ECE 410 / ECE 557 Laboratory

Professor S. A. Bortoff

August, 1996

Introduction to the Control Systems Laboratory

An important component of both ECE 410 and ECE 557 is the four laboratories. While the lectures emphasize *continuous-time* control theory, the labs emphasize *discrete-time, digital* control, i.e., computer control. Each laboratory builds upon results from the previous lab, a process that continues into ECE 411.

The first three labs concern digital control of a DC servomotor system. Servomotors are found throughout industry, used mainly as actuators in systems such as robotic manipulators, aircraft, automobiles, etc. Almost everything that requires precision movement is powered by a servomotor, so they are a very important system in themselves. Moreover, servomotors exhibit a number of characteristics found in other control systems, including unmodeled dynamics and sensor noise. Thus, they make an excellent laboratory test bed.

The fourth lab is a project emphasizing robust controller design using MATLAB. The lab does not involve hardware, and can be done in groups outside GB347. All of the ECF Sparc workstations support MATLAB, and can be used to complete Lab 4, and the pre-lab work for Labs 1-3.

Before coming to the first lab, you should become familiar with the laboratory equipment and policies by reading through this entire document, which is organized as follows. Section 1 lists some of the rules enforced in GB347. These are in place because the room is a very busy place when school is in session. Section 2 discusses the ECE 410 grading policy. The servomotor system used in Labs 1, 2 and 3 is discussed in Section 3, and must be well-understood before beginning Lab 1. Section 4 discusses the computer, the controller hardware and software, and the network's software, which includes the C compiler and MATLAB.

1 GB 347 Rules

Several undergraduate courses share the Systems Laboratory, located in GB347, for their labs. In addition, the computers in GB347 are available to students who are working on their fourth-year theses, or who require a computer for personal use. Because it is so heavily used, a few rules must be followed.

1. No food or drink is allowed into the lab.

2. Academic work has priority over personal work.

3. During assigned laboratory periods, **only those students assigned to that laboratory section will be allowed in GB347.** Ten minutes before a scheduled lab begins, all other students will be asked to leave, so the TAs can set-up the equipment. Laboratory schedules are posted on the door of GB347.

4. Lab equipment can be used only during assigned lab periods. **No equipment can be checked-out or used after hours.**

5. Lab equipment is to be set-up and put away by one of the TAs. Parts of the equipment are fragile and extremely easy to damage beyond repair. The TAs get paid to do this; make them earn their money.

6. Report any damaged equipment to either the TA on duty or Mr. George Owen, the laboratory manager, whose office is adjacent to GB347.

2 Grading Policies

Each of the four labs is worth 5% of your final course mark. The first three labs (Labs 1-3, not Lab 0) are done in GB347 using the servomotor systems. The 5 marks are assigned as follows: 4 marks for performance during lab, as judged by the lab TA, and 1 mark for the informal lab write-up, done in a lab notebook, which is submitted on the last day of class, December 3, 1997. Lab performance will be determined by the quality of the pre-lab, done in the lab notebook, the demonstrations, and the answers given to the TA's questions. There are no formal reports to submit for the first three labs!

The fourth lab is a MATLAB design project which can be completed by pairs of students outside the assigned lab time. It requires a "formal" report, typset, due December 3, 1997, at 3:30pm in Professor Bortoff's office, GB343A. This lab will be distributed at a later date.

All laboratory material is fair game on both the quizzes and the final. If you *do* the lab, and answer the questions, then any exam questions will be straightforward. For example, a final exam question may involve the design of a control system that is very similar to that done in Lab 3. If you have done lab 3, then you've already done that question, and it's always easier to do it the second time. In fact, Problem 4 on the 1995 final, worth 30 marks out of 100, was identical to a part of the 1995 edition of Lab 4. Writing C code will not be the subject of any exam questions.

3 The Servomotor System

The first three labs use the servomotor systems, which consist of the following four units.

1. **A servomotor unit**. This is the blue box with a wheel protruding from the top. The unit contains a small motor, which is connected through a gear to the wheel on top. Each of the servomotors has a different amount of inertia bolted to this wheel, and therefore, each of the servomotors is a *different* plant. *You must use the same servomotor for all three labs.* Of course, because each motor is different, each group will get different experimental results. Therefore, sharing of data, models, results, etc., is not recommended. A potentiometer (POT) is also connected to the shaft. The POT converts the motor's angular position to a voltage, which in turn drives the analog-to-digital (A/D) board in the computer.

2. **A power amplifier unit.** This is simply a power operational amplifier which can output more current than the digital-to-analog (D/A) board in the computer. For all the labs, it is wired as a non-inverting, unity-gain "buffer" amplifier, i.e., a wire connects the output to the negative input, and the positive input is driven by the D/A output.

3. **A networked PC containing the A/D and D/A card.** Not all of the PCs have this card. Those that do will be partially connected to the rest of the equipment when you arrive for lab. It is suggested that you use the same workstation for each laboratory.

4. **A bench-top board** for interfacing the D/A and A/D card with the servomotor unit and power amplifier. This is connected to the computer via a grey ribbon cable. **PLEASE DO NOT ATTEMPT TO CONNECT THIS CABLE TO THE COMPUTER. IT IS VERY EASY TO INSERT THE RIBBON CABLE BACKWARDS, WHICH DAMAGES THE A/D CARD, MAKING THE ENTIRE WORKSTATION UNUSABLE. LET THE TA CONNECT THE RIBBON CABLE TO THE COMPUTER.**

3.1 Servomotor Model

The servomotor system is diagrammed in Figure 1. Here, $y(t)$ is the motor's angular position,

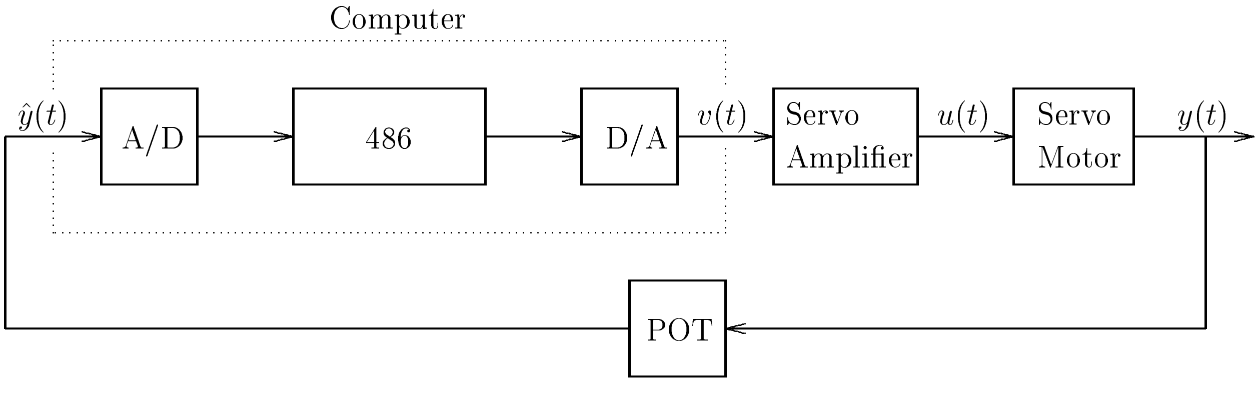

Figure 1: Physical block diagram of the servomotor system.

in degrees (degrees are used instead of the MKS units of radians, because the motors are labeled in degrees), $\hat{y}(t)$ is the output of the POT, which is proportional to $y(t)$, but has units of volts, $v(t)$ is the output of the D/A channel 0, in units of volts, $u(t)$ is the output of the power amplifier, in units of volts.

The POT is linear, so it can be modeled as

$$\hat{y}(t) = \hat{K}_s y(t) + \hat{K}_o$$

where $\hat{K}_o$ is a voltage offset, and $\hat{K}_s$ is the POT gain, in units of (volts/degree). (Thus, $0°$ corresponds to $\hat{K}_o$ volts, which is not necessarily zero.) The voltage $\hat{y}$ is periodically sampled by the A/D board, which converts it to a dimensionless integer $\bar{y}$ between 0 and 4095. Software running on the PC can read this integer from a fixed memory location, and manipulate it according to the control law. Although the conversion is discrete, i.e., $\hat{y}$ is a continuous voltage, while $\bar{y}$ is one of 4096 integers, we will model it as a linear map

$$\bar{y}(t) = \bar{K}_s \hat{y}(t) + \bar{K}_o.$$

Thus, the relationship between $\bar{y}$ and y is given by

$$\begin{aligned} \bar{y} &= \bar{K}_s(\hat{K}_s y + \hat{K}_o) + \bar{K}_o \\ &= K_3 y + K_0 \end{aligned} \tag{1}$$

where $K_3 = \bar{K}_s \hat{K}_s$ has units of (1/degree) and $K_0 = \bar{K}_s \hat{K}_o + \bar{K}_o$ is dimensionless.

The D/A channel converts an integer between 0 and 4095 to a set of discrete voltages. The relationship is linear, with 0 mapping to -5 volts, 2047 mapping to 0 volts, and 4095 mapping to +5 volts. We can therefore model the D/A channel as

$$v = K_{DA}\bar{v} - 5\,\text{volts}, \tag{2}$$

where v is the D/A output voltage, $\bar{v}$ is an dimensionless integer between 0 and 4095, and the gain $K_{DA} = 10/4095$ volts. The power amplifier is also linear, and can be modeled as a dimensionless gain K_{AMP}:

$$K_{AMP} = \frac{u(t)}{v(t)}.$$

The servomotor can be modeled with a second-order transfer function

$$P_1(s) = \frac{Y(s)}{U(s)} = \frac{K_M}{s(s + K_T)}, \tag{3}$$

where K_M is the servomotor gain, and $1/K_T$ is the servomotor time-constant. This can be understood as the product of a first-order system $\frac{K_M}{s+K_T}$, which maps input voltage to motor shaft velocity, and an integrator $\frac{1}{s}$ that takes velocity to position. Both of the constants depend upon the electrical characteristics of the motor, the inertia of the wheel, and the friction (damping) in the motor and gear. Since each of the servomotors has a different amount of inertia, the constants differ from motor to motor: Make sure to use the same servomotor for all three labs.

This second-order model ignores the inductance of the motor windings, i.e., the electrical dynamics. This is possible because the inductance L is small relative to the shaft inertia J. More precisely, the electrical time-constant of the motor winding, $\tau_E = L/R$, where R is the winding resistance, is much smaller (faster) then the mechanical time-constant $1/K_T$. A more precise model is

$$P_2(s) = \frac{K_M}{s(s + K_T)(1 + \tau_E s)}, \tag{4}$$

which has an additional pole at $s = -\frac{1}{\tau_E}$ to model the electrical dynamics of the motor, which is just an L-R circuit. Notice that for low frequencies, i.e., for $\omega << 1/\tau_E$, $P_1(j\omega) \approx P_2(j\omega)$; the two models differ only at high frequencies. Although we will use $P_1(s)$, and not $P_2(s)$, in all of our *designs*, it is important to remember that $P_1(s)$ only a simplified *model* of the actual servomotor. Unmodeled dynamics such as the winding inductance are present in the actual system, and affect closed-loop performance, as we shall see. Other unmodeled effects include motor shaft flexibility, static friction, and gear backlash.

All controllers are realized in software running on the PC. This controller software takes as input a dimensionless sequence (in time) of numbers from the A/D, and produces as output to the D/A a dimensionless sequence (in time) of numbers. This discrete-time controller realization is computed from the continuous-time $C(s)$ using one of several techniques discussed in lecture and the labs.

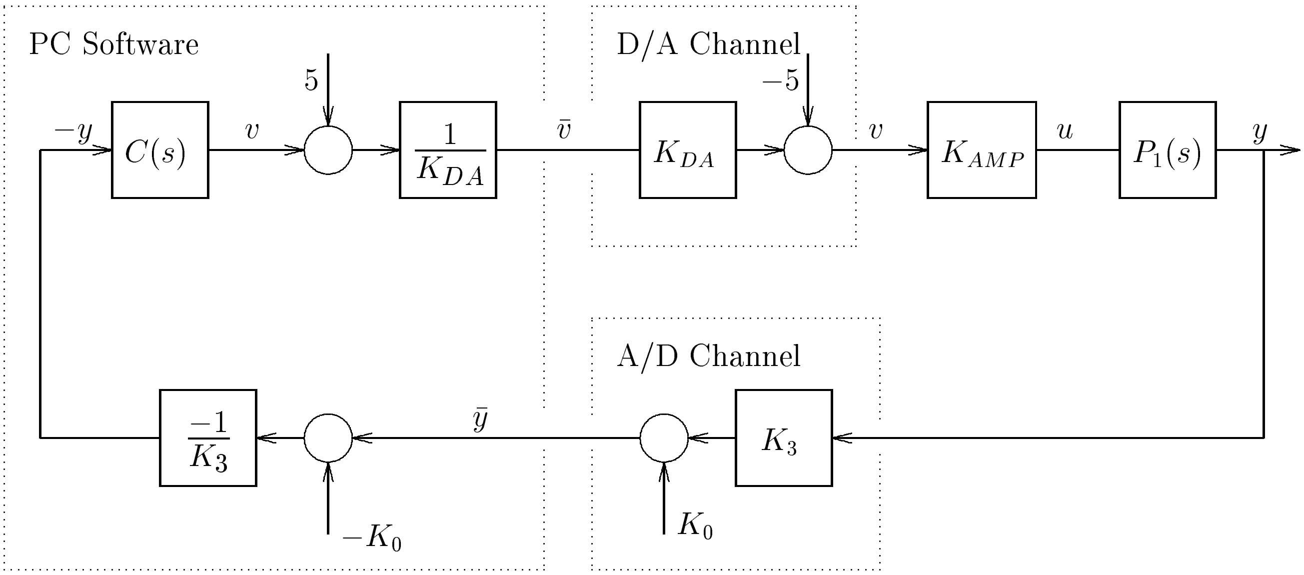

Figure 2: Block diagram of the servomotor system.

The control system block diagram is in Figure 2. Notice that the software does three things: 1. converts the A/D integer input $\bar{y}$ to degrees using equation (1); 2. computes v via a discrete-time realization of $C(s)$; and 3. converts v to the integer output $\bar{v}$, suitable for the D/A conversion, using equation (2).

In Figure 2, there is no reference input $r(t)$. An input could be added by using a second A/D channel (there are a total of 6). In this case, the A/D input to the computer would have to be scaled and the offset compensated, using equation (1), just as is done for y. Alternatively, the input $r(t)$ can be generated by the computer in software. This is preferred because a rich set of inputs can be generated easily in software, e.g., steps, ramps, sinusoids, etc. In this case, it is important that $r(t)$ have units of degrees. Why? Suppose, for example, that we are to design a controller $C(s)$ to cause the motor shaft to turn at a constant velocity of precisely 90 degrees per second, with no steady-state error. (What does the Final Value Theorem say about this design?) For this example, the input is

$$r(t) = K_r \cdot t,$$

where K_r is a constant (the slope), and the tracking error $e(t) = r(t) - y(t)$ is to converge to

0 as $t \to \infty$. To subtract them, both $r(t)$ and $y(t)$ must have identical units. The complete block diagram is shown in Figure 3.

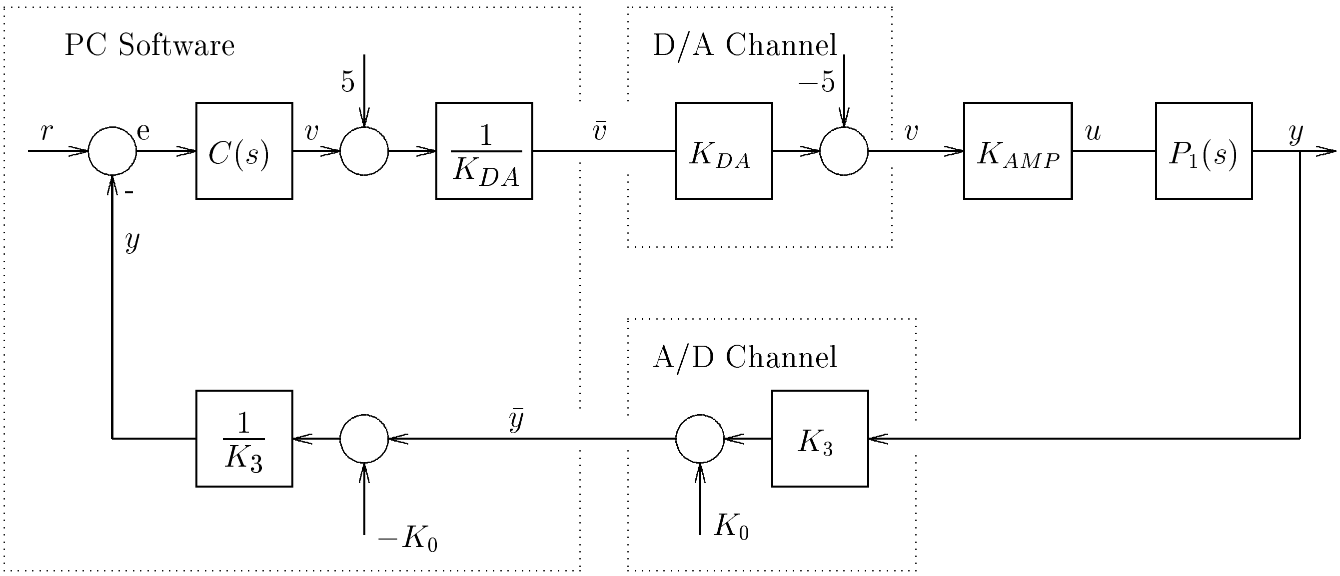

Figure 3: Servomotor system block diagram with computer-generated input.

Although the block diagram in Figure 3 may appear to be complex, most of the blocks are simple gains and cancel each other. Simply multiplying all the gains together, and substituting the definition of $P_1(s)$, gives the unity feedback system shown in Figure 4. Finally, defining

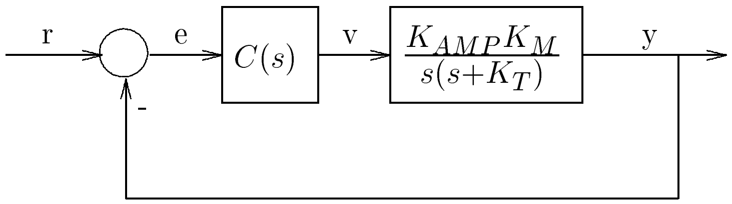

Figure 4: Servomotor block diagram with second-order model $P_1(s)$.

$$K_1 := K_{AMP}K_M \tag{5}$$

$$K_2 := K_T, \tag{6}$$

the block diagram can be redrawn as in Figure 5. The plant is completely determined by knowledge of K_0, K_1, K_2 and K_3. It is not necessary to measure each and every gain, e.g. K_{AMP}: It is only the product of the loop gains, K_1, the motor time constant, K_2, and the gain K_3 and offset K_0 from shaft angle through the POT and A/D, that are important. This observation considerably simplifies the system identification, done in Lab 1.

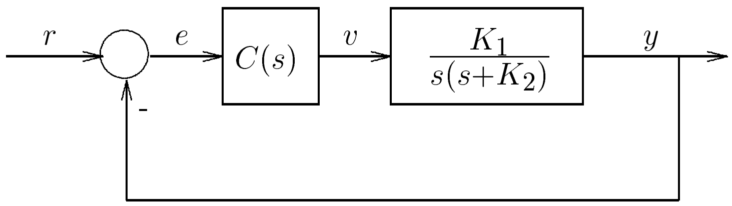

Figure 5: Simplified servomotor block diagram.

4 The Computer System

The role of the computer is threefold:

- to realize a real-time, discrete-time approximation to the continuous-time $C(s)$;
- to serve as a platform for real-time software development, i.e., a compiler for the real-time C code;
- to serve as a MATLAB workstation, making design and analysis possible.

4.1 Real-Time Computer Control

There are two common methods to implement real-time control software: Polling and Interrupts. Suppose the controller is a positive gain: $C(s) = K$. Thus, the computer is to sample the A/D converter, subtract the value from a desired value r, multiply the result by K, and output the result to the D/A. It is supposed to do this continuously in time, but the computer is a discrete-time device. A rather simple-minded approach to approximate the continuous-time gain using the discrete-time computer is called *polling*: The control is implemented as an infinite loop, as in the following pseudo-code.

1. `Sample the A/D giving` $y(t)$.
2. `Determine the value of` $r(t)$.
3. `Compute the tracking error` $e(t) = y(t) - r(t)$.
4. `Multiply result by` K.
5. `Output result to the D/A.`
6. `GOTO 1.`

There are several problems with this approach. First, the sample period, which is the time between samples, is the amount of time it takes to execute the loop. A designer has no way of knowing the sample period before compiling the code and experimentally measuring it. Although this will not greatly effect the simple gain controller, discrete-time implementations of dynamic controllers are a function of the sample rate. *In order to compute a discrete-time realization of a dynamic $C(s)$, the designer must know the sample rate.* Note that the time through the loop will change if the software is moved to a different machine, the code is modified, or just recompiled using different compiler options.

A second problem with polling is the fact that the code inside the loop will take a different amount of time to execute each time through the loop. This is because the execution time for a given operation depends upon the data. Computing 0×0 takes less time than $\pi \times e$, for example. This means that the sample times will not be uniform in time. However, digital implementation of a dynamic controller requires uniformity of the sample period. The situation is illustrated in Figure 6.

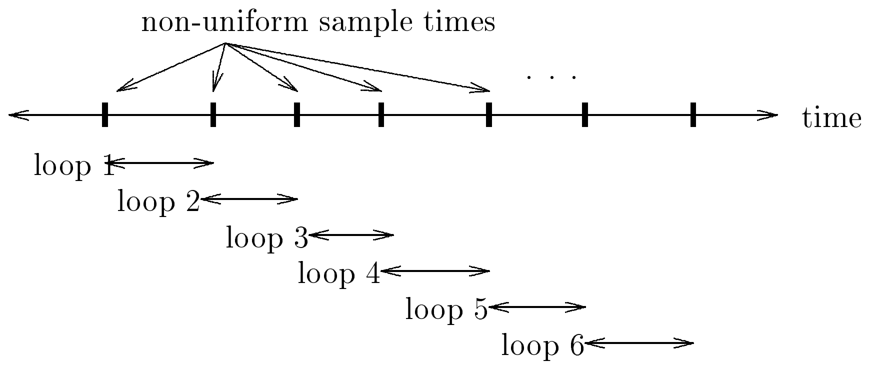

Figure 6: Timing of a polling controller. Note the non-uniform sampling times, due to the non-uniform amount of execution time spent for each loop.

A better way to realize timing-sensitive digital controllers uses *interrupts*. The software is designed so the processor remains in an timing-insensitive loop, polling the keyboard for input from the user, for example, until an externally-generated hardware interrupt occurs. The interrupt is a periodic signal applied to the processor, and can originate from a timer on the A/D card, or the DOS timer inside the computer, but external to the 80486 microprocessor. Once the 80486 receives the interrupt signal, it immediately jumps to a specified location of memory called the *interrupt vector*, and executes the operation stored there. This is (usually) another jump operation, pointing to the subroutine that realizes the controller. This subroutine is called an *interrupt service routine*, or ISR. The ISR computes exactly the same thing as the polling loop, although there are some restrictions on the types of things an ISR can do. (For example, the ISR cannot make DOS calls, so it is difficult to write a string to the computer screen from the ISR.) When the ISR is finished, it issues the *return-from-interrupt* command, and the 80486 goes back to doing what it was doing before. In our labs, that is just a loop that polls the keyboard, waiting for a key to be pressed, indicating the end of the experiment. In more complex systems, this may be a program that continuously monitors the control system for faults, but is not timing sensitive.

The timing of the ISR is depicted in Figure 7. Note that the ISR, each time it is called, may take a different amount of time to complete, but the times at which it is called, i.e., the sample times, are uniform. Also, the interrupt-driven software has idle times when the processor is not computing the control law, but could be doing something else.

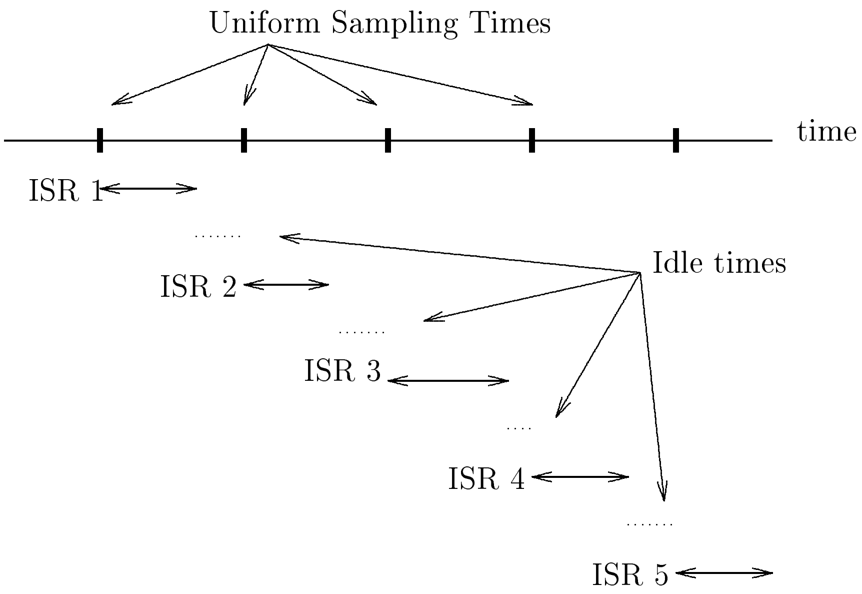

Figure 7: Timing of a interrupt-driven controller. Note the uniform sampling times, despite the fact that the ISR takes a variable amount of time to execute.

4.2 Real-Time Software Development

Writing an ISR is more difficult than writing a standard subroutine. The source code must do several things, including replacing the existing ISR vector with the new one, moving the ISR into memory, initializing the timer, and enabling the interrupt. When the program is finished, the old ISR vector is restored, and the program is removed from DOS memory. Most of this is rather mundane, so the core ISR software is distributed to you. A template for all three labs, called `lab.c`, is available on the WWW at

`http://www.control.toronto.edu/~bortoff/410.html.`

You should retrieve this software before going to Lab 1.

All three servomotor labs require modifications of this template. The modifications involve writing subroutines that implement desired reference trajectories, e.g., a step of 45 degrees, or implement the control law. However, the core code will remain unchanged. Although you will not have to write your own ISR, you should be able to read and understand the rest of code.

Borland's Turbo C compiler is installed on the PC network in GB 347, and should be used to compile all software. To compile, make your own directory on the C drive, and copy your version of `lab.c` into it. Lets assume your modified version is called `mylab1.c`. To compile the code, type

```
tcc mylab1
```

at the DOS prompt. This produces an executable, `mylab1.exe`, which you run by typing `mylab1` at the prompt. As distributed, `lab.c` issues no warnings or errors, so all warnings and errors will be due to your modifications :-).

You may edit the source code using one of several editors available on the network, e.g., `edit`. Alternatively, you can use the integrated Borland Turbo C environment to edit the source. From this environment, you can edit, compile and run the software. On-line help is useful. A common problem with the Borland integrated environment is improperly set the directory paths, which (incorrectly) default to the C: drive. The C libraries are on the networked F: drive, so you must change the library directory to F: from C:. Note: **You should come to lab with a floppy disk to save your work from the C: drive, because the C: drive is cleared after lab.**

4.3 A MATLAB Workstation

Starting MS Windows allows you to run MATLAB. MATLAB is a CAD program that simplifies the numerical aspects of controller design and system analysis. It can make root locus plots, Bode plots, and Nyquist plots. It can compute the bilinear transformation for Labs 2 and 3, and do pole placement for Lab 4. It can plot the experimental data from all the labs. In short, it is a very powerful tool that you will use often in ECE 410 and ECE 557.

Some of the more important commands are listed below. Typing "help command" will bring up a screen-full of information. So, if you don't know how to make a plot, type "help plot."

- **bilinear** does the bilinear transformation.
- **bode** plots a Bode plot.
- **cd** changes the current directory.
- **load** loads data, which is useful to plot experimental data.
- **lsim** simulates a continuous-time linear system.
- **margin** computes the gain and phase margins.
- **nyquist** plots a Nyquist plot.
- **plot** makes 2-dimensional plots.
- **rlocus** plots a root locus.
- **save** saves all variables in the environment to a file for later loading.
- **step** computes the step response of a linear system.
- **who** lists all defined variables.
- **quit** quits the program.