#define BUFSIZE 1000

char a[BUFSIZE],b[BUFSIZE],c;
int i=0;
c = 0xA5;        // the key

while (i < BUFSIZE)
 { a[i] = b[i] ^ c ; i++}
```

This however is rather weak for two reasons. First there are only 256 possible keys. Second, there are easy statistical methods to extract the key from `a`. A more secure method (but still weak) is to use a longer key as follows:

```
#define BUFSIZE 1000
#define KEYSIZE 5

char a[BUFSIZE],b[BUFSIZE];
char c[BUFSIZE]={0x12, 0xA3, 0x4F, 0xB9, 0x7C};

int j=0;i=0;


while (i < BUFSIZE)
 {
 a[i] = b[i] ^ c[i % KEYSIZE] ;    // use MOD operator (%)
                                   // sequence through the key bytes
 i++
 }
```

Here we are rotating through the key array and using one of the five key bytes on each data byte.

10.1.1 Key issues

- Length. length(key) $\neq$ length(msg).
  - Use fixed key length and repeat.
  - long keys secure, expensive.

  - short keys insecure, convenient.

- Key distribution. How does the receiver `get` the key?
- Key Quality.
  - short keys are "bad": 8 bit key has only 256 possibilities to try.
  - Not all keys are created equal.
  - Key = *all zeros* is a **BAD** key! (also all 1's).
  - Need key to have a mix of ones / zeros

10.2 Public Key Systems

10.2.1 One-way functions

- A function $f(x)$, for which $y = f(x)$ is easy to compute, but $x = f^{-1}(y)$ is very hard to compute.
- Based on large prime #'s.
- Computationally intensive.

One way function example:

$$y = f(a, b) = a \times b$$

where a and b are large integers.

Surprisingly this simple function can be one way. Consider when a and b are large (i.e. 50-100 digits) prime numbers. Now y is a *very* large integer but a and b can only be recovered if you can *factor* y.

10.2.2 Algorithm

Assume existence of a matched pair of one-way functions, $f_A(x, k)$ and $f_B(x, k)$.

Properties:

- $f_B(f_A(x, k_1), k_2) = x$ where k_1, and k_2 are a special pair of keys.
- f_A, f_B can be computed.
- $f^{-1}_{A/B}(y, k)$ is unknown.

10.2.3 Use

1. Key generator makes pair of keys for each user i: k_{i1} (public) and k_{i2} (private).
2. User i sends key k_{i1} to anyone or makes it public on the web.
3. If user j sends message x to user i:
   (a) User j sends $y = f_A(x, k_{i1})$ to i using i's public key, k_{i1}.
   (b) User i decodes by: $x = f_B(y, k_{i2})$ using their private key, k_{i2}.

Advantage: No more key distribution problem!

10.2.4 RSA Method

Rivest, Shamir, Adloeman, 1977 MIT.
Most Famous Public Key System

- RSA Keys: 200bits
- $k_1 = P_1 * P_2$ where P_1, P_2 are large prime numbers.
- can break code if you can factor k_1 into $P_1 \times P_2$.
- for $k_1 = 200$ bits, factoring takes 3.8B years at 1 op/musec.
- Encoding is expensive.
  - Block size 512 bits
  - encode 1 block: 3×10^6 multiplications.
  - at 1/musec/mult., 3 seconds per 64 bytes = 13hrs / Mbyte! (special hardware considerably faster).

10.3 Data Encryption Standard (DES)

US Government Standard developed by IBM.

1977 56 bit key

1990 112 bit key

64 bit block code with single encryption key.

Some claimed that the US National Security Agency (NSA) designed weaknesses into the DES in the 1970's so that it could have the ability to break it at will. However later research has seemed to reduce this concern.

Chapter 11

μ/C-OS-II Real Time Operating System

Micro-C OS/II Overview

Introduction to MicroC/OS-II

11.1 Overview

11.1.1 Information Sources

- http://www.uCOS-II.com/
  reference card
  cool animations
  Application notes
  Examples of commercial products.
- Book: "MicroC/OS-II, The Real-Time Kernel," by Jean J. Labrosse, CMP Books, 2nd edition, 2002.

11.1.2 Characteristics and Features

Ucos-II is a real time *kernel*.
"Operating System" is more than a kernel.
Kernel supports

- Scheduler
- Message passing (mailboxes)
- Synchronization (semphores)
- Memory Management

Does not support

- I/O Devices
- File System
- Networking

Features

- Source Code available (not "GPL").
- Free for educational use, license required for commercial use.
- Portable to multiple micro processors.
- ROMable.
- Scalable (only builds what you need)
- Multitasking (up to 56 tasks)
- Preemptive: 64 unique priority levels. Always runs highest priority task.
- Deterministic: ISRs etc. take known, fixed amount of time.
- Individual stacks for each task. Can be different sizes.
- Services: mailboxes, queues, semphores, dynamic memory allocation.
- Interrupt management. Scheduler can check for a task that should be activated after an interrupt.
- Mature. In use since 1992. Incorporated into commercial products.

11.2 μC/OS-II Tasks and Calls

11.2.1 Task Priorities

All tasks have a unique priority. Low priority numbers have the highest priority. Priority numbers 0,1,2,3, are reserved.

There are a *maximum* of 64 total priority levels, but if you need smaller memory useage, the system can be configured for fewer. Thus the lowest priority task is not always 64. Instead the OS defines four levels at the bottom:

`OS_LOWEST_PRIO, OS_LOWEST_PRIO-1, OS_LOWEST_PRIO-2, OS_LOWEST_PRIO-3`

(which would be 64, 63 ,62, and 61 in a full system). The lowest user level priority task is therefore: `OS_LOWEST_PRIO - 4` . The four highest priorities are also all reserved. The maximum number of priorities is thus 56.

Only one task may have each priority level so 56 is also the maximum number of tasks. Since task priorities are unique they are also used to identify the processes (like the pid in Unix). The max number of tasks can be reduced at compile time to save memory.

11.2.2 OS Function Calls (selected)

Tables 12.1 and 11.2 summarize key function calls in μC/OS-II.

Name	Prototype	Function
OSInit()	`void OSInit(void)`	Sets up OS internal data structures. Must be called first.
OSStart()	`void OSStart(void)`	Begins operation of the scheduler.
OSIntEnter()	`void OSIntEnter(void)`	Must be called at start of an ISR. Lets OS keep track of number of ISRs.
OSIntExit()	`void OSIntExit(void)`	Call at end of an ISR *before* restoring conext and `IRET`.
OSTaskCreate()	`INT8U OSTaskCreate( void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)`	Creates a new task. `task` is a pointer to the task's function, `pdata` points to optional parameters to the task, `ptos` is a pointer to the tasks stack top. `prio` is the task priority. Note that since each μCOS-II task must have a unique priority, priority is sometimes used to as the "task number" as well (see OSTaskDel().
OSTaskDel()	`INT8U OSTaskDel(INT8U prio)`	Deletes a task. The task is indicated by its priority number so any task can delete any other task. `OS_PRIO_SELF` is a predefined constant which a task can use to delete itself if it doesn't know its priority.
OSTaskDelReq()	`INT8U OSTaskDelReq(INT8U prio)`	Send a request to a task to delete itself. If a task is set up for this, it will *also* issue a OSTaskDelReq(OS_PRIO_SELF) and if that returns OS_TASK_DEL_REQ, then it should delete itself.
OSTaskSuspend()	`OSTaskSuspend(INT8U prio)`	Block the execution of the current task.
OSTaskResume()	`OSTaskResume(INT8U prio)`	Resume execution of a task which was blocked using OSTaskSuspend()
OSTaskChangePrio()	`INT8U OSTaskChangePrio( INT8U oldprio, INT8U newprio)`	Change the priority of a task. `oldprio` must exist and `newprio` must be free.
OSSchedLock()	`void OSSchedLock(void)`	Stops all task scheduling but not interrupts. Higher priority tasks will no longer take over.
OSSchedUnlock()	`void OSSchedUnlock(void)`	Resumes task scheduling.
OSTimeDly()	`void OSTimeDly(INT16U ticks)`	Delay the calling task by `ticks` clock ticks (i.e. timer0 interrupts). The scheduler will start other tasks during this delay
OSTimeDlyHMSM()	`void OSTimeDlyHMSM( INT8U hours, INT8U minutes, INT8U`	Same as ISTimeDly() but uses natural time units.

Name	Prototype	Function
OS_ENTER_CRITICAL()		Not actually a function. This *macro* inserts the machine instruction into your code to block all interrupts .
OS_EXIT_CRITICAL()		Not actually a function. This *macro* inserts the machine instruction to enable interrupts.
OSSemCreate()	`OS_EVENT *OSSemCreate( WORD value)`	Create a new semaphore and initialize it to `value`. Returns a pointer to the semphore (actually an *event control block*).
OSSemPost()	`INT8U OSSemPost(OS_EVENT *pevent`	Signal the semaphore pointed to by `pevent`. (enables other tasks to continue).
OSSemPend()	`void OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err`	Wait for access to resource controlled by semaphore `pevent`. `pevent` must have been previously created by OSSemCreate(). `timeout` is a time value (in ticks) which allows the task to wake up if the semaphore never becomes available. If this happens, the function will return `OS_TIMEOUT`. Do not use inside an ISR.

Table 11.2: Table of Selected μC/OS-II calls (cont.)

11.3 Code Example

```
//-----------------------------------------------------------------------------
// AUTHOR:    Jered Aasheim
// CREATED:   February 23, 2001
// MODIFIED:  February 23, 2001
// PROJECT:   EE472 Final Project
// MODULE:    Test Application using uC/OS RTOS
//
// Abstract:  The following program demonstrates how to use the uC/OS Real-Time
//            Operating System to solve a critical section problem.  Using the
//            uC/OS facilities, two different tasks are created and a binary
//            semaphore is used to protect a critical section (in this case the
//            first row on the LCD screen).
//
//            Unfortunately, this program only demonstrates a few facilities of
//            the uC/OS RTOS.  For further examples using uC/OS, refer to
//            "An Embedded Software Primer" by David E. Simon.
//
//            NOTE: In order to use the uC/OS RTOS, you will need to add the
//                  UCOS.LIB and UCOS.H files to your Paradigm C++ project.
//
//-----------------------------------------------------------------------------

#include "td40.h"
#include "ucos.h"

#define STACK_SIZE              1024
#define WAIT_FOREVER            0


#define STARTUP_PRIORITY         0
#define TASK1_PRIORITY          11
#define TASK2_PRIORITY          12

UWORD StartUp_Stk[STACK_SIZE];
UWORD T1_Stk[STACK_SIZE];
UWORD T2_Stk[STACK_SIZE];

OS_EVENT *pBinSem = NULL;               // Binary semaphore

static UBYTE err = 0;

void SetupTimers(void);

void far StartUpTask(void* data);

void far SimpleTask0(void* data);
void far SimpleTask1(void* data);

void interrupt far Timer0_ISR(void);
```

```
void main(void)
{
   ae_init();
   lcd_init();

   OSInit();                              // Initialize uC/OS

   //SetupTimers();                       // Configure the TD40 timers.
   setvect(uCOS, (void interrupt (*)(void))OSCtxSw);

   pBinSem = OSSemCreate(1);              // Initialize the semaphore

   OSTaskCreate(StartUpTask, (void*)0, (void*)&StartUp_Stk[STACK_SIZE-1], STARTUP_PRIORITY);

   OSStart();                             // Start the uC/OS
}


void far StartUpTask(void* data)
{
        OS_ENTER_CRITICAL();

   // Install the uC/OS timer ISR
   setvect(0x0008, (void interrupt (*)(void))OSTickISR);
   // Install applications timer ISR
   setvect(0x0081, Timer0_ISR);

   // Initialize Timer0
   outport(0xFF32, 0x0007);         // Unmask TIMER0 interrupt and set Timer0
                                                                                  // interrupt to

   outport(0xFF56, 4000);                        // Disable TIMER0

   outport(0xFF52, 10000);          // Initialize TIMER0's count

   outport(0xFF56, 0xE001);         // Enable TIMER0 to generate an interrupt,
                                    // use maxcount A, don't use prescaled value,
                                    // use internal clock, continuous operation

        OS_EXIT_CRITICAL();

   OSTaskCreate(SimpleTask0, (void*)'1', (void*)&T1_Stk[STACK_SIZE-1], TASK1_PRIORITY);
   OSTaskCreate(SimpleTask1, (void*)'2', (void*)&T2_Stk[STACK_SIZE-1], TASK2_PRIORITY);
   OSTaskDel(OS_PRIO_SELF);

}


//-----------------------------------------------------------------------
// SimpleTask() - demonstrates a simple task that uses semaphores.  In
```

```
//               this case, the LCD screen is the critical section.
//-----------------------------------------------------------------------------
void far SimpleTask0(void* data)
{
   for(;;)
   {
      // Get the semaphore (i.e. P())
      OSSemPend(pBinSem, WAIT_FOREVER, &err);

      // Critical section...
      lcd_movecursor(0x80);
      lcd_putstr("Task ");
      lcd_put((char)data);
      delay_ms(2000);

      // Give up the semaphore (i.e. V())
      OSSemPost(pBinSem);

      // Block for 5 ticks - forces a task switch
      OSTimeDly(5);
   }
}


void far SimpleTask1(void* data)
{
   for(;;)
   {
      // Get the semaphore (i.e. P())
      OSSemPend(pBinSem, WAIT_FOREVER, &err);

      // Critical section...
      lcd_movecursor(0x80);
      lcd_putstr("Task ");
      lcd_put((char)data);
      delay_ms(2000);

      // Give up the semaphore (i.e. V())
      OSSemPost(pBinSem);

      // Block for 5 ticks - forces a task switch
      OSTimeDly(5);
   }
}



//-----------------------------------------------------------------------------
// SetupTimers() - initializes the timers on the TD40 board.
//                 The uC/OS requires a timer to periodically
//                 generate an interrupt so that it can schedule
//                 tasks appropriately.  In most systems, however,
```

```
//                uC/OS can't just define the ISR for a Timer
//                for several reasons:
//
//                1) Defining a timer ISR is platform specific
//                   and uC/OS is intended to be platform independent.
//
//                2) The application may need to define its own
//                                                    ISR for the timer.  If uC/OS defines the ISR
//                   then the application can't define their own
//                   ISR.
//
//                The solution is simple: have the application
//                configure the timer to generate interrupts and
//                the uC/OS "timer" ISR.  Then, inside the uC/OS
//                ISR a software interrupt is generated (i.e. INT 81)
//                which runs the users "timer" ISR.  Using this
//                technique, after all is said and done, uC/OS will
//                have the required timer interrupt ISR setup correctly
//                and the application can define its own timer ISR.
//
//                NOTE: Even if the application doesn't use timer interrupts
//                      it still must define a "dummy" timer ISR (see below).
//
//-----------------------------------------------------------------------
void SetupTimers(void)
{
   // Install the uC/OS "task switch" ISR
   //setvect(uCOS, (void interrupt (*)(void))OSCtxSw);

   // Install the uC/OS timer ISR
   setvect(0x0008, (void interrupt (*)(void))OSTickISR);

   // Install applications timer ISR
   setvect(0x0081, Timer0_ISR);

   // Initialize Timer0
   outport(0xFF32, 0x0007);              // Unmask TIMER0 interrupt and set Timer0
                                         // interrupt to low priority

   outport(0xFF56, 4000);                // Disable TIMER0

   outport(0xFF52, 10000);               // Initialize TIMER0's count

   outport(0xFF56, 0xE001);              // Enable TIMER0 to generate an interrupt,
                                         // use maxcount A, don't use prescaled value,
                                         // use internal clock, continuous operation
}



//-----------------------------------------------------------------------
// Timer0_ISR() - "dummy" Timer0 ISR, see SetupTimers() above
```

```
//                for details
//---------------------------------------------------------------------------
void far interrupt Timer0_ISR()
{
   outport(0xFF22,0x0008);          // Non-specific EOI
}
```

Chapter 12

FreeRTOS Real Time Operating System

FreeRTOS Overview

Introduction to FreeRTOS

12.1 Overview

12.1.1 Information Sources

- `http://freertos.org`
-

12.1.2 Characteristics and Features

FreeRTOS is a real time *kernel*.
"Operating System" is more than a kernel.
Kernel supports

- Scheduler
- Message passing (mailboxes)
- Synchronization (semaphores)
- Memory Management

Does not support

- I/O Devices
- File System
- Networking

Features

- Source Code available (license: "modified GPL").

- Free for educational use and commercial use. Commercial support packages are available and a closed source version is available.
- Portable to multiple micro processors.
- ROMable.
- Scalable (only builds what you need)
- Multitasking (up to 56 tasks)
- Preemptive: Flexible number of priority levels. Multiple tasks can have same priority.
- Individual stacks for each task. Can be different sizes.
- Services: queues, semphores, recursive mutexes. dynamic memory allocation.

12.2 FreeRTOS and the Make Library

FreeRTOS is used widely *beyond* the Make Controller.

Make has written "wrapper functions" around the FreeRTOS calls.

We will use the Make library names for the calls (but also explain the underlying FreeRTOS).

12.3 FreeRTOS Tasks and Calls

12.3.1 Sleep

`void Sleep(int timems)`

Code for `Sleep()`

```
void Sleep( int timems )
{
  vTaskDelay( timems / portTICK_RATE_MS );
}
```

Thus `Sleep()` is just a new name for `vTaskDelay()`.

12.3.2 Why Sleep is important.

In a priority based scheduler, need to allow lower priority tasks to run:

```
void ExampleTask(void *p)
{
      while( 1 )
  {
    Led_SetState( 0 ); // turn the LED off
    Sleep( 900 ); // leave it off for 900 milliseconds
    Led_SetState( 1 ); // turn the LED on
    Sleep( 10 ); // leave it on for 10 milliseconds
  }
}
```

Sleep does **two** things:

1. Makes the LED flash with proper on and off durations.
2. Allows FreeRTOS to give CPU to **other** tasks.

Make Name	FreeRTOS Name	Prototype	Function
`Sleep()`	`vTaskDelay()`	`void Sleep(int timems)`	Delay task for `timems` milliseconds.
`TaskCreate()`	`xTaskCreate()`	`void* TaskCreate ( void(taskCode)(void *) , char * name, int stackDepth, void * parameters, int priority )`	Create a task, specify its name, stack size, and priority. Pass it a single void pointer parameter.
`TaskYield()`	`TaskYIELD()`	`void TaskYield(void)`	Tell scheduler that this task has nothing to do at this moment but will need to get back to work as soon as possible.
`TaskDelete()`	`vTaskDelete()`	`void TaskDelete( void* task )`	Delete this task from the schedule. It will no longer run.
`SemaphoreCreate()`	`vSemaphoreCreate Binary()`	`void* SemaphoreCreate()`	Create a binary semaphore. Returns a void pointer to the new semaphore.
`SemaphoreTake()`	`xSemaphoreTake()`	`int SemaphoreTake(void* semaphore, int blockTime`	"Take" (pend) the semaphore. Return 1. if waited past blockTime, return 0.
`SemaphoreGive()`	`xSemaphoreGive()`	`int SemaphoreGive(void* semaphore)`	Release (post) a semaphore. Return 1. If error (such as no such semaphore) return 0.

Table 12.1: Table of Selected Make Library RTOS calls

12.3.3 Task Priorities

{0-7}
High priority numbers have the highest priority.

12.3.4 OS Function Calls (selected)

Tables 12.1 and 11.2 summarize key function calls in μC/OS-II.

12.4 Code Example

```
 /*
make.c

make.c is the main project file.  The Run( ) task gets called on bootup, so stick any initialization stu
In Heavy, by default we set the USB, OSC, and Network systems active, but you don't need to if you aren
Furthermore, only register the OSC subsystems you need - by default, we register all of them.
*/

#include "config.h"

// include all the libraries we're using
#include "appled.h"
#include "dipswitch.h"
#include "digitalout.h"
#include "digitalin.h"



// EE472 Include Files

 //  if there are any EE472 includes you could put them here



#define   PIO_PA02          AT91C_PIO_PA2
#define   PIO_PA03          AT91C_PIO_PA3
#define   PIO_PA04          AT91C_PIO_PA4
#define   PIO_PA06          AT91C_PIO_PA6

#define   WIRE_BLUE         PIO_PA02
#define   WIRE_BLACK        PIO_PA03
#define   WIRE_RED          PIO_PA06
#define   WIRE_WHITE        PIO_PA04


#define   EE472_OUTPUT         1
#define   EE472_INPUT          0
// Local function/task prototypes
      // a prototype for each of the task functions (and other functions you write)

void display(int, int, int, int);
void error(int,int,int,int);
int pio_472_bit_setup (unsigned int, int);  // function prototype
void BlinkTask( void* parameters );
void ToneTask(void* parameters );
void IOSetup(void* p);

// task pointers as needed
      // declare a void pointer for each task you would like to refer to later.
void *blinktask, *tonetask, *iosetuptask;
```

```
void Run( ) // this task gets called as soon as we boot up.
{

// Create some tasks.
   blinktask     = TaskCreate( BlinkTask,   "Blink",  400, 0, 1 );
   if(blinktask == NULL) error(1,1,1,1);

   iosetuptask   = TaskCreate( IOSetup,   "IOSetup",  400, 0, 2 );
   if(iosetuptask == NULL) error(1,1,1,0);

// Show you got here.
   display(1,0,1,0);             // set initial pattern on the LEDs

}


void IOSetup(void* p)
  {
   pio_472_bit_setup(WIRE_RED,   EE472_OUTPUT);     // set up bit "DIGITAL OUT 3" for output
   tonetask      = TaskCreate( Tone,   "Tone",  400, 0, 2 );
      if(tonetask == NULL) error(1,1,0,1);
   TaskDelete(iosetuptask);  // delete yourself!
  }


void ToneTask(void* p)
  {
  while(1)        //   Should make a 250Hz tone on speaker
      {
      *AT91C_PIOA_SODR  = WIRE_RED;    // set red wire high
      Sleep(2);
      *AT91C_PIOA_CODR  = WIRE_RED;    //  clear red wire.
      Sleep(2);
      }
  }



// A very simple task...a good starting point for programming experiments.
// If you do anything more exciting than blink the LED in this task, however,
// you may need to increase the stack allocated to it above.
void BlinkTask( void* p )
{
 (void)p;
  Led_SetState( 1 );
  Sleep( 1000 );

  while ( true )
  {
    Led_SetState( 0 );
    Sleep( 950 );
```

```
    Led_SetState( 1 );
    Sleep( 50);
  }
}


int   pio_472_bit_setup( unsigned int bit,  int input_or_output)
  {
  //Allow the PIO to control this bit
  *AT91C_PIOA_PER  = bit;            //AT91C_PIOA_PER is addr of PER register
  if(input_or_output == EE472_OUTPUT)
       {
       // Enable the bit as an output
       *AT91C_PIOA_OER  = bit ;
       }
   else
      {
       // Enable the bit as an input / disable as output
       *AT91C_PIOA_ODR  = bit ;
      }
  return(0);
  }

//  set status of the 4 LEDs on the Make Applications Board
void display(int a, int b, int c, int d)
  {
  AppLed_SetState(0,d);
  AppLed_SetState(1,c);
  AppLed_SetState(2,b);
  AppLed_SetState(3,a);
  }



//  Any task can go here to display a flashing code and lock the system
//     (note that error() is not a task by itself, just a part of the task
//      which calls it.)
void error(int a, int b, int c, int d)
  {
      //vTaskSuspendAll(); // direct RTOS call to stop all other tasks.
  while(1) {
  display(a,b,c,d);
 Sleep(800);        ///* if vTaskSuspendAll() is used, execution halts here. .
  display(0,0,0,0);
 Sleep(200);
      }
  }
```

Chapter 13

Concurrency Problems, Critical Sections, and Threads

Critical Sections

Critical Sections, Interprocess Communication

13.1 Critical Sections

Introduction

When processes share a resource (such as blocks of memory or I/O devices) we have to think very carefully about coordinated use of the resource. Humans find this relatively easy (in simple cases of course) but time-shared processes in a computer are a different story.

We will illustrate these issues with the producer-consumer problem. We will simulate the operation of the tasks in-class using people in place of the CPU. We will use plastic buttons for data items, and the chalkboard will represent RAM (for counters flags etc).

Producer A task which generates blocks of data.

Consumer A task which does something with the data blocks and discards them or passes them to another consumer.

13.1.1 Producer-Consumer Example

Producer A task which generates blocks of data.

Consumer A task which does something with the data blocks and discards them or passes them to another consumer.

Human Example An in-class exercise to illuminate the problem.

Human instructions: Producer

```
If there is room
   add item to shared buffers
else
```

```
   wait;
```

Human instructions: Consumer

```
If an item available
   take item from shared buffers
else
   wait;
```

13.1.2 Pseudo Code Producer-Consumer

Globals

```
/* example.h   */
#define BSIZE = 5
typedef item {plastic button(!)};
struct item ConsIt,NextIt, buffer[BSIZE];
int in=0, out=0, count=0;
```

```
#####  PRODUCER ######
#include example.h
while(1)
   {
   while(count >= BSIZE);
   count++;
   NextIt = {produce an item};
   buffer[in] = NextIt;
   in++;
   if(in >= BSIZE) in = 0;
   }
```

```
#####  CONSUMER #####
#include example.h
while(1)
   {
   while(count==0); //"spinlock"
   count--;
   ConsIt = buffer[out];
   out++;
   if(out >= BSIZE) out = 0;
   Take(ConsIt);   /*Consume the item*/
   }
```

13.1.3 Procedures

- 3 volunteers:
  - Producer
  - Consumer
  - Memory Device (writes on chalkboard)
- Condition 1: Two Cooperating Humans (Human Instructions)
- Condition 2: Two Humans "executing" pseudocode.
- Condition 3: One Human Multitasking based on pseudocode. (pre-emptive context switches)
- Variations: try two producers or two consumers or both.

13.2 Intertask Communication and Data Sharing

13.2.1 Passing Data I: Shared Memory

- Global Variables
- Shared Buffer
- Ring Buffer
- FIFO

Shared Buffer(s)

Producer fills a buffer
Signals Consumer
Consumer clears buffer
Multiple buffers: Producer fills buffer A *while* Consumer clears buffer B.
switch pointers A and B
This scheme is called "double buffering". Allows producer and consumer to work simultaneously.

Ring Buffer / FIFO

One buffer. Two Pointers *in, *out. Also called "FIFO" — First-In-First-Out
Producer:

```
while(in != out) {
   buffer[in++] = {new data item}
   if(in >= BUFFER_SIZE) in = BUFFER;
   }
print "BUFFER OVERFLOW!!!!" ; halt;
```

Consumer:

```
while(out != in ) {
   data = buffer[out++];
   if(out >= BUFFER_END) out = BUFFER;
   }
print "BUFFER UNDERFLOW!!!!" ; halt;
```

LIFO

"Last-In-First-Out"
A stack
One buffer. One pointer *iop
Producer:

```
while(iop <= BUFFER_END) {
   buffer[iop++] = {new data item}
   }
print "LIFO OVERFLOW!!!!" ; halt;
```

Consumer:

```
while(iop != BUFFER ) {
   data = buffer[--iop];
   }
print "LIFO UNDERFLOW!!!!" ; halt;
```

13.2.2 Passing Data II: Message Passing

Problem with shared memory approaches is that processes are not protected from each others bugs.

OS can isolate processes better by supporting messages.

Two types:

- Message Passing
- Mailbox

Example:

```
/*  Process 1
 */
while(1) {
   compute & produce data;
   OS_send_message(Proc_ID);
   }
```

13.2.3 Passing Data III: Mailboxes

Example:

```
/*  Process 1
 */

mailboxname="mailbox1_2";
mailboxstatus = OS_mailbox_setup(mailboxname);
if(mailboxstatus != MB_SUCCESS)
       error_exit("Couldn't setup mailbox");
while(1) {
   compute & produce data;
   OS_post_message(mailboxname);
   }

/*  Process 2
 */
mailboxname="mailbox1_2";
// assume mailbox has been setup by Proc 1
while(1) {
   OS_pend_message(mailboxname);
   consume data;
   }
```

OS message calls invoke the scheduler.

Sender is set to `WAITING` until receiver gets the message.

Receiver is set to `WAITING` until mailbox contains a message.

13.2.4 Networked and Multiprocessor Environments

If OS supports it, can communicate across networks and between processors.

Message passing: `Proc_ID` → `host:Proc_ID`

Mailboxes: `mailboxname` → `"host:mailbox"`

13.3 Concurency Problem Statement

Fundamental Problem: make sure only one task accesses a resource during "Critical Section".

Critical Section: a short segment of code which must be done as a unit (also called "atomic" operation).

Above example: testing buffer counter and taking/putting token must be done together without interruption.

13.3.1 Remedies

Mask Interrupts

Make an OS call, or manipulate bits in interrupt controller to prevent any ISRs from running during Critical Section.

Example:

```
...
OS_INT_MASK();  // block all interrupts
 if(data_avail_flag) {
        // if ISR occurs here ---> Trouble!!
          process(buffer);
          data_avail_flag = FALSE;
          buffer_avail_flag = TRUE;
          }
OS_INT_ENABLE();
...
```

This prevents

1) pre-emption by scheduler which would start another task.

2) An ISR which may mess with buffers/ptrs.

Int Masking: pro/con

Pros:

Fast

Cons:

Too greedy: blocks all I/O devices and other tasks whether they use the key resource or not!

Test and Set

A Machine Language Instruction (first on IBM 360)

```
# assembler:
        TSR     R,FailAddr   # jump to FailAddr if R ne 0
# explanation:
        if(R==0) R = 1 ;
        else jump FailAddr ;
```

`TSR` is a single instruction and cannot be interrupted.

Example (pseudo assembler code):

```
x = 0;
while(1) {
Wait:   TSR  X, Wait ;  // wait for x==0
        //
        // Critical Section Code
        //
        MOVI    X, 0 //  Release Flag X
        ...
        }
```

Test and Set Pros and Cons

Pros:

Extremely fast

Can be specialized for many resources without blocking unnecessarily.

Cons:

Very low-level. Who decides what "`X`" is for each resource?

Only allows one user per resource. What if resource permits `N` users?

13.3.2 Semaphores

Term Origin: Flags used in signaling.

An OS facility which guarantees exclusive access.

Three OS Calls:

```
semaphore os_get_semaphore(int N);
    // establish a new semaphore
    // initialize to N

void os_pend(semaphore S);
    // wait for resource protected by S

void os_post(semaphore S);
    // free up resource protected by S
```

Semaphores: Pend and Post

Identify the critical section in your code.

"protect" it with `Pend(S)` and `Post(S)`:

```
...
pend(S);

  critical section code

post(S)
```

Implementing Pend and Post inside a kernel

Notes:

S is typically a small integer.

S==0 $\rightarrow$ wait, S $\geq$ 1 $\rightarrow$ go

Pend(S):

1. disable interrupts
2. `if (S>0) { S--; Enable Interrupts; return}`
3. else {`Enable Interrupts; set task to 'WAIT'; store S in TCB.sem; start scheduler`}

Post(S):

1. S++
2. start scheduler

Critical Section Summary

The **Contention Problem** can occur under two conditions:

1. There is preemption due to interrupts.
2. Two or more processes share a single resource.

A **Critical Section** is the part of code which, if interrupted, could cause a bug in sharing a resource between two processes.

Example: Increment a counter and then put data in buffer.

Semaphores can be used to protect a critical section.

1. One semaphore, S, is established for each shared resource.
2. pend(S) operation is used at start of critical section.
3. post(S) operation is used at end of critical section.

Pend means "wait" (i.e. "patent *pend*ing").

Scheduler makes sure that tasks waiting for a resourse (i.e. pending a semaphore) are set to "WAITING" and that other tasks can run instead.

Semaphores: Pros and Cons

Pros:

Specialized, one semaphore per resource, no unnecessary blocking.

Widely used standard.

Low to medium implementation overhead in O/S.

Cons:

Can cause *Piority Inversion*

Priority Inversion

Consider the following scenario.

(low numbers = high priority)

```
Task A    Priority  5
Task B    Priority 10
Task C    Priority 15
```

Suppose tasks A, and C, both want to use a buffer protected by semaphore `S`, but task B does not.

1. Task C is running and uses `OS_pend(S)` to get the buffer. OS grants the buffer to C and C computes slowly on buffer.
2. Task A starts. Task A also requests buffer by `OS_pend(S)`. It has to wait for C to finish with `S`.
3. With Task A still waiting, Task B starts.
4. C is pre-empted by an interrupt.
5. Scheduler runs B. Although B does not even want the buffer (semaphore S), it blocks C because it has higher priority, thereby blocking A.

Priority Inversion Remedies

- Priority Inheritance. O/S has to check each semaphore `pend`.

  If a higher priority process is blocked by a `pend`, process *having* resource temporarily gets priority equal to blocked process.

- Priority Ceiling Protocol.

  When getting a resource, process temporarily gets priority equal to highest process sharing that resource.

Chapter 14

USB

14.1 USB Overview

14.1.1 Goals

- Ease of Use (for consumer)
- Low cost
- up to 12Mb/sec
- support real-time data / multimedia
- flexibility for different devices/needs
- Compatible with laptop & desktop
- Highly standardized for easy adoption by industry

14.1.2 Applications

(as anticipated in 1996)
Low Speed: 1.5 Mbit/s Keyboard, mouse, stylus, game peripherals.
Full Speed: 12Mbit/s Phone/Audio.
NOT: High Speed: Video, Disk Drive

14.1.3 Architecture

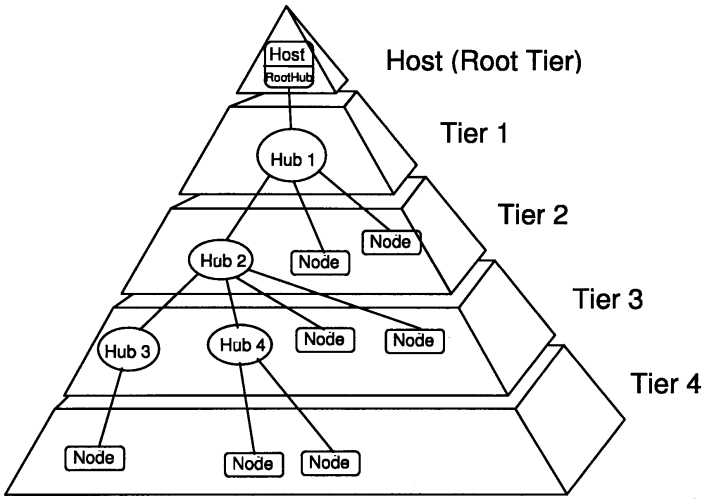


Figure 4-1. Bus Topology

(Figure 4-1, page 16)

"Function" — a USB peripheral device

A function can have one or more "endpoints"

"Hub" — a node that can join USB wires

"Compound Device" — a HUB and one or more Functions in a single package.

Compound Device

Figure 4-4 illustrates how hubs provide connectivity in a typical computer environment.

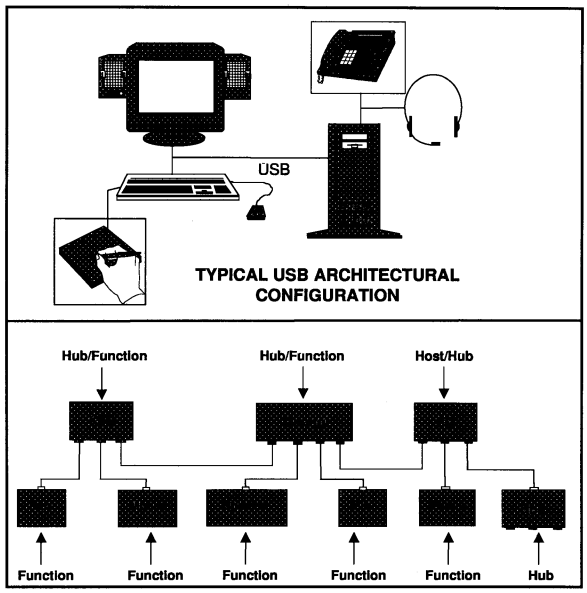


Figure 4-4. Hubs in a Desktop Computer Environment

(Figure 4-4, page 23)

14.2 Bus Protocol

Low speed: 1.5Mb / sec
Full speed: 12.5 Mb/sec (8x faster)
(we'll ignore low speed mode ...)

14.2.1 Frames and Polling

Time is divided into 1ms "Frames". Each frame time the Host transmits a "Start of Frame" (SOF) packet.

SOF packet structure

SYNC	PID	Frame Number (11bits)	CRC (5bits)

Frame Number rolls over.

PID

ALL packets have a packet ID (PID)

PID (4 bits)	$\overline{\text{PID}}$ (4 bits)

The 4-bit PID is repeated, complemented, inside the PID byte for error detection.

32 PID values including:

Type	Code
ACK	0010
NAK	1010
IN	1001
OUT	0001
SOF	0101
DATA0	0011
DATA1	1011

`TOKEN` packets are for `SOF`, polling, and pipe control

Polling Sequence

1. Host sends "token packet"

PID	ADDR (7 bits)	Endpoint (4 bits)	CRC (5 bits)

PID: `OUT (0001)` = host asks endpoint to receive data.
`IN (1001)` = host asks endpoint to send data.

2. source sends data packet to destination

PID	DATA (0-1023 bytes)	CRC (16 bits)

3. destination sends handshake packet (ACK/NAK)

PID

There can be many transactions in each frame. 10% of each frame is reserved for Control packets.

14.2.2 Pipes

Connections are called "pipes". Attributes of pipes:

- bandwidth
- data flow type
- endpoint characteristics (direction, buffer size).

Pipes are established when a device is plugged in and (auto) configured.
"Default Control Pipe" always exists for each device to set and query status and control of device.

14.2.3 Data Flow Types

Each pipe can be one of the following types

1. Control Transfer. configure the device when it is attached. Other control functions.
2. Bulk Data Transfer. non-realtime transfer of big file. Example: Printer
3. Interrupt Data Transfer. Small data amounts with short latency requirements. Example: Mouse
4. Isochronous Data Transfer. Fixed (negotiated) bandwidth amount with constant latency. Example: Audio.

Characteristics of Data Flow Types

1. Packet format
2. Direction of flow
3. Packet size (or size ranges)
4. Bus Access constraints
5. Latency Constraints
6. Error Handling

Control Transfer

10% of the bus bandwidth is reserved for control transfers.

Bulk

Low priority in the sense that Interrupt and Isochronus get the bandwidth first, Bulk gets what is left.

No data format requirement.

Max packet size is 8, 16, 32, or 64 bytes.

Packet Overhead

SYNC (3)	PID (3)	Endpoint	CRC (3)

Counts () indicate total for three transactions:

1. Poll (host to function)
2. Data (either direction)

SYNC	PID	Data	CRC (2)

3. `ACK/NAK`

Total overhead, 13 bytes packet header, 3 byte delay between packets.
Access granted as bandwidth is available.
If errors are detected, transfers are re-tried.
Guaranteed data delivery. No bandwidth or latency guarantee.
Transfer completion is detected when endpoint:
Has sent all bytes expected
Sends a packet with less than max payload size.

Interrupt

typically keystrokes and mouse movements.
Endpoints specify max packet size when configured.
Negotiated based on min(source,dest,bus avail).
Max packet size 64 bytes, max 108 transactions per frame.
A mouse typically sends packets only when moved by user. Each packet consists of a sync byte, and one byte each for ΔX, ΔY, and button/wheel status.

Isochronus

Periodic bursts of data at a pre-set rate. Host makes sure bus is available (quiet) at scheduled times for this transfer.
Endpoint specifies packet size when it is configured.
Packet overhead:

SYNC (2)	PID (2)	Endpoint	CRC (3)

Total overhead, 8 bytes packet header, 1 byte delay between packets.
No *length* field in header but length can vary(!)
Payload is up to 1023 bytes. max 150 1-byte transfers per frame.
1 packet per frame per endpoint.
Because latency must be maintained low, there is no time for `ACK/NAK` and re-transmission.
Hardware error rate is expected to be very low. But 0 error rate is not expected in this mode.

14.2.4 Devices

Devices are divided into classes (similar to C++ idea).
Devices must be able to *report* info on their identity and configuration.

- Standard Info: All USB devices: vendor ID, Class, Pwr Mgt., endpoints.
  USB Control and status info.
- Class Info: same for all devices of the same class. Example: Specific info for keyboards.
- Vendor: Proprietary info for the device vendor. Example: Firmware revision number.

Devices must support endpoint 0 for Default Control Pipe.

14.2.5 Speed

Type	Payload Size	Frame Utilization	Max bytes/sec
Bulk	16	2%	816,000
Bulk	64	5%	1,216,000
Interrupt	4	1%	352,000
Isochronus	16	2%	960,000
Isochronus	512	35%	1,024,000

Notes:
1,000,000 bytes / sec = 8 MBaud(!)
Isochronus — only 1 transfer per frame. BW above shows *total* for ALL Isochronus connections.

14.3 Electrical

Four wires:

1. VBUS
   +5V DC power. May power devices.
2. D+ Data line 1
3. D- Data line 2
4. GND

Software support for power management (suspend/resume)

14.3.1 Robustness

- differential line drivers and shielding
- CRC protection of all parts of packet. (1 and 2 bit errors 100% detected).
- timeouts for lost packets — self recovery
- Flow control for low priority streams prevents buffer overflow/underflow.

14.3.2 Attachment

Devices can be plugged in to hub at any time. Hub has status bits which let host know which ports have devices plugged in.

- host queries hub
- hub reports status bits
- if host detects a new device, it
- assigns a unique USB address to the new port.
- sends a control packet to enable the port.

- if the new device is a hub, repeat process for new hub.
- if new device is a function, notify software driver on host that a new instance of this function has been attached to the bus.

14.3.3 Voltage Levels and Rise Times:

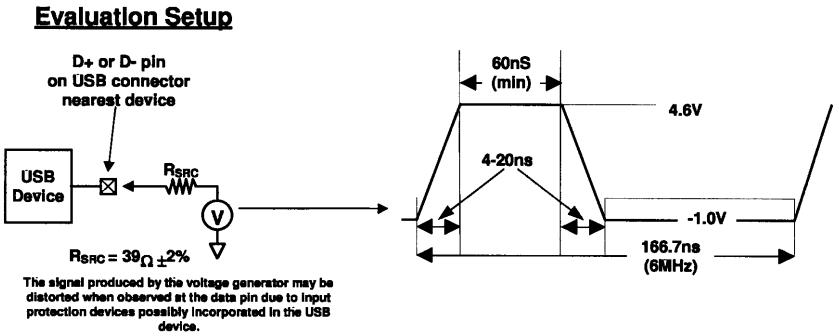


Figure 7-1. Maximum Input Waveforms for USB Signaling

(Figure 7-1, page 107)

Driver Low, VOL, must be < 0.3v (1.5kΩ to 3.6V)
Driver High, VOH, must be > 2.8v (15kΩ to 0V)
Receiver High, VIH, > 2.0V must be detected as High
Receiver Low, VIL, < 0.8V must be detected as Low
Worst case inputs: [-1.0V — +4.6V]
90Ω characteristic impedance (high speed)

14.3.4 Differential Drive

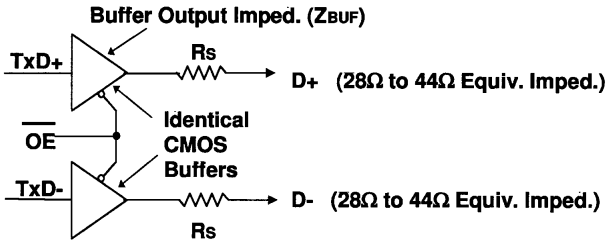


Figure 7-2. Example Full-speed CMOS Driver Circuit

(Fig 7-2 page 108)
D+ and D- usually signal by their *difference*
This eliminates common mode noise
`OE` enables the output drivers.

14.3.5 Signaling Levels (high speed)

Differential "1": (D+ − D-) > 200mV
Differential "0": (D- − D+) > 200mV
Idle: Differential "1"
Start of Packet (SOP): Idle → Differential "0"
("J" and "K" states (!))
Reset: D+ and D- < 0.8V for >= 10 ms.

14.4 Modulation and Channel Coding

14.4.1 NRZI

Non Return to Zero:

Logic zero is sent by toggling

Logic one is sent by not toggling

`Data` → `NRZI Signal`

`00000000` → `01010101`

`11111111` → `11111111`

14.4.2 Bit Stuffing

Packets don't have as many start and stop bits as RS-232

Receiver needs to see 1→0 or 0→1 transitions at least every seven bits.

Bit-stuffing inserts a zero whenever there are 6 ones in a row.

`11111111` → `111111011`

Now the receiver sees at least a few transitions for *any* data stream and so it can keep its clock phase locked.

Receiver hardware automatically removes stuffed bits.

14.4.3 Update: USB 2.0

Released April 2000.

Logically the same as 1.1, backwards compatible.

Provides additional speed mode: **Hi-Speed** 480 Mbit/s

Caution: if a device is labeled "Full Speed USB 2.0" you are really getting USB 1.1 (!)

Only "High Speed" or "480 Mbit/s" is a "real" USB 2.0 device.