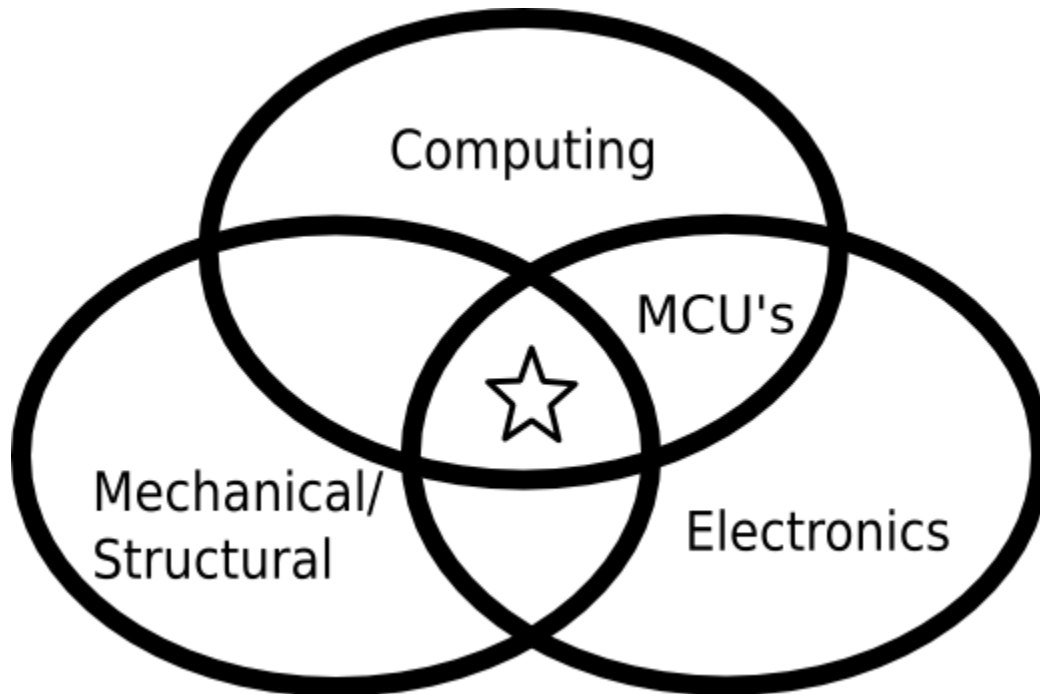


Class Notes for ATS 3203 Digital Microsystems

Instructor Ed Bennett

Kinetic art probably began in 1920 with Marcel Duchamp's experiments with his "Rotorelief". From a physical point of view, what made the Rotorelief novel in the world of fine art was the use of power or energy to operate the artwork. Beyond being able to set the speed of rotation the Rotoreliefs (there were several) there was no concept of control employed. Kinetic art uses movement, light, and sound as



Mechatronics: The nexus of electronics, computer control, and mechanical and structural form. Microcontrollers (MCU's) overlap Computing and Electronics.

effectors and those energies might or might not be controlled beyond being turned on when the gallery opens and off at the end of the day. Over the years since the appearance of the Rotorelief, every form of electrical and mechanical control available to business and industry has been applied to kinetic artworks.

The result of applying control to a device which uses or transforms energy is behavior. Control systems can be made of electrical, mechanical, electro-mechanical, pneumatic, electronic, and computing devices. The choice of controller in a kinetic artwork is often not dictated by cost or efficiency. Many times the aesthetics of an artwork relate to not only what the object might be in name or appearance, but also what it is made of and how it works. The artist defines the visual and behavioral language of the piece and chooses technology appropriate for evoking a viewer or participant's response. If control is an element of the piece's design, that control can come from any field of science or technology, and from any era of history or invention as long as it is conceptually congruent with the artist's strategy. (Engineers and designers work on a different value system which always chooses technologies to maximize control capability and minimize cost.) Generally what drives forward the technology of control in kinetic art is a quest for more options in the design of device behavior.

The more flexible a controller is, the more versatile the behavior it can produce. The ultimate flexibility

is software programming because device behavior can be changed at the artist's discretion by changing the source code rather than the hardware. Controller interoperability with other software-based devices including wired and wireless networks, hand held devices, and interactive multimedia through sensor input is making the hardware-software mash-up approach to experimental art (and design) ever more common.

Kinetic art became very popular in the late 1960's and early 1970's. After that time it returned to the background of art and culture, never completely going away. In the late 1990's kinetic art began a return to a more conspicuous position in popular culture under the heading of "robotics" or robotic art. The traditional definitions of robotics would normally make reference to vehicles or manipulators (robotic arms). In current usage however, robotic art often just means kinetic art, with a strong implication of control or behavior as an element of design. This is due in very large part to easy availability of cheap re-programmable microcontrollers and cheap or free software development tools to go with them.

A "black box" containing a microcontroller and support circuitry for running a sculpture or installation is an example of an *embedded system*. Embedded controllers need not be small microcontrollers. Some are quite powerful computers. What makes them embedded is that they (1) are computers of some flavor, and (2) don't have the normal keyboard/screen/mouse that a PC does. Other terms around this subject are *embedded computer* and *dedicated computer*.

In engineering and academia there is a term which describes the nexus of disciplines which gives rise to computer-controlled devices: *mechatronics*. The term is used more in Europe than the U.S. (this is changing), but its message has implications for all kinds of manufactured items.

Mechatronics addresses what things are made of, how they behave, and how they are designed and built. Implicit in the commercial practice of mechatronic design is the tightest possible integration of design software and concept flow between disciplines. Generally, people in industry and academia using the word mechatronics to describe their work are trained in engineering disciplines which start at "first principles". That is, physical systems are modeled and manipulated as systems of equations which describe interactions of matter, energy, and time. The field of robotics could be thought of as a subset of mechatronics. But the word mechatronic is also used by hobbyists, artists, and other informal practitioners. The common denominator is that the product is some kind of computer controlled "thing" that exists outside of a keyboard, screen and mouse.

The word mechatronic has been criticized as being "ugly" or sounding contrived. It doesn't exactly trip off the tongue. The "mech-" also emphasizes the connection to things mechanical and this is seen by some belonging to the Machine Age or prioritizing the section of the field which is reputed to be the most difficult part to get right, and so is a turn-off. The desire to break out of the box and bring computing into the real world in order to expand the creative options in the various fields of New Media is fueling an explosion of materials and methods for linking hardware and multimedia systems. Much of this work uses some sort of embedded controller to facilitate sensor input into existing multimedia authoring platforms. A smaller portion of the current effort goes to programming stand-alone embedded controllers from within multimedia environments and creating physical things - sculptures, machines, and installations - which are controlled by or use input generated in multimedia software. Several terms are being used to describe these efforts. The most popular is *physical computing*. Seen also are the terms *tangible computing*, *ubiquitous computing (ubicomp)*, and *smart objects*.

The material presented here assumes a hardware perspective from the outset rather than a multimedia perspective. Multimedia is important and relevant, and linkages to multimedia practice will be provided

in later sections. The level of abstraction will be kept deliberately low so much of what is said here can be applied to other systems of embedded control for art making.

The Toolchain

Our topic is programming microcontrollers and any discussion of such has to start by identifying the MCU or MCU's being used, and the software tools being used to write the programs. Microcontrollers exist on a diet of 1's and 0's. We don't write programs in 1's and 0's (unless we're very strange or very bored) so whatever language we write in has to be masticated by some piece of software into a form the MCU can load and execute. The programming language we will use is called C, and the program which chews our C code so that the MCU can digest it is called a *compiler*.

We will mention a few different microcontroller chips along the way. They will all be members of Microchip® Corporation's (www.microchip.com) mid-range PIC family of MCU's and have between eight and 40 pins, depending on the type. Programming will be done in C, using the CCS (www.ccsinfo.com) PCM compiler. Microchip makes many dozens of sizes and styles of MCU's. The CCS PCM compiler can produce code which will run on around 140 different models of PIC chip. The M in PCM stands for Mid-range.

To program a PIC chip:

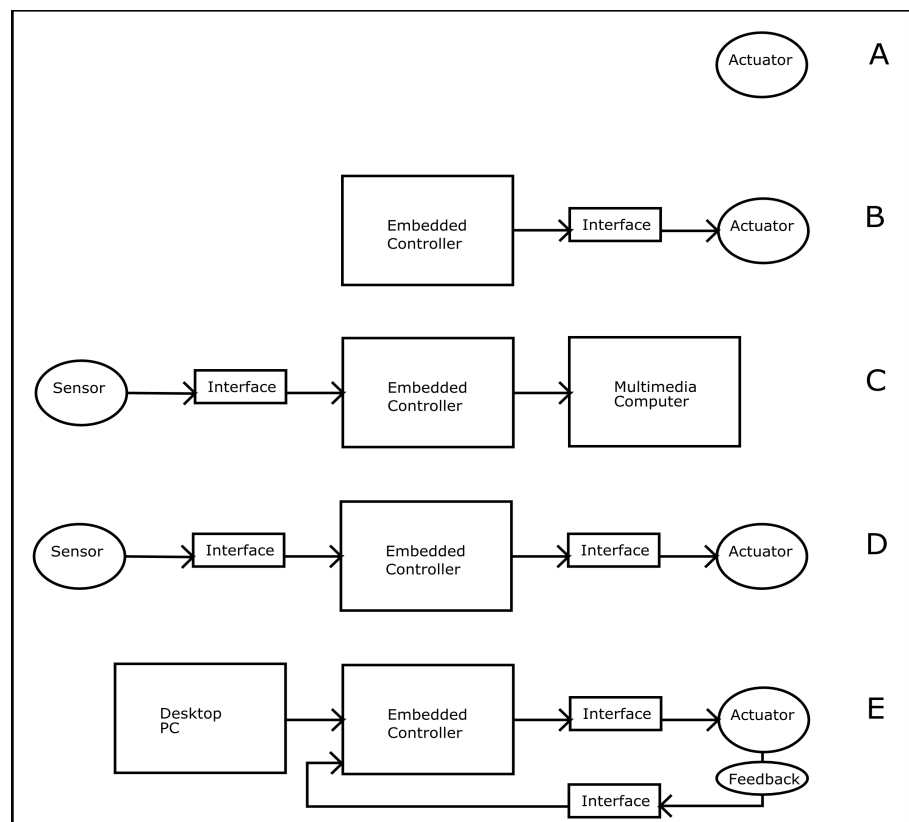
1. You write your C program in a text editor. Almost any text editor can be made to work, but ones for programming are easier to use for the task. Save the file as unformatted text with the extension ".c". (Ex: myprogram.c) This is called the *source file*. The program you've written in the text editor is called the *source code*, or just the *source*. So the source file contains the source code. The source file myprogram.c should be in a folder called "myprogram". This is important. Each program gets its own folder.
2. Point the compiler at the C file. Usually this is done in a gui and you don't have to see how it works. I hope it's not too confusing to interject this here, but if I wanted to compile a C program for my Linux desktop machine (not a PIC, obviously), the way I would point the GCC Linux compiler at the C source file, would be to open a terminal window and type the command "cc myprogram.c". Which means "Hey, C compiler, compile myprogram.c". When it's done the compiler writes a new file to the hard disk. The new file is the *executeable*. This is the file in 1's and 0's that the computer can run. The executables produced by the CCS compiler have names that end in ".hex". So "myprogram.c" would compile to "myprogram.hex".
3. Now we have to put myprogram.hex into the PIC. This is called "burning the chip". Burning requires some hardware and a program running on the PC to control that hardware. There are many ways of doing this step, that is to say, there are many kinds of burners and burning software. We will use the "PICkit 2" programmer device, by Microchip. The PICkit 2 plugs into the PC with a USB cable and it plugs into the circuitboard containing the PIC to be programmed (the *target*). We activate the PICkit 2 by clicking the correct button in our GUI, and after a few seconds the lights on the PICkit 2 stop blinking and we're done. Now when the PIC chip is powered up it will go straight to running our program.

Some Common Examples of Systems Using Control

An actuator is some device which consumes power to move a load. Actuators are output devices. In a table fan the actuator is a motor and the load is the fan blades. In a table lamp the actuator and the load are both the light bulb. Other actuators would be hydraulic cylinders and loudspeakers.

An interface is a circuit which is a “bridge” connecting two devices which don't have the same kinds of input/output. When thinking of interfaces, I'm reminded of the Roman god Janus. Janus has two faces and is “... used to symbolize change and transitions such as the progression of past to future, of one condition to another, of one vision to another, the growing up of young people, and of one universe to another.” [Wikipedia]. An interface has two faces, each face having the characteristics of the device into which it looks. To properly construct an interface, the characteristics of both devices being connected must be understood.

In the set of diagrams A through E, “A” represents an actuator with no controlled behavior. A normal table lamp or fan, or the Roto-relief would be examples. “B” would be a non-interactive kinetic sculpture which has some kind of controlled behavior. Interactive as I'm using the term here means either interacting with people, or its environment, or both. “C” is a sensor with signal conditioning for use as input to a multimedia computer. “D” is a fully interactive kinetic artwork with controlled behavior. “E” is a servo system. The command input from a computer or human enters the controller on the left.



An MCU is a rather complete little computer on a chip. What makes an MCU most different from other microprocessors is that the program the MCU will run is stored in the same silicon as the computing circuitry. MCU's also have built-in hardware for interfacing to sensors, timing things, controlling power, communication, and so on.

Bits and Logic

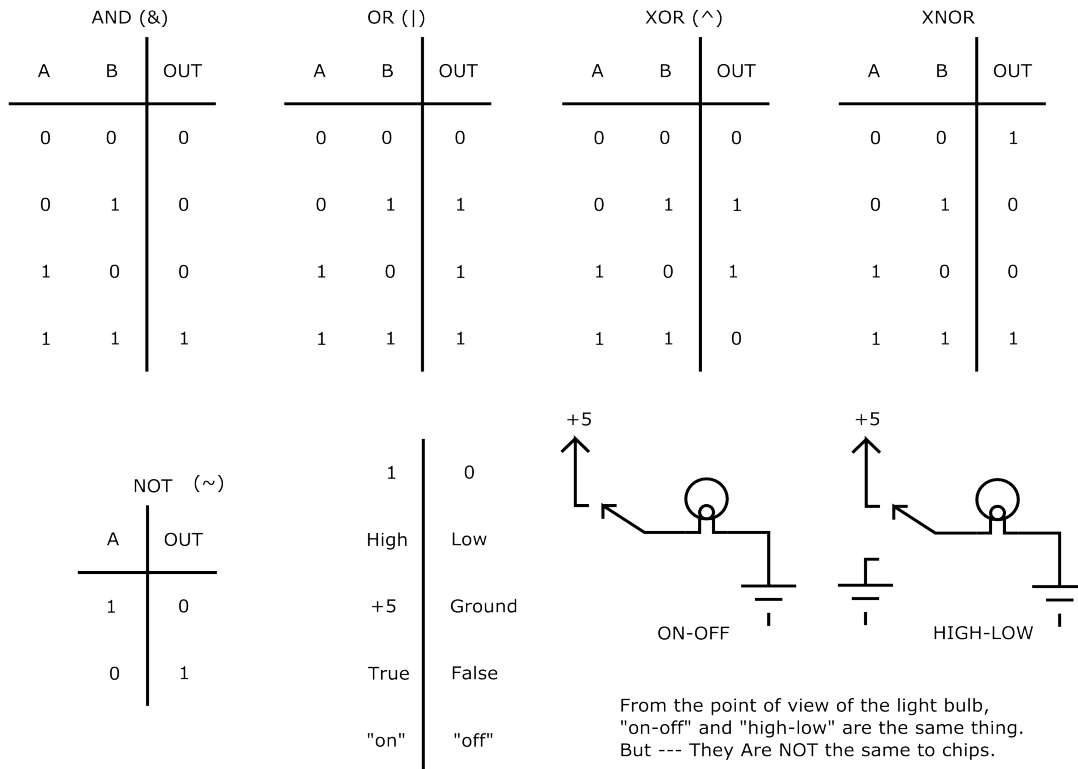
Bit. **B**inary **d**ig**IT**. The smallest possible piece of information: “Is a thing, or is it not?” A bit can be 1 or 0. If a bit is not 1 then it is 0, and so on.

In C, logical FALSE is equal to zero. Anything else is TRUE. It's often convenient to say that 1 equals TRUE, but 3.141, “cat” , and (7>5) are TRUE as well. If you tell something to be TRUE, it will be

made equal to 1.

Bitwise operators.

Statements of boolean logic can be written with the bitwise operators. These are more prominent in microcontroller work than desktop PC programming because MCU's often read and write their I/O (Input/Output) lines as bits when reading switches or turning things on and off.



The most basic bitwise operators are AND, OR, and NOT. For an AND statement to be TRUE, both of its inputs must be TRUE. For an OR statement to be TRUE, at least one of its inputs must be TRUE. NOT, or negation outputs a FALSE when its input is TRUE and a TRUE when its input is FALSE. XOR, eXclusive OR, outputs a TRUE when one or the other of its inputs is TRUE. In other words, the inputs must be *different* to give a TRUE output. The XOR is also called a half-adder. The XNOR, eXclusive NOT OR, produces a TRUE output when its inputs are *the same*.

A happy aside:

These decisions (AND, OR, NOT) form the fundamental basis of mechanical logic, and by extension, computing. Most of our experience using and programming computers involves layers of metaphor and abstraction which separate us from what goes on in the hardware. The bitwise operations are where there's really no abstraction left and you start to think like the machine.

NOTE WELL:

The bitwise operators and the logical operators are different in C. The ~ is a bitwise NOT (The CCS manual calls it a one's complement operator). It inverts all the bits in a byte. The ! is a logical NOT. It is used to invert the logical condition of a value which evaluates to TRUE or FALSE. The symbol & is the bit-wise AND operator which is rarely used in C except by we microcontroller programmers. The symbol && is the logical AND, and is used all the time in evaluating statements like

```
if(myVar1 && myVar2){
    something happens if myVar1 and myVar2 are both true
}
```

Comments: something you need to know now: We can hide things in our C file from the compiler by commenting them out. There are two ways to do this: // and /* */

```
// I'm a short little comment. The compiler can't see me.
```

```
/* I'm a longer
   comment spanning
   across several lines!
   The compiler can't see me, either!!
*/
```

It's very common to comment out lines of code which are suspected of being “broken” in order to try something else without deleting the original code. We might want to put it back the way it was before we messed with it.

Now, back to the show. Our first program.

```
// boolean.c
// shows syntax for bitwise AND, OR, NOT, and XOR
//
// connections: LED and resistor on 12F683 pin 5 (PIN_A2),
// a momentary pushbutton to +5 on 12F683 pin 2 (PIN_A5) and
// a momentary pushbutton to +5 on 12F683 pin 3 (PIN_A4)
// also 1K pull-down resistors on 12F683 pins 2 and 3.
//

#include <12F683.h>
#include <STDLIB.H>
#fuses INTRC_IO,NOWDT,NOPROTECT,NOMCLR,NOBROWNOUT,NOIESO,NOFCMEN

boolean result;
boolean first_input;
boolean second_input;

void main(void){

    while(1){
        first_input = input(PIN_A4); // Read the inputs!
        second_input = input(PIN_A5);

        //uncomment (remove the //) one of these to make it go:

        /* AND - both must be TRUE for result to be TRUE */
        //result = first_input & second_input;

        /* OR - either (or both) must be TRUE for result
         * to be TRUE */
        //result = first_input | second_input;

        /* NOT - Inversion or negation of input */
        //result = !first_input;
```

```
/* XOR -eXclusive OR- one or the other, but not
 * both - inputs must be DIFFERENT for TRUE output */
//result = first_input ^ second_input;

/* XNOR - eXclusive NOT OR- inputs must be SAME for TRUE
 * output */
//result = !(first_input ^ second_input);

output_bit(PIN_A2,result);
```

```
}
```

```
}
```