MICROCOMPUTER APPLICATIONS

This article reviews the field of microcomputer applications. We will discuss basic concepts and provide examples of microcomputers used in the design of embedded systems. We begin with an overall discussion of the topic and introduce relevant terminology. Next, we present the fundamental hardware and software building blocks required to construct a microcom-

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

puter system. Then, we customize our computer system by interfacing specific devices to create the desired functionality. We conclude with a systems-level approach to microcomputer applications by presenting a few case studies that illustrate the spectrum of applications which employ microcomputers.

OVERVIEW OF MICROCOMPUTER APPLICATIONS

The term *embedded microcomputer system* refers to a device that contains one or more microcomputers inside. To get a better understanding, we break the expression "embedded microcomputer system" into pieces. In this context, the word *embedded* means "hidden inside so we can't see it." A computer is an electronic device with a processor, memory, and input/output ports, as shown in Fig. 1. The processor performs operations (executes software). The processor includes registers (which are high-speed memory), an arithmetic logic unit (ALU) (to execute math functions), a bus interface unit (which communicates with memory and I/O), and a control unit (for making decisions.) Memory is a relatively high-speed storage medium for software and data. Software consists of a sequence of commands (functions) which are usually executed in order. In an embedded system, we use read only memory (ROM) (for storing the software and fixed constant data,) and random access memory (RAM) (for storing temporary information.) The information in the ROM is nonvolatile, meaning the contents are not lost when power is removed. I/O ports allow information to enter via the input ports and exit via the output ports. The software, together with the I/O ports and associated interface circuits, give an embedded computer system its distinctive characteristics.

The term *microcomputer* means a small computer. Small in this context describes its size not its computing power, so a microcomputer can refer to a very wide range of products from the very simple (e.g., the PIC12C08 is an 8-pin DIP microcomputer with 512 by 12 bit ROM, 25 bytes RAM, and 5 I/O pins) to the most powerful Pentium. We typically restrict the term embedded to systems which do not look and behave like a typical computer. Most embedded systems do not have a keyboard, a graphics display, or secondary storage (disk). In the context of this article we will focus on the microcomputers available as single chips, because these devices are more suitable for the embedded microcomputer system.

We can appreciate the wide range of embedded computer applications by observing existing implementations. Examples of embedded microcomputer systems can be divided into categories:

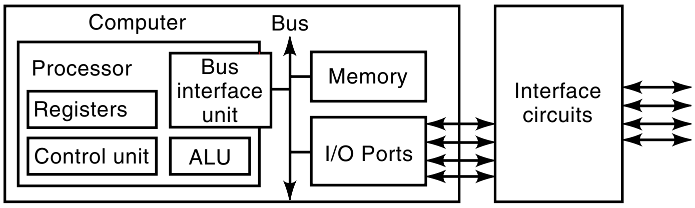


Figure 1. An embedded computer system performs dedicated functions.

1. Consumer
   - Washing machines (computer controls the water and spin cycles)
   - Exercise bikes (computer monitors the workout)
   - TV remotes (computer accepts key touches and sends IR pulses)
   - Clocks and watches (computer maintains the time, alarm, and display)
   - Games and toys (computer entertains the child)
   - Audio/video (computer interacts with the operator and enhances performance)
2. Communication
   - Telephone answering machines (record and play back messages)
   - Cellular phones and pagers (provide a wide range of features)
   - Cordless phones (combine functionality and security)
   - ATM machines (provide both security and banking convenience)
3. Automotive
   - Automatic braking (optimizes stopping on slippery surfaces)
   - Noise cancellation (improves sound quality by removing background noise)
   - Theft-deterrent devices (keyless entry, alarm systems)
   - Electronic ignition (controls spark plugs and fuel injectors)
   - Power windows and seats (remember preferred settings for each driver)
   - Instrumentation (collects and provides the driver with necessary information)
4. Military
   - Smart weapons (don't fire at friendly targets)
   - Missile-guidance systems (direct ordnance at the desired target)
   - Global positioning systems (can tell you where you are on the planet)
5. Industrial
   - Set-back thermostats (adjust day/night thresholds, saving energy)
   - Traffic-control systems (sense car positions and control traffic lights)
   - Robot systems used in industrial applications (computer controls the motors)
   - Bar code readers and writers for inventory control
   - Automatic sprinklers for farming (control the wetness of the soil)
6. Medical
   - Monitors (measure important signals and generate alarms if patient needs help)
     - Apnea (monitor breathing and alarms if baby stops breathing)
     - Cardiac (monitor heart functions)
     - Renal (study kidney functions)
   - Therapeutic devices (deliver treatments and monitor patient response)
     - Drug delivery
     - Cancer treatments (radiation, drugs, heat)
   - Control devices (take over failing body systems providing life-saving functions)

- Pacemakers (help the heart beat regularly)
- Prosthetic devices (increase mobility for the handicapped)
- Dialysis machines (perform functions normally done by the kidney)

MICROCOMPUTER COMPONENTS

Hardware Components

Digital Logic. There are many logic families available to design digital circuits. Each family provides the basic logic functions (and, or, not), but differ in the technology used to implement these functions. This results in a wide range of parameter specifications. Some of the basic parameters of digital devices are listed in Table 1. Because many microcomputers are high-speed CMOS, typical values for this family are given. In general, it is desirable to design digital systems using all components from the same family.

Speed. There are three basic considerations when using digital logic. The first consideration is speed. For simple combinational logic, speed is measured in propagation delay or the time between changes in the input to resulting changes in the output. Another speed parameter to consider is the rise time of the output (time it takes an output signal to go from high to low or from low to high). A related parameter is slew rate (dV/dt on outputs during transitions). For memory devices, speed is measured in read access time, which is how long it takes to retrieve information. For communication devices, we measure speed in bandwidth, which is the rate at which data are transferred.

Power. The second consideration is power. Many embedded systems run under battery power or otherwise have limited power. High-speed CMOS is often used in embedded applications because of its flexible range of power supply voltages and low power supply current specifications. It is important to remember that CMOS devices require additional current during signal transitions (e.g., changes from low to high or from high to low). Therefore, the power supply current requirements will increase with the frequency of the digital signals. A dynamic digital logic system with many signal transitions per second requires more current than a static system with few signal transitions.

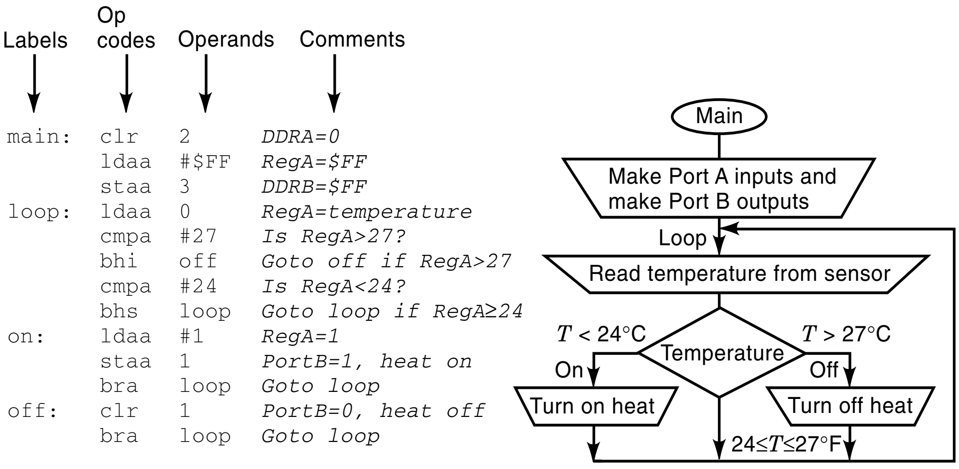
Loading. The third consideration is signal loading. In a digital system, where one output is connected to multiple inputs, the sum of the I_{IL} of the inputs must be less than the available I_{OL} of the output which is driving those inputs. Similarly, the sum of the I_{IH}'s must be less than the I_{OH}. Using the above data, we might be tempted to calculate the fanout (I_{OL}/I_{IL}) and claim that one 74HC04 output can drive 4000 74HC04 inputs. In actuality, the input capacitance's of the inputs will combine to reduce the slew rate (dV/dt during transitions). This capacitance load will limit the number of inputs one CMOS output gate can drive. On the other hand, when interfacing digital logic with external devices, these currents (I_{OL}, I_{OH}) are very important. Often in embedded applications we wish to use digital outputs to control non-CMOS devices like relays, solenoids, motors, lights, and analog circuits.

Table 1. Some Typical Parameters of a High-Speed CMOS 74HC04 Not Gate

Parameter	Meaning	Typical 74HC04 Value
V_{cc}	Power supply voltage	2 V to 6 V
I_{cc}	Power supply current	20 μA max (with V_{cc} = 6 V)
t_{pd}	Propagation delay	24 ns max (with V_{cc} = 4.5 V)
V_{IH}	Input high voltage	3.15 V min (with V_{cc} = 4.5 V)
I_{IH}	Input high current	1 μA max (with V_{cc} = 6 V)
V_{IL}	Input low voltage	0.9 V max (with V_{cc} = 4.5 V)
I_{IL}	Input low current	1 μA max (with V_{cc} = 6 V)
V_{OH}	Output high voltage	4.4 V min (with V_{cc} = 4.5 V)
I_{OH}	Output high current	4 mA max (with V_{cc} = 4.5 V)
V_{OL}	Output low voltage	0.33 V max (with V_{cc} = 4.5 V)
I_{OL}	Output low current	4 mA max (with V_{cc} = 4.5 V)
C_I	Input capacitance	10 pF

Application-Specific Integrated Circuits. One of the pressures which exist in the microcomputer embedded systems field is the need to implement higher and higher levels of functionality into smaller and smaller amounts of space using less and less power. There are many examples of technology developed according to these principles. Examples include portable computers, satellite communications, aviation devices, military hardware, and cellular phones. Simply using a microcomputer in itself provides significant advantages in this faster-smaller race. Since the embedded system is not just a computer, there must also be mechanical and electrical devices external to the computer. To shrink the size and power required of these external electronics, we can integrate them into a custom IC called an application-specific integrated circuit (ASIC). An ASIC provides a high level of functionality squeezed into a small package. Advances in integrated circuit design allow more and more of these custom circuits (both analog and digital) to be manufactured in the same IC chip as the computer itself. In this way, systems with fewer chips are possible.

Microprocessor. In the last 20 years, the microprocessor has made significant technological advances. The term microprocessor refers to products ranging from the oldest Intel 8080 to the newest Pentium. The processor, or CPU, controls the system by executing instructions. It contains a bus interface unit (BIU), which provides the address, direction (read data from memory into the processor or write data from processor to memory), and timing signals for the computer bus. The registers are very high-speed storage devices for the computer. The program counter (PC) is a register which contains the address of the current instruction which the computer is executing. The stack is a very important data structure used by computers to store temporary information. It is very easy to allocate temporary storage on the stack and deallocate it when done. The stack pointer (SP) is a register which points into RAM specifying the top entry of the stack. The condition code (CC) is a register which contains status flags describing the result of the previous operation and operating mode of the computer. Most computers have data registers which contain information and address registers which contain pointers. The arithmetic logic unit (ALU) performs arithmetic (add, subtract, multiply, divide) and logical (and, or, not, exclusive or, shift) operations. The inputs to the ALU come from registers and/or memory, and the outputs go to registers or memory. The CC register contains status information from the previous ALU operation. Typical CC bits include:



```
        Op
Labels  codes  Operands  Comments

main:   clr    2         DDRA=0
        ldaa   #$FF      RegA=$FF
        staa   3         DDRB=$FF
loop:   ldaa   0         RegA=temperature
        cmpa   #27       Is RegA>27?
        bhi    off       Goto off if RegA>27
        cmpa   #24       Is RegA<24?
        bhs    loop      Goto loop if RegA≥24
on:     ldaa   #1        RegA=1
        staa   1         PortB=1, heat on
        bra    loop      Goto loop
off:    clr    1         PortB=0, heat off
        bra    loop      Goto loop
```

Figure 2. This program implements a bang-bang temperature controller by continuously reading temperature sensor on port A (location 0), comparing the temperature to two thresholds, then writing to the heater connected to port B (location 1) if the temperature is too hot or too cold.

- Z result was zero
- N result was negative (i.e., most significant bit set)
- C carry/borrow or unsigned overflow
- V signed overflow (some computers do not have this bit)

Software is a sequence of commands stored in memory. The control unit (CU) manipulates the hardware modules according to the software that it is executing. The CU contains an instruction register (IR), which holds the current instruction. The BIU contains an effective address register (EAR) which holds the effective address of the current instruction. The computer must fetch both instructions (op codes) and information (data). Both types of access are controlled by the bus interface unit.

When an instruction is executed, the microprocessor often must refer to memory to read and/or write information. Often the I/O ports are implemented as memory locations. For example, on the Motorola 6812, I/O ports A and B exist as locations 0 and 1. Like most microcomputers, the I/O ports can be configured as inputs or outputs. The 6812 Port A and B have direction registers at locations 2 (DDRA) and 3 (DDRB), respectively. The software writes 0's to the direction register to specify the pins as inputs, and 1's to specify them as outputs. When the 6812 software reads from location 0 it gets information from Port A, and when the software writes to location 1, it sends information out Port B. For example, the Motorola 6812 assembly language program, shown in Fig. 2, reads from a sensor which is connected to Port A, if the temperature is above 27 °C, it turns off the heat (by writing 0 to Port B). If the temperature is below 24 °C, then it turns on the heat by writing 1 to Port B.

Microcomputer. The single-chip microcomputer is often used in embedded applications because it requires minimal external components to make the computer run, as shown in Fig. 3. The reset line (MCLR on the PIC or RESET on the 6805) can be controlled by a button, or a power-on-reset circuit.

During the development phases of a project, we often would like the flexibility of accessing components inside the single-chip computer. In addition, during development, we are often unsure of the memory size and I/O capabilities that will be required to complete the design. Both of these factors point to the need for a single-board computer like the one shown in Fig. 4. This board has all of the features of the single-chip computer but laid out in an accessible and expandable manner. For some microcomputer systems, the final product is delivered using a single-board computer. For example, if the production volume is small and the project does not have severe space constraints, then a single-board solution may be cost-effective. Another example of a final product delivered with a single-board occurs when the computer requirements (memory size, number of ports, etc.) exceed the capabilities of any single-chip computer.

Choosing a Microcomputer

The computer engineer is often faced with the task of selecting a microcomputer for the project. Figure 5 presents the relative market share for the top twelve manufacturers of 8 bit microcontrollers. Often the choice is focused only on those devices for which the engineers have hardware and software experience. Because many of the computers overlap in their cost and performance, this is many times the most appropriate approach to product selection. In other words, if a microcomputer that we are familiar with can implement the desired functions for the project, then it is often efficient to bypass that more perfect piece of hardware in favor of a faster development time. On the other hand, sometimes we wish to evaluate all potential candidates. It may be cost-effective to hire or train the engineering personnel so that they are proficient in a wide spectrum of potential computer devices. There are many factors to consider when selecting an embedded microcomputer:

- Labor costs include training, development, and testing
- Material costs include parts and supplies
- Manufacturing costs depend on the number and complexity of the components
- Maintenance costs involve revisions to fix bugs and perform upgrades
- ROM size must be big enough to hold instructions and fixed data for the software
- RAM size must be big enough to hold locals, parameters, and global variables

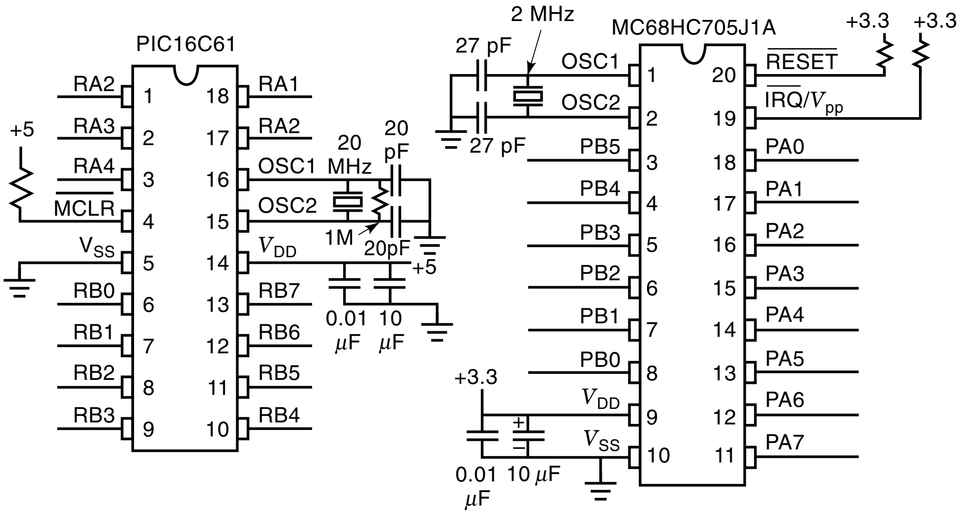


Figure 3. These PIC and 6805 single-chip microcomputer circuits demonstrate that to make the computer run, usually all we need to add is an external crystal for the clock.

- EEPROM to hold nonvolatile fixed constants which are field configurable
- Speed must be fast enough to execute the software in real time
- I/O bandwidth affects how fast the computer can input/output data
- 8, 16, or 32 bit data size should match most of the data to be processed
- Numerical operations, like multiply, divide, signed, floating point
- Special functions, like multiply&accumulate, fuzzy logic, complex numbers
- Enough parallel ports for all the input/output digital signals
- Enough serial ports to interface with other computers or I/O devices

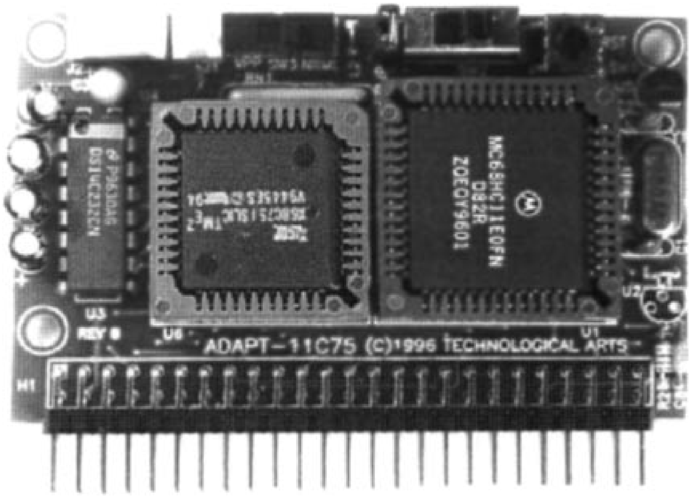


Figure 4. The Adapt-11C75 board from Technological Arts is a typical example of a single-board microcomputer used to develop embedded applications. It is based on the Motorola MC68HC11 computer, and has 8 K of external EEPROM. Additional I/O ports and memory can be easily added to the 50-pin connector.

- Timer functions generate signals, measure frequency, measure period
- Pulse width modulation for the output signals in many control applications
- ADC is used to convert analog inputs to digital numbers
- Package size and environmental issues affect many embedded systems
- Second source availability
- Availability of high-level language cross-compilers, simulators, emulators
- Power requirements, because many systems will be battery operated

When considering speed it is best to compare time to execute a benchmark program similar to your specific application, rather than just comparing bus frequency. One of the difficulties is that the microcomputer selection depends on the speed and size of the software, but the software cannot be written without the computer. Given this uncertainty, it is best to select a family of devices with a range of execution speeds and memory configurations. In this way a prototype system with large amounts of memory and peripherals can be purchased for software and hardware development and, once the design is in its final stages, the specific version of the

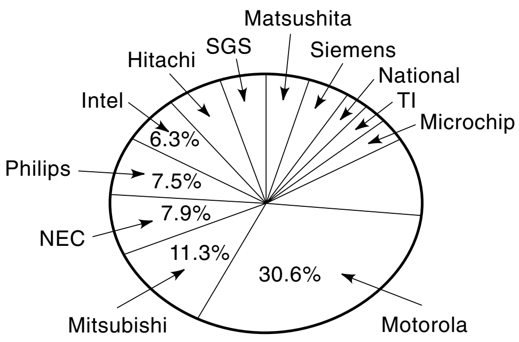


Figure 5. 1995 worldwide market share in dollars for 8 bit microcontrollers (from 1997 Motorola University Symposium, Austin, TX).

computer can be selected, knowing the memory and speed requirements for the project.

Software

Assembly Language. An assembly language program, like the one shown in Fig. 2, has a 1 to 1 mapping with the machine code of the computer. In other words, one line of assembly code maps into a single machine instruction. The label field associates the absolute memory address with a symbolic label. The op code represents the machine instruction to be executed. The operand field identifies the data itself or the memory location for the data needed by the instruction. The comment field is added by the programmer to explain what, how, and why. The comments are not used by the computer during execution, but rather provide a means for one programmer to communicate with another, including oneself at a later time. This style of programming offers the best static efficiency (smallest program size), and best dynamic efficiency (fastest program execution). Another advantage of assembly language programming is the complete freedom to implement any arbitrary decision function or data structure. One is not limited to a finite list of predefined structures as is the case with higher level languages. For example one can write assembly code with multiple entry points (places to begin the function).

High-Level Languages. Although assembly language enforces no restrictions on the programmer, many software developers argue that the limits placed on the programmer by a structured language, in fact, are a good idea. Building program and data structures by combining predefined components makes it easy to implement modular software, which is easier to debug, verify correctness, and modify in the future. Software maintenance is the debug, verify, and modify cycle, and it represents a significant fraction of the effort required to develop products using embedded computers. Therefore, if the use of a high-level language sacrifices some speed and memory performance, but gains in the maintenance costs, most computer engineers will choose reliability and ease of modification over speed and memory efficiency.

Cross-compilers for C, C++, BASIC, and FORTH are available for many single-chip microcomputers, with C being the most popular. The same bang-bang controller presented in Fig. 2 is shown in Fig. 6 implemented this time in C and FORTH.

One of the best approaches to this assembly versus high-level language choice is to implement the prototype in a high-level language, and see if the solution meets the product specifications. If it does, then leave the software in the high-level language because it will be easier to upgrade in the future. If the software is not quite fast enough (or small enough to fit into the available memory), then one might try a better compiler. Another approach is to profile the software execution, which involves collecting timing information on the percentage of time the computer takes executing each module. The profile allows you to identify a small number of modules which, if rewritten in assembly language, will have a big impact on the system performance.

Software Development

Simple Approach. Recent software and hardware technological developments have made significant impacts on the software development for embedded microcomputers. The simplest approach is to use a cross-assembler or cross-compiler to convert source code into the machine code for the target system. The machine code can then be loaded into the target machine. Debugging embedded systems with this simple approach is very difficult for two reasons. First, the embedded system lacks the usual keyboard and display that assist us when we debug regular software. Second, the nature of embedded systems involves the complex and real-time interaction between the hardware and software. These real-time interactions make it impossible to test software with the usual single-stepping and print statements.

Logic Analyzer. A logic analyzer is a multiple channel digital oscilloscope. For the single-board computer which has external memory, the logic analyzer can be placed on the address and data bus to observe program behavior. The logic analyzer records a cycle-by-cycle dump of the address and data bus. With the appropriate personality module, the logic analyzer can convert the address&data information into the corresponding stream of executed assembly language instructions. Unfortunately, the address and data bus singles are not available on most single-chip microcomputers. For these computers, the logic analyzer can still be used to record the digital signals at the microcomputer I/O ports. The advantages of the logic analyzer are:

- Very high bandwidth recording (100 MHz to 1 GHz)
- Many channels (16 to 132 inputs)
- Flexible triggering and clocking mechanisms
- Personality modules, which assist in interpreting the data

```
// bang-bang controller in C
void main(void) { unsigned char T;
   DDRA=0;    // Port A is sensor
   DDRB=0xFF; // Port B is heater
   while(1){
      T=PORTA; // read temperature
      if(T>27)
         PORTB=0;     // too hot
      else
         if(T<24)
            PORTB=1; // too cold
   }
}
```

```
\ bang-bang controller in FORTH
: main ( - )
   DDRA 0 !    \ Port A is senso
   DDRB 0xFF ! \ Port B is heate
   begin
     PORTA @  \ read temperature
     dup 27 > if
        PORTB 0 !      \ too hot
     else
        dup 24 < if
           PORTB 1 !   \ too col
        then
then drop 0 until ;
```

Figure 6. Bang-bang controllers implemented in C and FORTH, showing that both languages have well-defined modular control structures and make use of local variables on the stack.

Simulation. The next technological advancement which has greatly affected the manner in which embedded systems are developed is simulation. Because of the high cost and long times required to create hardware prototypes, many preliminary feasibility designs are now performed using hardware/software simulations. A simulator is a software application which models the behavior of the hardware/software system. If both the external hardware and software program are simulated together, even although the simulated time is slower than the actual time, the real-time hardware software interactions can be studied.

In-Circuit Emulator. Once the design is committed to hardware, the debugging tasks become more difficult. One simple approach, mentioned earlier, is to use a single-board computer which behaves similarly to the single-chip. Another approach is to use an in-circuit emulator. An in-circuit emulator (ICE) is a complex digital hardware device which emulates (behaves in a similar manner to) the I/O pins of the microcomputer in real time. The emulator is usually connected to a personal computer, so that emulated memory, I/O ports, and registers can be loaded and observed. Figure 7 shows that to use an emulator we first remove the microcomputer chip from the circuit, then attach the emulator pod into the socket where the microcomputer chip used to be.

Background Debug Module. The only disadvantage of the in-circuit emulator is its cost. To provide some of the benefits of this high-priced debugging equipment, some microcomputers have a background debug module (BDM). The BDM hardware exists on the microcomputer chip itself and communicates with the debugging personal computer via a dedicated 2- or 3-wire serial interface. Although not as flexible as an ICE, the BDM can provide the ability to observe software execution in real time, the ability to set breakpoints, the ability to stop the computer, and the ability to read and write registers, I/O ports, and memory.

Segmentation. Segmentation is when you group together in physical memory information which has similar logical properties. Because the embedded system does not load programs off disk when started, segmentation is an extremely important issue for these systems. Typical software segments include global variables, local variables, fixed constants, and machine instructions. For single-chip implementations, we store different types of information into the three types of memory:

1. RAM is volatile and has random and fast access
2. EEPROM is nonvolatile and can be easily erased and reprogrammed
3. ROM is nonvolatile but can be programmed only once

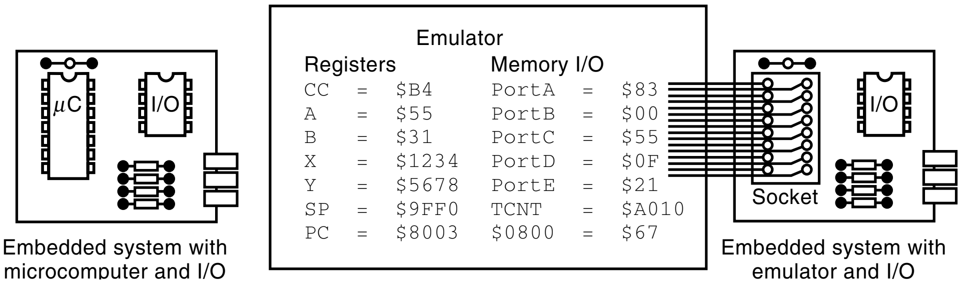
In an embedded application, we usually put structures which must be changed during execution in RAM. Examples include recorded data, parameters passed to subroutines, global and local variables. We place fixed constants in EEPROM because the information remains when the power is removed, but can be reprogrammed at a later time. Examples of fixed constants include translation tables, security codes, calibration data, and configuration parameters. We place machine instructions, interrupt vectors, and the reset vector in ROM because this information is stored once and will not need to be reprogrammed in the future.

Real-Time Systems. The microcomputer typically responds to external events with an appropriate software action. The time between the external event and the software action is defined as the latency. If we can guarantee an upper bound on the latency, we characterize the system as real time, or hard real time. If the system allows one software task to have priority over the others, then we describe it as soft real time. Since most real-time systems utilize interrupts to handle critical events, we can calculate the upper bound on the latency as the sum of three components: (1) maximum time the software executes with interrupts disabled (e.g., other interrupt handlers, critical code); (2) the time for the processor to service the interrupt (saving registers on stack, fetching the interrupt vector); and (3) software delays in the interrupt handler before the appropriate software action is performed. Examples of events which sometimes require real-time processing include:

- New input data ready to when the software reads the new input
- Output device is idle to when the software gives it more data
- An alarm condition occurs until the time the alarm is processed

Sometimes the software must respond to internal events. A large class of real-time systems involve performing software tasks on a fixed and regular rate. For these systems, we employ a periodic interrupt which will generate requests at fixed intervals. The microcomputer clock guarantees that the interrupt request is made exactly on time, but the software response (latency) may occur later. Examples of real-time systems which utilize periodic interrupts include:

- Data acquisition systems, where the software executes at the sampling rate
- Control systems, where the software executes at the controller rate


μC
I/O
Embedded system with microcomputer and I/O
Emulator
Socket
I/O
Embedded system with emulator and I/O

```
Registers          Memory I/O
CC  =  $B4         PortA  =  $83
A   =  $55         PortB  =  $00
B   =  $31         PortC  =  $55
X   =  $1234       PortD  =  $0F
Y   =  $5678       PortE  =  $21
SP  =  $9FF0       TCNT   =  $A010
PC  =  $8003       $0800  =  $67
```

Figure 7. To use an in-circuit emulator, remove the microcomputer chip from the embedded system, and place the emulator connector into the socket.

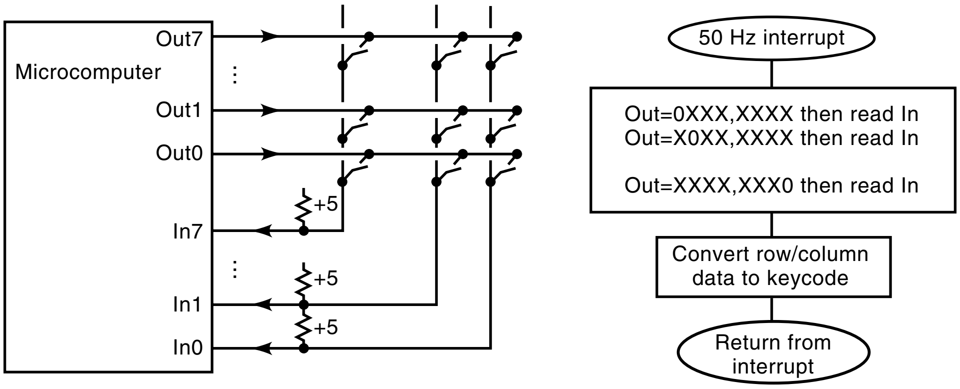

Microcomputer
Out7
Out1
Out0
In7
In1
In0
+5
+5
+5
50 Hz interrupt
Out=0XXX,XXXX then read In
Out=X0XX,XXXX then read In
Out=XXXX,XXX0 then read In
Convert row/column data to keycode
Return from interrupt

Figure 8. The matrix-scanned keyboard allows many keys to be interfaced using a small number of I/O pins.

- Time-of-day clocks, where the software maintains the date and time

MICROCOMPUTER INTERFACING AND APPLICATIONS

Keyboard Inputs

Individual buttons and switches can be interfaced to a microcomputer input port simply by converting the on/off resistance to a digital logic signal with a pull-up resistor. When many keys are to be interfaced, it is efficient to combine them in a matrix configuration. As shown in Fig. 8, 64 keys can be constructed as an 8 by 8 matrix. To interface the keyboard, we connect the rows to open collector (or open drain) microcomputer outputs, and the columns to microcomputer inputs. Open collector means the output will be low if the software writes a zero to the output port, but will float (high impedance) if the software writes a one. Pull-up resistors on the inputs will guarantee the column signals will be high if no key is touched in the selected row. The software scans the key matrix by driving one row at a time to zero, while the other rows are floating. If there is a key touched in the selected row, then the corresponding column signal will be zero. Most switches will bounce on/off for about 10 ms to 20 ms when touched or released. The software must read the switch position multiple times over a 20 ms time period to guarantee a reliable reading. One simple software method to use a periodic interrupt (with a rate slower than the bounce time) to scan the keyboard. In this way, the software will properly detect single key touches. One disadvantage of the matrix-scanned keyboard is the fact that three keys simultaneously pressed sometimes “looks” like four keys are pressed.

Finite State Machine Controller

To illustrate the concepts of programmable logic and software segmentation, consider the simple traffic light controller illustrated in Fig. 9. The finite state machine (FSM) has two inputs from sensors in the road which identify the presence of cars. There are six outputs, red/yellow/green for the north/south road and red/yellow/green for the east/west road. In this FSM, each state has a 6 bit output value, a time to wait in that state, and four next states, depending on if the input is 00 (no cars), 01 (car on the north/south road), 10 (car on the east/west road), or 11 (cars on both roads).

In the software implementation, presented in Fig. 10, the following three functions are called but not defined: *InitializeHardware();* is called once at the beginning to initialize the hardware. The function *Lights()* outputs a 6 bit value to the lights. The function *Sensor()* returns a 2 bit value from the car sensors. The software implementation for this system exhibits the three classic segments. Since the global variable *Pt* and the local variable *Input* have values which change during execution, they must be defined in RAM. The finite state machine data structure, *fsm[4],* will be defined in EEPROM, and the program *main()* and its subroutines *InitializeHardware(); Lights()* and *Sensor()* will be stored in ROM. You should be able to make minor modifications to the finite state machine (e.g., add/delete states, change input/output values) by changing the linked list data structure in EEPROM without modifying the assembly language controller in ROM.

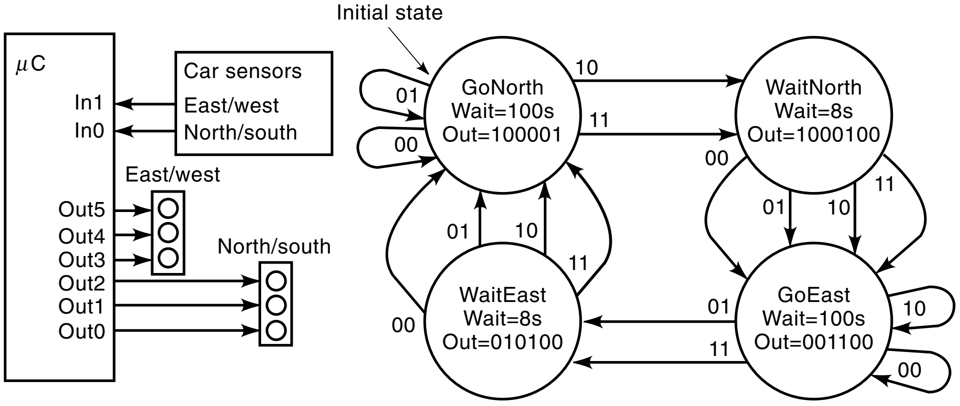


Figure 9. A simple traffic controller has two inputs and six outputs, and is implemented with a finite state machine.

```
struct State
{   unsigned char Out;      /* 6 bit Output                    */
    unsigned char Time;     /* Time to wait in seconds         */
    struct State *Next[4];} /* Next state if input=00,01,10,11 */
typedef struct State StateType;
typedef StateType * StatePtr;
StatePtr Pt;  /* Current State                                         RAM */
#define GoNorth &fsm[0]
#define WaitNorth &fsm[1]
#define GoEast &fsm[2]
#define WaitEast &fsm[3]
StateType fsm[4]={                                     /*             EEPROM*/
   {0x21,100,{GoNorth, GoNorth,WaitNorth,WaitNorth}},  /* GoNorth     EEPROM*/
   {0x22,  8,{ GoEast,  GoEast,   GoEast,   GoEast}},  /* WaitNorth   EEPROM*/
   {0x0C,100,{ GoEast,WaitEast,   GoEast, WaitEast}},  /* GoEast      EEPROM*/
   {0x0C,100,{GoNorth, GoNorth,  GoNorth,  GoNorth}}}; /* WaitEast    EEPROM*/
void Main(void){                                       /*                ROM*/
   unsigned char Input;                                /*                RAM*/
   Pt=GoNorth;           /* Initial State                                ROM*/
   InitializeHardware(); /* Set direction registers, clock               ROM*/
   while(1){                                           /*                ROM*/
      Lights(Pt->Out);   /* Perform output for this state                ROM*/
      Wait(Pt->Time);    /* Time to wait in this state                   ROM*/
      Input=Sensor();    /* Input=00 01 10 or 11                         ROM*/
      Pt=Pt->Next[Input];}};                           /*                ROM*/
```

Figure 10. C implementation of the finite state machine and controller.

Two advantages of segmentation are illustrated in this example. First, by placing the machine instructions in ROM, the software will begin execution when power is applied. Second, small modifications/upgrades/options to the finite state machine can be made by reprogramming the EEPROM without throwing the chip away. The RAM contains temporary information which is lost when the power is shut off.

Current-Activated Output Devices

Many external devices used in embedded systems activate with a current, and deactivate when no current is supplied. Examples of such devices are listed in Table 2. The control element describes the effective component through which the activating current is passed. dc motors which are controlled with a pulse width modulated (PWM) signal also fall into this category and are interfaced using circuits identical to the EM relay or solenoid. Figure 11 illustrates the similarities between the interface electronics for these devices.

The diode-based devices (LED, optosensor, optical isolation, solid-state relay) require a current-limiting resistor. The value of the resistor determines the voltage (V_d), current (I_d) operating point. The coil-based devices (EM relay, solenoid, motor) require a snubber diode to eliminate the large back EMF (over 200 V) that develops when the current is turned off. The back EMF is generated when the large dI/dt occurs across the inductance of the coil. The microcomputer output pins do not usually have a large enough I_{OL} to drive these devices directly, so we can use an open collector gate (like the 7405, 7406, 75492, 75451, or NPN transistors) to sink current to ground or use an open emitter gate (like the 75491 or PNP transistors) to source current from the power supply. Darlington switches like the ULN-2061 through ULN-2077 can be configured as either current sinks (open collector) or sources (open emitter). Table 3 provides the output low currents for some typical open collector devices. We need to select a device with an I_{OL} larger than the current required by the control element.

Stepper Motors

The unipolar stepper motor is controlled by passing current through four coils (labeled as B′ B A′ A in Fig. 12) exactly two at a time. There are five or six wires on a unipolar stepper motor. If we connect four open collector drivers to the four coils, the computer outputs the sequence 1010, 1001, 0101, 0110 to spin the motor. The software makes one change (e.g., change from 1001 to 0101) to affect one step. The software repeats the entire sequence over and over at regular time intervals between changes to make the motor spin at a constant rate. Some stepper motors will move on half-steps by outputting the sequence 1010, 1000, 1001, 0001, 0101, 0100, 0110, 0010. Assuming the motor torque is large enough to overcome the mechanical resistance (load on the shaft), each

Table 2. Output Devices Which Can Be Controlled by an Open Collector Driver

Device	Control Element	Definition	Applications
LED	Diode	Emits light	Indicator light, displays
EM relay	Resistor + inductor coil	μC-controlled switch	Lights, heaters, motors, fans
Solid-state relay	Diode	μC-controlled switch	Lights, heaters, motors, fans
Solenoid	Resistor + inductor coil	Short binary movements	Locks, industrial machines

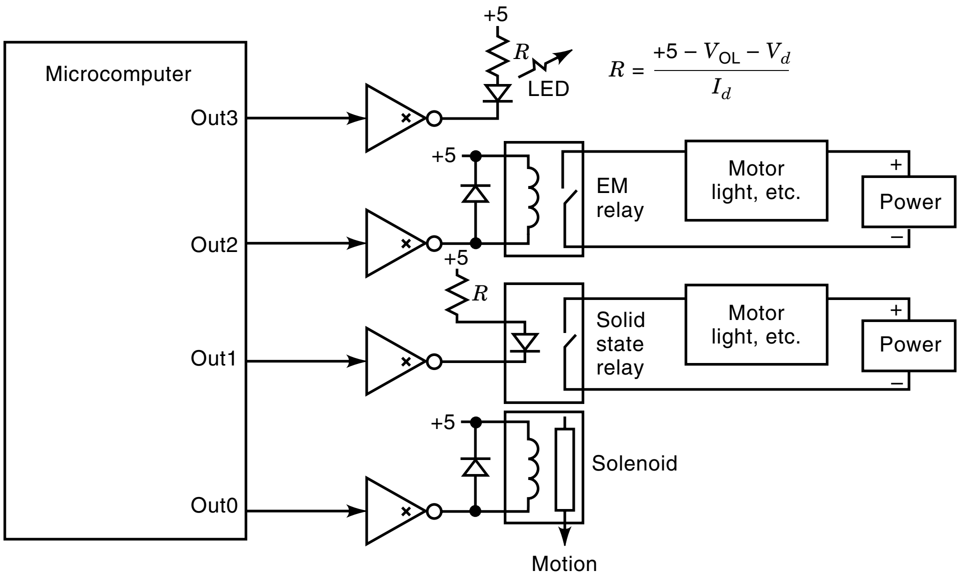


Figure 11. Many output devices are activated by passing a current through their control elements.

output change causes the motor to step a predefined angle. One of the key parameters which determine whether the motor will slip (a computer change without the shaft moving) is the jerk, which is the derivative of the acceleration (i.e., third derivative of the shaft position). Software algorithms which minimize jerk are less likely to cause a motor slip. If the computer outputs the sequence in the opposite order, the motor spins in the other direction. A bipolar stepper motor has only two coils (and four wires.) Current always passes through both coils, and the computer controls a bipolar stepper by reversing the direction of the currents. If the computer generates the sequence (positive, positive) (negative, positive) (negative, negative) (positive, negative), the motor will spin. A circular linked list data structure is a convenient software implementation which guarantees the proper motor sequence is maintained.

Microcomputer-Based Control System

Basic Principles. A control system, shown in Fig. 13, is a collection of mechanical and electrical devices connected for the purpose of commanding, directing, or regulating a physical plant. The real state variables are the actual properties of the physical plant that are to be controlled. The goal of the sensor and data-acquisition system is to estimate the state variables. Any differences between the estimated state variables and the real state variables will translate directly into controller errors. A closed-loop control system uses the output of the state estimator in a feedback loop to drive the errors to zero. The control system compares these estimated state variables, $X'(t)$, to the desired state variables, $X^*(t)$, in order to decide appropriate action, $U(t)$. The actuator is a transducer which converts the control system commands, $U(t)$, into driving forces, $V(t)$, which are applied the physical plant. The goal of the control system is to drive $X(t)$ to equal $X^*(t)$. If we define the error as the difference between the desired and estimated state variable:

$$E(t) = X^*(t) - X'(t) \tag{1}$$

then the control system will attempt to drive $E(t)$ to zero. In general control theory, $\boldsymbol{X}(t)$, $\boldsymbol{X}'(t)$, $\boldsymbol{X}^*(t)$, $\boldsymbol{U}(t)$, $\boldsymbol{V}(t)$, and $\boldsymbol{E}(t)$ refer to vectors (multiple parameters), but the example in this article controls only a single parameter. We usually evaluate the effectiveness of a control system by determining three properties: (1) steady-state controller error, (2) transient response, and (3) stability. The steady-state controller error is the average value of $E(t)$. The transient response is how long does the system take to reach 99% of the final output after X^* is changed. A system is stable if steady-state (smooth constant output) is achieved. An unstable system may oscillate.

Table 3. Output Low Voltages and Output Low Currents Illustrate the Spectrum of Interface Devices Capable of Sinking Current

Family	Example	V_{OL}	I_{OL}
Standard TTL	7405	0.4 V	16 mA
Schottky TTL	74S05	0.5 V	20 mA
Low-power Schottky TTL	74LS05	0.5 V	8 mA
High-speed CMOS	74HC05	0.33 V	4 mA
High-voltage output TTL	7406	0.7 V	40 mA
Silicon monolithic IC	75492	0.9 V	250 mA
Silicon monolithic IC	75451 to 75454	0.5 V	300 mA
Darlington switch	ULN-2074	1.4 V	1.25 A
MOSFET	IRF-540	Varies	28 A

Pulse Width Modulation. Many embedded systems must generate output pulses with specific pulse widths. The internal microcomputer clock is used to guarantee the timing accuracy of these outputs. Many microcomputers have built-in hardware which facilitate the generation of pulses. One classic example is the pulse-width modulated motor controller. The motor is turned on and off at a fixed frequency (see the *Out* signal in Fig. 14). The value of this frequency is chosen to be too fast for the motor to respond to the individual on/off signals. Rather, the motor responds to the average. The computer controls the power to the motor by varying the pulse width or duty cycle of the wave. The IRF540 MOSFET can sink up to 28 A. To implement Pulse Width Modulation

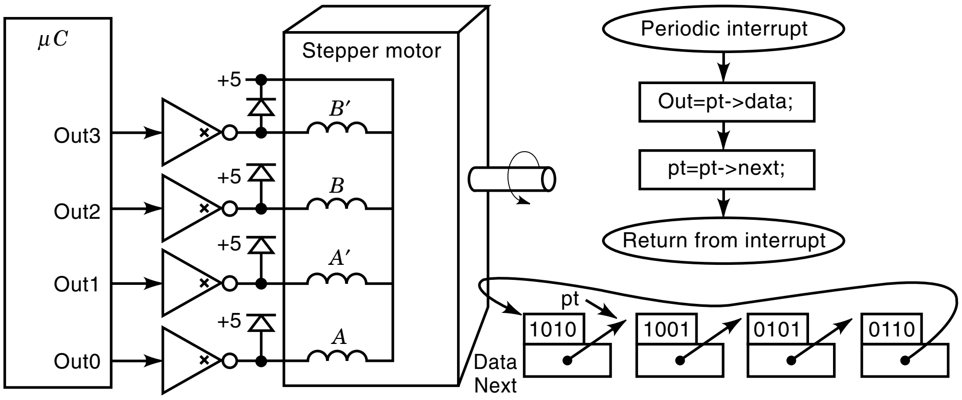


Figure 12. A unipolar stepper motor has four coils, which are activated using open collector drivers.

(PWM), the computer (either with the built-in hardware or the software) uses a clock. The clock is a simple integer counter which is incremented at a regular rate. The Out signal is set high for time T_h then set low for time T_l. Since the frequency of *Out* is to be fixed, $(T_h + T_l)$ remains constant, but the duty cycle $[T_h/(T_h + T_l)]$ is varied. The precision of this PWM system is defined to be the number of distinguishable duty cycles that can be generated. Let n and m be integer numbers representing the number of clock counts the *Out* signal is high and low, respectively. We can express the duty cycle as $n/(n + m)$. Theoretically, the precision should be $n + m$, but practically the value may be limited by the speed of the interface electronics.

Period Measurement. In order to sense the motor speed, a tachometer can be used. The ac amplitude and frequency of the tachometer output both depend on the shaft speed. It is usually more convenient to convert the ac signal into a digital signal (*In* shown in the Fig. 14) and measure the period. Again, many microcomputers have built-in hardware which facilitate the period measurement. To implement period measurement the computer (either with the built-in hardware or the software) uses a clock. Period measurement simply records the time (value of the clock) of two successive rising edges on the input and calculates the time difference. The period measurement resolution is defined to be the smallest difference in period which can be reliably measured. Theoretically, the period measurement resolution should be the clock period, but practically the value may be limited by noise in the interface electronics.

Control Algorithms

Incremental Control. There are three common approaches to designing the software for the control system. The simplest approach to the closed-loop control system uses incremental control, as shown in Fig. 15. In this motor control example, the actuator command, U, is the duty cycle of the pulse-width modulated system. An incremental control algorithm simply adds or subtracts a constant from U, depending on the sign of the error. To add hysteresis to the incremental controller, we define two thresholds, X_H X_L, at values just above and below the desired speed, X^*. In other words, if $X' < X_L$ (motor is spinning too slow) then U is incremented and if $X' > X_H$ (motor is spinning too fast), then U is decremented. It is important to choose the proper rate at which the incremental control software is executed. If it is executed too many times per second, then the actuator will saturate resulting in a bang-bang system like Fig. 6. If it is not executed often enough, then the system will not respond quickly to changes in the physical plant or changes in X^*.

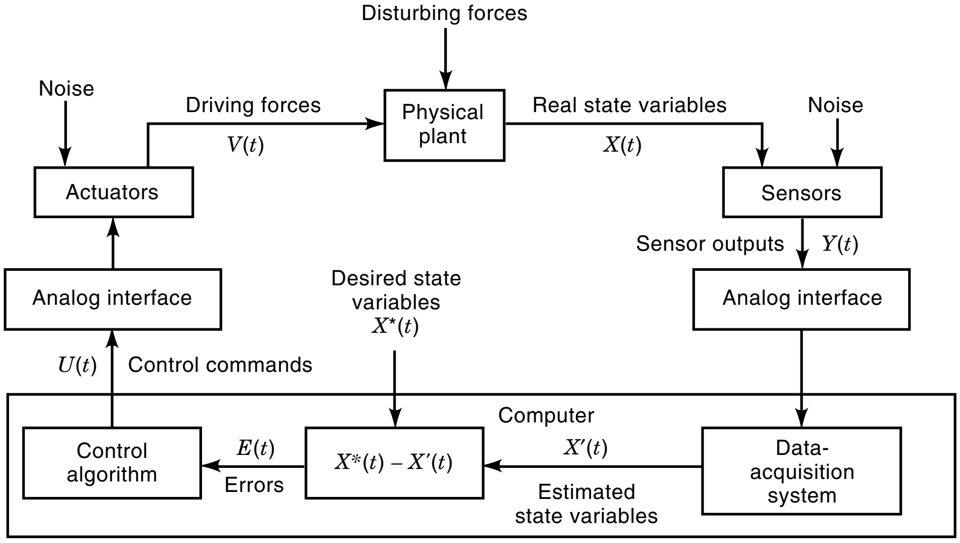


Figure 13. The block diagram of a closed-loop control system implemented with an embedded computer shows that the computer: (1) estimates the state variable, (2) compares it with the desired values, then (3) generates control commands which drive the physical plant to the desired state.

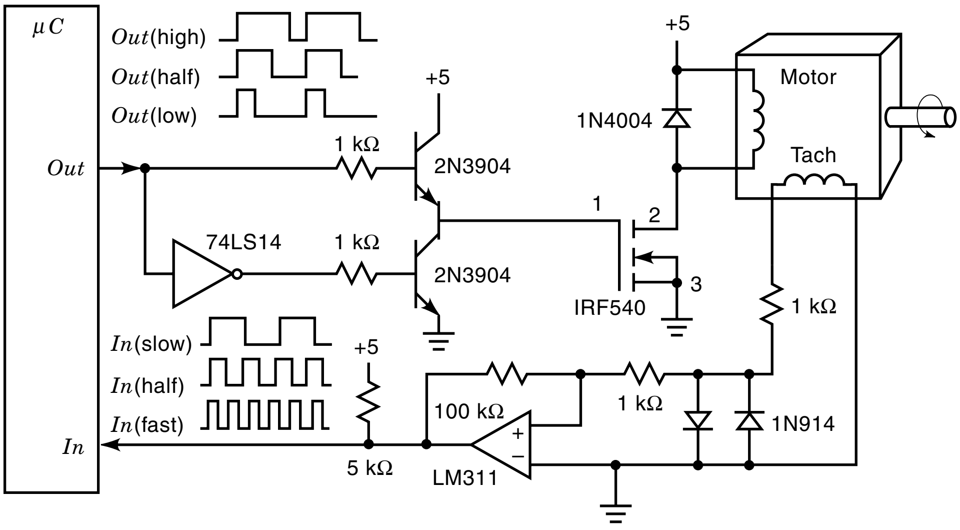


Figure 14. A dc motor can be controlled by varying the duty cycle, and the computer can sense the shaft speed by measuring the frequency or period from the tachometer.

Proportional Integral Derivative (PID) Control. The second approach, called proportional integral derivative, uses linear differential equations. We can write a linear differential equation showing the three components of a PID controller.

$$U(t) = K_P E(t) + K_I \int_0^t E(\tau)\,d\tau + K_D \frac{dE(t)}{dt} \tag{2}$$

To simplify the PID controller, we break the controller equation into separate proportion, integral and derivative terms, where $P(t)$, $I(t)$ and $D(t)$ are the proportional, integral, and derivative components, respectively. In order to implement the control system with the microcomputer, it is imperative that the digital equations be executed on a regular and periodic rate (every Δt). The relationship between the real time, t, and the discrete time, n, is simply $t = n\ \Delta t$. If the sampling rate varies, then controller errors will occur. The software algorithm begins with $E(n) = X'(n) - X^*$. The proportional term makes the actuator output linearly related to the error. Using a proportional term creates a control system which applies more energy to the plant when the error is large. To implement the proportional term we simply convert the above equation into discrete time.

$$P(n) = K_P \cdot E(n) \tag{3}$$

The integral term makes the actuator output related to the integral of the error. Using an integral term often will improve the steady-state error of the control system. If a small error accumulates for a long time, this term can get large. Some control systems put upper and lower bounds on this term, called anti-reset-windup, to prevent it from dominating the other terms. The implementation of the integral term requires the use of a discrete integral or sum. If $I(n)$ is the current control output, and $I(n - 1)$ is the previous calculation, the integral term is simply

$$I(n) = K_I \cdot \sum_{1}^{n} [E(n) \cdot \Delta t] = I(n-1) + K_I \cdot E(n) \cdot \Delta t \tag{4}$$

The derivative term makes the actuator output related to the derivative of the error. This term is usually combined with either the proportional and/or integral term to improve the transient response of the control system. The proper value of K_D will provide for a quick response to changes in either the set point or loads on the physical plant. An incorrect value may create an overdamped (very slow response) or an underdamped (unstable oscillations) response. There are a couple of ways to implement the discrete time derivative. The simple approach is

$$D(n) = K_D \cdot \frac{E(n) - E(n-1)}{\Delta t} \tag{5}$$

In practice, this first-order equation is quite susceptible to noise. In most practical control systems, the derivative is cal-

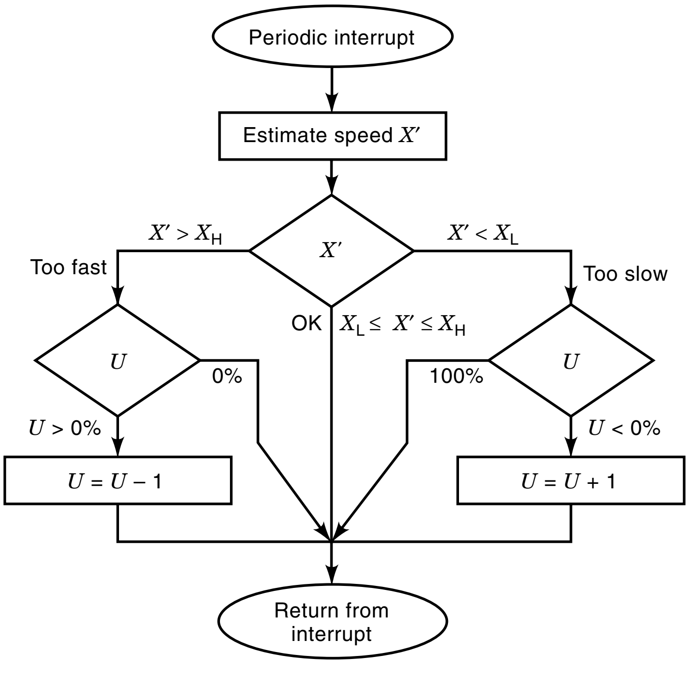


Figure 15. An incremental controller simply adds or subtracts a constant to the actuator control, depending on whether the motor is too fast or too slow.

culated using a higher-order equation like

$$D(n) = K_D \cdot \frac{E(n) + 3E(n-1) - 3E(n-2) - E(n-3)}{6\Delta t} \qquad (6)$$

The PID controller software is also implemented with a periodic interrupt every Δt. The interrupt handler first estimates the state variable, $X'(n)$. Finally, the next actuator output is calculated by combining the three terms.

$$U(n) = P(n) + I(n) + D(n) \qquad (7)$$

Fuzzy Logic Control. The third approach uses fuzzy logic to control the physical plant. Fuzzy logic can be much simpler than PID. It will require less memory and execute faster. When complete knowledge about the physical plant is known, then a good PID controller can be developed. That is, if you can describe the physical plant with a linear system of differential equations, an optimal PID control system can be developed. Since the fuzzy logic control is more robust (still works even if the parameter constants are not optimal), then the fuzzy logic approach can be used when complete knowledge about the plant is not known or can change dynamically. Choosing the proper PID parameters requires knowledge about the plant. The fuzzy logic approach is more intuitive, following more closely to the way a "human" would control the system. If there is no set of differential equations which describe the physical plant, but there exists expert knowledge (human intuition) on how it works, then a fuzzy system can be developed. It is easy to modify an existing fuzzy control system into a new problem. So if the framework exists, rapid prototyping is possible. The approach to fuzzy design can be summarized as

- The physical plant has real state variables (like speed, position, temperature, etc.).
- The data-acquisition system estimates the state variables.
- The preprocessor calculates relevant parameters, called crisp inputs.
- Fuzzification will convert crisp inputs into input fuzzy membership sets.
- The fuzzy rules calculate output fuzzy membership sets.
- Defuzzification will convert output sets into crisp outputs.
- The postprocessor modifies crisp outputs into a more convenient format.
- The actuator system affects the physical plant based on these outputs.

The objective of this example is to design a fuzzy logic microcomputer-based dc motor controller for the above dc motor and tachometer. Our system has two control inputs and one control output. S^* is the desired motor speed, S' is the current estimated motor speed, and U is the duty cycle for the PWM output. In the fuzzy logic approach, we begin by considering how a "human" would control the motor. Assume your hand were on a joystick (or your foot on a gas pedal) and consider how you would adjust the joystick to maintain a constant speed. We select crisp inputs and outputs on which to base our control system. It is logical to look at the error and the change in speed when developing a control system. Our fuzzy logic system will have two crisp inputs. E is the error in motor speed, and D is the change in motor speed (acceleration).

$$E(n) = S^* - S'(n) \qquad (8)$$

$$D(n) = S'(n) + 3S'(n-1) - 3S'(n-2) - S'(n-3) \qquad (9)$$

Notice that if we perform the calculations of D on periodic intervals, then D will represent the derivative of S', dS'/dt. To control the actuator, we could simply choose a new duty cycle value U as the crisp output. Instead, we will select, ΔU which is the change in U, rather than U itself because it better mimics how a "human" would control it. Again, think about how you control the speed of your car when driving. You do not adjust the gas pedal to a certain position, but rather make small or large changes to its position in order to speed up or slow down. Similarly, when controlling the temperature of the water in the shower, you do not set the hot/cold controls to certain absolute positions. Again you make differential changes to affect the "actuator" in this control system. Our fuzzy logic system will have one crisp output. ΔU is the change in output:

$$U = U + \Delta U \qquad (10)$$

Next we introduce fuzzy membership sets which define the current state of the crisp inputs and outputs. Fuzzy membership sets are variables which have true/false values. The value of a fuzzy membership set ranges from definitely true (255) to definitely false (0). For example, if a fuzzy membership set has a value of 128, you are stating the condition is half way between true and false. For each membership set, it is important to assign a meaning or significance to it. The calculation of the input membership sets is called fuzzification. For this simple fuzzy controller, we will define six membership sets for the crisp inputs:

1. *Slow* will be true if the motor is spinning too slow.
2. *OK* will be true if the motor is spinning at the proper speed.
3. *Fast* will be true if the motor is spinning too fast.
4. *Up* will be true if the motor speed is getting larger.
5. *Constant* will be true if the motor speed is remaining the same.
6. *Down* will be true if the motor speed is getting smaller.

We will define three membership sets for the crisp output:

1. *Decrease* will be true if the motor speed should be decreased.
2. *Same* will be true if the motor speed should remain the same.
3. *Increase* will be true if the motor speed should be increased.

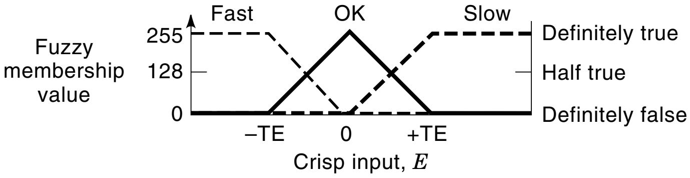


Figure 16. These three fuzzy membership functions convert the speed error into the fuzzy membership variables *Fast, OK,* and *Slow.*

The fuzzy membership sets are usually defined graphically (see Fig. 16), but software must be written to actually calculate each. In this implementation, we will define three adjustable thresholds, TE, TD, and TN. These are software constants and provide some fine-tuning to the control system. If TE is 20 and the error, E, is -5, the fuzzy logic will say that *Fast* is 64 (25% true), *OK* is 192 (75% true), and Slow is 0 (definitely false.) If TE is 20 and the error, E, is $+21$, the fuzzy logic will say that *Fast* is 0 (definitely false), *OK* is 0 (definitely false), and *Slow* is 255 (definitely true.) TE is defined to be the error above which we will definitely consider the speed to be too fast. Similarly, if the error is less than $-$TE, then the speed is definitely too slow.

In this fuzzy system, the input membership sets are continuous piecewise linear functions. Also, for each crisp input value, *Fast, OK, Slow* sum to 255. In general, it is possible for the fuzzy membership sets to be nonlinear or discontinuous, and the membership values do not have to sum to 255. The other three input fuzzy membership sets depend on the crisp input, D, as shown in Fig. 17. TD is defined to be the change in speed above which we will definitely consider the speed to be going up. Similarly, if the change in speed is less than $-$TD, then the speed is definitely going down.

The fuzzy rules specify the relationship between the input fuzzy membership sets and the output fuzzy membership values. It is in these rules that one builds the intuition of the controller. For example, if the error is within reasonable limits and the speed is constant, then the output should not be changed, [see Eq. (11)]. If the error is within reasonable limits and the speed is going up, then the output should be reduced to compensate for the increase in speed. If the motor is spinning too fast and the speed is constant, then the output should be reduced to compensate for the error. If the motor is spinning too fast and the speed is going up, then the output should be reduced to compensate for both the error and the increase in speed. When more than one rule applies to an output membership set, then we can combine the rules using the *or* function.

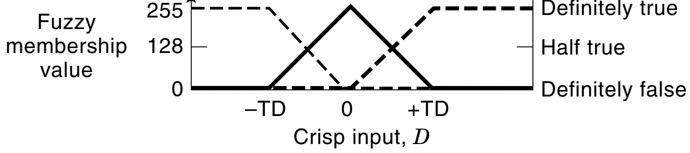


Figure 17. These three fuzzy membership functions convert the acceleration into the fuzzy membership variables *Down, Constant,* and *Up.*

$$\text{Same} = \mathit{OK} \text{ and } \mathit{Constant} \tag{11}$$

$$\mathit{Decrease} = (\mathit{OK} \text{ and } \mathit{Up}) \text{ or } (\mathit{Fast} \text{ and } \mathit{Constant}) \text{ or } (\mathit{Fast} \text{ and } \mathit{Up}) \tag{12}$$

$$\mathit{Increase} = (\mathit{OK} \text{ and } \mathit{Down}) \text{ or } (\mathit{Slow} \text{ and } \mathit{Constant}) \text{ or } (\mathit{Slow} \text{ and } \mathit{Down}) \tag{13}$$

In fuzzy logic, the *and* operation is performed by taking the minimum and the *or* operation is the maximum. The calculation of the crisp outputs is called defuzzification. The fuzzy membership sets for the output specifies the crisp output, ΔU, as a function of the membership value. For example, if the membership set *Decrease* were true (255) and the other two were false (0), then the change in output should be $-$TU (where TU is another software constant). If the membership set *Same* were true (255) and the other two were false (0), then the change in output should be 0. If the membership set *Increase* were true (255) and the other two were false (0), then the change in output should be $+$TU. In general, we calculate the crisp output as the weighted average of the fuzzy membership sets:

$$\Delta U = [\mathit{Decrease} \cdot (-\text{TU}) + \mathit{Same} \cdot 0 + \mathit{Increase} \cdot \text{TU}]/(\mathit{Decrease} + \mathit{Same} + \mathit{Increase}) \tag{14}$$

A good C compiler will promote the calculations to 16 bits, and perform the calculation using 16 bit signed math, which will eliminate overflow on intermediate terms. The output, ΔU, will be bounded in between $-$TU and $+$TU. The Motorola 6812 has assembly language instructions which greatly enhance the static and dynamic efficiency of a fuzzy logic implementation.

Remote or Distributed Communication

Many embedded systems require the communication of command or data information to other modules at either a near or a remote location. We will begin our discussion with communication with devices within the same room, as presented in Fig. 18. The simplest approach here is to use three or two wires and implement a full duplex (data in both directions at the same time) or half duplex (data in both directions but only in one direction at a time) asynchronous serial channel. Half-duplex is popular because it is less expensive (two wires) and allows the addition of more devices on the channel without change to the existing nodes. If the distances are short, half-duplex can be implemented with simple open collector TTL-level logic. Many microcomputers have open collector modes on their serial ports, which allow a half-duplex network to be created without any external logic (although pull-up resistors are often used). Three factors will limit the implementation of this simple half-duplex network: (1) the number nodes on the network, (2) the distance between nodes; and (3) presence of corrupting noise. In these situations a half-duplex RS485 driver chip like the SP483 made by Sipex or Maxim can be used.

To transmit a byte to the other computers, the software activates the SP483 driver and outputs the frame. Since it is half-duplex the frame is also sent to the receiver of the computer which sent it. This echo can be checked to see if a collision occurred (two devices simultaneously outputting.) If more than two computers exist on the network, we usually

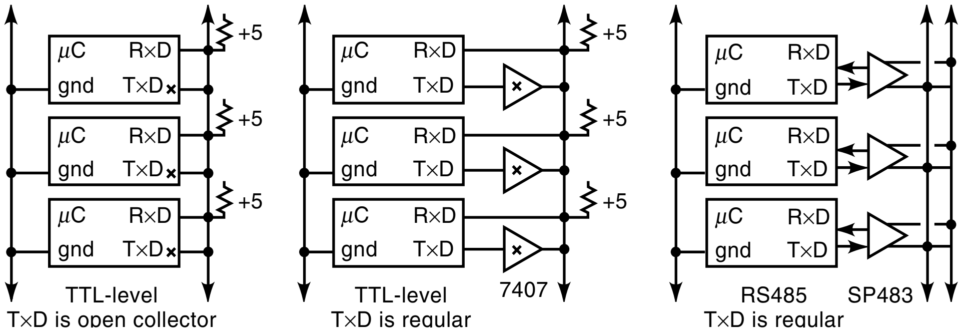


Figure 18. Three possibilities to implement a half-duplex network. The first network requires that the serial transmit output be open collector.

send address information first, so that the proper device receives the data.

Within the same room, infrared (IR) light pulses can be used to send and receive information. This is the technology used in the TV remote control. In order to eliminate background EM radiation from triggering a false communication, the signals are encoded as a series of long and short pulses which resemble bar codes.

There are a number of techniques available for communicating across longer distances. Within the same building the X-10 protocol can be used. The basic idea is to encode the binary stream of data as 120 kHz pulses and mix them onto the standard 120 V 60 Hz ac power line. For each binary one, a 120 kHz pulse is added at the zero crossing of the first half of the 60 Hz wave. A zero is encoded as a 120 kHz pulse in the second half of the 60 Hz wave. Because there are three phases within the ac power system, each pulse is repeated also 2.778 ms, and 5.556 ms after the zero crossing. It is decoded on the receiver end. X-10 has the flexibility of adding or expanding communication capabilities in a building without rewiring. The disadvantage of X-10 is that the bandwidth is fairly low (about 60 bits/s) when compared to other techniques. A typical X-10 message includes a 2 bit start code, a 4 bit house code, and a 5 bit number code requiring 11 power line cycles to transmit. A second technique for longer distances is RF modulation. The information is modulated on the transmitted RF, and demodulated at the receiver. Standard telephone modems and the internet can also be used to establish long-distance networks.

There are two approaches to synchronizing the multiple computers. In a master/slave system, one device is the master, which controls all the other slaves. The master defines the overall parameters which govern the functions of each slave and arbitrates requests for data and resources. This is the simplest approach but may require a high-bandwidth channel and a fast computer for the master. Collisions are unlikely in a master/slave system if the master can control access to the network.

The other approach is distributed communication. In this approach each computer is given certain local responsibilities and certain local resources. Communication across the network is required when data collected in one node must be shared with other nodes. A distributed approach will be successful on large problems which can be divided into multiple tasks that can run almost independently. As the interdependence of the tasks increase, so will the traffic on the network. Collision detection and recovery are required due to the asynchronous nature of the individual nodes.

Data-Acquisition Systems

Before designing a data-acquisition system (DAS) we must have a clear understanding of the system goals. We can classify system as a quantitative DAS, if the specifications can be defined explicitly in terms of desired range, resolution, precision, and frequencies of interest. If the specifications are more loosely defined, we classify it as a qualitative DAS. Examples of qualitative DAS include systems which mimic the human senses where the specifications are defined, using terms like "sounds good," "looks pretty," and "feels right." Other qualitative DAS involve the detection of events. In these systems, the specifications are expressed in terms of specificity and sensitivity. For binary detection systems like the presence/absence of a burglar or the presence/absence of cancer, we define a true positive (TP) when the condition exists (there is a burglar) and the system properly detects it (alarm rings). We define a false positive (FP) when the condition does not exist (there is no burglar) but the system thinks there is (alarm rings). A false negative (FN) occurs when the condition exists (there is a burglar) but the system does not think there is (alarm is silent). Sensitivity, TP/(TP + FN), is the fraction of properly detected events (burglar comes and alarm rings) over the total number of events (number of burglars). It is a measure of how well our system can detect an event. A sensitivity of 1 means you will not be robbed. Specificity, TP/(TP + FP) is the fraction of properly detected events (burglar comes and alarm rings) over the total number of detections (number of alarms.) It is a measure of how much we believe the system is correct when it says it has detected an event. A specificity of 1 means when the alarm rings, the police will arrest a burglar when they get there.

Figure 19 illustrates the basic components of a data-acquisition system. The transducer converts the physical signal into an electrical signal. The amplifier converts the weak transducer electrical signal into the range of the ADC (e.g., −10 V to +10 V). The analog filter removes unwanted frequency components within the signal. The analog filter is required to remove aliasing error caused by the ADC sampling. The analog multiplexer is used to select one signal from many sources. The sample and hold (S/H) is an analog latch used to keep the ADC input voltage constant during the ADC conversion. The clock is used to control the sampling process. Inherent in digital signal processing is the requirement that the ADC be sampled on a fixed time basis. The computer is used to save and process the digital data. A digital filter may be used to amplify or reject certain frequency components of the digitized signal.

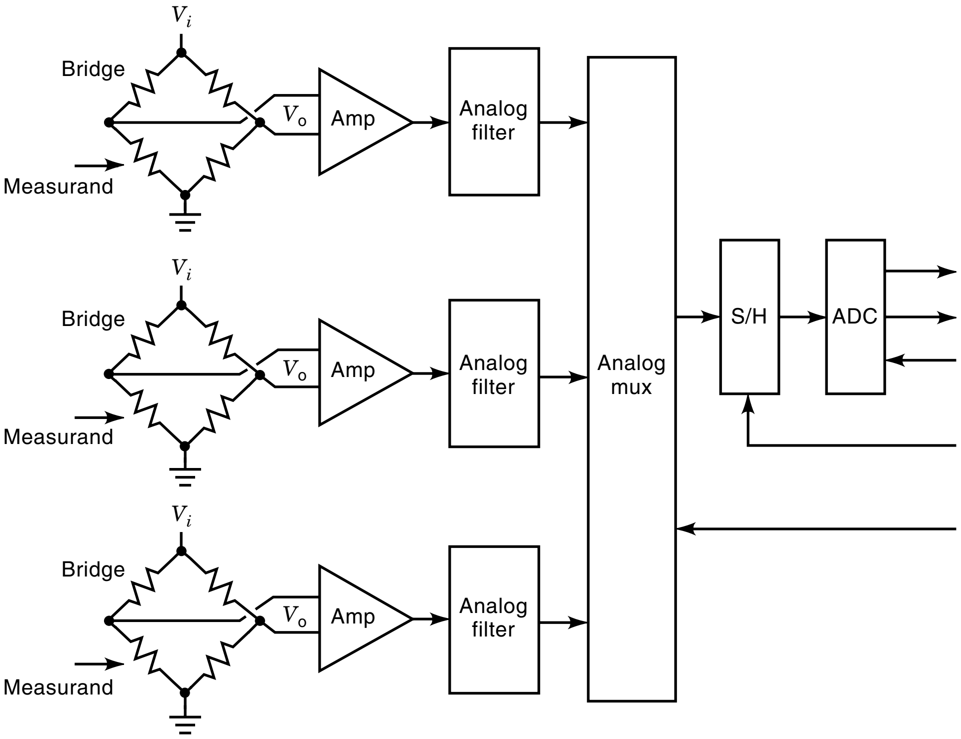


Figure 19. Block diagram of a multiple-channel data-acquisition system, where the transducer and bridge convert the measurands into electrical signals (V_o), the analog circuits amplify and filter the signals, and the multiplexer-ADC system converts the analog signals into digital numbers.

The first decision to make is the ADC precision. Whether we have a qualitative or quantitative DAS, we choose the number of bits in the ADC so as to achieve the desired system specification. For a quantitative DAS this is a simple task because the relationship between the ADC precision and the system measurement precision is obvious. For a qualitative DAS, we often employ experimental trials to evaluate the relationship between ADC bits and system performance.

The next decision is the sampling rate, f_s. The Nyquist Theorem states we can reliably represent, in digital form, a band-limited analog signal if we sample faster than twice the largest frequency that exists in the analog signal. For example, if an analog signal only has frequency components in the 0 Hz to 100 Hz range, then if we sample at a rate above 200 Hz, the entire signal can be reconstructed from the digital samples. One of the reasons for using an analog filter is to guarantee that the signal at the ADC input is band-limited. Violation of the Nyquist Theorem results in aliasing. Aliasing is the distortion of the digital signal which occurs when frequency components above $0.5\,f_s$ exist at the ADC input. These high-frequency components are frequency shifted or folded into the 0 to $0.5\,f_s$ range.

The purpose of the sample and hold module is to keep the analog input at the ADC fixed during conversion. We can evaluate the need for the S/H by multiplying the maximum slew rate (dV/dt) of the input signal by the time required by the ADC to convert. This product is the change in voltage which occurs during a conversion. If this change is larger than the ADC resolution, then a S/H should be used.

BIBLIOGRAPHY

1. H. M. Dietel and P. J. Dietel, *C++ How to Program,* Englewood Cliffs, NJ: Prentice-Hall, 1994.
2. R. H. Barnett, *The 8951 Family of Microcomputers,* Englewood Cliffs, NJ: Prentice-Hall, 1995.
3. Brodie, *Starting FORTH,* Englewood Cliffs, NJ: Prentice-Hall, 1987.
4. G. J. Lipovski, *Single- and Multiple-Chip Microcomputer Interfacing,* Englewood Cliffs, NJ: Prentice-Hall, 1988.
5. J. B. Peatman, *Design with Microcontrollers,* New York: McGraw-Hill, 1988.
6. J. B. Peatman, *Design with PIC Microcontrollers,* New York: McGraw-Hill, 1998.
7. C. H. Roth, *Fundamentals of Logic Design,* Boston, MA: West, 1992.
8. J. C. Skroder, *Using the M68HC11 Microcontroller,* Upper Saddle River, NJ: Prentice-Hall, 1997.
9. K. L. Short, *Embedded Microprocessor Systems Design,* Upper Saddle River, NJ: Prentice-Hall, 1998.
10. P. Spasov, *Microcontroller Technology The 68HC11,* Upper Saddle River, NJ: Prentice-Hall, 1996.
11. H. S. Stone, *Microcomputer Interfacing,* Reading, MA: Addison-Wesley, 1982.
12. R. J. Tocci, F. J. Abrosio, and L. P. Laskowski, *Microprocessors and Microcomputers,* Upper Saddle River, NJ: Prentice-Hall, 1997.
13. J. W. Valvano, *Real Time Embedded Systems,* Pacific Grove, CA: Brooks/Cole, 1999.
14. J. G. Webster (ed.), *Medical Instrumentation,* Application and Design, 3rd ed., New York: Wiley, 1998.
15. W. C. Wray and J. D. Greenfield, *Using Microprocessors and Microcomputers,* Englewood Cliffs, NJ: Prentice-Hall, 1994.

Reading List

L. Steckler (ed.), *Electronics Now,* Boulder, CO: Gernsback, 1993–current.

S. Ciarcia (ed.), *Circuit Cellar INK—The Computer Applications J.,* Vernon, CT: Circuit Cellar Inc., 1991–current.

JONATHAN W. VALVANO
University of Texas at Austin