CHAPTER 1

SYNCHRONOUS PROGRAMMING

Real-time computer systems must interact with the outside world on terms that are dictated by events taking place there. The computations that are done in response to those events must not only produce the correct results, but they must also produce those results at the right time. Unlike a real-time system, the success of a scientific or engineering computation is rarely related to when the result appears, although the user's patience and total computing expenses are related to the computation time. A further distinction in real-time computing is that the total computing environment consists of many semi-independent tasks that must be synchronized properly.

Many varieties of computers and systems qualify as "real-time." In this text, our concerns will focus on engineering systems in which there are interactions between a computer and some form of physical system. There are also often interactions with an operator. The physical system usually contains several measuring devices, which the computer must interrogate to get information, and several actuators, which receive signals from the computer to control their actions. Some systems have only one or the other of sensors or actuators, while most have both (Fig. 1.1). The computer (or computers) used can range from thumbnail size to room size (microprocessors to superminis), but the basic techniques for designing effective real-time systems are the same: careful conceptual design, systematic implementation, exhaustive validation, and thoughtful choice of software and hardware development tools. A major focus here will be on the use of high-level computing languages for implementation of real-time systems.

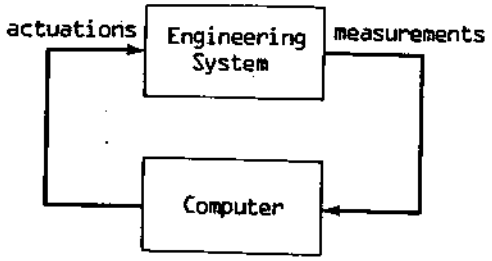

Figure 1.1

1.1 MOTOR SPEED CONTROL

We have chosen the control of electric motors as our theme. Motors are widely used and appear in so many different kinds of engineering systems that they cross virtually all disciplinary boundaries. When different methods of actuation and speed and position measurement are considered, motors also offer examples of situations that are typical of almost any real-time system. Motor systems are also easy and inexpensive to build in a laboratory, and so offer an excellent learning environment. On the other hand, the programs developed in the course of exploring the theme of motor control are generic to other control problems, and could be applied to many of them with little or no change.

A simple motor control system is shown schematically in Fig. 1.2. From the point of view of real-time system design, the simplicity of the job, even for this very simple-looking physical system, will depend on how much we demand of the computer. If the analog-to-digital (A/D) and digital-to-analog (D/A) converters can operate with little or no intervention from the computer, if the only interaction with the operator takes place at the beginning and end of an experiment, and if the algorithm chosen for computing the output signal to the power amplifier as a function of the measured motor speed depends only on the most recent measurement, then the real-time system will also be quite simple. With these restrictions, we can embark on our first example.

1.2 THE CONTROL ALGORITHM

At the heart of most real-time computation systems there are usually some key calculations. This could be a trend analysis of incoming data, spectral analysis for recognizing changes in system characteristics, generation of waveforms for system excitation, or, in this case, computation of the actuation signal on the basis of the measured motor velocity. Although these calculations are absolutely critical to proper system operation, the actual amount of program code devoted to them is usually embarrassingly small!

Control of motor speed is accomplished by increasing the voltage to the power amplifier if the speed is too low, and decreasing it if the speed is too high. A simple rule for doing this is to make the change in actuation voltage proportional to the velocity error, the difference between the actual velocity and the

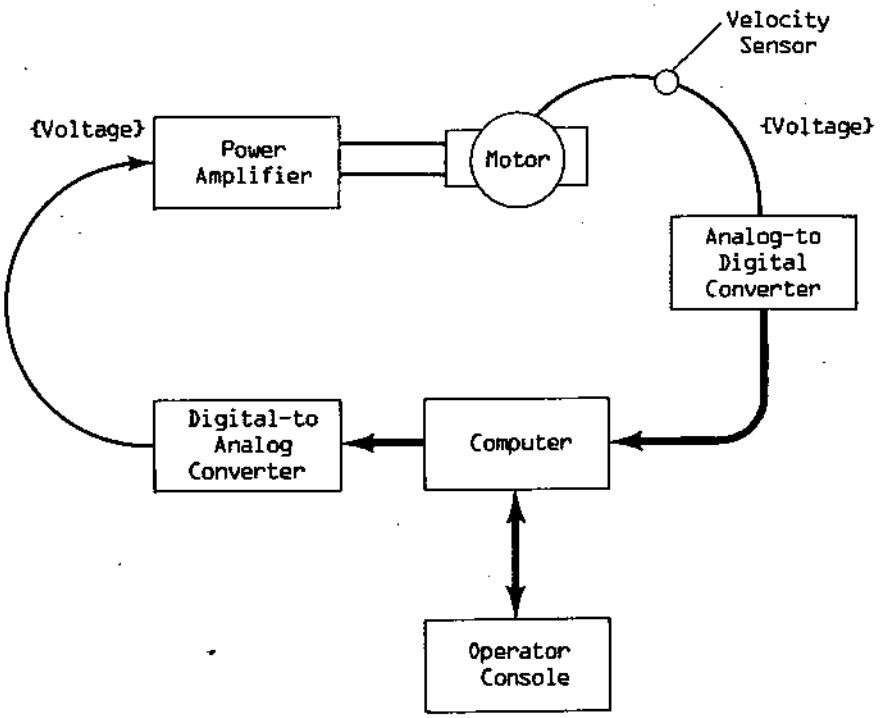

Figure 1.2

desired value

$$m = k_p \varepsilon + c \tag{1.1}$$

where m is the controller output, k_p is the proportionality constant ("gain"), ε is the error

$$\varepsilon = r - v_m \tag{1.2}$$

and c is a constant bias that is applied to the output voltage, often to compensate for steady loads such as gravity or friction. r is the desired ("reference") voltage, and v_m is the motor speed (scaled to the same units as r).

1.3 PROGRAM STRUCTURE

The equations expressing the control algorithm, when coded for computer implementation, must be embedded in a program that will interact properly with the system (motor) being controlled. If, in this case, the external environment is limited to a single motor, a relatively simple program structure emerges. Because the calculation of the controller output, m, does not depend either implicitly or ex-

plicitly on time, the strategy for the real-time portion of the program is to run the control calculation as often as possible. The basic real-time constraint in this case is whether the length of time required to complete one controller calculation cycle is short enough to maintain effective control. If it isn't, the only solution is to find some way to make it run faster, a more expensive processor, a better compiler, use of assembly language in key places, and so on.

Real-time programs consist of sets of semi-independent modules, often called *tasks*. Each of these tasks is a sequence of program instructions written in a standard programming language. To illustrate the structure of the tasks, we will use a form of "pseudocode," that is, free-form statements about actions to be taken, but organized to look very much like a program in one of the structured languages (C, Pascal, etc.). The convention we will use is to identify program blocks by indentation, with no further beginning or end of block marker. No formal syntax will be used for loops or conditionals; whatever form suits the need will be used. Blocks that consist of functionally grouped statements but do not require any formal blocking will be set off with blank lines. No GO-TO's will be used within tasks.

The motor control outlined above makes a good first example because its simple structure has only one task. That task has three parts: the beginning, in which the user interaction and initialization activity takes place, the middle part, which contains the real-time portion, and the final section, where performance reports are made (the final section can be omitted). The full program could be diagrammed as shown in Fig. 1.3, with each of the three sections represented as a program block. (The details of how these program structures are implemented with specific computers and compilers are given in the appendices in the form of fully commented programs, with additional explanatory material if necessary. These programs show the implementation for the final program structure selected for each of the problems.)

```
Get user input (gains, setpoint, number of iterations, ...)

Initialize I/O ports, internal control variables

For specified number of iterations:
        Read the velocity
        Compute the controller output
        Send output to the motor

Report results
```

Figure 1.3

This structure is simple and will work, but leaves no control to the operator once the parameters are set. Particularly in laboratory environments, it is very common to want to "play" with a system as a means of understanding its behavior and finding the best control parameters ("tuning" the gains). In that situation, the user likes to make one change at a time, see its effect, then make another change. An alternate structure to accommodate such repetitive activity would be to make all actions depend on a user command. These actions would include changing parameters, initializing, and initiating the actual control. This leads to the following, more complex, but more useful structure (Fig. 1.4).

```
Main:
        Repeat continuously:
                Get user command (single letter commands)
                Interpret and execute the command

Command Interpreter:
        Execute block that matches the single letter command:
                e       exit from program
                k       set controller gain
                s       set the setpoint (desired velocity)
                g       control for specified number of iterations
                i       initialize the system
                other   print an error message
```

Figure 1.4

The control module, initializing module, and results reporting module (which isn't included in the menu) now become subsidiary to the command interpreter. Otherwise, they look very much like the version in Fig. 1.3.

Further protocol is required to establish the way in which information can be passed to the various modules that are called by the command interpreter. Each module can do its own prompting and ask for the data from the user, but that usually leads to a lack of uniformity in the user interface, particularly if all the modules were not written by the same person or if they were written at different times. A simple solution to this is to put the information to be passed to the lower level program on the same line as the command itself. When the command interpreter calls a function, it also passes the remainder of the line to it as an argument. To specify the number of iterations for control, for example, the "g" command, the user would type:

```
g 50
```

As long as the protocol is similar for each of the commands, i.e., one or more numbers following the command letter, the user interface remains consistent. Likewise, the programming necessary to decode the additional numbers is not too difficult because almost all high-level computer languages provide for a format decoding that works on an internal buffer of characters.

1.4 DEBUGGING: SIMULATING REAL-TIME

Debugging of real-time programs is a difficult problem because the computer and the real world to which it is relating are operating asynchronously. This means that it is often not possible to duplicate the situations that are causing problems. This is in distinct contrast to debugging a computational program. In the latter case, all the action takes place in the computer, which is a strictly sequential machine, and, unique among engineering systems, will always duplicate its actions *exactly*. Although debugging is still a major problem, the ability to debug by repeatedly reproducing the same error is a boon!

We would like to take advantage of this property of computers as long as possible in debugging real-time problems. Extensive planning and program design remain the main bulwarks against excessive debugging time, but some amount of debugging always seems to be necessary. Before entering the real-time phase of operation, it would be nice to be reasonably sure that the computational part of the program is correct. This can be done by operating the program with another program (or program section) that simulates the real-time part of the system. The extra work involved in producing the simulation will pay off handsomely in overall productivity. It might also add some additional insight into the nature of the problem leading to improved solution methods.

At this stage, an advantage of using a high-level language is that with proper choice of the language and compiler, it becomes possible to do the simulation studies on one computer and then transport the program to a different computer for real-time operation when the simulation studies are complete. This could not be done efficiently with assembly language programs because the languages differ from one computer to another.

The structure of the real-time control module and the simulator are shown in Fig.1.5. The simulator has two functions. It has to increment "time," and it has to compute the change in the controlled system that would take place during that time. The interesting property of the simulated real-time environment is that it makes the control computer appear to have an infinitely fast computing speed. For example, in this system, the control program must first read the value of the velocity from the analog-to-digital converter, then compute the voltage to be sent to the digital-to-analog converter for transmission to the motor's power amplifier. All of that computation takes a finite amount of time. There is some delay between the time that the velocity measurement process is started and the time that the new voltage command signal gets to the amplifier. This introduces a

certain amount of delay into the system, and that delay can affect the quality of control. In the simulated system, however, "time," that is, simulated time, is suspended while the computation takes place. By breaking the process into finer divisions, some of these effects can be simulated also; however, simulated time will always stand still while the control computation is taking place.

```
Control:
        For specified number of iterations
                Read velocity instrument
                Compute controller output
                Send value to power amplifier
                Simulate one increment in time

Simulation:
        Compute motor velocity change for this time step
        Increment time
```

Figure 1.5

In most cases, since it is the computational validity that is being tested rather than whether the particular control algorithm will successfully control the motor, it is usually satisfactory at this stage to substitute a very simple model of the system under control. For the motor, the simplest useful model is of a pure rotational inertia with no friction, no compliance in the drive train, and no lag in the power amplifier. This can be expressed in differential equation form as:

$$\frac{dv}{dt} = \frac{T}{m} \tag{1.3}$$

where T is the torque applied to the motor and m is its mass (rotational inertia). A simple approximation of this equation for computer solution is:

$$v(t+\Delta t) = v(t) + (\frac{T}{m})\,\Delta t \tag{1.4}$$

If the control sample time is long, it may be necessary to use a Δt that is a fraction of the sample time and solve Eq. (1.4) several times to advance the solution one full sample time. Fig. 1.6 shows the program structure for solving the differential equation.

```
For n iterations (where n = T/Δt)
    v = v + (T * Δt ≠ m)
```

Figure 1.6

The emphasis in this model of real-time is on producing an environment in which the numerical validity of the program can be tested. Errors due to timing conflicts must be debugged in actual real-time operation. To accomplish that, the model of the control object (i.e., the motor) does not require a great deal of fidelity with the real thing. If the simulation were to be used to tune the control gains, for example, a more accurate model would have to be used. In most circumstances, it would be better to use a simulation language for that purpose.

1.5 INTEGER AND FLOATING POINT: REAL-TIME CONSIDERATIONS

In the discussion thus far, we have not made any reference to the way in which numbers are represented internally in the computer's memory, or to the way in which the basic arithmetical operations are carried out. These details are of critical importance to real-time programs.

There are two common ways that numbers can be dealt with in computers: as integers, or as floating point quantities. For computational purposes, the integer representation includes positive and negative values (with zero usually counted as positive), that is, signed integers. Floating point numbers, more properly called scientific notation, are represented in two parts, a mantissa and an exponent, both signed. The value of the number is the mantissa times the base of the number system raised to the power given by the exponent:

$$value = mantissa * base^{exponent} \tag{1.5}$$

For ordinary numbers, the base is 10. In internal computer representations, the base can be either 2 or 10, with 2 being more common in engineering applications. A normalized floating point number is one for which the magnitude of the mantissa is always within a range given by:

$$(base^{r-1}) \leq mantissa < base^{r} \tag{1.6}$$

For decimal numbers, for example, a normalized mantissa might be constrained to the range $0.1 \leq mantissa < 1$.

Both of these are finite precision representations. For the integers, the precision is expressed by the range of integer values that can be represented. Some integer examples are: a 3-bit signed integer can represent numbers from −4 to +3; an 8-bit integer can represent numbers from −128 to +127; a 16-bit integer can represent numbers from −32,768 to +32,767; a 32-bit signed integer can represent −2,147,483,648. For integers, the precision is expressed as maximum positive and maximum (absolute value) negative numbers. For floating point numbers, the precision limitations are expressed as a number of significant digits for the mantissa, and a maximum positive and negative integer value for the exponent.

By contrast, neither precision nor range is normally considered when expressing an engineering quantity. The normal assumption is that whatever

computing instrument is used, it will have sufficient precision and range to deal with the numbers involved. As a result, quantities in engineering units can have vastly different ranges in order to use common units. The floating point number representation falls closest to common usage. Using this representation, as long as the quantity does not fall into the extreme edge of the range of exponents available, the overall computing precision will be independent of the value. In most cases, there will be some round-off error associated with every calculation.

Integer calculations, on the other hand, pose much greater computing difficulty. The precision associated with an integer value depends on its magnitude. Small (absolute value) numbers have very low precision, measuring precision in an intuitive way as percentage change required to move from the current value to the next allowable value. For an integer of value 1, for example, it takes a 100% change to get to its nearest neighbors. All quantities used must therefore be scaled if integer representations are to be used. The internal value will be related to the actual value (in engineering units) by an arbitrary scale factor. The scaling process must compromise between two problems: (1) to maintain sufficient precision; and (2) to avoid exceeding the allowable range, even in intermediate results of calculations. For example, the controller output equation requires that the error be multiplied by the proportional gain and then added to the bias to get the controller output value. If all quantities are scaled for integer arithmetic, they will all have approximately the same range of allowable values. For many control applications, that will mean that the control gain will be a number near unity. That is precisely the range, however, where integer arithmetic has its least precision, and so will not allow for fine tuning the controller's gain. To avoid that, the gain can be represented as a ratio of numbers, so the calculation becomes:

$$output = (k_{num} * error) \ / \ k_{den} + bias \qquad (1.7)$$

The position of the parentheses in this calculation is not arbitrary; the parentheses control the computing order. The multiplication must be done before the division so that the round-off error will be minimized. If, for example, k_{num}, *error*, and k_{den} are all about the same magnitude, if the division were done first, the result would be near unity and all precision would be lost. On the other hand, the scaling of the problem must be done in such a way that the product, $k_{num} * error$, does not ever exceed the integer range. If it does, in most cases the result will have a very large error and, often, will have the opposite sign of the true product.

Floating point is clearly much easier to work with. Except for extreme cases, engineering units can be used directly with no problems of precision or range. The catch is that floating point is much slower in computing time than integer, as much as 100 times slower, or, alternatively, to get computing times for floating point even close to integer times (but usually still slower) requires the use of additional computing hardware that is quite expensive.

The suggested procedure for approaching the integer/floating point decision is to start in the simulated real-time mode using floating point. Because

scaling is almost never required for programs using floating point numeric representation, the validation and debugging can proceed without the need to worry about scaling. Because time is being simulated, the computing time is not a factor either. When this step is complete, a set of time trials can be made to find out the execution time for the key modules in the program (some compilers provide links to "profilers" that make it very easy to get these statistics). If the performance is within the system specifications, it may be possible to leave the floating point in place and proceed with the system development. If the performance is close to specification, it might be possible to identify one or two key modules and recode them for integer calculation, leaving the rest of the program intact. If, however, it appears that integer calculation will be necessary throughout the program, the simulated real-time system can be used to great advantage in doing the conversion.

The first step in the conversion is to decide what class of integer to use. All processors have a "most natural" word size, that is, the number of bits that can be processed in parallel. This determines the natural integer precision for that processor. For microprocessor systems, the most common are 8-bit and 16-bit word sizes. Eight-bit precision means that the integer number range is −128 to +127 (for a given word size, there are a total of 2^n possible numbers, where n is the word size). This range is inadequate for most control-type problems. Sixteen-bit precision implies a range of −32,768 to 32,767. This range is adequate for many control problems, but scaling must be done very carefully to preserve precision and avoid overflows. The next most common integer size is 32-bit, providing a range of approximately $\pm 2.15 \times 10^9$. This range is large enough to make the problem of scaling for specified precision while avoiding overflow relatively simple for most control tasks.

The use of a high-level language insulates the programmer from the details of how the arithmetic is implemented on a specific processor. Any processor can implement arithmetic operations with any desired precision level, at the cost of increased computational time. In making the choices, therefore, it is important to know the characteristics of the computer to know what precision level is likely to work well. Most processors implement some arithmetic operations at one level above their most natural level. Eight-bit processors have some 16-bit operations and 16-bit processors have some 32-bit operations. Most compilers allow at least limited 8-bit operations, and full 16- or 32-bit integer arithmetic.

Using a combination of the overall desired precision level and the characteristics of the processor in use (or the target processor if the simulation is being run on a computer different than the one that will actually be used for control), a first choice can be made for the integer arithmetic precision level. If possible, the choice should be the most natural and therefore most efficient integer mode of the target computer. Using a high-level language, changes can easily be made to other modes, but using a larger word size than the natural size will cause a significant computing time penalty. Before changing all of the program variable declarations, however, the original, floating point version of the program should

be "instrumented" with print statements to give the values of internal variables and intermediate computation results throughout the program. This might require breaking program statements into several parts in order to get at the intermediate results.

At this point, the simulated real-time program will be turned into a simulated integer arithmetic program. This is done by scaling all of the variables as if they were integers, but leaving them in floating point form. The program should then be run over as wide a range of operating conditions as possible, recording all of the data from the "print" statements that have been introduced. This data is used primarily to check that no overflows occur anywhere in the program, for any of its operating ranges. The overall results can be compared with the original floating point simulation program to make sure that overall precision requirements are being met.

The final step in the integer conversion is the actual substitution of new declarations for all the appropriate variables. With the integer mode simulation already done, there should be few further conversion problems. The "instrumented" version of the program can be run to verify the intermediate and final results.

1.6 RUNNING IN REAL-TIME

The final version of the program, either integer or floating point, is now ready for real-time operation. For this motor control problem, the changes that must be made are: (1) Remove the simulated motor module; (2) Insert a function to read the analog-to-digital converter; and (3) Insert a function to send results to the digital-to-analog converter.

A/D converters typically are available with conversion word widths of 8 to 16 bits, with 8, 10, and 12 bits being the most common. The output of the converter is an integer, so the same scaling conditions discussed above in connection with integer variables apply to it. The conversion precision is usually chosen on the basis of problem requirements since the measurement input will be a primary precision-limiting step. The narrower the conversion word, the cheaper and faster the converter will be. Once a conversion precision has been chosen, the voltage range at the input to the converter must be set so that the full range of the converter is used. This is done by setting gains in the analog circuit at the input to the converter or in the converter itself. The A/D converter function used in the simulated real time studies should have had the same conversion factor (motor speed -> voltage -> converter output) as the real converter, so the program ranges and scaling factors should not be affected by the change to a real A/D converter.

Operating the converter requires consideration of multiplexing and speed of conversion. In order to save cost, A/D converters are often multiplexed, that is, an electronic switch is placed ahead of the converter so that it can be attached to any of a number of different signals. The conversion speed is a factor because the

program must wait, or do something else, while the conversion is taking place. Many converters can complete a conversion in less than 100 microseconds, so it is usually not worth trying to do anything else in that time. The logic for getting a value from a specified multiplexor channel is shown in Fig. 1.7.

```
Get A/D value:
        Set multiplexor channel
        Start the conversion
        Wait for conversion to finish
        Read the result
```

Figure 1.7

Digital-to-analog converters (D/As) are much easier to use than A/Ds. No multiplexor is involved and the conversion is done very quickly. The programming required is just sending a value to the D/A's output port. The A/D and D/A modules should be coded and tested separately. With these modules installed in place of the real-time simulation functions, the program is ready to control an actual motor. The sample time that the controller will achieve depends entirely on the speed with which the computer can make a full cycle through the program. This is most easily checked by using an oscilloscope to measure the width of the "staircase" on the D/A output.

Functional testing can now begin to check that the control algorithm is working properly and then to tune the controller gains.

INTRODUCTION TO EXAMPLE PROGRAMS

1.7 ABOUT EXAMPLE PROGRAMS

The programs in this section are designed to provide examples of how one might implement the concepts discussed in the various chapters. The programs are designed to run on the ubiquitous IBM-PC, and almost all are written in the C programming language. There are a few modules written in 808x assembly language, but one cannot write real-time programs to control devices without having to resort to assembly language once in a while. Not all readers are familiar with the C programming language, and some of you may be just learning the language. Because of this, the descriptive text for the example programs in the earlier chapters will devote more space to descriptions of language features. In addition to describing the function and organization of the program modules, we also try to point out why certain language features are used and why certain functions are implemented the way they are. Although execution speed is always important, we believe that good programming practices such as structured design, input checking, data hiding, and encapsulation can be equally important, especially in a large application. Such practices need not adversely affect performance, but we do recognize that real-time process control programs operate under different conditions from, say, a spreadsheet, and sometimes rules may have to be bent to meet performance requirements.

Almost all real-time control programs have to deal with hardware; it may be an analog-to-digital converter, a parallel port, or a motor. It is impossible to write stand-alone example programs that can accommodate all the possible interface devices that a reader may need. It is also not realistic to expect readers to obtain the hardware setup required by the example programs. In view of these difficulties, some of the example programs include a module that simulates the external hardware environment. Simulation is a very useful tool for debugging program logic and developing new applications where it may be difficult if not impossible to run tests on the actual hardware.

We hope that readers who are implementing computer-based real-time control projects will find these example programs useful as starting points or as modules that can be incorporated into their applications.

1.8 THE C LANGUAGE STANDARD

At this time, the C programming language is undergoing a transition from the informal de facto standard established by the book *The C Programming Language* by Kernighan and Ritchie (commonly referred to as K&R), and the ANSI draft

standard proposed by the X3J11 committee. There are several new language features in the draft standard, and many C compiler vendors have already implemented them. Some of the features have been stable for quite some time now and are very unlikely to change in the future.

The example programs makes use of some ANSI C features such as the *void* type and function prototyping. The *void* type is used to declare functions that do not return a value,[1] and as a generic pointer type, an absence that is sorely felt in the old K&R standard. Function prototyping is a feature that allows a function's return type and the type of its arguments to be declared before the function is used or outside of the file where the function resides. This allows the compiler to check if the correct number of arguments are passed to the function and if the arguments are of the correct type.

1.9 COMPILERS

The example programs were compiled with Microsoft C, version 5.0, and linked using the Microsoft linker, version 3.61. The code and compilation commands suggested here are totally compatible with the version 4.00 compiler and the version 3.51 linker. It is unlikely that you will have to modify the code in order to compile the programs since we have tried to avoid compiler dependent features. The few assembly language modules were assembled using the version 4.00 Microsoft MASM assembler.

1.10 PROGRAMMING PRACTICES

Program organization and certain programming practices common to all the programs are described in this section.

1.10.1 FILE MODULES

The module concept organizes a program into groups of related functions. For example, a device driver module for an analog-to-digital converter contains functions that initialize and otherwise control the ADC, and maintains data required by all the routines[2] in the module. The module provides a set of interface routines that allows other routines outside of the module to set the channel and perform

[1] This is known as a *procedure* in Pascal.

[2] We use the term *routine* to refer to both functions that return a value and those that do not. In C, the term *function* is commonly used to denote both types of functions; a function in C is assumed to return an integer unless declared otherwise.

analog-to-digital conversion. Routines outside of the module cannot call on routines or access data not specifically made *visible* outside of the module.[3] Thus, a module is a sort of black box with a defined interface.

The module concept encourages one to conceptualize the program as a collection of functional blocks that communicate with other blocks through well defined-interface routines. As long as the module interface is unchanged, its contents can be replaced without affecting the rest of the program. Although this may not seem important in small programs, large programs are very difficult to manage and maintain unless some form of modular organization is used. This will become obvious in the later chapters as the example programs increase in size and complexity.

The module concept is central to the **Modula-2** language, and a variant of it is implemented in **Ada** as *packages*. Unfortunately, C does not directly support modules, though many of the concepts can be implemented in C by treating a file as a module. In C, a routine that is declared to be *static* can only be called by name by other routines within the same file; it is not visible outside of the file. Such routines can be thought of as being "private" to the module.

A data variable defined outside of any routine is visible to all routines. However, as with routines, the variable can be made to be visible only within a file in which it is defined by declaring it to be static. Routines and variables outside of the module can have the same name without fear of accidental name conflicts (a particularly insidious type of bug).

How do we make the interface routines (and perhaps data) known to other functions outside the module? The simplest way is by means of a header file declaring the names, return type, and arguments of the interface routines. Visible routines and data are declared to be *extern*, which means that the code and data are probably not in the current file, i.e., external to the current file. Thus, each module has its own header file, and we have adopted the convention that the header file has the same name as the module file, but with the .h extension instead of the .c extension. Other program modules wishing to use the services of a particular module merely have to include the appropriate header file.

C automatically assumes that a routine returns an integer unless declared otherwise. Thus, all functions that do not return integers should be declared before they are called, preferably with a function prototype. Declaring a function before it is used or defined is also known as a *forward declaration*. In the module file, there is a section near the beginning of the file for forward declarations of functions private to the module. Strictly speaking, forward declarations for private routines should use the static keyword, but some compilers may not accept the use

[3] The term *visible* in this context means that the item can be referenced directly by name; a function is visible if it can be called by name, a data variable is visible if it can be read and assigned-to by name.

of the static keyword in a forward declaration.[4] Because of this, we have avoided the static keyword in forward declarations for private routines.

Function prototyping is a more advanced form of forward declaration that not only declares the return type, but also the type and order of the function arguments. This allows compilers to check that the correct types are being passed to functions. Since not all compilers have implemented this feature yet, function prototyping is only enabled if the symbol ANSI is defined. The file modules in the example programs are organized into sections:

header files and imported declarations
global data visible outside of the module
forward declarations
private data
interface or entry functions
private functions
initialization functions

Grouping related items together makes it easier to locate specific items for modification, and there are fewer chances of introducing bugs due to multiple and possibly different declarations for the same item.

Each routine in the module is preceded by a short comment block describing the routine. The type of routine is indicated by the words PROCEDURE or FUNCTION, used in the Pascal sense. This is followed by the name of the routine in capital letters; this makes it easier to search for the place where the routine is defined using a program or text editor. The return type and arguments are described next. There is an optional REMARK section that describes any special features or algorithm used in the routine.

1.10.2 THE ENVIRONMENT FILE—ENVIR.H

Many of the example program modules include a header file named *envir.h*. This header file specifies the environment under which the program is compiled and executed. This is useful if the program is to run on different computers or has to be compiled by different compilers. This is not uncommon, especially in "embedded systems," as real-time control programs are often developed and debugged on large computers or workstations and later transferred to the microcomputer that actually controls the hardware. Code that depends on certain features of the environment, such as the computer, operating system, or compiler can be isolated and conditionally compiled using the information in the *envir.h* file.

[4] We know of a compiler that quietly accepts static forward declarations, but generates incorrect code!

1.10.3 INPUT AND OUTPUT PORTS—INOUT.H

Computers based on Intel CPUs, such as the IBM-PC family, have special instructions to access i/o ports, and most compiler vendors supply functions to read from and write to i/o ports. Unfortunately, there is no consensus on the names of these functions. To simplify matters, we have chosen to use the function names *in()* and *out()* for routines that read a byte from a port and write a byte to a port, respectively. These names are mapped to the appropriate compiler library names by macros defined in *inout.h*; here is an excerpt:

```
#if CIC86           /* Computer Innovations C86 */
#define in(port)            inportb((unsigned)(port))
#define out(port, value)    outportb((unsigned)(port), value)
#endif

#if MICROSOFT       /* Microsoft C Version 4.00 & 5.00 */
#define in(port)            inp((unsigned)(port))
#define out(port, value)    outp((unsigned)(port), value)
#endif
```

1.10.4 FLOATS AND DOUBLES

C has two floating point types: *float* for single precision and *double* for double precision floating point types. For historical reasons, the C language specifies that floats are always expanded into doubles in expressions and when passed to a function.[5] The example program uses doubles rather than floats for floating point variables, as we feel that in most cases, the reduction in conversion overhead more than compensates for the slight increase in storage requirement and data transfer time. There is also another reason relating to the use of function prototypes in the proposed ANSI standard. The proposed standard in its current form allows single precision calculations for greater speed without automatic expansion into double precision. Function prototypes that declare arguments as floats may have problems with functions that expect float type parameters to be expanded into doubles. Using doubles for all floating point variables avoids this problem during the transition from K&R C to ANSI C.

[5]The historical reason for this convention of automatically converting floats to doubles is that C was developed on early PDP-11 computers. Programs running on early PDP-11s cannot determine whether the floating point unit is in single or double precision mode; this presents an obvious problem for multi-tasking systems since the precision mode of the floating point unit could have been changed by another program. The solution the creators of C came up with was to always use double precision.

CHAPTER 2

TIME

The mathematical function used for the control algorithm in the previous chapter made no reference to time, so the program could be run as often as the processor was able to. This is very simple and convenient for programming, but unusual. Most real-time programs require some synchronization with "real" time. In this chapter, we shall consider an algorithm that makes use of data sampled at known instants of time. It is still the only task present, so synchronous programming can still be used, but some mechanism must be added to determine time.

2.1 PROPORTIONAL PLUS INTEGRAL (PI) CONTROL

The proportional control algorithm has the disadvantage in that, when the error is zero, its output is fixed by the constant bias term. In general, there will only be a single setpoint for which the preset bias is correct. For all other setpoints, the inability of the controller to provide different biases for different setpoints will mean that after all of the transient behavior has died away, the output will not be at its desired value but will have some steady-state error. Whether or not this happens depends on the nature of the behavior of the system under control. If we imagine a frictionless motor, once it has reached its desired speed it will coast at that speed forever if no energy is removed from the system. If a controller is built in such a way that the control output represents torque applied to the motor, then maint-

enance of setpoint speed will not require any torque, and an offset bias of zero will work for all setpoint values.

Even without violating the assumption of no friction, whether the system will work satisfactorily with zero for an offset bias depends on the type of power amplifier that is used. A power amplifier that controls voltage across the motor will allow current to flow when its voltage is zero, thus dissipating energy and causing the motor to slow down. A non-zero bias voltage is needed to maintain speed, so, with only proportional (P) control, there will be a steady-state error in speed. A power amplifier that controls the current flow to the motor will act very much like a torque source, so will come closer to having no steady-state error. All real systems have some dissipation, however, so even a current-controlled system will have some steady-state offset.

Rather than use a constant bias voltage in the controller, the control algorithm can be designed to automatically adjust its bias voltage to whatever level is necessary to ensure that there is no steady-state error. This is done by substituting a term that acts on accumulated error for the constant bias. This can be expressed as an integral

$$m=k_p\varepsilon+k_i\int\varepsilon dt \qquad (2.1)$$

As long as the error is not zero, the integral will continue to increase. When the error finally reaches zero, the value of the integral will remain, providing the needed bias to hold the error at zero. For computer control systems, the integral is approximated as a summation

$$m=k_p\varepsilon+k_i\sum\varepsilon \qquad (2.2)$$

From an implementation point of view, it is simpler to separate the accumulation process from the control output computation. This can be done with the general form shown in Fig. 2.1.

```
error = setpoint - velocity
accumulation = accumulation + error
m = kp * error + ki * accumulation
```

Figure 2.1

The rate at which the accumulation term builds up depends on how often the process is sampled. It is thus necessary to control the timing of the sampling process, so that the sampling is done uniformly, and at a specified interval. A further point that must be considered is that the addition of the integral term (to give a PI control) reduces the system stability margin. If the integral gain, k_i is too high, it is possible to produce unstable behavior. k_i is therefore usually kept as small as possible consistent with reducing the error to zero in a reasonable amount of time.

2.2 CLOCKS

To a computer, a clock is a device that generates a sequence of pulses with a constant time interval between the pulses. To "keep time," it is necessary to count each of the pulses and keep track of the count. Within this context, there are several ways of recording the passage of time. The clock/calendar model is to maintain a record of "absolute" time. Absolute time, however, implies that the count can go on forever, which, in a finite precision machine such as a computer, can lead to difficulties in defining the means of storing the number. In ordinary timekeeping, we switch from the clock to the calendar for long time periods, but rarely maintain a precision of better that one second. For real-time problems, the precision level of interest is usually somewhere between a microsecond (10^{-6} sec) and a millisecond (10^{-3} sec). Cumulatively keeping track of microseconds means 3.6×10^{9} counts per hour! Since real-time computers can run continuously for days, weeks, or even years, the bookkeeping problem is substantial. Furthermore, use of such a large format number to keep track of time can use up significant amounts of computing time if the time has to be updated frequently.

Fortunately, many real-time tasks only need to know relative time rather than absolute time, so the model of an interval timer can be used instead of the clock/calendar model. In these cases, the nature of the task is such that at the time it is run, the interval to the next time it must run again is already known. The interval timer is then set for that time, and when it runs out, the task is run again.

A third view of time is the stopwatch model. In this case, an event will happen at some indeterminate time in the future, but, when that event happens, something must be done (perhaps only record the time of the event). The interval timer and stopwatch models do not have as severe a precision problem as the absolute timekeeper, but there can still be problems with precision and word size. A 16-bit (unsigned) integer can keep track of 65535 counts. Even at a millisecond precision level, that is only about one minute. 32-bit integers can keep track of about 4 billion counts (4×10^{9}) which is adequate for most problems, but dealing with 32-bit integers can be slow for many computers.

2.3 CLOCK IMPLEMENTATION

The original source for the pulse train that is the fundamental timekeeper must be a physical device, usually a crystal oscillator, but, if the accuracy and stability of a crystal is not needed, it could be a tuned circuit. The rest of the clock can be implemented in either hardware or software, depending on the precision and duration requirements. When implemented in hardware, the basic pulse interval is usually from around one microsecond to around one millisecond. Devices called *programmable clocks* usually have several operating modes so that they can be used in either interval timing mode or stopwatch (event detection) mode.

Programmable timers usually also allow for variable count rate by dividing the basic pulse rate by a user-specified amount (every other pulse, every third, etc.). Because they are hardware devices, the duration is limited by the word size of the counter; 16-bits is common, but others are available.

Clock/calendar hardware clocks are also available, mainly for maintaining time and date information for operating systems. They usually have much cruder precision and are not as useful for real-time applications as programmable clocks.

At the other end of the spectrum, if the pulse interval is in the range of a millisecond or more, it is possible to implement the rest of the clock entirely in software. Hardware solutions are necessary for faster pulse trains to avoid using a large fraction of the computing time for the clock software. As pulse intervals get down toward a microsecond, it becomes impossible for a computer to keep up at all. Software solutions are much more flexible than hardware clocks because changes in precision, word size, and so on, can all be taken care of with programming changes. Mode changes and special needs for event detection can also be accommodated more easily.

The program structure for an interval timer implemented in software is shown in Fig. 2.2. The clock-set module is executed to start the interval timer by presetting a counter to the desired number of "ticks." Each pulse represents one tick of the clock. The clock module is executed whenever a pulse from the clock is detected. The clock module must also contain some means of communicating with the other parts of the program that use the time information. This can be done by providing a function that returns the current value of the counter. Another function normally supplied returns a clock-done *flag*, a logic variable that indicates whether the clock has run out yet. Its advantage is that its form is the same regardless of the means of implementing the clock, so programs using that information do not have to "know" the form of the internal counter used in the clock program.

```
clock-set(interval):
        counter = interval * scaling-factor
        clock-done = FALSE
        turn on the pulse generator and
                enable the detection circuitry
        return

clock:          (This module is called whenever a clock
                pulse is detected)
        counter = counter - 1
        if(counter <= 0)
                clock-done = TRUE
        return
```

Figure 2.2

It would also be possible to make the counter and clock-done flag available directly to programs as global variables. This method is less attractive, even though it requires slightly less computing time, because it compromises the isolation of the clock service module. Imagine, for example, a programming error resulting in a statement that changed the value of the clock counter in a module that was only supposed to use the clock, not change it. All other parts of the program that use the clock would then operate improperly, implying that the error is somewhere in the clock service functions when it is not. Careful modularization and "protection" of variables that are local to a module can go a long way toward more reliable and easy-to-debug programs.

The programming logic for using a hardware programmable timer is very similar. The clock-set program must transmit the interval information to the timer hardware and start it running. The maintenance of the count is done by the hardware timer, so the only further software requirement is a function that can tell if time has run out.

2.4 USING TIME IN CONTROL—PARALLEL PROCESSES

The primary interaction between the control program and the timekeeper is the flag variable, clock-done, which is available by a function call. For the purpose of designing the control program, it is useful to assume that time is being kept by a process that is completely independent of the control program, and running parallel with it. The parallel process assumption implies that nothing going on in the control program will interfere with the timekeeping function, and no constraints on execution time have to be applied to the control program other than the requirement that it be able to complete its work by the time of the next sample. With this structure in mind, the control program shown in Fig. 2.3 will behave the same as the control program developed for proportional control, except that sampling and control will only take place at specified times.

```
clock-set(sample-interval)

For number of iterations specified
        Do control
        Wait until clock-done = TRUE
        clock-set(sample-interval)
                                    (Reset the clock for the next
                                    sample)
```

Figure 2.3

One feature of this program is noted in the comment at the top of Fig. 2.3, that is, the time interval must be entered in units of ticks. Although a seemingly minor detail, this requirement makes the use of the high-level program dependent on the details of the low-level implementation. To run the program, the user must be aware of clock implementation information. Most users have no need for such information, so a better implementation might be to call the clock-set function with the sample time in units of seconds (or, perhaps, milliseconds) and convert to ticks in clock-set. The decisions on how to best insulate the user from unnecessary detail must be made early in the program design cycle. Once they become embedded, changes might have to be made throughout the program to alter the level at which information can be accessed.

Such decisions, however, have a variety of consequences. In this case, the conversion of sample interval from seconds to ticks normally would only have to be done once, during the setup phase of the program. By moving that conversion to a lower level (in clock-set), the user is indeed insulated from that detail, but the conversion must now be performed for every sample interval. Thus, convenience and portability may affect efficiency. A further difficulty is in deciding what "convenience" really is. Users of packaged programs are constantly frustrated on the one hand by features that are buried too deeply in the program for them to change and, on the other hand, by an overwhelming choice of features and parameters, which often interact with each other.

2.5 ACHIEVING PARALLELISM

If the clock is implemented with a hardware timer, the parallel operation assumed above is achieved naturally. Suitable circuitry must be provided so that the clock can be set and interrogated. In all other respects, though, the clock and the computer are independent devices. When the counting part of the clock is implemented in software, both the clock software and the control software must be run in the same computer. Since the computer is a strictly sequential device, true parallel operation cannot be achieved. If the computer is fast enough, however, both tasks can be carried out with an appearance of concurrent operation even though they run in sequence. If this is to be done in a manner that is "invisible" to the control program, a facility must be available that can suspend the execution of the control program whenever a pulse is detected, run the clock counting function, and then resume execution of the suspended program. This is called an *interrupt* mechanism and is present on most microprocessors.

The interrupt provides for pseudo-parallel operation as long as the computation that takes place during the interrupt does not significantly interfere with the *background* calculation. In most cases, this is accomplished by making the interrupt routine (the *foreground*) short enough and infrequent enough so that the appearance to the user is that the background calculation is just running on a slightly slower processor.

The mechanism of the interrupt is that the hardware device causing the interrupt (the input port where the pulse train signal is connected in our case) sends a signal to the processor requesting an interrupt. If the processor's operating mode is such that the signal can be recognized, it initiates the interrupt by suspending the execution of its current program and saving whatever internal processor information is necessary to restart that task when the interrupt has been completed. The processor then starts the execution of the foreground task, the "clock" module described above. When the foreground task has completed its work, the process is reversed. A signal is sent to the device that caused the interrupt indicating that interrupt processing is complete, the saved information is used to restore the processor state to where it was when the interrupt first occurred, and the background process is resumed. This sequence of events is illustrated in Fig. 2.4.

1. Hardware device requests an interrupt
2. Processor recognizes request
3. Execution of existing task is continued to the end of the current machine instruction
4. Internal processor status information is saved
5. Foreground task is started
6. Foreground task finishes
7. Background task's status information is restored
8. Background resumes

Figure 2.4

2.6 INTERRUPT HARDWARE

There are three main functions that must be accomplished by the interrupt control hardware:

1. Maintain *masking* information to decide whether or not an interrupt request should be honored,
2. Establish the *priority* of the current interrupt relative to already active interrupts, and
3. Determine the identity of the interrupt (what device caused the interrupt) and communicate the location (in memory) of the interrupt service function to the processor (*vectoring*).

The hardware that is used to perform these functions is often separate from the CPU, so it is possible to use different interrupt control schemes with the same CPU hardware.

The communication between the interrupt control hardware and the CPU to set the various modes, parameters, etc., is done through input and output ports. Most interrupt controllers allow for several modes of operation; they can be complex, and the operating instructions must be studied carefully!

The 3 functions described above, while always present in some form, are not always implemented to the same degree of sophistication. Masking, for example, implies the ability to selectively *enable* or *disable* individual interrupts. In simple interrupt processors, there may only be a global enable/disable present while others may allow for control of groups of devices.

The priority function can also have several levels of implementation. The simplest level disables all interrupts as soon as an interrupt request has been honored. This method gives all interrupts equal priority. Once the interrupt service function starts, the decision can be made in software whether (or when) to re-enable interrupts so that other interrupts can be allowed before processing of the current interrupt is complete. At the other end of the spectrum, a fully prioritized interrupt control maintains a separate priority for each interrupt. When an interrupt service routine starts running, it sets a CPU register to indicate its operating priority level. If another interrupt device requests service, the priority of that interrupt is compared to the priority level. If the requesting interrupt has a higher priority, its request is honored. Otherwise, it is held in abeyance.

A middle, and fairly common, priority control establishes priorities with groups of devices and stores the priority level information about the currently operating interrupt. The operating priority may not be accessible from the CPU, so remains fixed at the level set by the device. In order for the priority control to work, there must be appropriate mechanisms to signal to the interrupt controller that a particular interrupt service function has completed its work so that that priority level can be cleared.

The simplest possible vectoring method is for all interrupts to cause execution of the same interrupt service routine. This leaves to the software the task of determining the source of the interrupt and then executing the associated service function. This method requires a minimum of hardware, but is costly in computation time. It is not used frequently anymore, because interrupts are normally used for servicing time-critical tasks. Dedicated interrupt request lines between the interrupt controller and devices can be used to establish a time efficient compromise. There can be as many unique devices as there are wires provided for the interconnection. Beyond that, each wire must be shared. Because of the unique connection, however, the interrupt controller can determine the identity of the interrupting device very quickly and communicate the associated vector (i.e., memory address) to the CPU.

When a system has many devices, however, a more flexible method—a fully vectored interrupt—is necessary. To achieve full vectorization, each individual interrupt must be able to specify a unique vector. An interchange between the interrupt controller and the device is often used for this purpose. When the interrupt controller recognizes an interrupt, it sends a signal to the

device asking for its interrupt vector. The device then sends the vector address to the interrupt controller, which passes it on to the CPU. The interchange between the interrupt controller and the device interface takes some time, but very much less time than would be needed to do the same interchange in software.

2.7 XIGNAL: A SOFTWARE INTERRUPT CONTROLLER

Interrupt hardware is usually complex. Many details must be attended to in order to set up and use the interrupt controller. The "xignal" facility is a software package that, for almost all real-time problems, allows that setup to be done once, then used for many programs in a way that makes the interrupts very simple to implement.

"Xignal" looks to the user like a fully vectored interrupt, except that communication with it only requires knowledge of some code names for the available interrupts. The name "xignal" is used to avoid conflicts with programs named "signal" that are part of several operating systems and compiler packages. Figure 2.5 shows a sample of using "signal" to set up an interrupt using a clock (the syntax is patterned after the "signal" facility of the UNIX operating system). The first call shown will cause the function timer-service to be called whenever the clock interrupts. The second call resets the interrupt hardware back to its original state after the real-time portion of the program is over.

```
xignal(SIGTMR, timer-service)
        .
        .
        .
xignal(SIGTMR, signal-default)
```

Figure 2.5

A general purpose package such as "signal" ("xignal") is never quite as time efficient as a module written for a specific purpose, but it should be fast enough for most needs.