PID AUTO-TUNE CONTROL

A Practical Implementation

by

STEVEN BATES

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science in Engineering

in the

Electrical Engineering

Program

Robert H. Foulkes 6-3-94

Advisor Date

6/6/94

Dean of the Graduate School Date

YOUNGSTOWN STATE UNIVERSITY

June, 1994

ABSTRACT

PID AUTO-TUNE CONTROL

STEVEN BATES

MASTER OF SCIENCE IN ENGINEERING

YOUNGSTOWN STATE UNIVERSITY, 1994

This thesis investigates the design and implementation of a simplified standard PID controller, a method of tuning these types of controllers and the implementation of an auto-tuning controller. The implementation includes programming the control software in "C" language on a personal computer, interfacing the software and hardware to analog-to-digital, digital-to-analog converters, and programming a plant simulation on an analog computer. A method of manually tuning the PID controller, an explanation of the "C" programming code, the hardware configuration, and the automatic tuning of PID controllers are discussed and presented.

ACKNOWLEDGEMENTS

I wish to express gratitude to several people who have provided support and inspiration for this work. Dr. Robert H. Foulkes, Jr. imparted his extensive knowledge of control design and practical hardware implementation. Professor Samuel J. Skarote has contributed genius in programming "C", and has served on my thesis review committee. I am grateful to Dr. Salvatore R. Pansino for being my advisor throughout the entire course of study for the master degree and for serving on my thesis committee. His help in all areas of this endeavor have been invaluable.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF SYMBOLS

SYMBOL	DEFINITION	UNITS OR REFERENCE
"a"	Process amplitude or plant 2 feedback parameter	none
A2D	Analog-to-digital converter	none
"b"	Setpoint weighting factor	none
"d"	Amplitude of square wave in relay feedback test	volts
dB	Decibels	none
D(k)	Controller derivative term, discrete time	volts/sec
D(k-1)	Controller derivative term, previous value, discrete time	volts/sec
D2A	Digital-to-analog converter	none
e	Error signal	volts
e(t)	Error signal	volts
$e^{i(\pi+\phi)}$	Exponential representation of complex phasor, Nyquist plots	none
de/dt	Rate of change of error	volts/second
G(s)	Plant or process Laplace transfer function	none
G(jω)	Plant or process frequency transfer function	none

LIST OF SYMBOLS

SYMBOL	DEFINITION	UNITS OR REFERENCE
G_p	Plant or process transfer function	none
h	Sampling time variable	seconds
i	Imaginary number $\sqrt{(-1)}$	none
I(k)	Controller integral term, discrete time	none
I(k+1)	Controller integral term, next value, discrete time	none
I/P	Current-to-pressure converter	ma to psi
K_c	Proportional gain	none
K_u	Ultimate gain	none
QDR	Quarter decay ratio response	none
r_p	Plant or process gain	none
r_r	Regulator or controller gain	none
r_s	Overall system gain	none
r(k)	Controller setpoint, discrete time	% of range
P	Proportional Controller	none
PI	Proportional, integral controller	none
PID	Proportional, integral, derivative controller	none
P(k)	Controller proportional term, discrete time	% of range
PLC5	Programmable logic controller	none

LIST OF SYMBOLS

SYMBOL	DEFINITION	UNITS OR REFERENCE
N	Noise gain limiting constant	none
S	Overall system transfer function	none
SISO	Single input single output	none
TIC	Temperature indicating controller	none
T_d	Controller derivative time	seconds
T_i	Integral time parameter	seconds
TT	Temperature transmitter	degrees - ma
T_u	Ultimate period	seconds
u(t)	Control output signal	volts
u(k)	Controller output signal, discrete time	volts
y(t)	Process output signal	volts
y(k)	Process output signal, discrete time	volts
y(k-1)	Process output signal, previous value, discrete time	volts
α	Ratio of T_d/T_i	none
ϕ_p	Plant or process phase margin	degrees
ϕ_s	Overall system phase margin	degrees
ϕ_r	Regulator or controller phase margin	degrees
ω	Frequency	radians/sec

LIST OF FIGURES

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION AND BACKGROUND

I chose this subject for my thesis in order to learn more about the methods of tuning proportional, integral, derivative (PID) controllers. I work as a consulting engineer for Fluor Daniel and provide clients with control system design and startup services. As an example of this, I have worked for Frito Lay in their Dorito manufacturing plant in Perry, Georgia, starting up their kitchen automation. This automation consisted of a programmable controller (PLC5) system, color graphic operator interfaces, and instrumentation. Imbedded in the PLC5 system was PID software that required tuning to provide satisfactory control. An example of the use of this PID control is illustrated in figure 1. The process under control consisted of a vegetable fryer that maintained the vegetable oil at 365 degrees Fahrenheit. A steam valve was modulated to provide steam heating of the vegetable oil. The Dorito chips were conveyed into the vegetable oil and circulated around the fryer to a conveyor that slowly elevated them out of the oil, allowing the oil to drain from the chips. The chips then proceeded to the next processing stage. The controller had to maintain the oil temperature under the disturbances of room temperature makeup oil entering the fryer and also as the room temperature chips entered the fryer. I had no method at my disposal to properly tune this process and

therefore had to manually tune the controller for an acceptable control response.

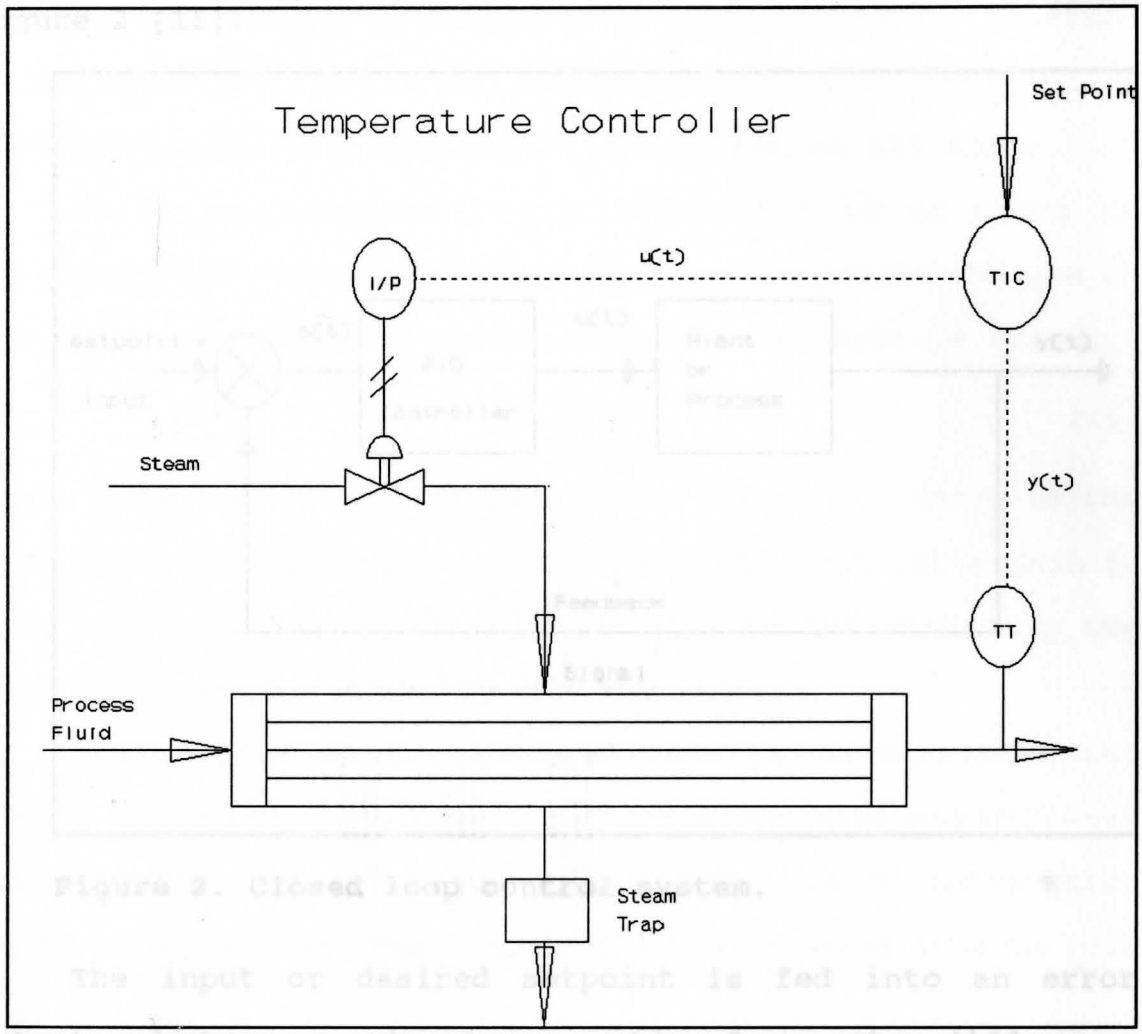


Figure 1. Vegetable fryer.

Proportional, integral, derivative controllers are used extensively in industry in all types of systems. PID controllers are sufficient for many control problems especially processes where there are modest performance requirements. Even though PID controllers are well known, they are often poorly tuned [1], [2], [4].

This thesis is about the design and tuning of a

proportional, integral, derivative (PID) type of closed loop control system. A typical closed loop system is illustrated in figure 2 [11].

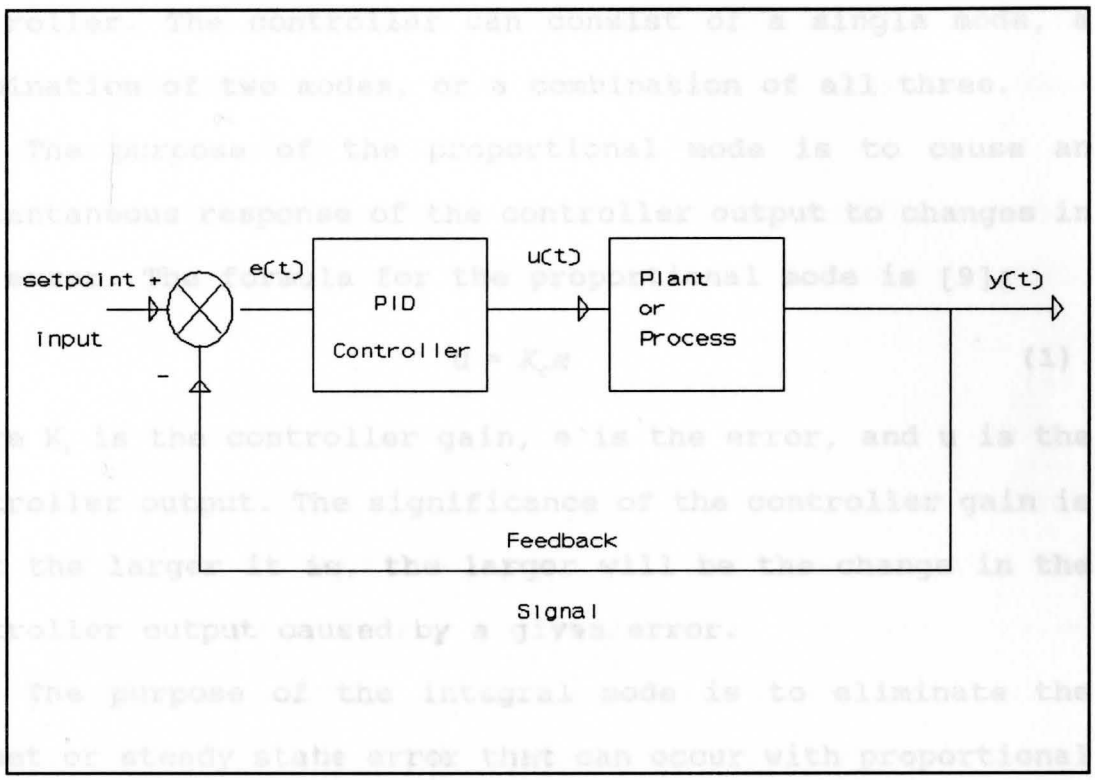


Figure 2. Closed loop control system.

The input or desired setpoint is fed into an error detector, where a signal proportional to the difference between the input and output is generated. The controller drives the plant input to produce an error of zero. Any differences between the actual output and desired output (setpoint) are automatically corrected in a closed-loop control system. A typical example would be a temperature controller for a room. The desired temperature would be the setpoint, the actual temperature would be the measured output and the difference between the two signals would be the error.

The three modes of control of feedback are proportional, integral, and derivative. Each of these modes introduces an adjustable parameter into the tuning operation of the feedback controller. The controller can consist of a single mode, a combination of two modes, or a combination of all three.

The purpose of the proportional mode is to cause an instantaneous response of the controller output to changes in the error. The formula for the proportional mode is [9]:

$$u = K_c e \tag{1}$$

where K_c is the controller gain, e is the error, and u is the controller output. The significance of the controller gain is that the larger it is, the larger will be the change in the controller output caused by a given error.

The purpose of the integral mode is to eliminate the offset or steady state error that can occur with proportional control used alone. The controller does this by integrating the error over time. The formula for the integral mode is [9]:

$$u = \frac{K_c}{T_i}\int e dt \tag{2}$$

where t is time and T_i is the integral tuning parameter for the integral mode. The smaller the integral time T_i the faster the controller output will change for a given error. Although integral mode is effective in eliminating offset, it is slower than the proportional mode because it must act over a period of time.

The derivative mode responds to the rate of change of

error with time. The formula for the derivative action is [9]:

$$u = K_c T_d \frac{de}{dt} \tag{3}$$

where T_d is the derivative time and de/dt is the rate of change of the error. Note that the derivative acts only when the error is changing with time.

The "textbook" equation for the PID algorithm has the following form [10], [11], [12].

$$u(t) = K_c \left[e(t) + \frac{1}{T_i} \int e(t)\, dt + T_d \frac{de}{dt} \right] \tag{4}$$

where u is the control variable and e is the control error, the difference between the setpoint and the output. The control variable is the sum of three terms: A P-term, (which is proportional to error), an I-term (which is proportional to integral of error), and the D-term (which is proportional to the derivative of error. Again, the controller parameters are the proportional gain K_c, the integral time constant Ti, and the derivative time constant Td.

PID controllers are implemented in many forms, such as, stand alone controllers that can handle multiple loops and as software in programmable controllers. These controllers are very "robust" and have been around for a long time, originally designed using pneumatics, later using electron tubes, transistors, integrated circuits, and now microprocessors [1], [14].

The adjustment or tuning of single input single output

(SISO) controllers is one of the least understood, poorly practiced, yet extremely important aspects of the application of automatic control theory. In many control rooms the derivative action is switched off for the simple reason that it is difficult to tune properly [8].

The first problem encountered in tuning controllers is to define "good" control. This unfortunately differs from process to process. It would seem that the best way to present this subject in detail would be to discuss only the best way, but there is no general agreement as to which method is the best method of tuning controllers. Some methods lean heavily on experience while others rely on mathematical considerations.

One of the key concepts of this project is the assumption that little or nothing is known about the transfer function of the actual plant and this tuning process must find optimum values for the controller without the benefit of mathematical analysis. This includes the plants that have slowly changing parameters where the PID control would slowly degrade over time if there were no way to automate the parameter adjustment features.

PID CONTROLLER TUNING

One of the methods proposed for tuning controllers was the ultimate method, reported by Ziegler and Nichols in 1942 [1], [4], [9]. The term "ultimate" was attached to the name of this method because it requires the determination of the

ultimate gain and ultimate period of a process. The ultimate gain is defined as the gain of a proportional controller at which the control loop oscillates with constant amplitude, and the ultimate period is the period of the oscillations. By its definition it can be deduced that the ultimate gain is the gain at which the control loop is at the threshold of instability. At gains below the ultimate gain the control loop signals will oscillate with decreasing amplitude, as shown in figure 3; at gains above the ultimate, the amplitude of the oscillations will increase with time. It is therefore, very important when determining the ultimate gain of an actual feedback control loop to ensure that it is not exceeded by much or the result would be a violently unstable system. The procedure for obtaining the ultimate gain and period is carried out with the controller in "auto" (automatic output, that is, with the loop closed) and with the integral and derivative modes turned off. To do this the following steps are recommended [4]:

1. Set the integral and derivative modes to off by setting the parameters to zero.
2. Set the proportional gain K_c of the controller to some arbitrary value.
3. Run the controller in automatic and watch the response on the oscilloscope.
4. Carefully increase the proportional gain of the controller in steps until the output of the process oscillates without a decreasing amplitude (or

approximately constant). Record the K_c of the controller at this point as the ultimate gain Ku. Measure and record the period of oscillations as Tu.

Two methods of tuning the controller were proposed by Ziegler and Nichols. The first was for a specific response in the time domain: the quarter-decay ratio response (QDR). The second was a frequency response method based upon a simple characterization of the process transfer function (the relationship between the process input and output) using the Nyquist curve. The QDR response is illustrated in figure 3 for a step change in the setpoint. Its characteristic is that each oscillation has an amplitude that is one fourth that of the previous oscillation.

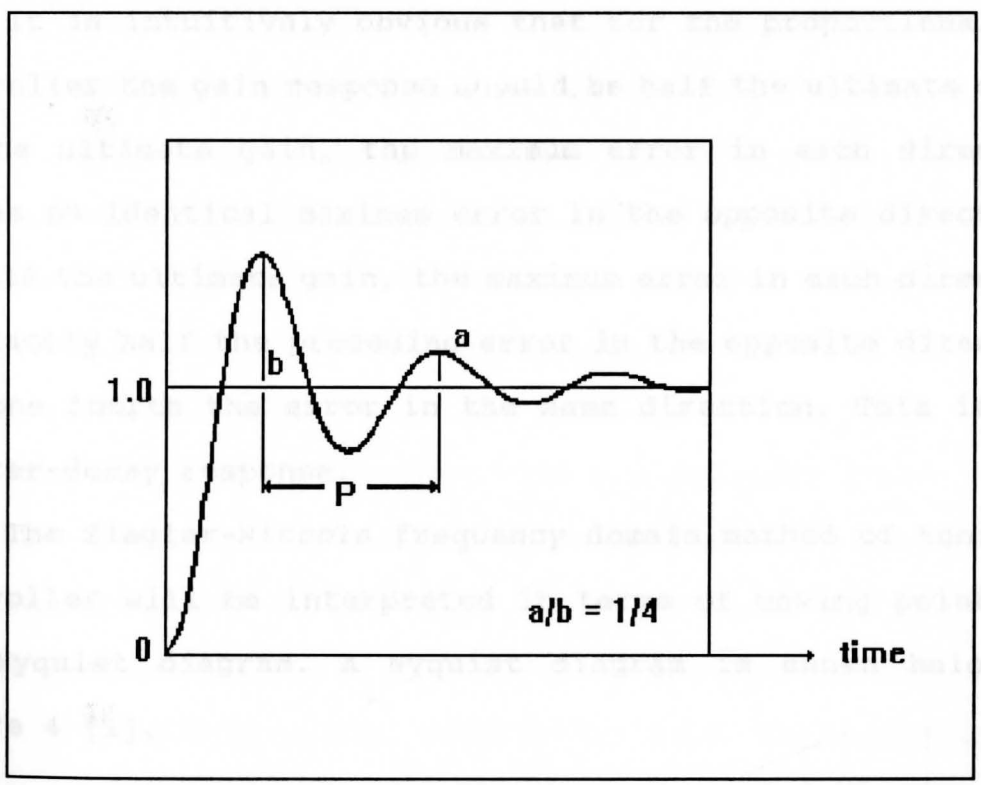


Figure 3. Quarter decay ratio response.

The formulas proposed by Ziegler and Nichols for calculating the QDR tuning parameters of P, PI, and PID controllers from the ultimate gain K_u and the ultimate period T_u are summarized in table 1 below [4].

TABLE 1. Quarter decay tuning formulas.

Controller	Gain	Integral Time	Derivative Time
P	$K_c = .5\ K_u$	-	-
PI	$K_u = .45\ K_u$	$T_i = T_u / 1.2$	-
PID	$K_c = .75\ K_u$	$T_i = T_u / 1.6$	$T_d = T_u / 10$

It is intuitively obvious that for the proportional (P) controller the gain response should be half the ultimate gain. At the ultimate gain, the maximum error in each direction causes an identical maximum error in the opposite direction. At half the ultimate gain, the maximum error in each direction is exactly half the preceding error in the opposite direction and one fourth the error in the same direction. This is the quarter-decay response.

The Ziegler-Nichols frequency domain method of tuning a controller will be interpreted in terms of moving points on the Nyquist diagram. A Nyquist diagram is shown below in figure 4 [1].

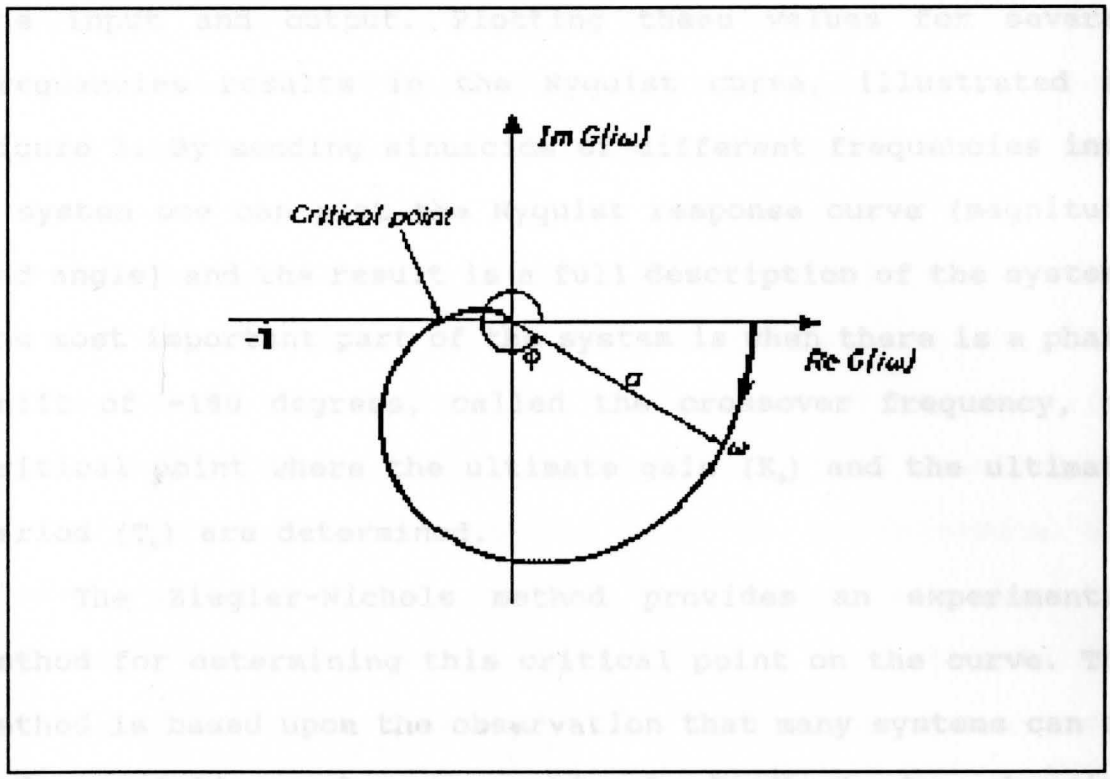


Figure 4. Nyquist diagram.

The Nyquist curve is a polar plot of the magnitude and phase of the open-loop transfer function of the process in the complex plane. The tuning method starts with the determination of the point $(-1/K_u, 0)$ where the Nyquist curve intersects the negative real axis. The concepts behind this method can be explained as follows.

Consider a linear process with a sinusoidal input. After a transient period, the output of the process is a sinusoid of the same frequency as the input. Only the phase and magnitude of the output will be different from the input. This means that under steady state conditions only two numbers are required to describe the output, the quotient "r" between the input and the output amplitude and the phase shift "a" between

the input and output. Plotting these values for several frequencies results in the Nyquist curve, illustrated in figure 3. By sending sinusoids of different frequencies into a system one can plot the Nyquist response curve (magnitude and angle) and the result is a full description of the system. The most important part of the system is when there is a phase shift of -180 degrees, called the crossover frequency, or critical point where the ultimate gain (K_u) and the ultimate period (T_u) are determined.

The Ziegler-Nichols method provides an experimental method for determining this critical point on the curve. The method is based upon the observation that many systems can be made unstable under proportional feedback by choosing sufficiently high gain in the proportional feedback. The feedback control loop is used to generate sinusoids by increasing the gain to where the process oscillates. The control variable and the measured variable are then sinusoids with a phase shift of -180 degrees and are therefore are related by [1]

$$u = -ky \tag{5}$$

because of the proportional feedback. The gain around the loop must be unity in order to maintain the oscillation, [1]

$$K_u|G(j\omega)| = 1 \tag{6}$$

where the gain, which brings the system to the stability limit is called the ultimate gain (Ku). The advantages of the Ziegler-Nichols method is that it is based upon a simple

experiment. The disadvantage is that some processes cannot tolerate this kind of disturbance without becoming dangerous and is therefore not a useful method for auto-tuning of many processes.

Another method for tuning PID controllers uses the process reaction curve, which requires an open loop step test on the process [4]. Process gain, time constant and dead time can be determined from the results of this test. The purpose of an open loop step response test is to determine the transfer function of the process. This method determines the process dynamic parameters by performing a test with the controller in "manual output". This thesis does not present this test and will not be discussed further.

Finally, with the advent of microprocessors new methods are starting to appear in products offered on the market with auto-tuning capabilities as well as some adaptive and heuristic type controllers [3], [4], [5], [7], [13].

The project includes the following:

- Design and implement a simple PID controller.
- Design the software in "C" on a microcomputer.
- Provide a user interface for adjusting parameters.
- Test and adjust the controller with the ultimate method.
- Design and test an auto-tuning software routine.

Figure 5 shows a block diagram of the hardware.

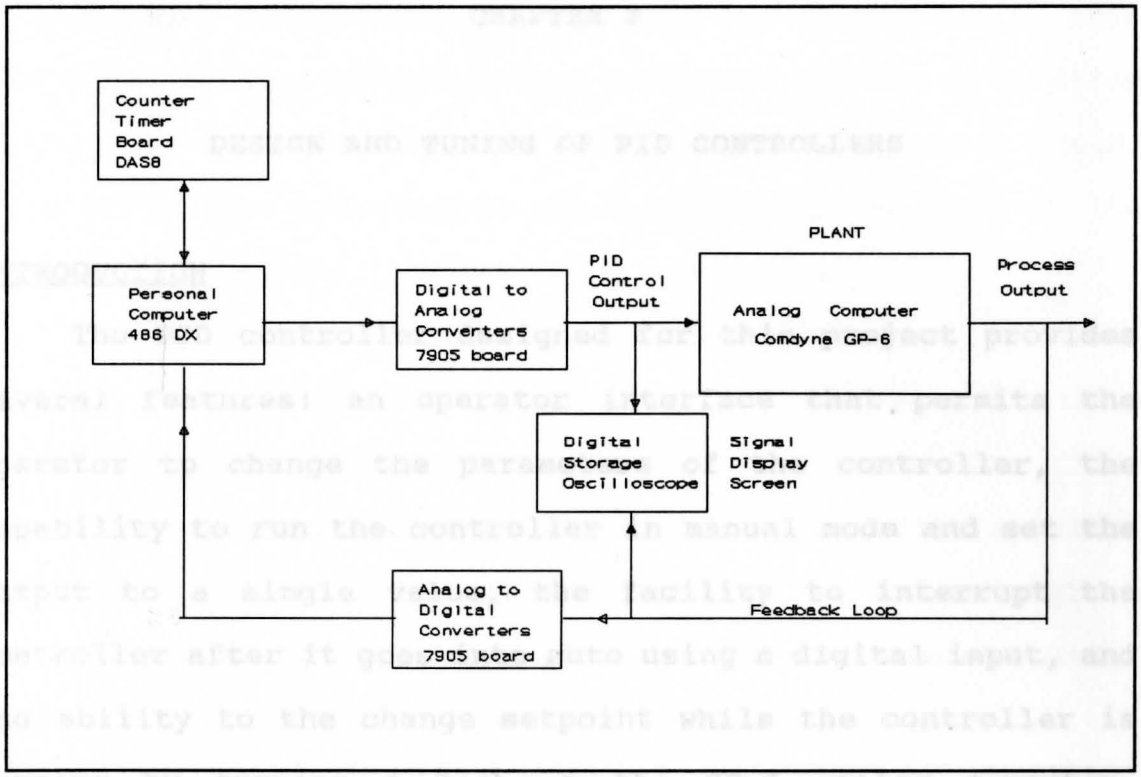


Figure 5. PID controller diagram.

CHAPTER 2

DESIGN AND TUNING OF PID CONTROLLERS

INTRODUCTION

The PID controller designed for this project provides several features: an operator interface that permits the operator to change the parameters of the controller, the capability to run the controller in manual mode and set the output to a single value, the facility to interrupt the controller after it goes into auto using a digital input, and the ability to the change setpoint while the controller is running by turning a knob on the GP-6 analog computer, allowing observation of the response to setpoint changes (see appendix A for additional details of the hardware).

OPERATOR INTERFACE

The operator interface was programmed on a personal computer using "C" language and provides the following selections:

Enter 1 if you wish to enter parameters
Enter 2 if you wish to run in manual
Enter 3 if you wish to run in auto
Enter 4 if you wish to run auto tune
Enter 5 if you wish to quit the program

Running in manual mode means the operator manually sets the value of the control output variable and the controller holds that value constant until the operator changes it again. This allows the operator to run a step response test of the process if desired.

The auto mode provides a PID controller that runs the digital control algorithm described in this chapter in a closed loop feedback control system.

The auto-tune selection allows the operator to initiate the PID controller's automatic tuning feature, that determines the parameters of the PID controller for the process under control.

PROGRAM STRUCTURE

The operator may change the following parameters that the software uses to control the operation of the controller. Below are listed the parameters and their default values.

```
setpoint = 0      initial setpoint value % range
control_uk = 39   initial u(k) control variable % range
control_uk1 = 0   initial u(k-1) control variable % range
output_yk = 0     initial y(k) output variable % range
output_yk1 = 0    initial y(k-1) output variable % range
pro_gain_Kc = .2  initial proportional gain variable
parameter_b = 1   initial setpoint weight factor
integral_Ti = 10  initial integral time variable
integral_Ik = 0   initial I(k) integral control variable
```

```
integral_Ik1 = 0 initial I(k-1) integral control variable
sample_h = 0.2   initial sample period in seconds
noise_N = 1      initial noise level variable
derivative_Td = 1 derivative time variable
derivative_Dk = 0 derivative D(k) control variable
derivative_Dk1 = 0 derivative D(k-1) control variable
relay_step = 10   relay step size variable % range
output_addr = 788 control output address (LDAC = 788)
process_addr = 2  process output address (OP AMP2 = 2)
setpoint_A2D = 8  analog input A4 = 8 setpoint ctl
```

The controller is designed for future expansion if desired. Some of the variables with "(k-1)" are not used, and therefore, can be utilized by others investigating some of the ideas presented in chapter 4 - "recommendations for future study". The controller design provides flexibility in its design with the ability to modify the output addresses and multiplexer addresses from which and to which signals are processed by the computer [16].

The PID controller is designed with a global PID data structure that contains all the parameters for controlling the process. Each function that is called by other functions utilizes this data structure and affects those variables it needs to affect.

The following functions are used in the design of the controller.

data_entry()	get parameters from operator
manual()	manual control mode function
automatic()	auto control mode function
relay_test()	run relay response test
ini_timer()	initialize timer function
ck_timer()	check for timeout function
A2D_input()	update value of output A2D
D2A_output()	control output function D2A
display_data()	manual ctl display variables function

Appendix B contains a complete listing of the "C" program written for this project. A brief discussion of some of the above functions follows.

The function "ini_timer()", as its name implies initializes the timing for the controller. The software actually communicates with the DAS8 board which has an Intel 8254 timer/counter providing 3 x 16-bit count down registers, deriving its clock from the IBM PC system clock. These counters are used in a unique way to detect the rising edge of the period for control (see appendix A for further details). The counters are configured in such a way that the count must be loaded as two bytes, requiring the software to break the word into parts for loading into the registers. This is accomplished by masking and shifting (division by 256).

The function "ck_timer()", checks the DAS8 board for the rising edge of the clock period counter pulse (see appendix A for details) and returns a one (1) if the timeout is true and a zero (0) if it is not true.

The "A2D_input()" communicates with the 7905 A2D / D2A interface board, which is connected to the Comdyna GP-6 Analog Computer. This function (A2D_input) uses the address stored in the PID structure discussed above (operator entered parameter) as an indirect pointer to the output address of the plant for the multiplexer to control the analog-to-digital (A2D) conversion. In the process of designing this software it was determined that every time the multiplexer address was changed that a minimum time for settling was required before starting a conversion, as bad results would be obtained otherwise. The following code provided this delay,

```
/* delay for sample & hold amp */
while(count < 80)
{
count ++;
}
```

A count of 70 was found to be insufficient for accurate results.

A similar routine was used, "D2A_output()" , to send digital data to the 7905 board for digital-to-analog conversion (D2A). The signal appears on the Model 767 Analog / Digital Position Control Panel at one of the D2A converter outputs. This output was wired to the input of the plant

simulator on the Comdyna GP-6 Analog Computer as the control signal.

CONTROLLER DESIGN EQUATIONS

A number of modifications to the "textbook" equation have to be made to provide a digital implementation of the PID controller. These changes are detailed below.

The controller equations utilized in this design were selected from a source in the bibliography [1]. The control law utilized in this controller is: u = P(k) + I(k) + D(k), where P is the proportional term, I the integral term, and D the derivative term.

The proportional term has an equation as follows:

$$P(k) = K_c[br(k) - y(k)] \qquad (7)$$

where r(k) is the sampled setpoint value, y(k) is the sampled process output, K_c is the proportional gain contained in the PID structure discussed above. The parameter b provides an additional degree of freedom used to provide a different control response to load and setpoint changes. This particular parameter was not studied in this project.

The integral term is calculated from the equation:

$$I(k+1) = I(k) + \frac{K_c h}{T_i} e(k) \qquad (8)$$

Note that the "next" value of the integral term is calculated here. "e(k)" is the sampled value of the difference between the setpoint and the current output, h is the sampling time in

seconds, T_i is the integral time in seconds, and I(k) is the current value of the integral term.

The derivative term is calculated from the following equation:

$$D(k) = \frac{2T_d - hN}{2T_d + Nh} D(k-1) - \frac{2K_c N T_d}{2T_d + hN} [y(k) - y(k-1)] \qquad (9)$$

The above is Tustin's algorithm [1] that is used most often in practical controllers and is quite close to the continuous time case. T_d is the derivative time in seconds, N is a gain limiting constant to limit the noise amplification to N. Note that the derivative is not calculated from the error between the setpoint and the output as is usually shown in the "text book" versions of PID software. This is because of the problem generated by abrupt changes in the setpoint the de/dt term would be very large. Therefore it is common practice to use only the process output to apply the derivative term.

CHAPTER 3 IMPLEMENTATION AND EXPERIMENTAL RESULTS

AUTO-TUNE CONTROLLER DESIGN

INTRODUCTION

Since the Ziegler-Nichols method of determining PID controller parameters has limitations to processes that can be disturbed without causing dangerous or other unacceptable conditions, another method has been proposed [1, 8] for determining these parameters, that of applying a square wave to the process input or control and using relay feedback to generate a small oscillation.

CONCEPTS

Figure 6 illustrates the controller block diagram for the relay control.

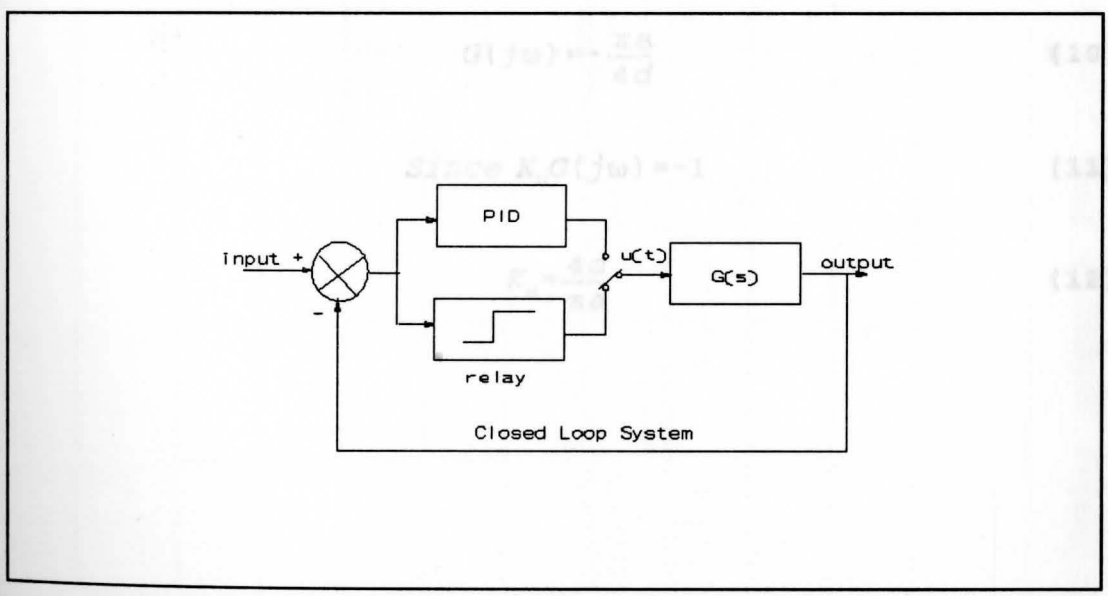


Figure 6. Relay control diagram.

Refer to the figure 7. For many systems the application of a square wave input will produce a nearly sinusoidal process output variation. Notice in the plot that the process input and output are out of phase and that the amplitude of the oscillation is proportional to the amplitude of the relay amplitude.

The relay feedback method is based upon generating an oscillation in the process output that is 180 degrees out of phase with a square wave input causing the response. If we can assume that the process attenuates the higher order harmonics of the square wave, then the first harmonic of the Fourier series expansion of the relay input square wave, is a sinusoid with an amplitude of $4d/\pi$. If the amplitude of the process response is "a", then the process gain and ultimate gain are [1]

$$G(j\omega) = -\frac{\pi a}{4d} \tag{10}$$

$$Since\ K_u G(j\omega) = -1 \tag{11}$$

$$K_u = \frac{4d}{\pi a} \tag{12}$$

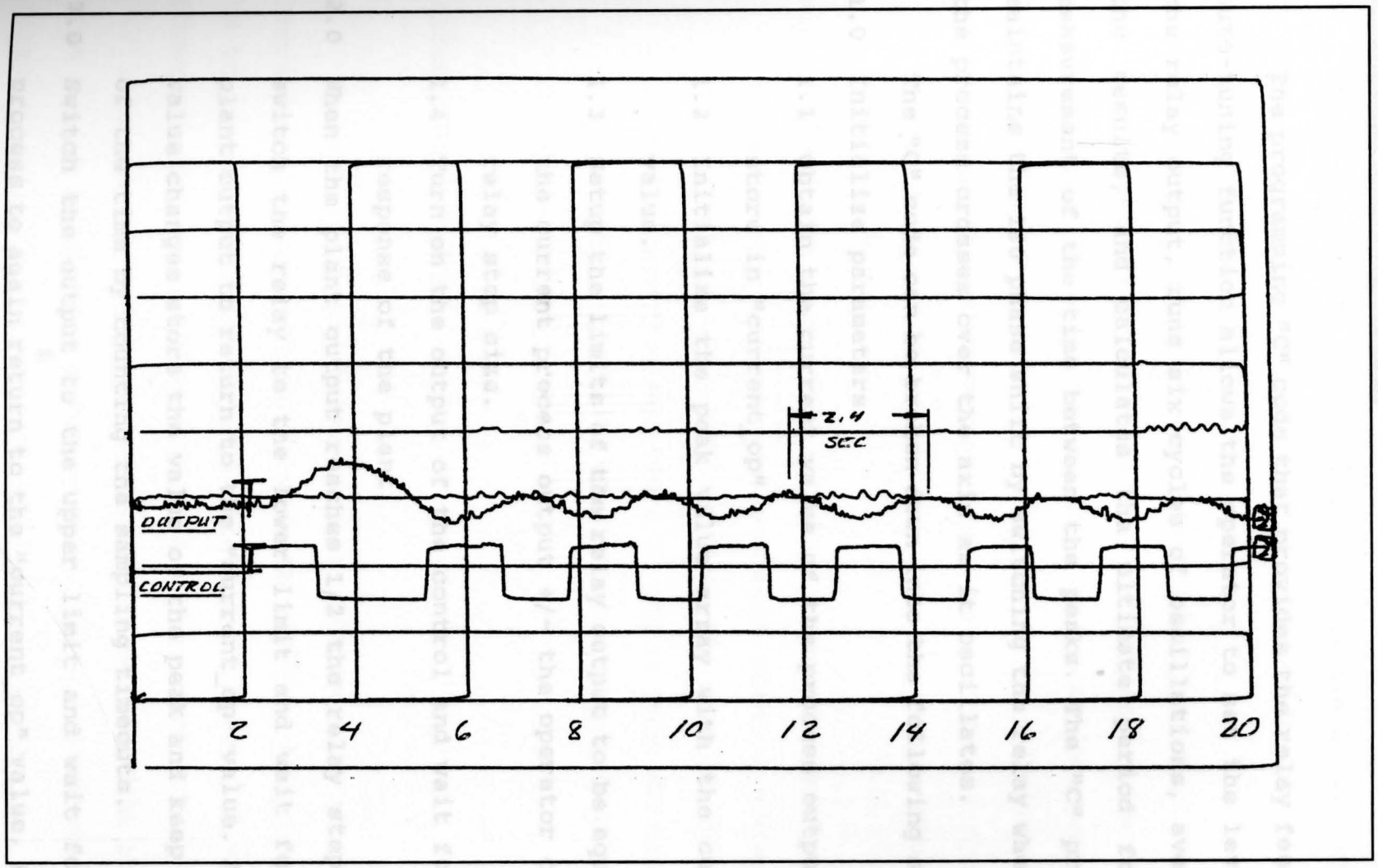


Figure 7. Relay output @ 20%, plot 1, plant 1. (K_u = 7.59, T_u = 2.40, output = 2 v/division, control = 5 v/division, vertical axis - volts, horizontal axis - seconds.)

CONTROLLER SOFTWARE DESIGN

The programming "C" code that provides the relay feedback auto-tuning function allows the operator to set the level of the relay output, runs six cycles of oscillations, averages the results, and calculates the ultimate period from a measurement of the time between the peaks. The "C" program maintains the 180 phase shift by switching the relay whenever the process crosses over the axis as it oscillates.

The "C" code can be broken down into the following steps.

1.0 Initialize parameters

- 1.1 Obtain the current value of the process output and store in "current_op".
- 1.2 Initialize the peak value array with the current value.
- 1.3 Setup the limits of the relay output to be equal to the current process output +/- the operator chosen relay step size.
- 1.4 Turn on the output of the control and wait for the response of the plant.

2.0 When the plant output reaches 1/2 the relay step size switch the relay to the lower limit and wait for the plant output to return to the "current_op" value. As the value changes store the value of the peak and keep track of the time by counting the sampling timeouts.

3.0 Switch the output to the upper limit and wait for the process to again return to the "current_op" value. As the

value changes store the value of the peak (neg peak) and keep track of the time by counting the sampling timeouts. When the process output returns to the "current_op" value goto step 4.0.

4.0 Switch the output to the lower limit and wait for the process to again return to the "current_op" value. As the value changes store the value of the peak (pos peak) and keep track of the time by counting the sampling timeouts. Increment a cycle counter, check to see if this cycle is the last (= 6) if not go to step 4.0 above. Otherwise goto step 5.0.

5.0 Analyze the data. The computer takes the peak values stored during the run and averages them, averages the periods between all peaks and uses the data to calculate the parameters from equations 10 and 11 (see the appendix B for additional details of the actual implementation of the software).

EXPERIMENTAL RESULTS

In appendix A two analog computer diagrams are shown that represent plant 1 and plant 2 used to simulate two processes to be tested under PID control. Figures 7 through 16 are for plant 1 and figures 17 through 30 are for plant 2.

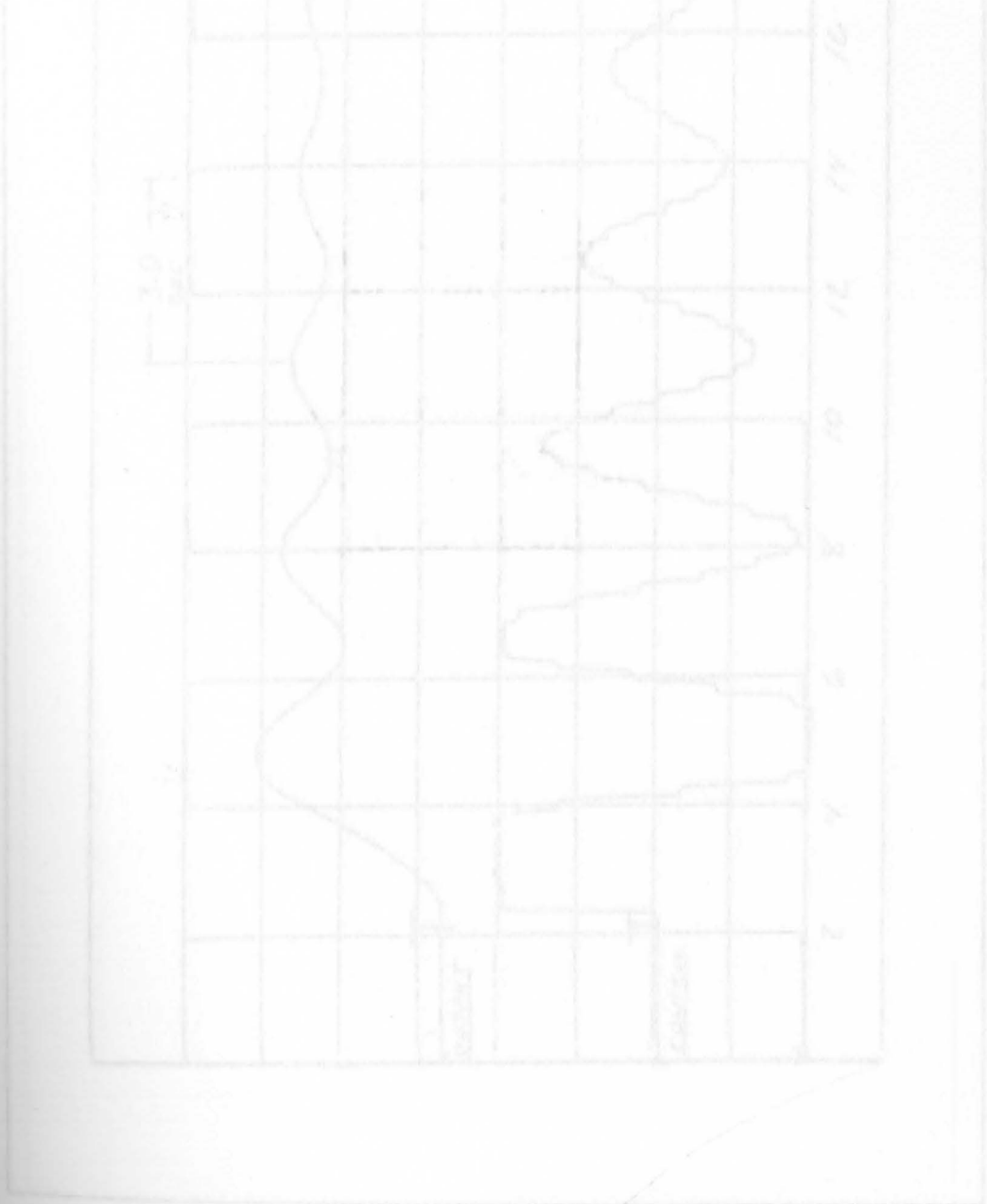
The 1st plant represents an open loop unstable process that has an integrator on the output. Table 2 summarizes the data:

Table 2. Plant 1 data.

Fig.	Control	Settling Time Sec	K_c/K_u	T_i/T_u	T_d
7	Relay 20%	N/A	/7.6	/2.4	
8	P	> 20	5.0	/3.0	/0.0
9	P	> 20	6.0	/2.8	0.0
10	P	>> 20	7.0	/2.6	0.0
11	PID auto-tune	12	2.6	2.7	.48
12	P	15	2.6	0.0	0.0
13	Relay 30%	N/A	/9.3	/2.2	
14	QRD P	20	3.8	0.0	0.0
15	QRD PI	>>20	3.5	2.0	0.0
16	QRD PID	20	5.8	1.5	.24

The first three plots represent an attempt to find the ultimate period of this process by varying the gain until a constant oscillation is obtained. The figure 8 was produced by the PID controller using a gain of 5, shows a damped response from the plant settling out in greater than 20 seconds. Figure 9 was produced from the controller with a gain of 6 and as can

be observed the damping is less but still prominent. Figure 10 is with plot of a gain of 7, where it is extremely difficult to tell if damping is occurring. This appears to be approximately the ultimate gain we are looking for from the above description. From the graphs the period is seen to be decreasing from 3 seconds, to 2.8 seconds, and finally to 2.6 seconds at the final gain of 7.



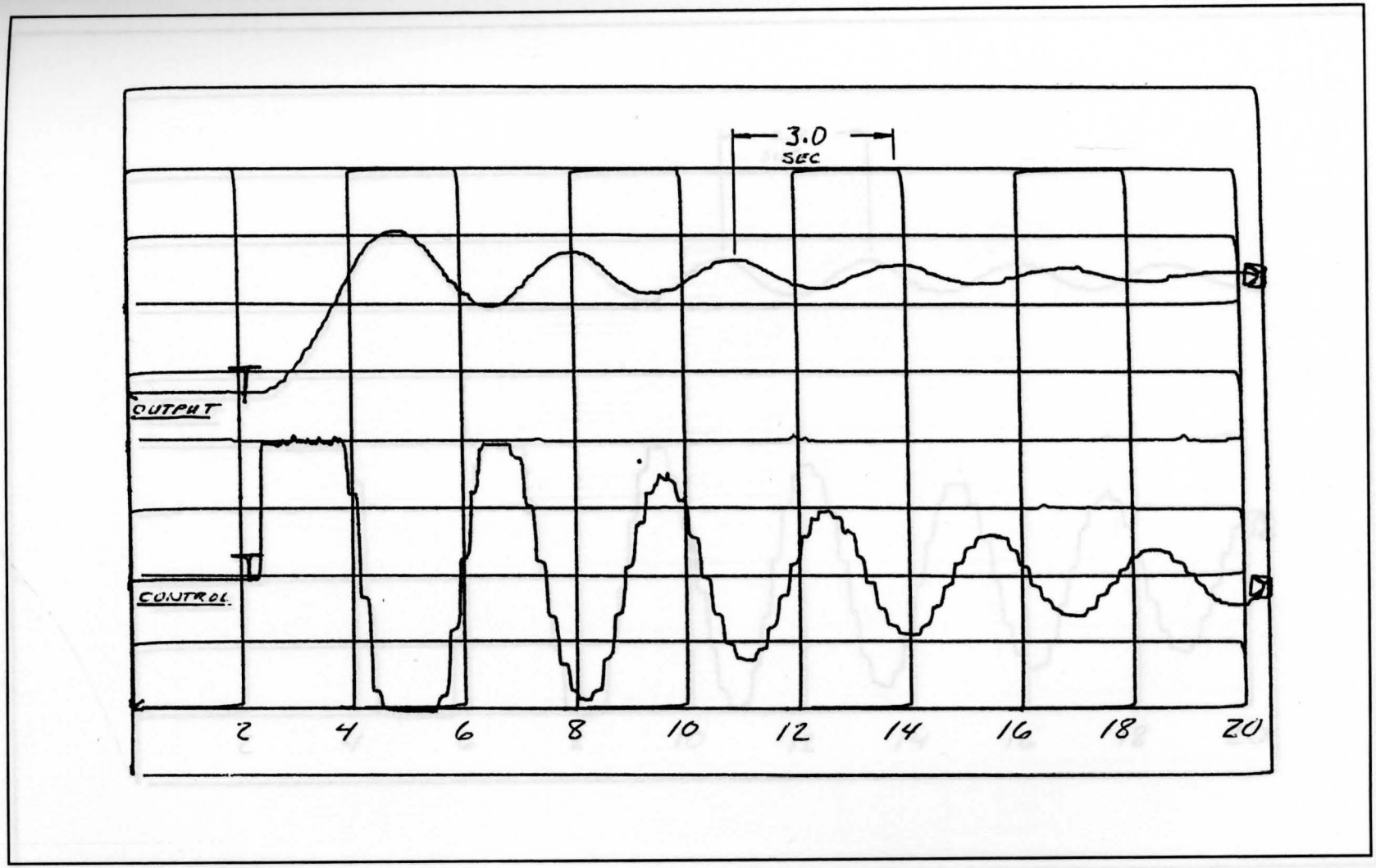


Figure 8. P controller plot 2, gain of 5, plant 1. (T_u = 3.0, setpoint = 2.6, output = 5 v/division, control = 5 v/division, vertical axis - volts, horizontal axis - seconds).

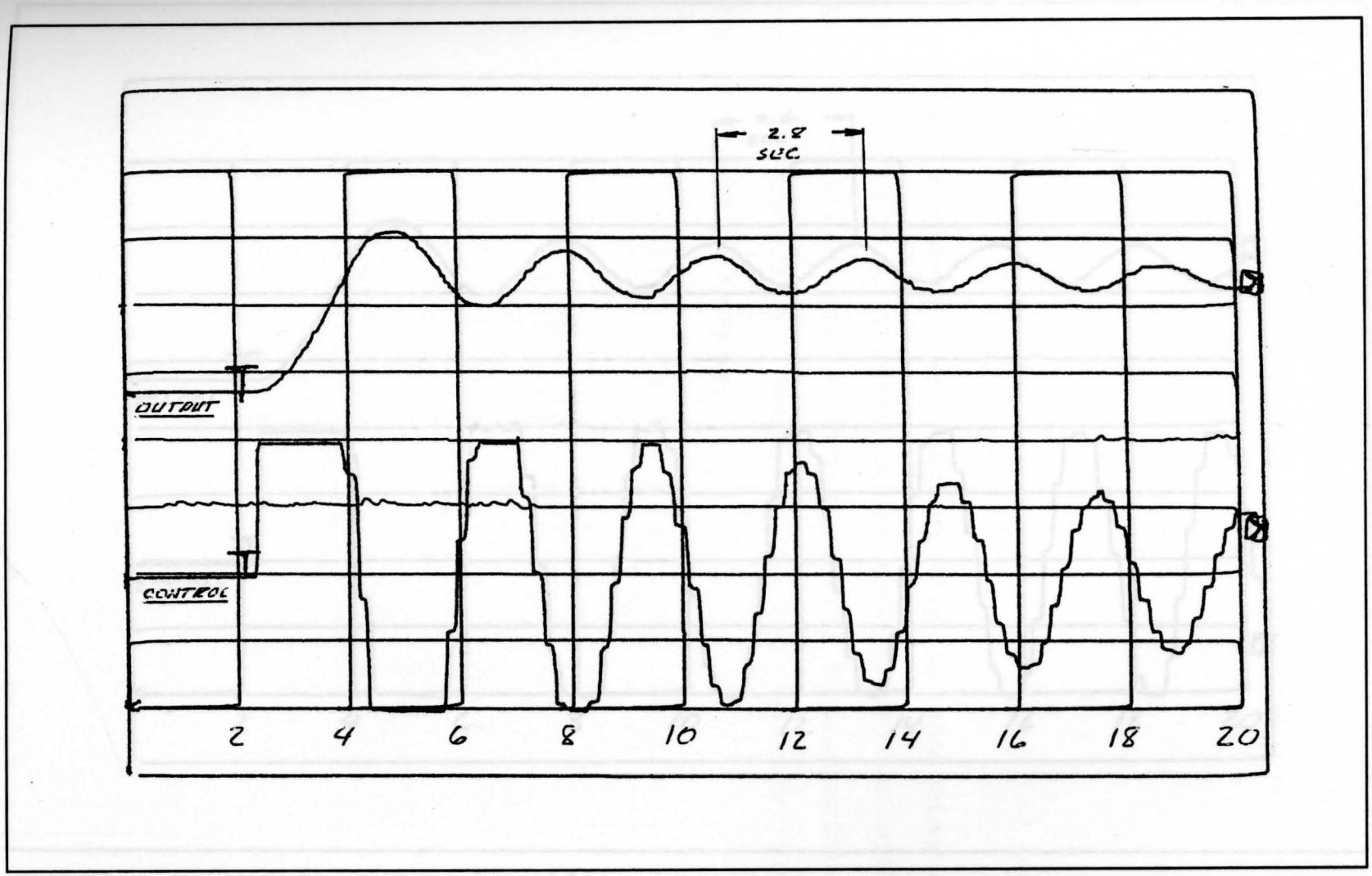


Figure 9. P controller plot 3, gain of 6, plant 1. (T_u = 2.8, setpoint = 2.6, output = 5 v/division, control = 5 v/division, vertical axis - volts, horizontal axis - seconds).

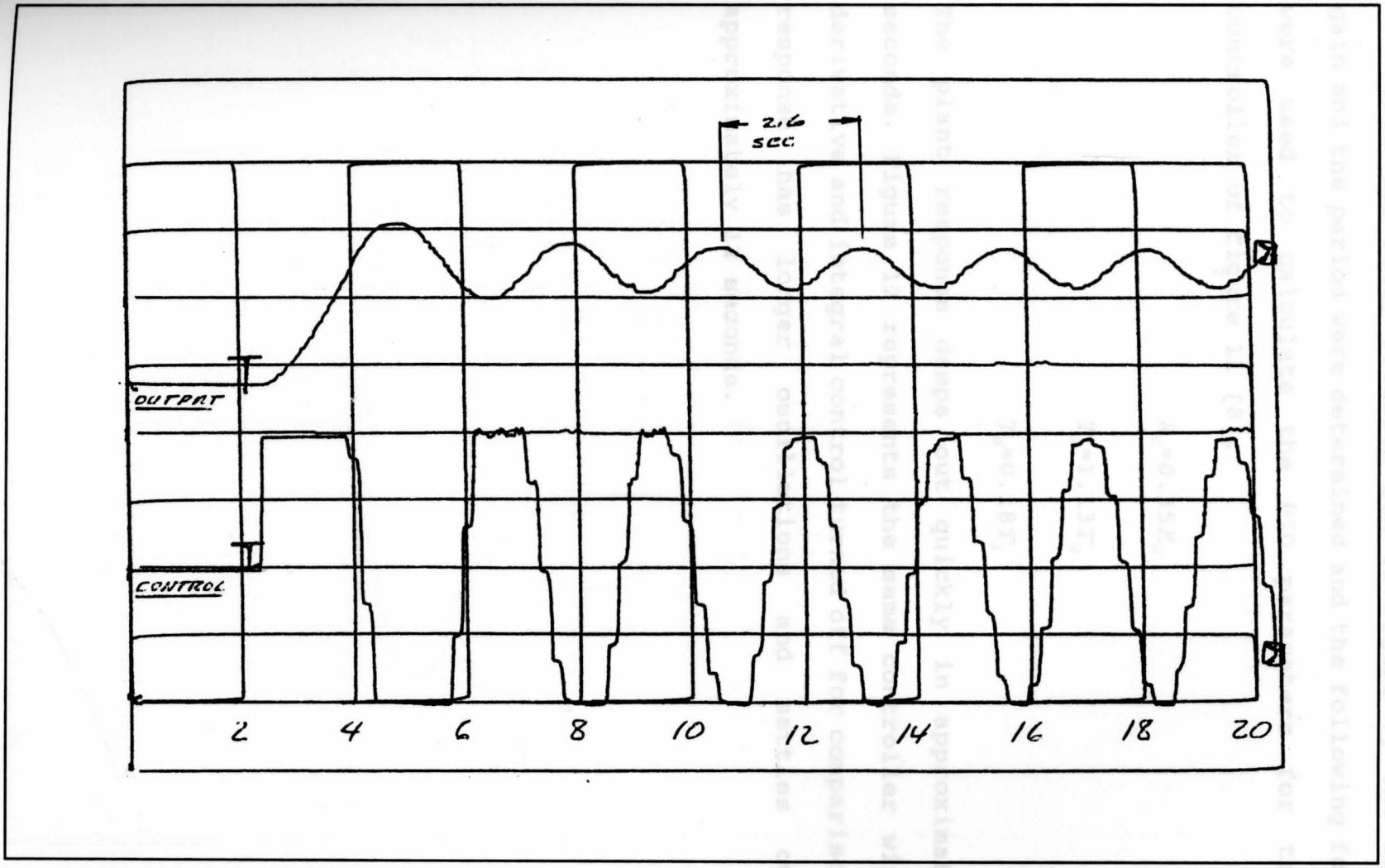


Figure 10. P controller plot 4, gain of 7, plant 1. (T_u = 2.6, setpoint = 2.6, output = 5 v/division, control = 5 v/division, vertical axis - volts, horizontal axis - seconds).

From the relay feedback run of figure 7, the ultimate gain and the period were determined and the following formulas were used to calculate the PID parameters for the PID controller of figure 11 [8].

$$k_c=0.35K_u \quad \textbf{(13)}$$

$$T_i=1.13T_u \quad \textbf{(14)}$$

$$T_d=0.18T_i \quad \textbf{(15)}$$

The plant response damps out quickly in approximately 8 seconds. Figure 12 represents the same controller with the derivative and integral control turned off for comparison. The response has longer oscillations and settles out in approximately 12 seconds.

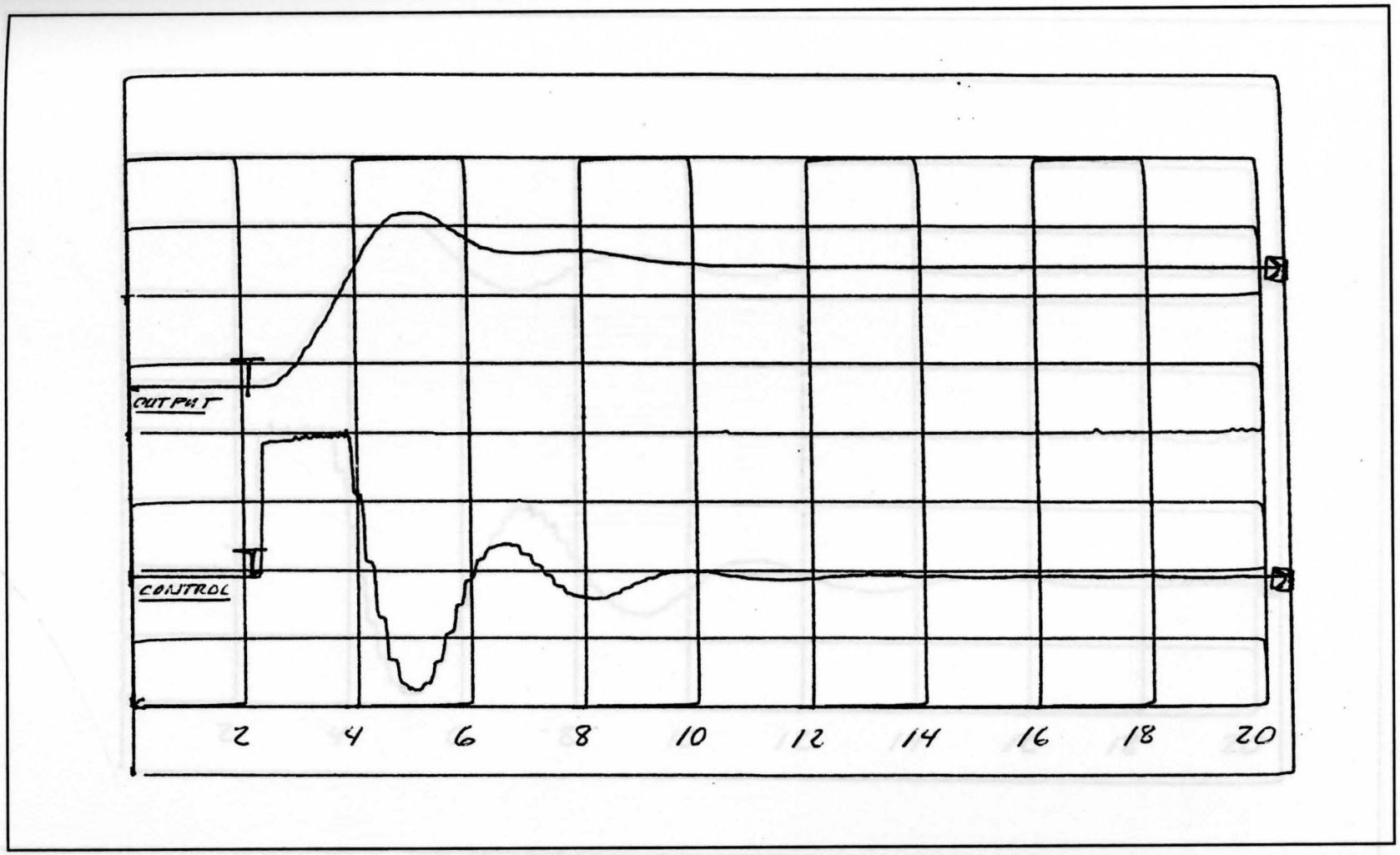


Figure 11. PID auto-tune controller plot 5, plant 1. (K_c = 2.65, T_i = 2.7, T_d = 0.49, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

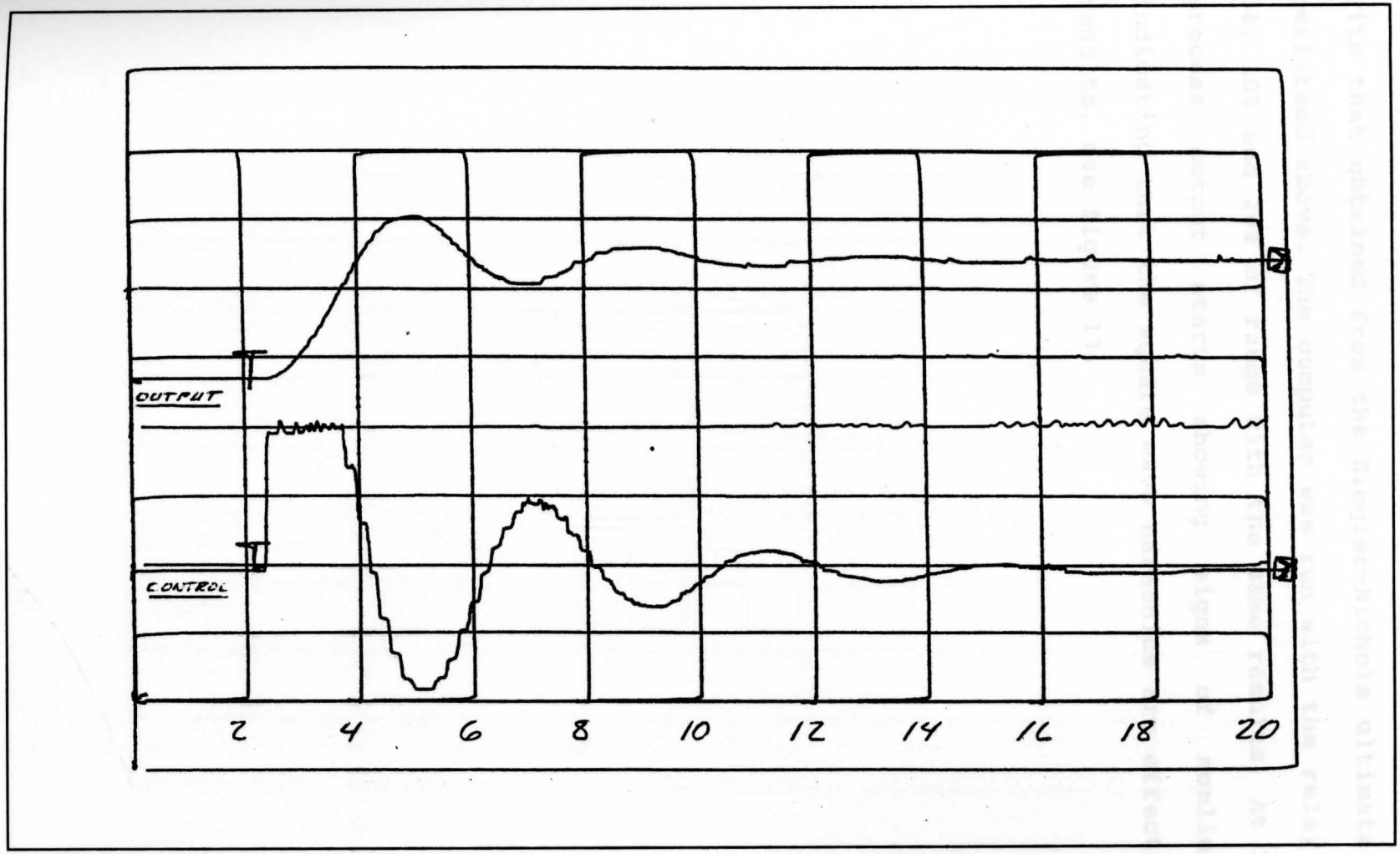


Figure 12. P controller plot 6, plant 1. (K_c = 2.65, T_i = 0.0, T_d = 0.0, setpoint = 2.6, output = 5 v/division, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

The results of the relay auto-tuning agree very closely with that obtained from the Ziegler-Nichols ultimate method performed above. The computer was run with the relay set at 5%, 10% and 20% of range with the same results. At 30% the process output starts showing signs of nonlinearity, indicating that the square wave harmonics are effecting the results, see figure 13.

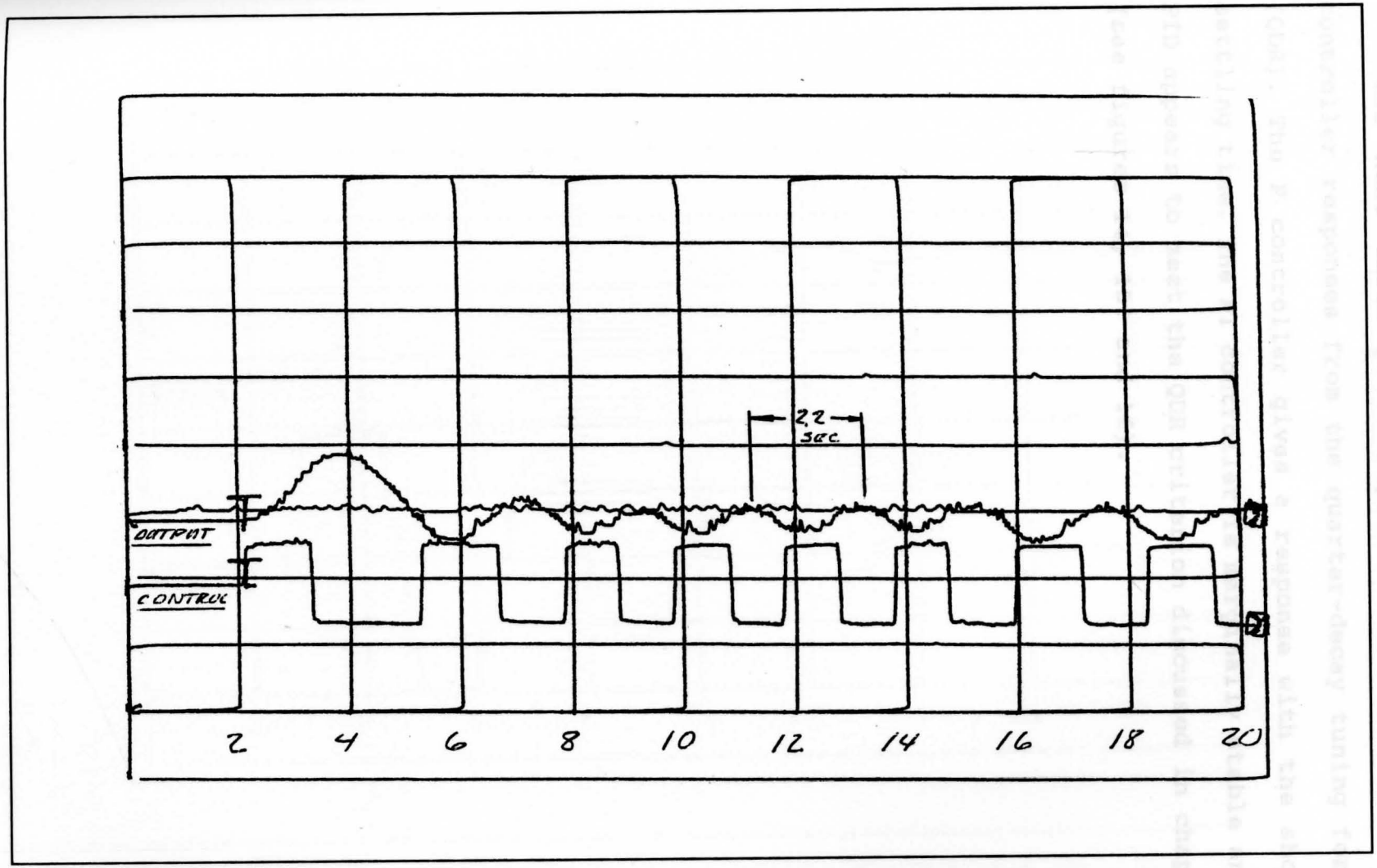
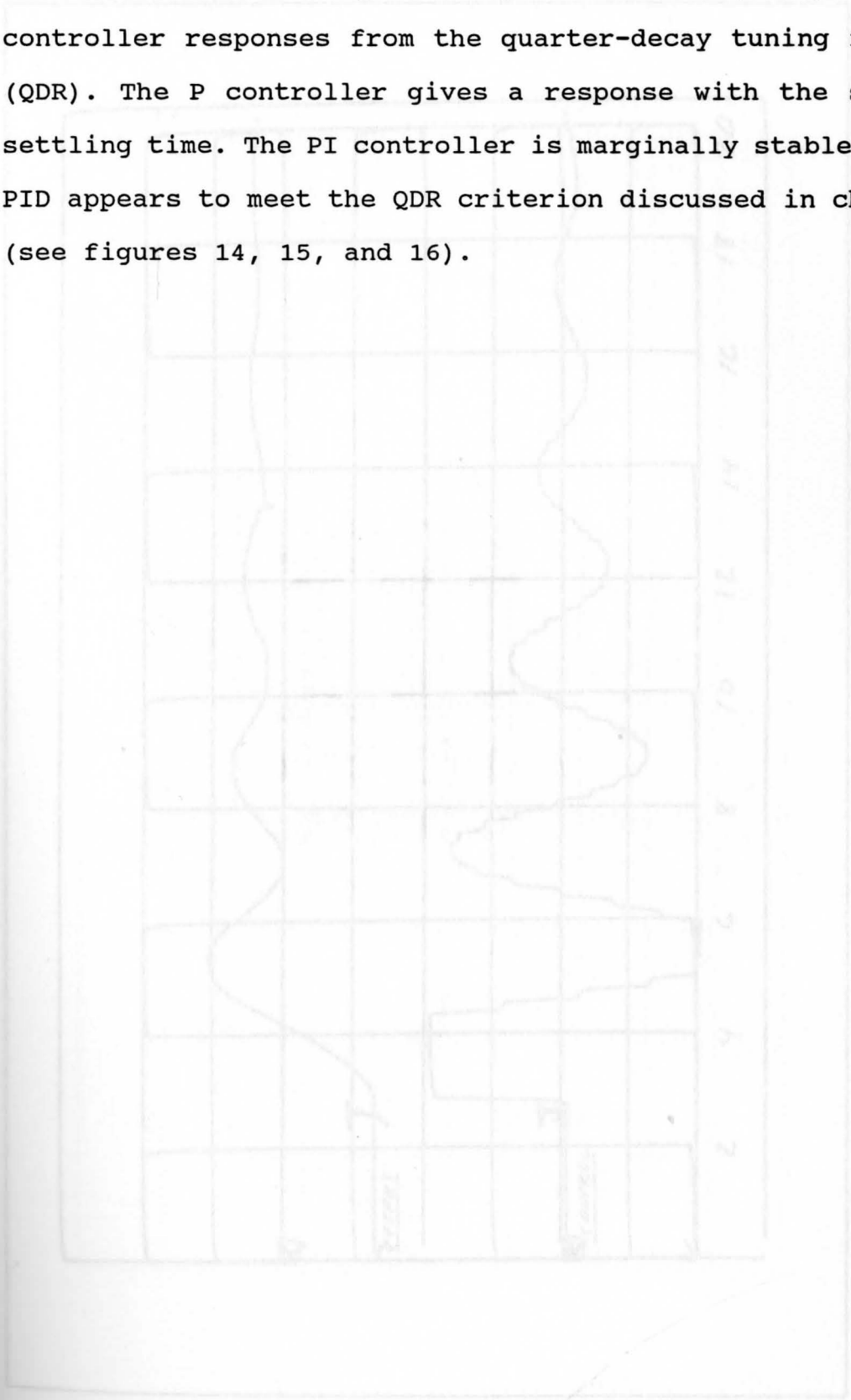


Figure 13. Relay output @ 30%, plot 7, plant 1. (K_u = 9.29, T_u = 2.2, output = 2 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

The next three plots represent the P, PI, and PID controller responses from the quarter-decay tuning formulas (QDR). The P controller gives a response with the shortest settling time. The PI controller is marginally stable and the PID appears to meet the QDR criterion discussed in chapter 1 (see figures 14, 15, and 16).



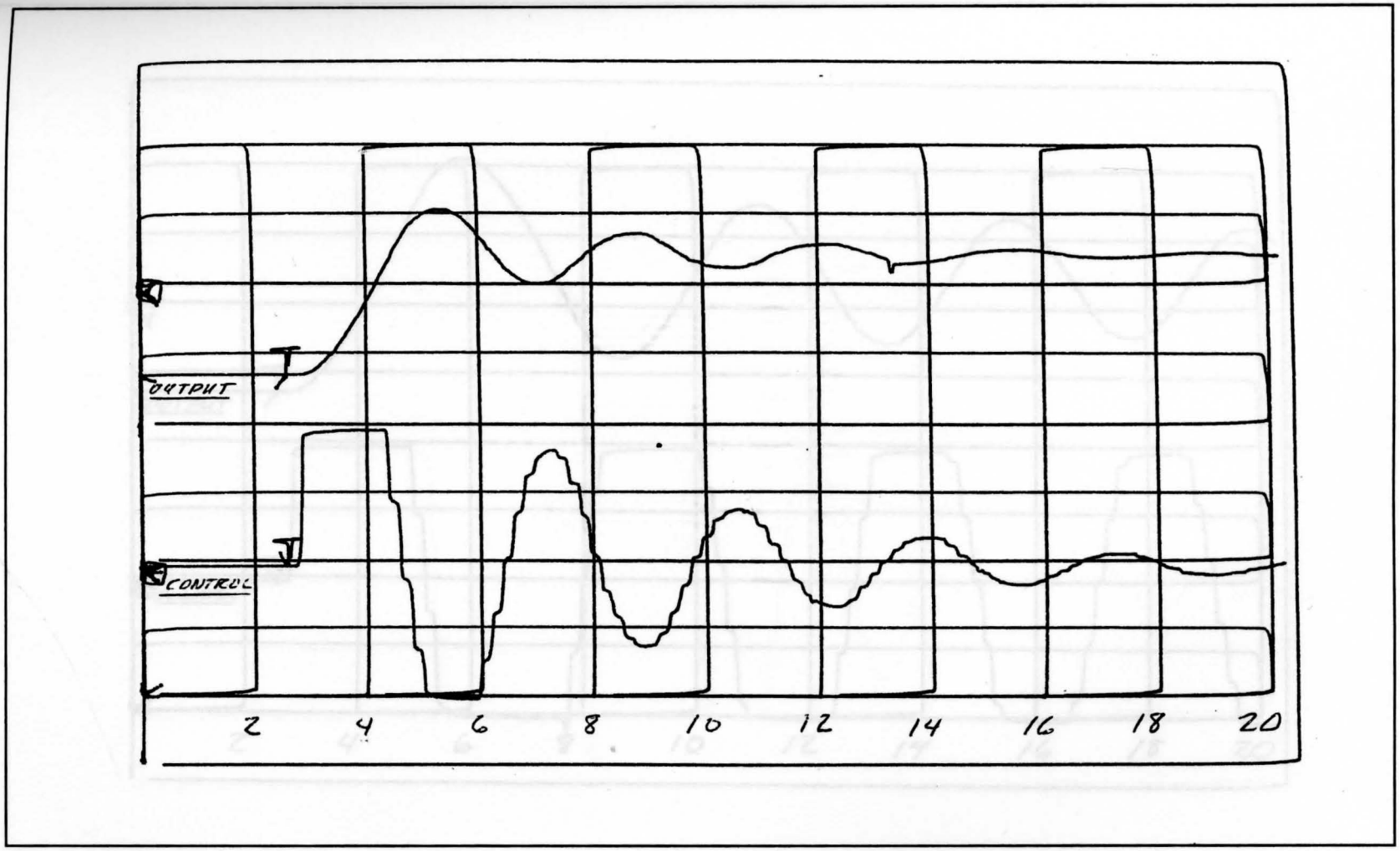


Figure 14. QRD P controller plot 8, plant 1. (K_c = 3.84, T_i = 0.0, T_d = 0.0, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

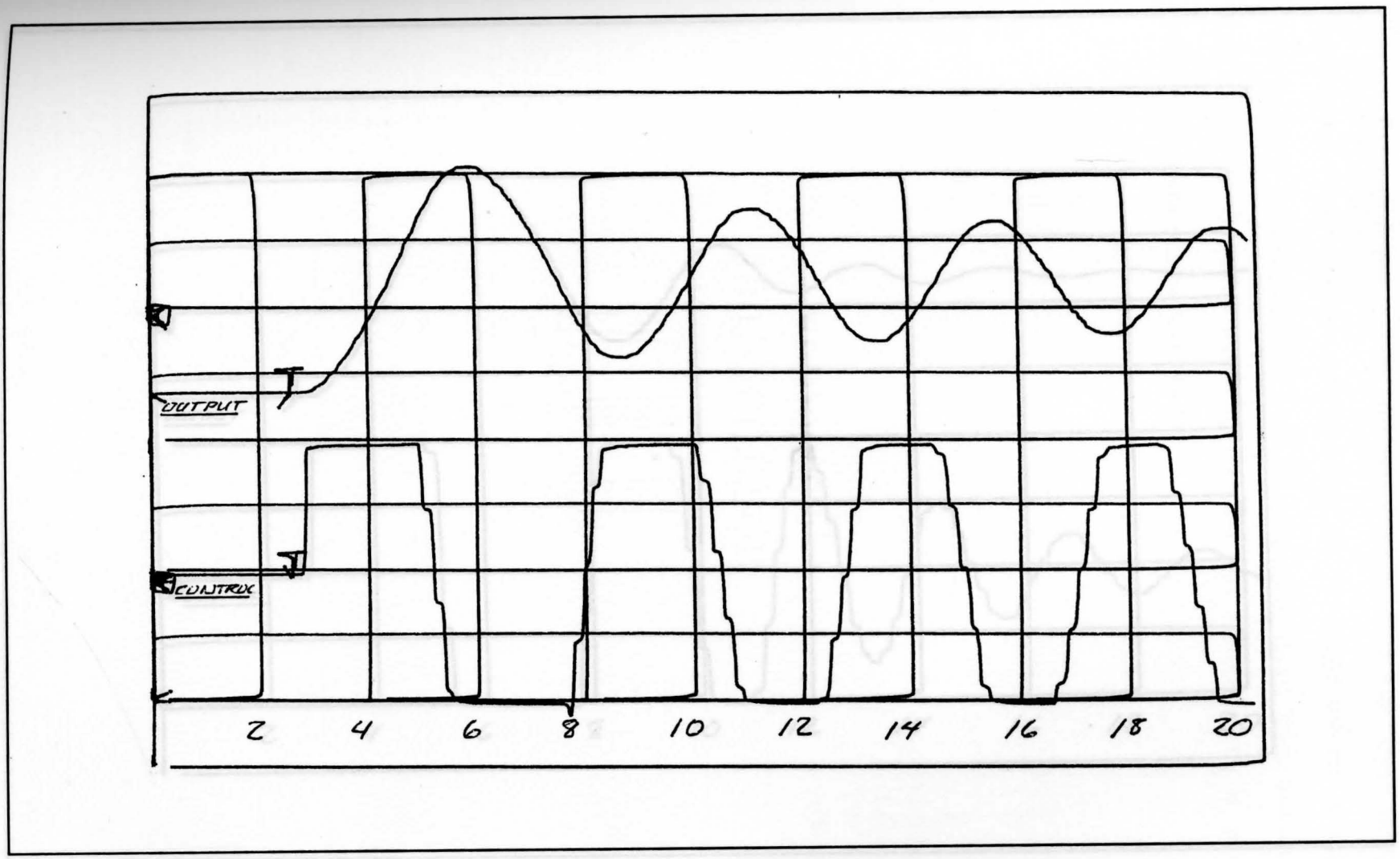


Figure 15. QRD PI controller plot 9, plant 1. (K_c = 3.458, T_i = 2.0, T_d = 0.0, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

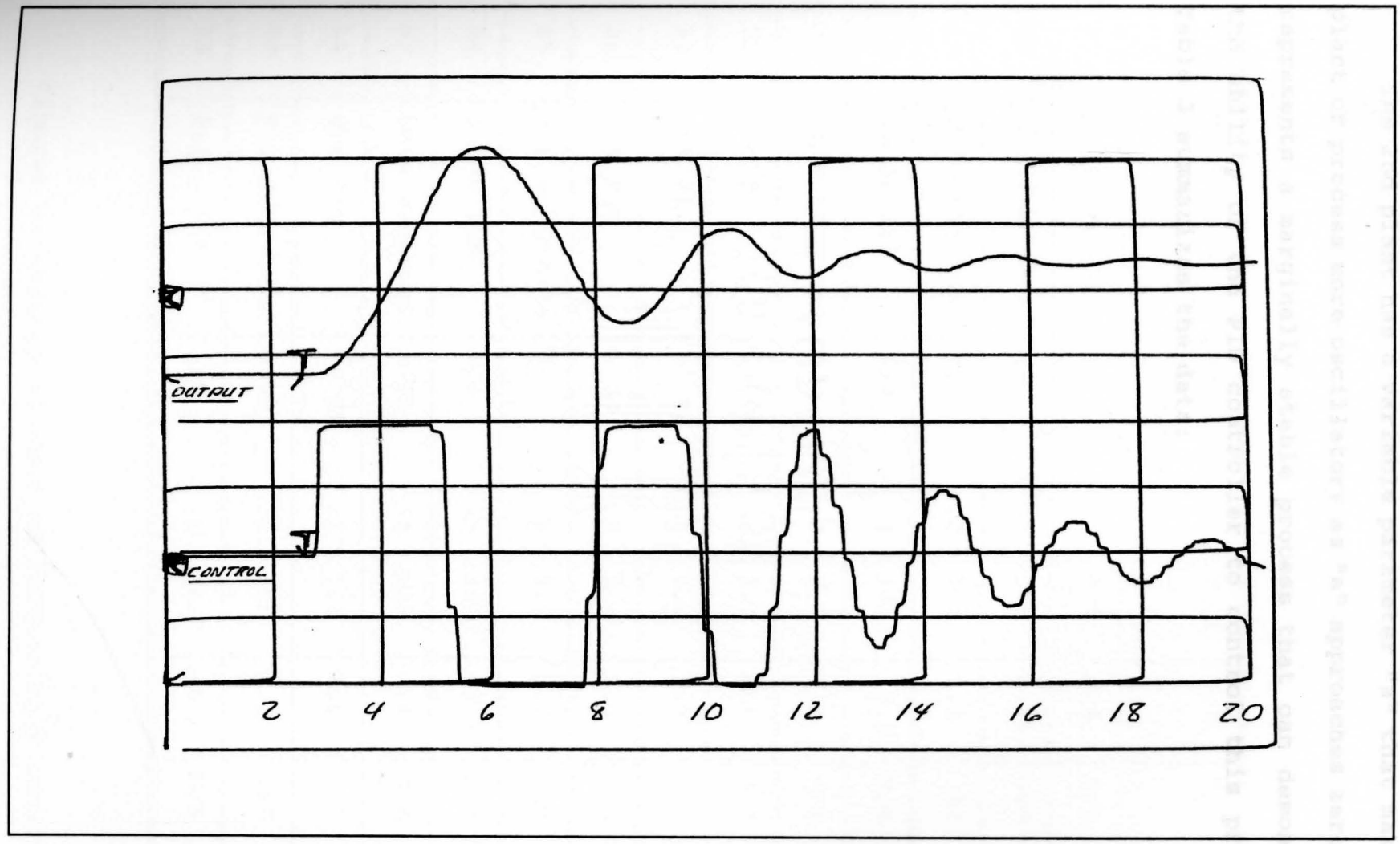


Figure 16. QDR PID controller plot 10, plant 1. (K_c = 5.76, T_i = 1.5, T_d = .24, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

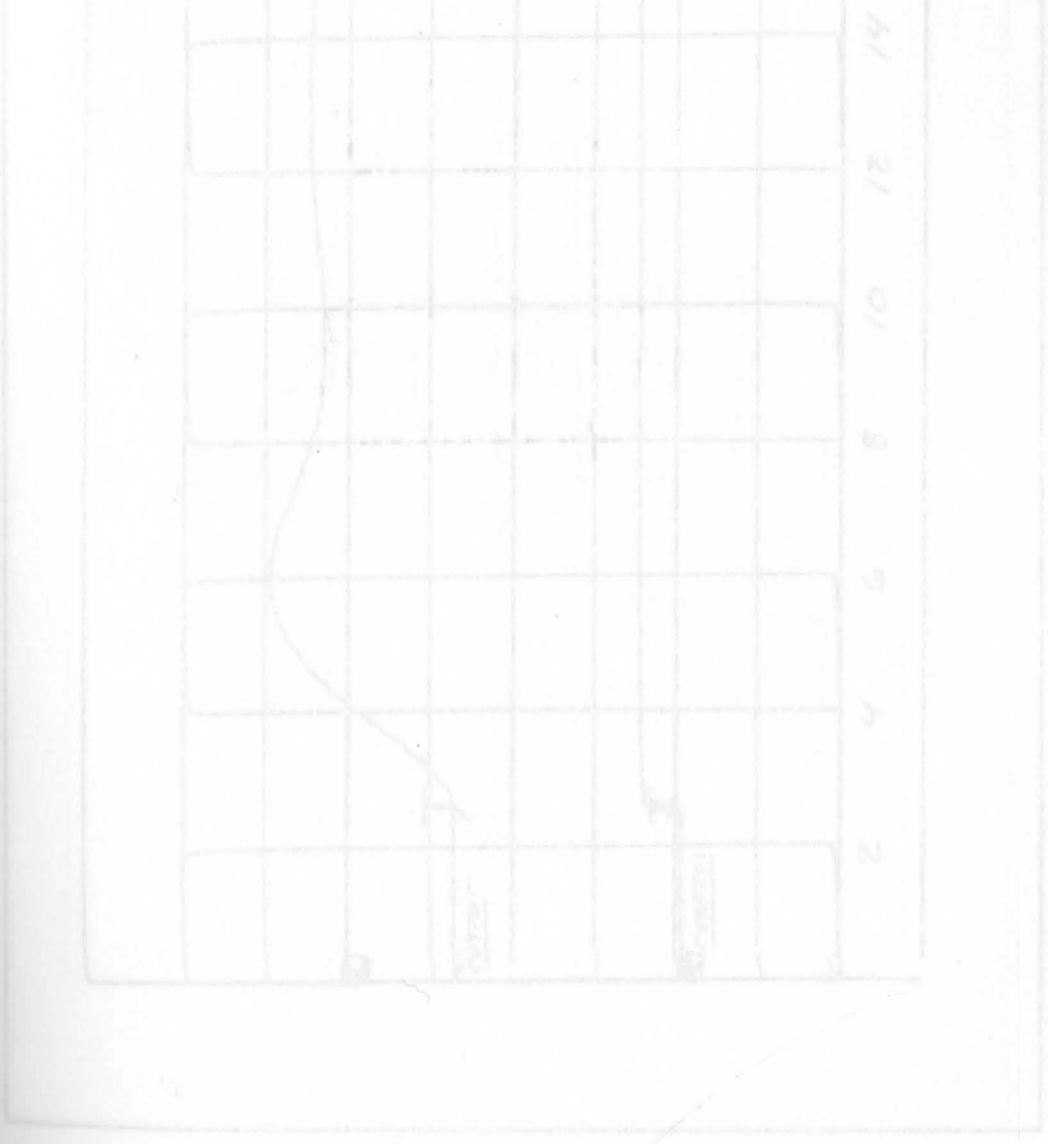
The 2nd plant has a variable parameter "a" that makes the plant or process more oscillatory as "a" approaches zero. This represents a marginally stable process that can demonstrate the ability of the PID controller to control this process. Table 3 summarizes the data:

Table 3. Plant 2 data.

Fig.	Control	Settling Time Sec	a	K_c/K_u	T_i/T_u	T_d
17	Step response	16	.7	N/A	N/A	N/A
18	QDR PID	20	.7	3.6	1.75	.28
19	PID auto-tune	15	.7	1.7	3.2	.57
20	Relay 10%	N/A	.7	/4.8	/2.8	N/A
21	Step response	40	.3	N/A	N/A	N/A
22	QDR P	>> 50	.3	1.6	0.0	0.0
23	QDR PI	>> 50	.3	1.5	2.7	0.0
24	QRD PID	> 50	.3	2.4	2.0	.32
25	PID auto-tune	40	.3	1.1	3.6	.65
26	Relay 10%	N/A	.3	/3.2	/3.2	N/A
27	Step response	> 50	.1	N/A	N/A	N/A
28	QDR PID	>> 50	.1	1.0	2.6	.42
29	PID auto-tune	> 50	.1	.48	4.8	.85
30	Relay 10%	N/A	.1	/1.4	/4.2	N/A

Figures 17 through 30 show an increasingly oscillatory

plant with the decreasing factor α. The auto-tuning software finds the ultimate gain and period to be decreasing, as α decreases. This indicates that a lower gain is required to obtain a stable control. In all cases the PID auto-tune controller is stable and controls the process in less time than the step response. However, the QDR tuning parameters do not provide this type of control. In figure 28 the QDR controller is not gaining control of this process in a reasonable time.



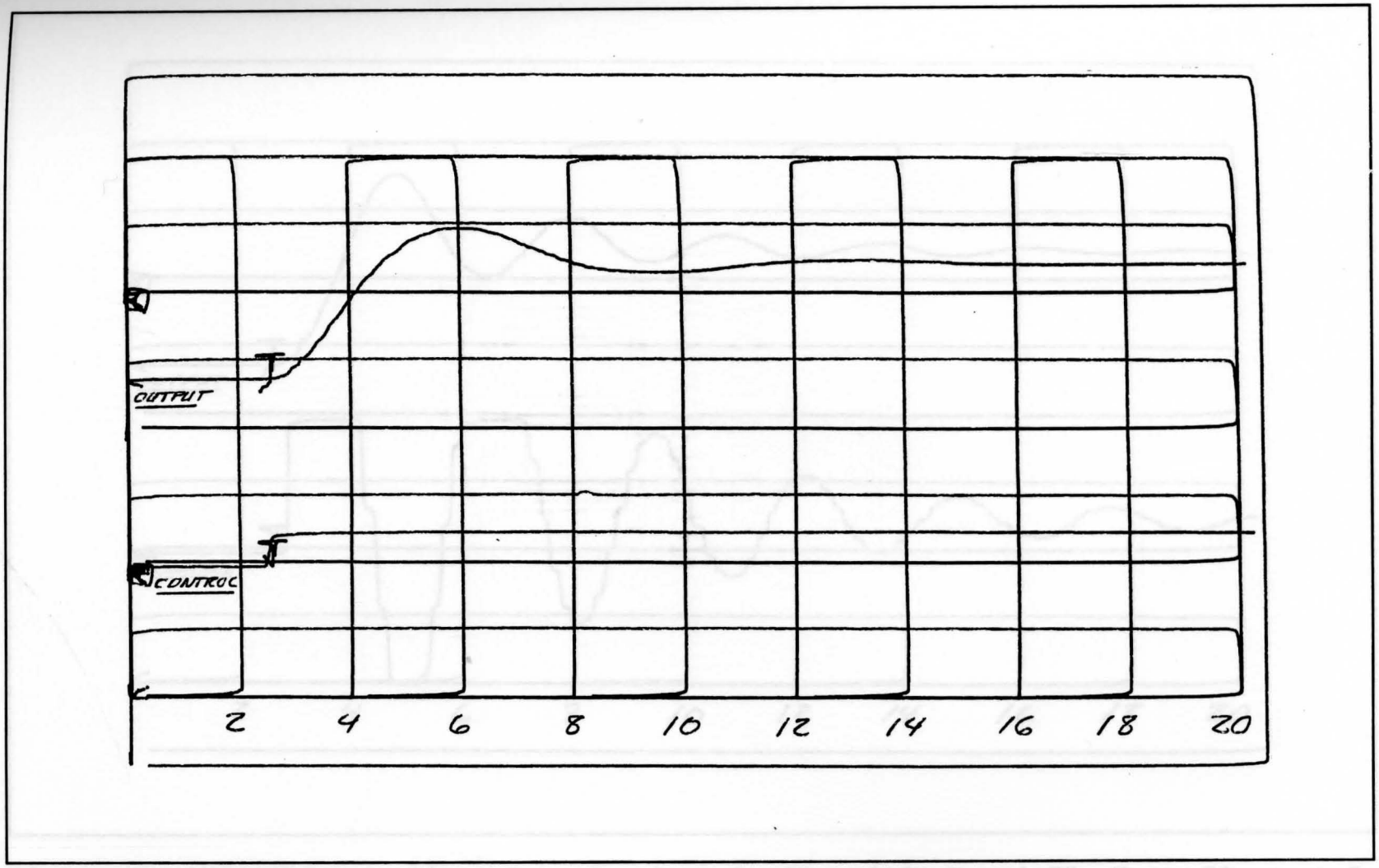


Figure 17. Step response α = .7, plot 11, plant 2. (Step = 2.59, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

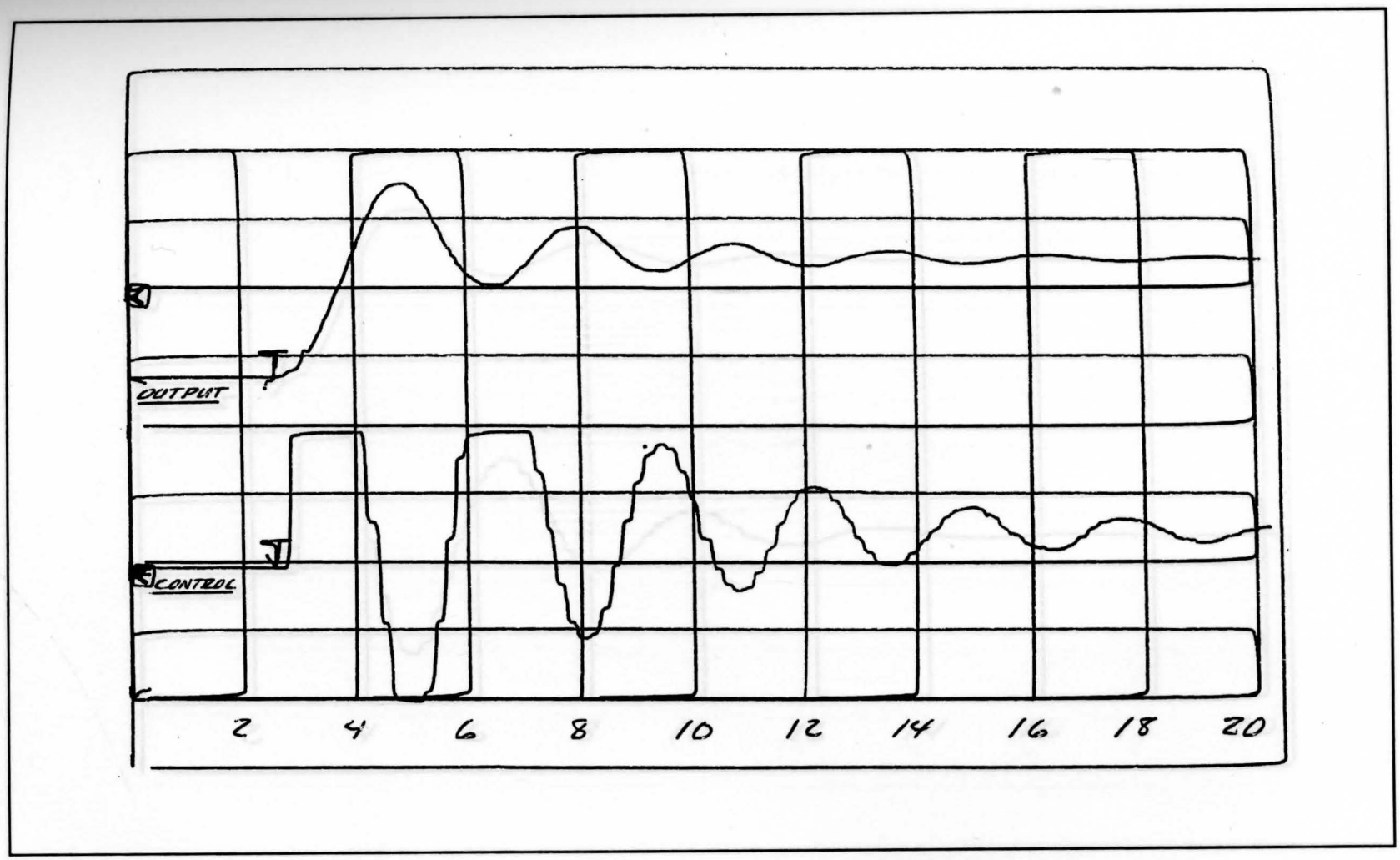


Figure 18. QDR PID controller plot 12, plant 2. (K_c = 3.6, T_i = 1.75, T_d = .28, α = .7, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

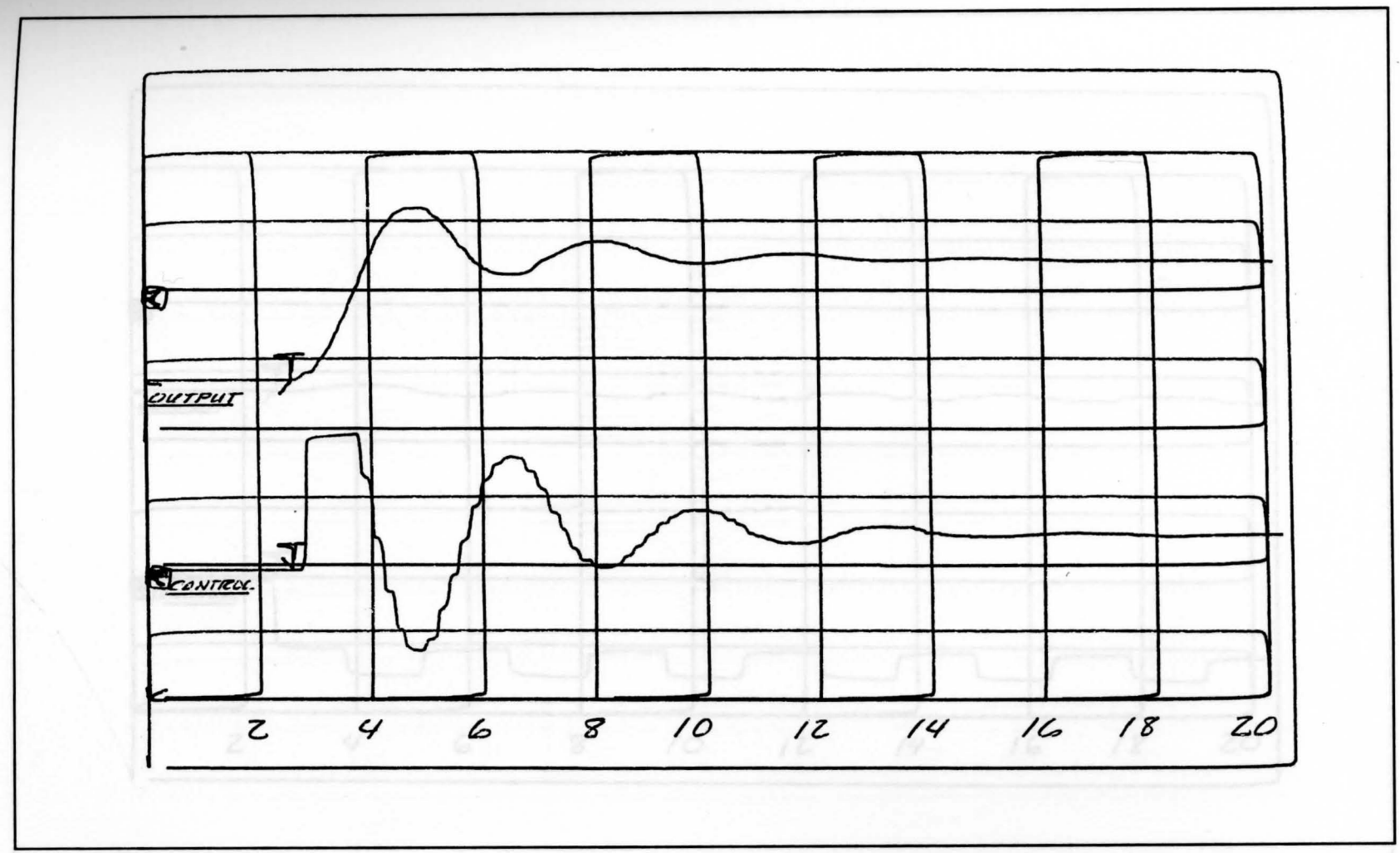


Figure 19. PID Auto-tune controller plot 13, plant 2. (K_c = 1.68, T_i = 3.16, T_d = .57, α = .7, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

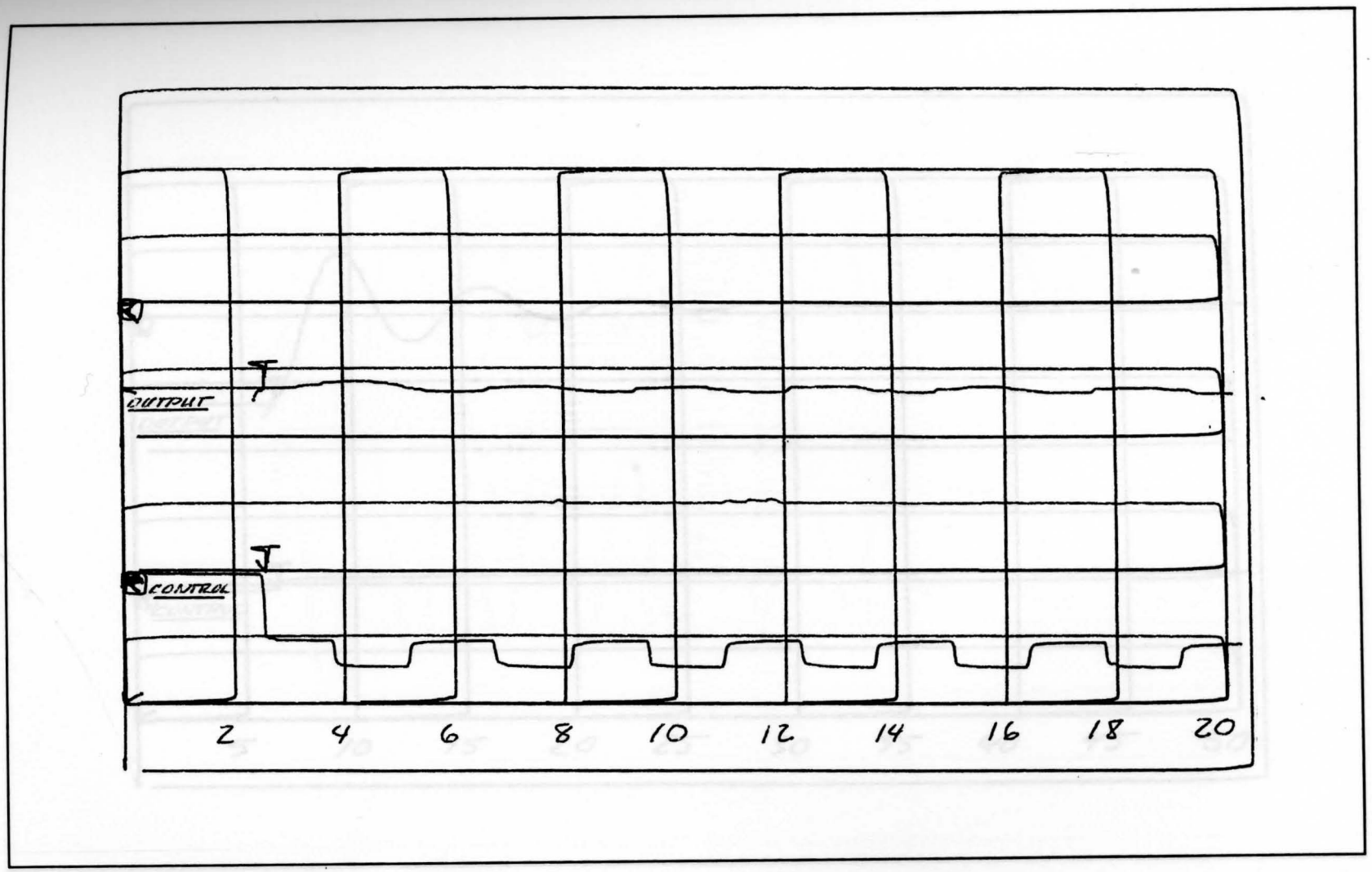


Figure 20. Relay output 10%, plot 14, plant 2. (K_u = 4.8, T_u = 2.8, α = .7, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

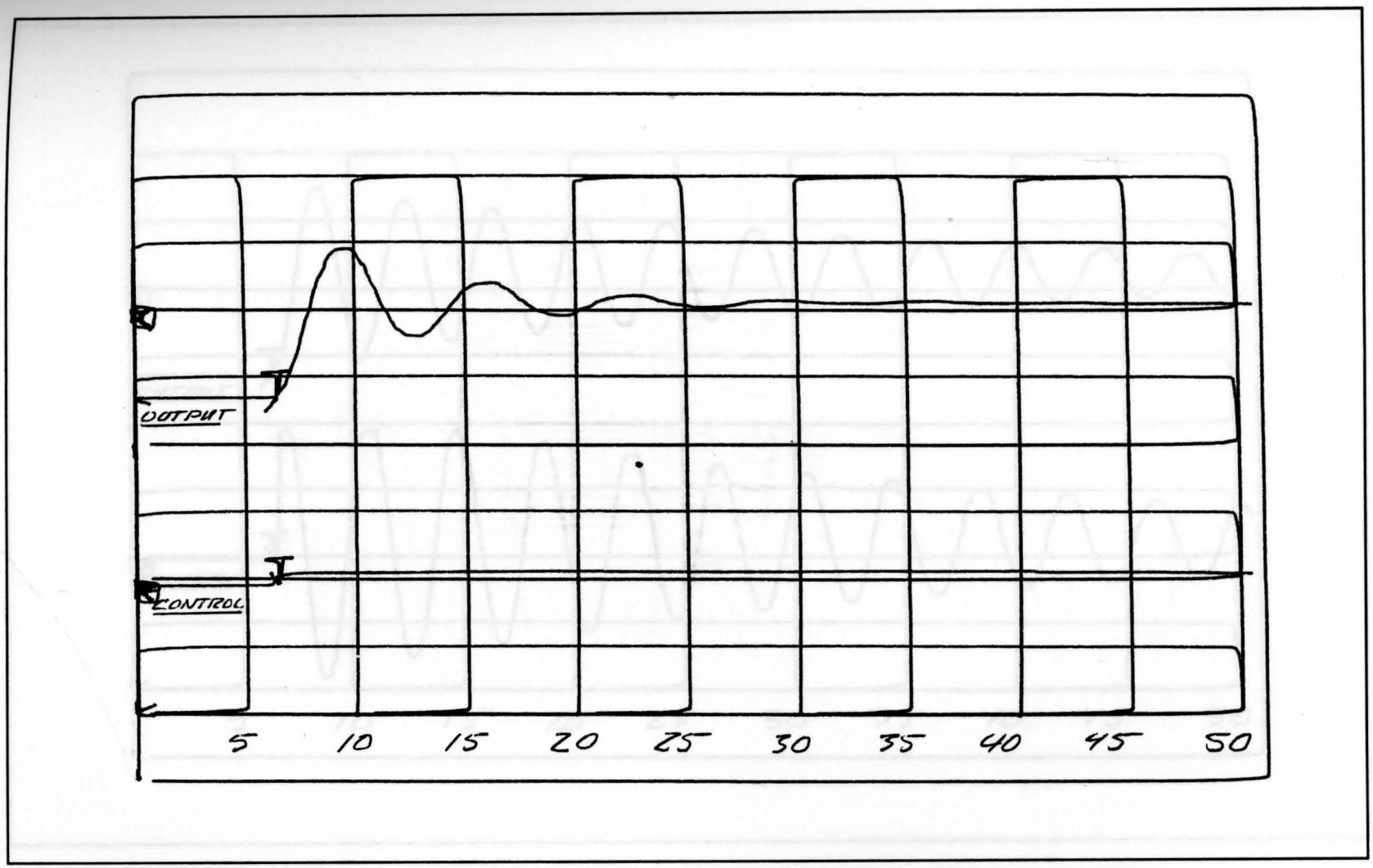


Figure 21. Step response α = .3, plot 15, plant 2. (Step = 1.0, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

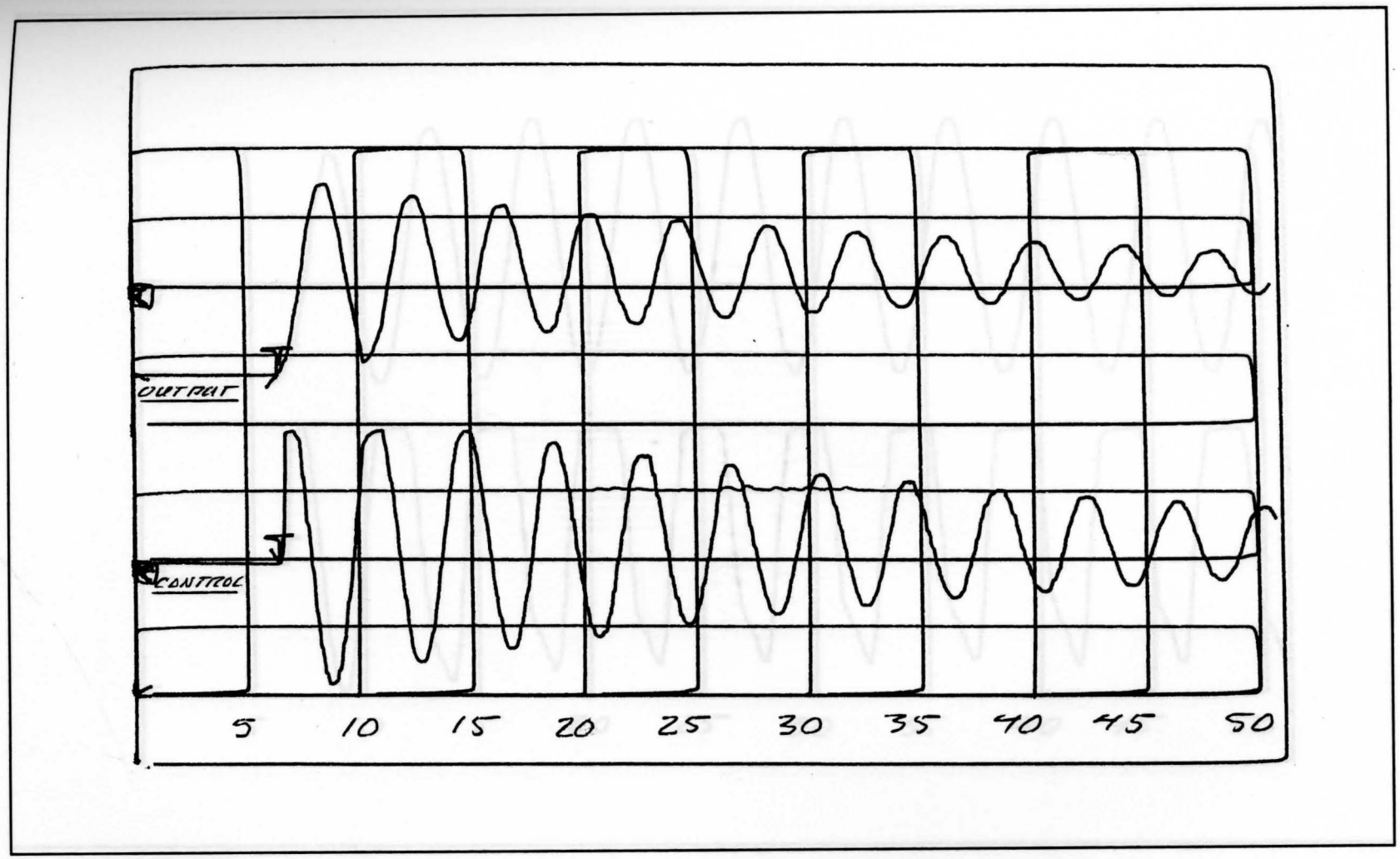


Figure 22. QDR P controller plot 16, plant 2. (K_c = 1.6, T_i = 0.0, T_d = 0.0, α = .3, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

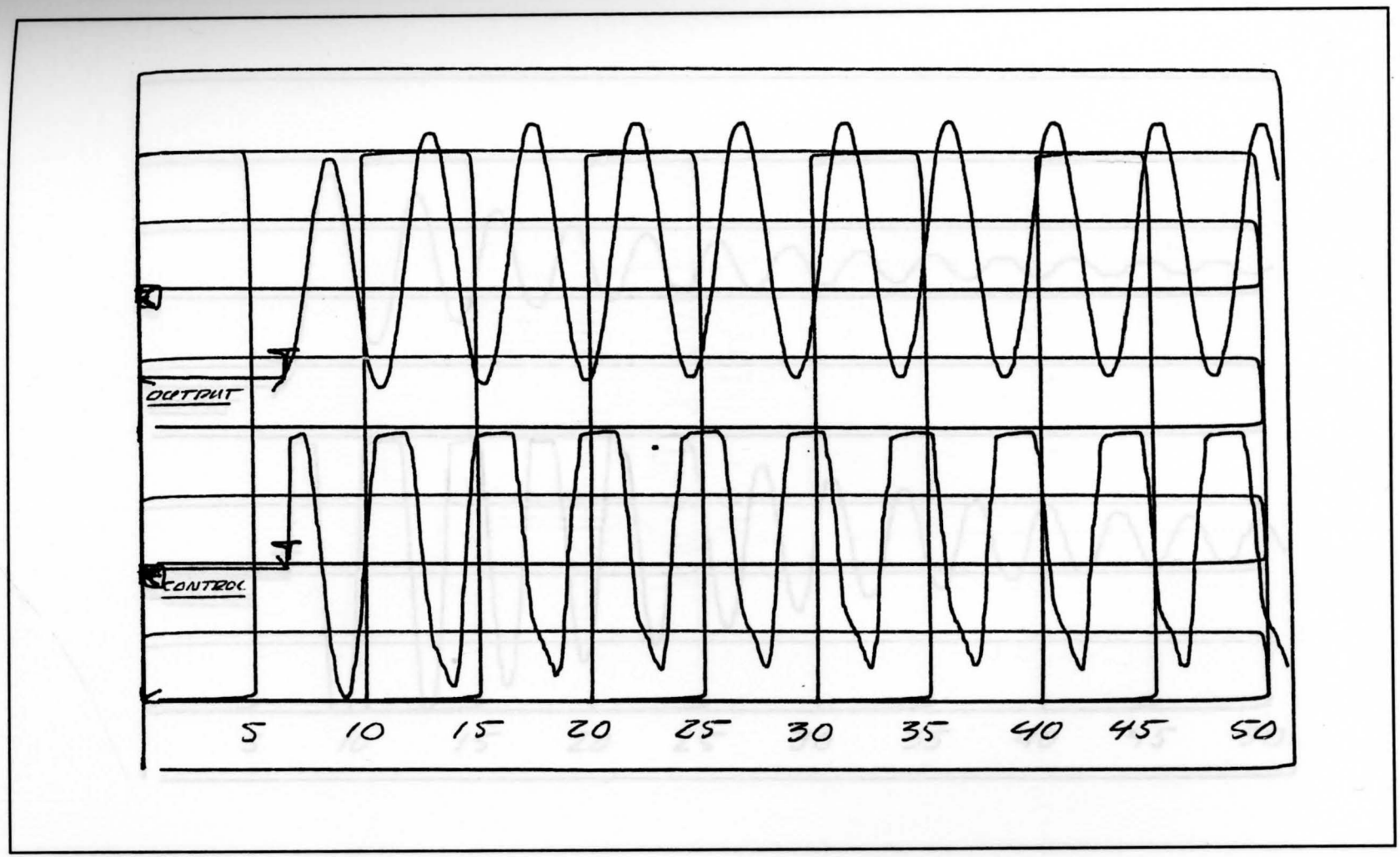


Figure 23. QDR PI controller plot 17, plant 2. (K_c = 1.45, T_i = 2.67, T_d = 0.0, α = .3, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

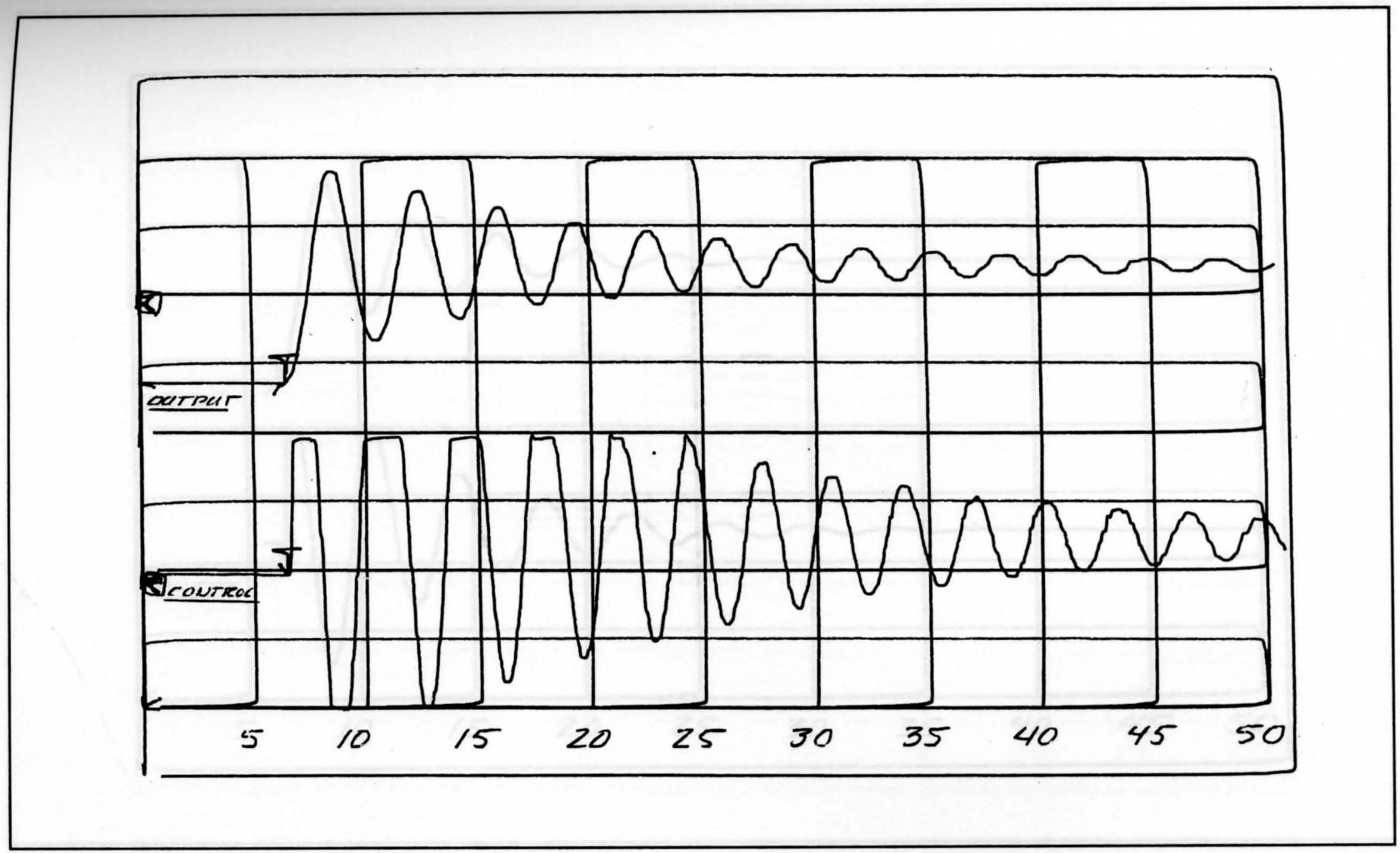


Figure 24. QDR PID controller plot 18, plant 2. (K_c = 2.42, T_i = 2.0, T_d = .32, α = .3, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

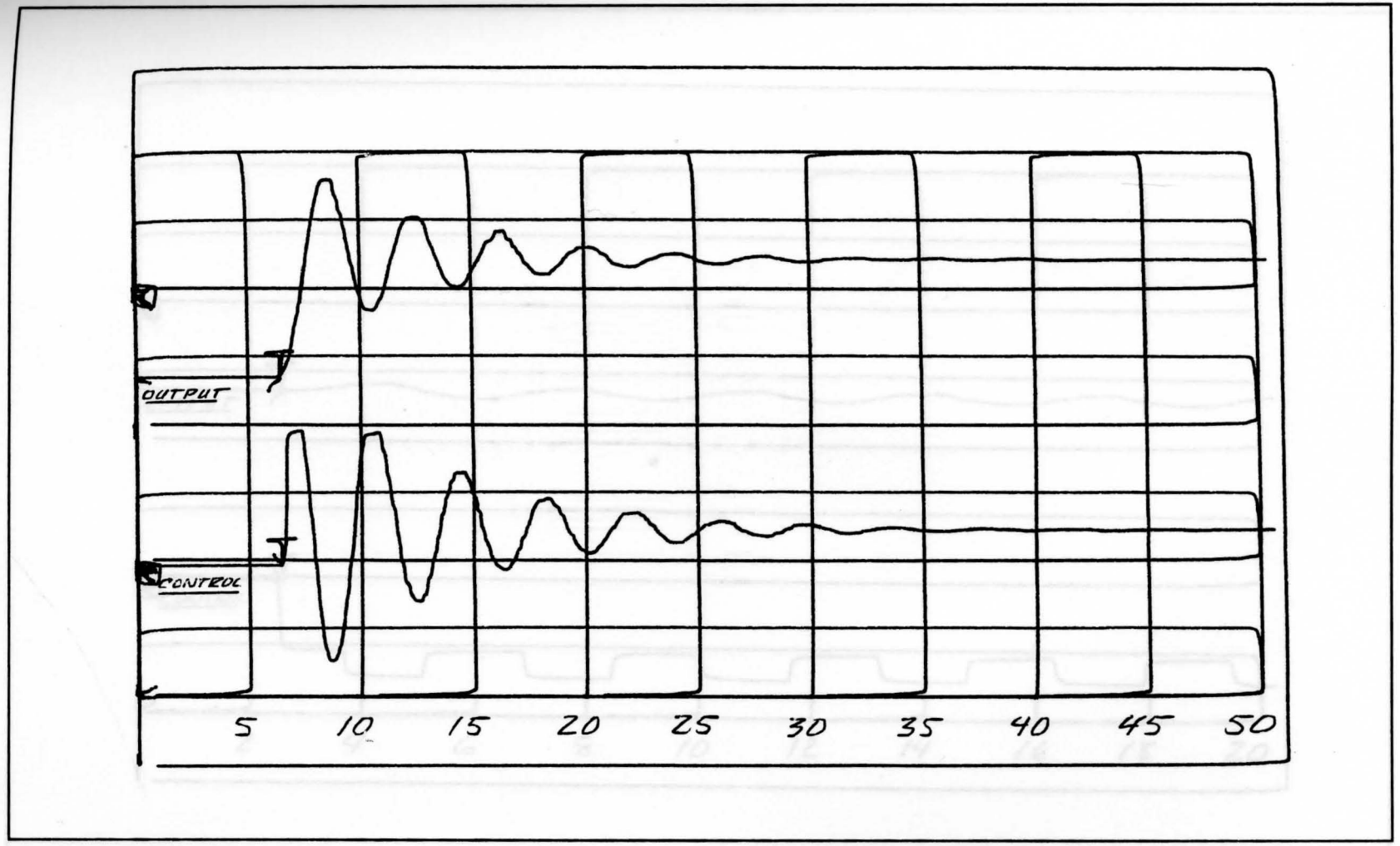


Figure 25. Auto-tune PID controller, plot 19, plant 2. (K_c = 1.13 T_i = 3.6, T_d = .65, α = .3, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

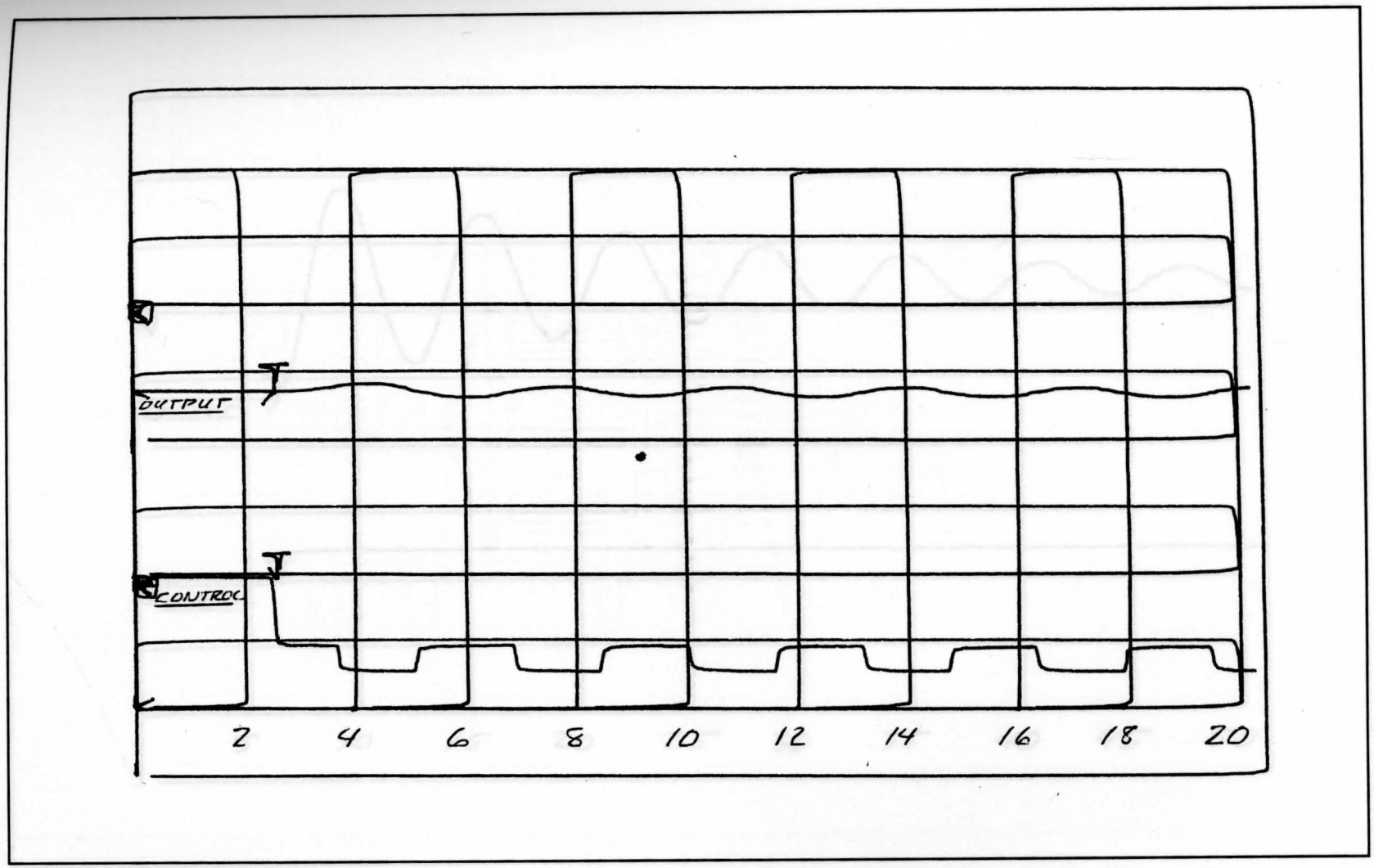


Figure 26. Relay output 10%, plot 20, plant 2. (K_u = 3.23, T_u = 3.2, α = .3, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

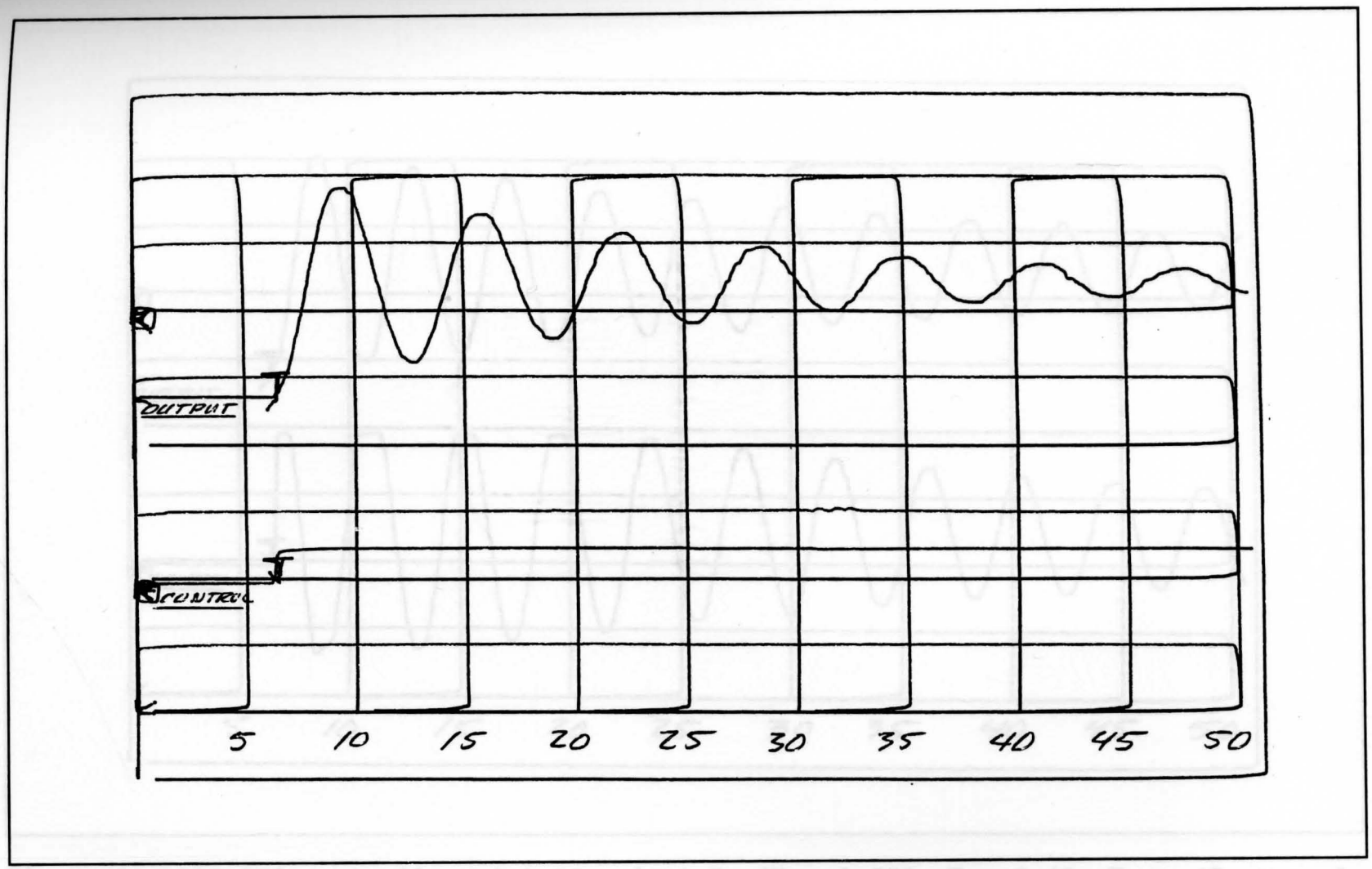


Figure 27. Step response α = .1, plot 21, plant 2. (Step = 2.59, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

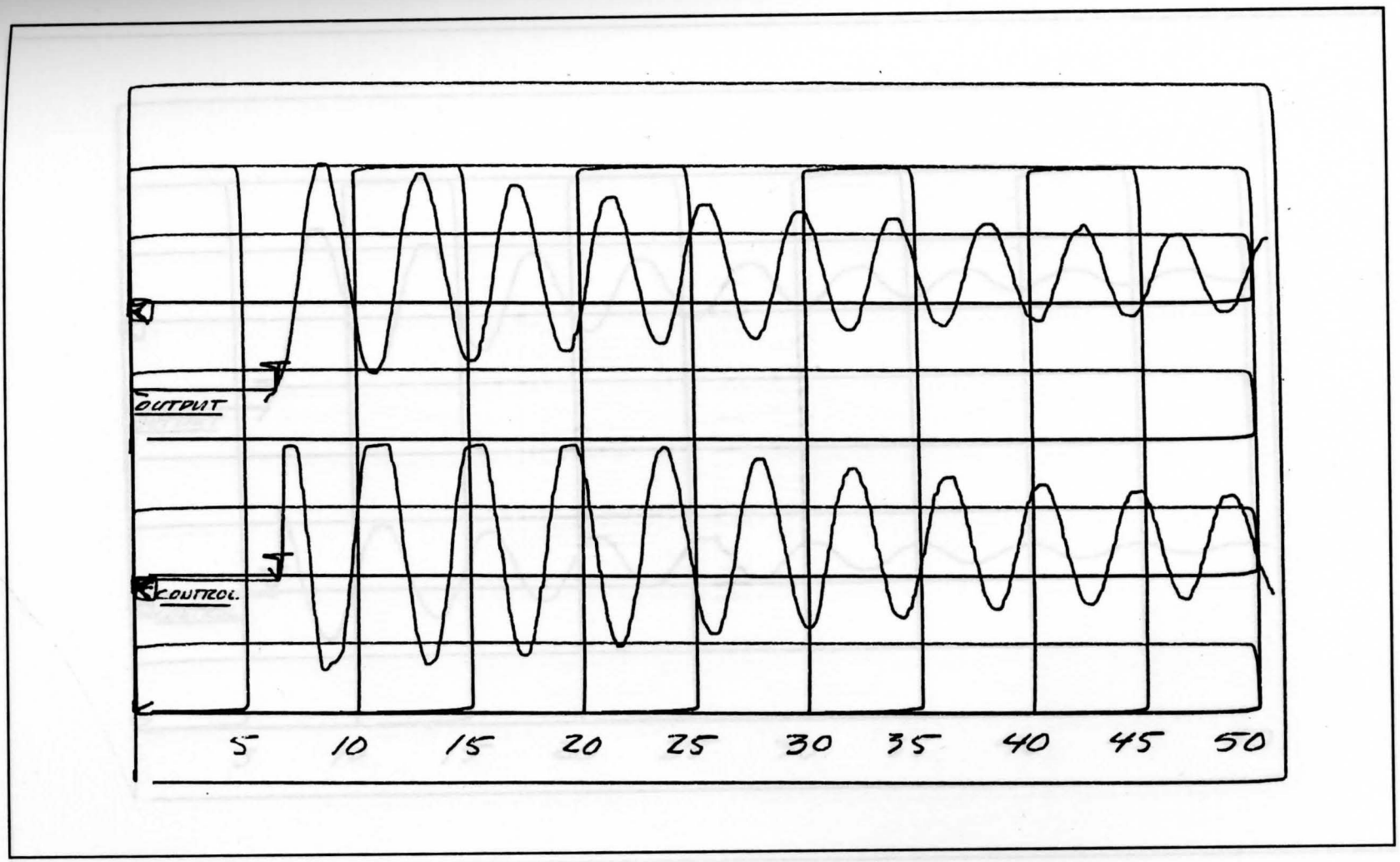


Figure 28. QDR PID controller plot 22, plant 2. (K_c = 1.022, T_i = 2.63, T_d = .42, α = .1, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

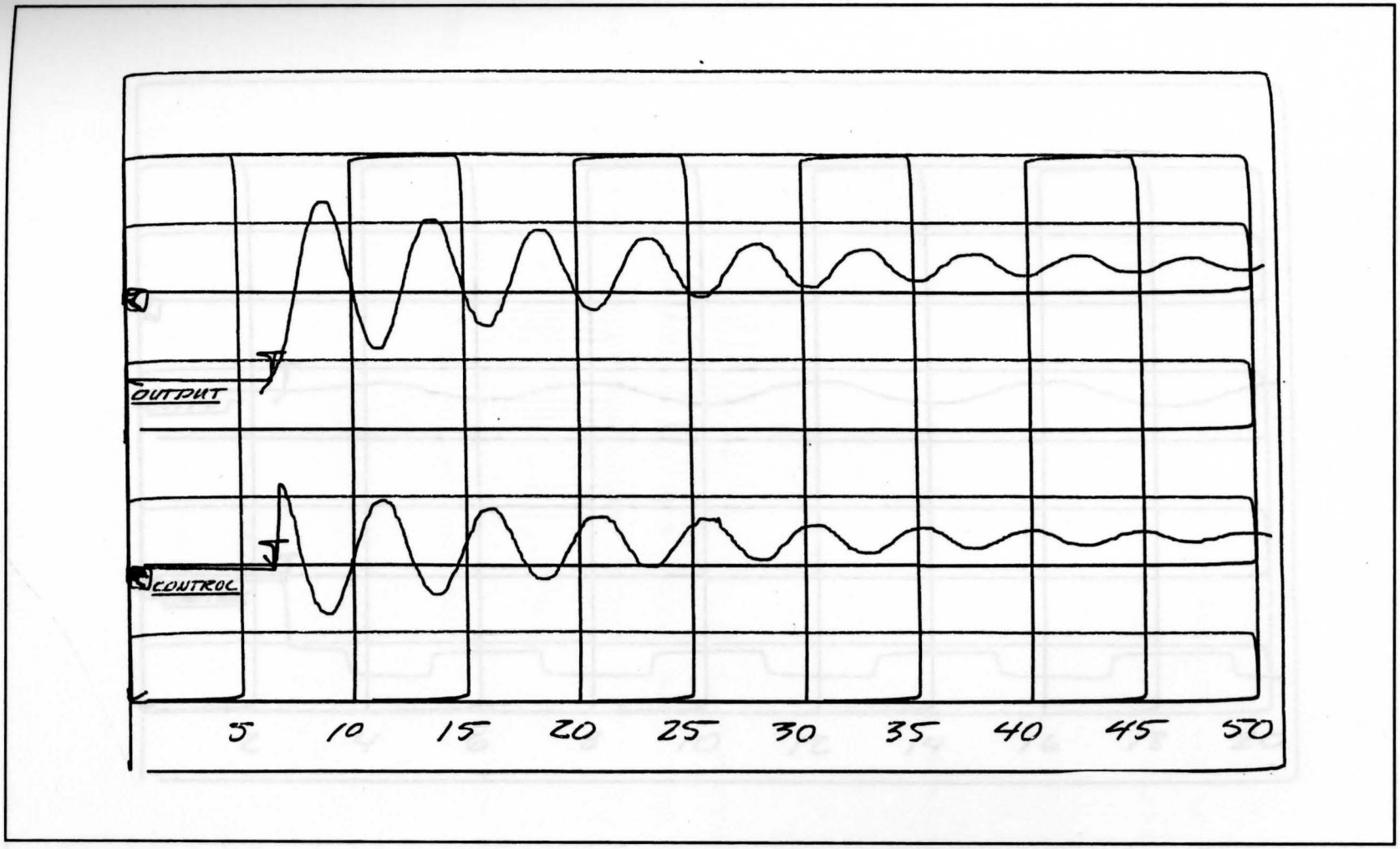


Figure 29. Auto-tune controller plot 23, plant 2. (K_c = .477, T_i = 4.75, T_d = .854, α = .1, setpoint = 2.6, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

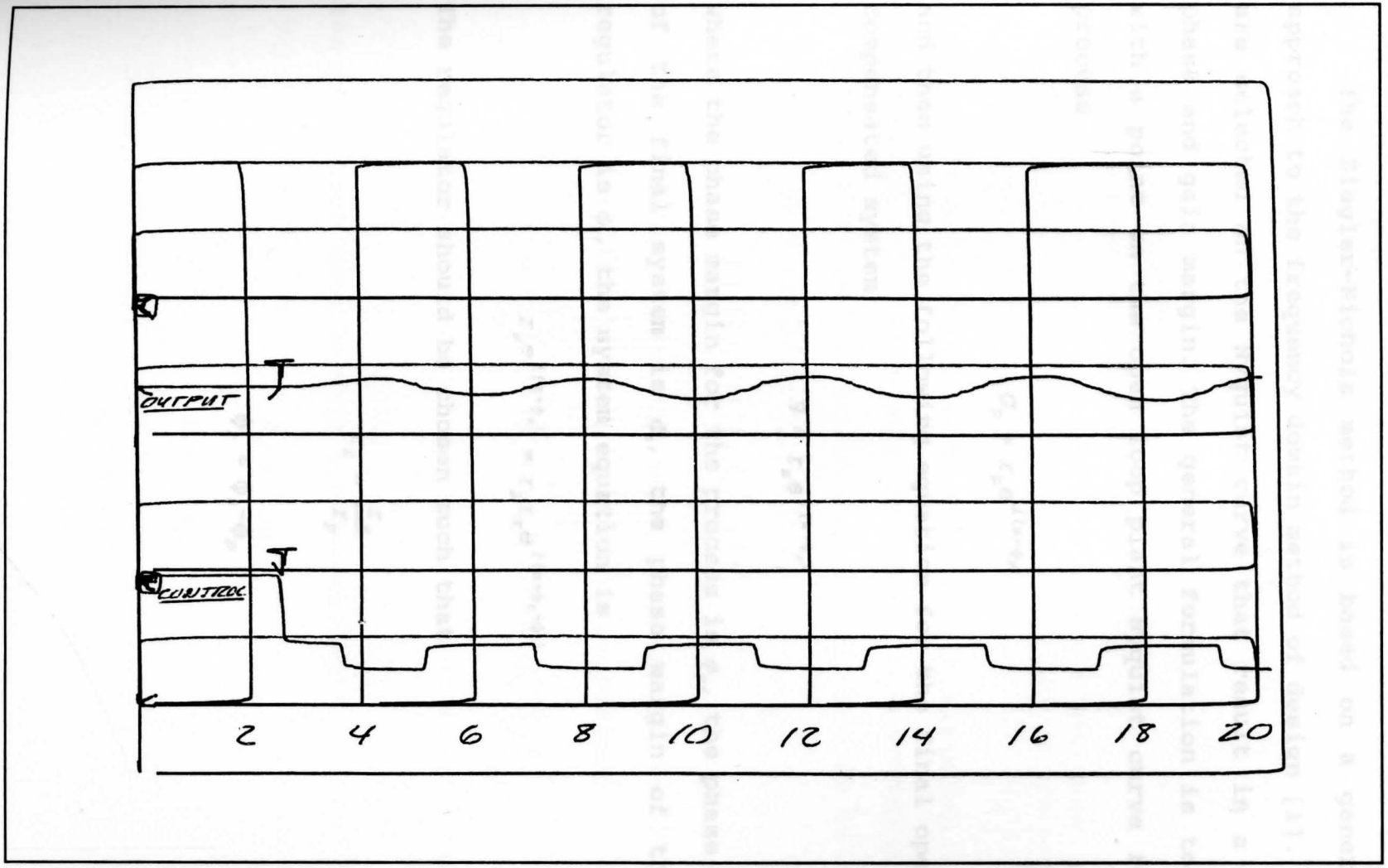


Figure 30. Relay output 10%, plot 24, plant 2. (K_u = 1.36, T_u = 4.2, α = .1, output = 5 v/div, control = 5 v/div, vertical axis - volts, horizontal axis - seconds).

ANALYSIS

The Ziegler-Nichols method is based on a generalized approach to the frequency domain method of design [1]. Points are selected on the Nyquist curve that result in a system phase and gain margin. The general formulation is to start with a point on the open loop plant Nyquist curve for the process

$$G_p = r_p e^{i(\pi+\phi_p)} \tag{16}$$

and then using the following equation for the final open loop compensated system

$$S = r_s e^{i(\pi+\phi_s)} \tag{17}$$

where the phase margin for the process is ϕ_p, the phase margin of the final system is ϕ_s, the phase margin of the PID regulator is ϕ_r, the system equation is

$$r_s e^{i(\pi+\phi_s)} = r_p r_r e^{i(\pi+\phi_p+\phi_r)} \tag{18}$$

The regulator should be chosen such that

$$r_r = \frac{r_s}{r_p} \tag{19}$$

$$\phi_r = \phi_s - \phi_p \tag{20}$$

resulting in Kc, the gain of the PID controller, the integral, and derivative parameters as follows:

$$K_c = \frac{r_s \cos(\phi_s - \phi_p)}{r_p} \tag{21}$$

$$\omega T_d - \frac{1}{\omega T_i} = \tan(\phi_s - \phi_p) \tag{22}$$

$$\omega = \frac{2\pi}{T_u} \tag{23}$$

The gain is uniquely given from the above equation, but the parameters T_i and T_d have only one equation. A common practice to provide a phase lead controller [4, 11, 12] is to specify T_i in terms of a ratio to T_d, such as,

$$T_d = \alpha T_i \tag{24}$$

where α is chosen to be some value such as .18. The following equations give the T_i and T_d parameters as follows:

$$T_d = \frac{1}{2\omega}\left[-\tan(\phi_p - \phi_s) + \sqrt{4\alpha + \tan^2(\phi_p - \phi_s)}\right] \tag{25}$$

$$T_i = \frac{1}{\alpha} T_d \tag{26}$$

the parameters used in this PID controller,

$$k_c = 0.35 K_u \tag{27}$$

$$T_i = 1.13 T_u \tag{28}$$

$$T_d = 0.18 T_i \tag{29}$$

will result in the following gain and phase margins:

$$P_m = \phi_s = 48.6 \; degrees \tag{30}$$

$$A_m = 20\log(\frac{1}{r_s}) = 6dB \tag{31}$$

where $\phi_p = 0$ is chosen for the ultimate point and $r_p = 1/K_u$ the reciprocal of the ultimate gain.

For example, to illustrate the calculation, let the desired phase margin be $\phi_s = 45$ degrees, the plant phase margin $\phi_p = 0$, the desired amplitude margin of 2 ==> $r_s = 1/2$ (gain margin = 20 log(2) = 6 dB), $\alpha = .25$, then

$$K_c = r_s K_u \cos(\phi_p - \phi_s) = .354K_u \tag{32}$$

$$T_d = \frac{T_u}{4\pi}\left[-\tan(\phi_p-\phi_s) + \sqrt{4\alpha + \tan(\phi_p-\phi_s)^2}\right] = .192T_u \tag{33}$$

$$T_i = \frac{T_d}{\alpha} = .768T_u \tag{34}$$

This Ziegler-Nichols frequency response method based upon moving one point on the Nyquist curve to a desired position, provides simple design rules and is generally sufficient for most processes. Control engineers have found in practice that a gain margin above 5 dB and a phase margin greater than 30 degrees generally provides a sufficiently stable system.

CHAPTER 4

SUMMARY

In this thesis a PID controller with auto-tuning capabilities was presented. "C" software for control and tuning of general processes was written and tested and proven highly reliable. The controller that resulted produced satisfactory control and capabilities. The first plant was open loop unstable and was made stable under closed loop control. Both tuning methods, QDR and auto-tune, provided satisfactory control for this plant. The second plant, however, provided some difficulty for both methods in the more oscillatory state. The QDR method proved less than satisfactory due to the fact that the control was worse than the natural settling time of the process.

Relay auto-tuning was demonstrated as a good method of tuning the two plants and provided a stable control with minimum disturbance to the process. This method in all cases was superior to the QDR tuning method. See figures 11 and 16, the two methods provide equal rise times but the auto-tune control provides less overshoot (5v versus 10v) and a settling time of 10 seconds versus 16 seconds. Again consider figures 18 and 19, rise times are equal but the auto-tune method provides less overshoot (5v versus 7v) and a settling time of ten seconds as compared to fourteen seconds for the QDR method. Similar remarks can be made regarding figures 24, 25

and 28, 29.

The project demonstrates the feasibility of using a personal computer to control a process, automatically tune the parameters, and provide an operator interface for control of the PID controller parameters.

Additional goals of this project included learning about the physical implementation of a practical PID controller and answering questions relating to the application of theory to a practical application. The answers to these questions is that the theory closely fits with the practice and produces a controller that performs as expected.

RECOMMENDATIONS FOR FUTURE STUDY

Students of controls are welcome to take this work and improve upon it or experiment with it in additional ways. An interesting project for this PID controller would be to construct an actual process of some type, such as a flow, pressure or temperature control process and implement PID control. Study the dead time / transport delay effects and how to handle them in the design of the controller. Study step response and ultimate gain tuning and the relay auto-tuning to compare the differences from that obtained with a simulation.

APPENDIX A

TECHNICAL DETAILS

DAS8 ADDRESSES AND OTHER DETAILS[15], [16], [17]

Important addresses found on page 2 of the Model 767 A2D manual (note these addresses do not agree with the manual):

Table 4. Model 767 and GP-6 addresses.

Item	Multiplexer Address
GP-6 Amplifier 1	00
GP-6 Amplifier 2	02
GP-6 Amplifier 3	04
GP-6 Amplifier 4	06
767 A4	08
767 A5	10
767 A6	12
767 A7	14

Note: the following instructions are required to cause the Comdyna GP-6 computer to operate from the computer. The wait is required because this looks for the operate pushbutton to be pushed in by the operator on the Comdyna.

```
outp(785,0);  /* start GP-6 operate mode */
GP_6 = inp(786);
while(GP_6 != 4)
 {
  GP_6 = inp(786);
  GP_6 &=4;
 }
```

The following instruction is required to put the Comdyna in the IC mode:

```
outp(785,8); /* set C3 high for GP_6 IC mode */
```

TIMER / COUNTER DETAILS

In order to implement the "C" code, software routines that access the DAS8 board must be operating properly. The MetraByte manual provides data relating to addresses on the DAS8 board that are summarized here. The following control words must be loaded into the control register for each counter to be properly initialized. See the "C" program "ini_timer(PID_ptr); /* initialize timer function */" for additional details regarding the initialization (appendix B).

Table 5. Counter details.

Control Word								
D7	D6	D5	D4	D3	D2	D1	D0	VALUE
SC1	SC0	RL1	RL0	M2	M1	M0	BCD	
0	1	1	1	0	1	1	0	CTR 1 = 118
1	0	1	1	0	1	1	0	CTR 2 = 182
0	0	1	1	0	0	0	0	CTR 0 = 48

Base address + 7 (768 + 7 = 775) ==> counter control register.

SC1 & SC0 ==> COUNTER NUMBER

RL1 & RL0 ==> DATA TRANSFER OPERATION = WORD

M1, M2, M0 ==> MODE 011 ==> SQUARE WAVE

MODE 000 ==> PULSE

SETUP FOR 200 MILLISECOND ==> #395 FOR COUNTER 2:

The following high byte and low bytes are required for the software to proper load the counters. The software obtains these values by ANDing and shifting data as can be seem from the function "ini_timer() initialize timer".

WORD VALUE = 395 ==> 18B HEX

HB = 1 LB = 139

0000 0001 1000 1011

SETUP FOR COUNTER 1 200 X 10 = 2000

2000 = 7D0 HEX

0000 0111 1101 0000 ==> HB = 7, LB = 208

SETUP FOR COUNTER 0 200 X 5 = 1000

1000 = 3E8 ==> HB = 3, LB = 232

Description of Counter Operation:

Counter 2 is setup as the main counter at the frequency required to give a period equal to 2000 times the sample period. For example, a count of 395 yields a frequency of 10,000 Hz or .1 milliseconds per cycle. The output of counter 2 is fed to counter 1 where the count is divided by 2000, resulting in a 5Hz (10000/2000) or .2 seconds (200 milliseconds) cycle. The output of counter 2 is also fed to counter 0, where it is divided by 1000. This counter counts out and goes high allowing the detection of the rising edge of counter 1. Counter 0 is reloaded and restarted with every timeout. The wave forms of counter 1 and counter 0 are input to the digital inputs IP1 and IP2 for reading by the computer. Whenever the IP1 and IP2 are equal to three a time out has occurred and the rising edge of counter 2 has been detected, indicating the beginning of another timing interval. Below is a timing diagram:

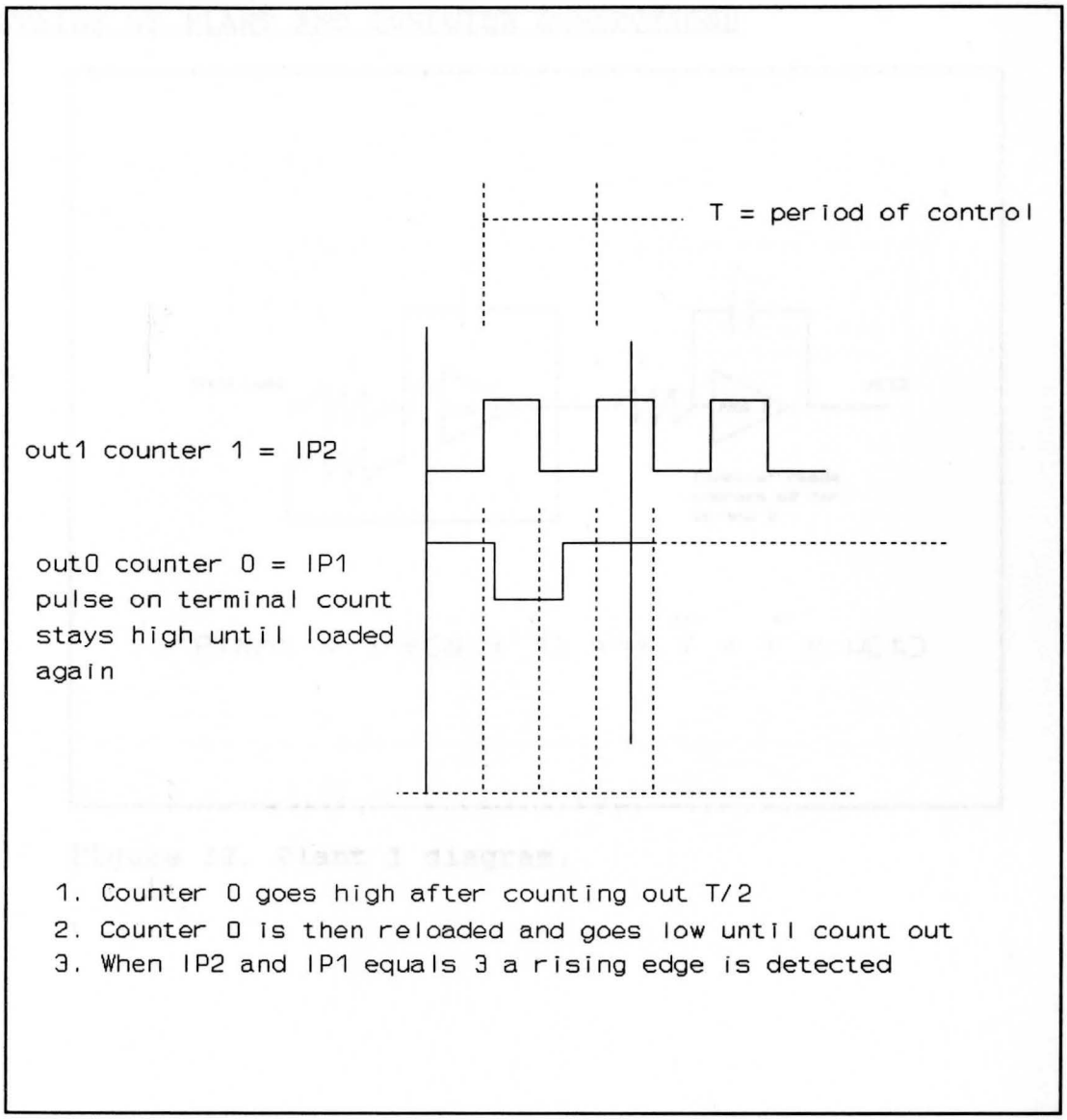


Figure 31. Counter board timing diagram.

DETAILS OF PLANT AND COMPUTER CONNECTIONS

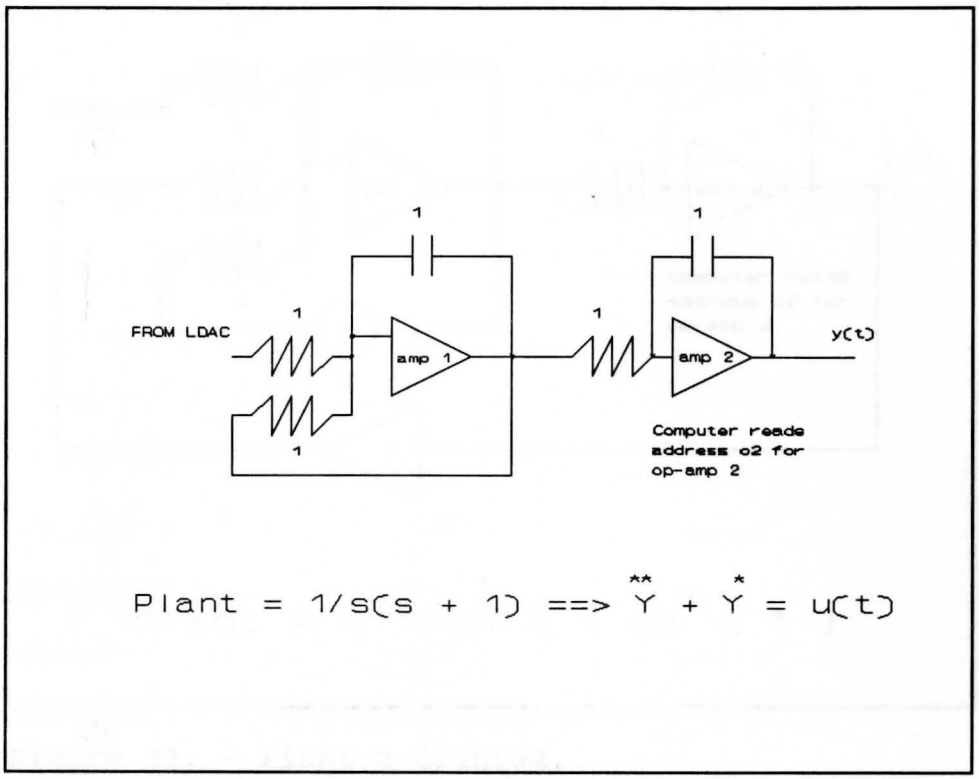


Figure 32. Plant 1 diagram.

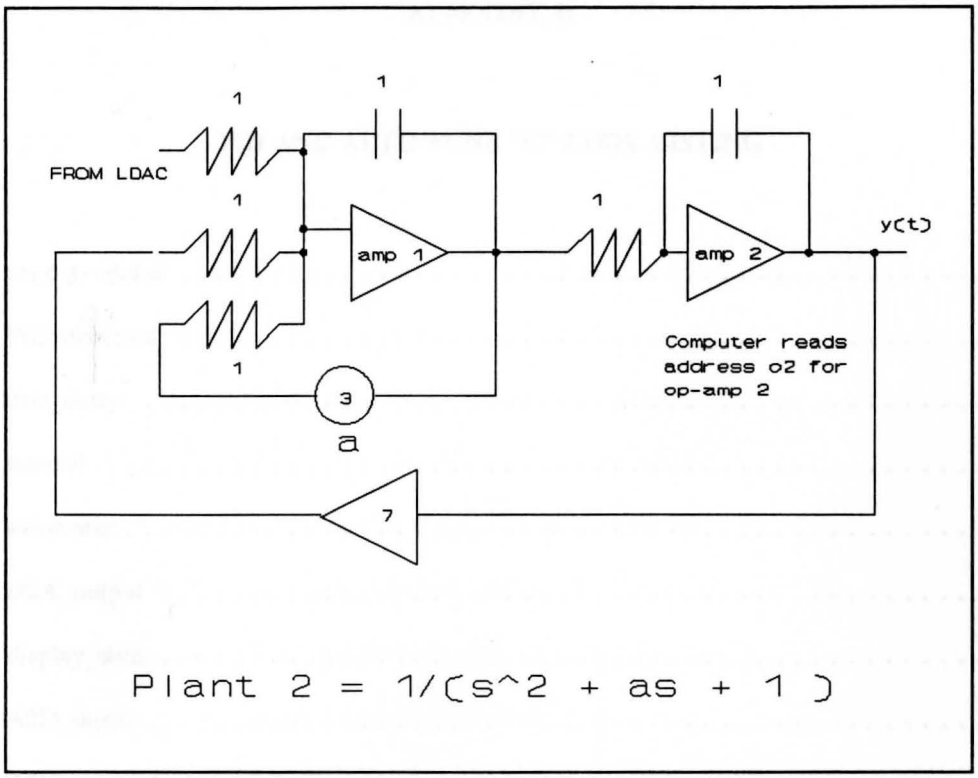


Figure 33. - Plant 2 diagram.

APPENDIX B

PID AND AUTO-TUNE "C" CODE LISTING

```
                    /* Standard PID program module */

#include <stdio.h>

#include <stdlib.h>

struct PID

{

  float setpoint;       /* setpoint value variable % of range */

  float control_uk;     /* value of u(k) control variable % of range */

  float control_uk1;    /* value of u(k-1) control variable % of range */

  float output_yk;      /* value of y(k) output variable % of range */

  float output_yk1;     /* value of y(k-1) output variable % of range */

  float pro_gain_Kc;    /* value of proportional gain variable */

  float parameter_b;    /* value of setpoint weight factor */

  float integral_Ti;    /* integral time variable, seconds */

  float integral_Ik;    /* value of I(k) integral control variable */

  float integral_Ik1;   /* value of I(k+1) integral control variable */

  float sample_h;       /* sample period in seconds */

  float noise_N;        /* noise level variable */

  float derivative_Td;  /* derivative time variable */

  float derivative_Dk;  /* derivative D(k) control variable */

  float derivative_Dk1; /* derivative D(k-1) control variable */

  float relay_step;     /* relay variable output % of range*/

  int *output_addr;     /* control output address D2A pointer (788, 789 LDAC) */

  int *process_addr;    /* process output address A2D pointer (mux no 0, 2, 4, 6) */

  int *setpoint_A2D;    /* analog input address for A2D for setpoint control */

  };

  /* global variables */

  float VOLTS = 10;
```

```
int RANGE = 2047;
struct PID standard_PID; /* declare a structure variable */
        /* functions required for the standard PID control */
void data_entry(struct PID *PID_ptr);   /* get parameters from operator */
void manual(struct PID *PID_ptr);       /* manual control mode function */
void automatic(struct PID *PID_ptr);    /* auto control mode function */
void relay_test(struct PID *PID_ptr);   /* run relay response test */
void ini_timer(struct PID *PID_ptr);    /* initialize timer function */
int ck_timer(struct PID *PID_ptr);      /* check for timeout function */
void A2D_input(struct PID *PID_ptr);    /* update value of output A2D */
void D2A_output(struct PID *PID_ptr);   /* control output function D2A */
void display_data(struct PID *PID_ptr); /* manual ctl display variables function */
    /* main program */
main()
{
        /* other variables */
int select;             /* operator selection */
struct PID standard_PID; /* declare a structure variable */
struct PID *PID_ptr;    /* declare pointer to PID structure */
PID_ptr = &standard_PID; /* assign address of PID structure to pointer */
/* initialize PID structure to default values */
PID_ptr->setpoint = 0;          /* initial setpoint value */
PID_ptr->control_uk = 39;       /* initial u(k) control variable */
PID_ptr->control_uk1 = 0;       /* initial u(k-1) control variable */
PID_ptr->output_yk = 0;         /* initial y(k) output variable */
PID_ptr->output_yk1 = 0;        /* initial y(k-1) output variable */
PID_ptr->pro_gain_Kc = .2;      /* initial proportional gain variable */
```

```
PID_ptr->parameter_b = 1;      /* initial setpoint weight factor */
PID_ptr->integral_Ti = 10;    /* initial integral time variable */
PID_ptr->integral_Ik = 0;      /* initial I(k) integral control variable */
PID_ptr->integral_Ik1 = 0;     /* initial I(k-1) integral control variable */
PID_ptr->sample_h = 0.2;         /* initial sample period in seconds */
PID_ptr->noise_N = 1;           /* initial noise level variable */
PID_ptr->derivative_Td = 1;    /* derivative time variable */
PID_ptr->derivative_Dk = 0;    /* derivative D(k) control variable */
PID_ptr->derivative_Dk1 = 0;   /* derivative D(k-1) control variable */
PID_ptr->relay_step = 10;        /* relay step size variable % range */
*PID_ptr->output_addr = 788;    /* control output address (LDAC = 788) */
*PID_ptr->process_addr = 2;  /* process output address (OP AMP2 = 2) */
*PID_ptr->setpoint_A2D = 8; /* analog input A4 = 8 setpoint ctl */
/* determine operator desires */
while (select != 5)
{
  printf("Enter 1 if you wish to enter parameters\n");
  printf("Enter 2 if you wish to run in manual\n");
  printf("Enter 3 if you wish to run in auto\n");
  printf("Enter 4 if you wish to run auto tune\n");
  printf("Enter 5 if you wish to quit the program\n");
  scanf("%d", &select);
  switch(select)
  {
      case 1:
        /* enter parameters function */
        data_entry(&standard_PID);
```

```
            break;
        case 2:
            /* go to manual mode */
            manual(&standard_PID);
            break;
        case 3:
            /* go to auto mode function */
            automatic(&standard_PID);
            break;
        case 4:
            /* relay feedback function */
            relay_test(&standard_PID);    /* run relay response test */
            break;
        }
    }
}
void data_entry(struct PID *PID_ptr) /* get parameters from operator */
{
    int select, address;
    float value;

    while (select != 12)
    {
    printf("Select the parameter to be changed\n\n");
    printf("1. Setpoint = %f\n", PID_ptr->setpoint);
    printf("2. Proportional Gain = %f\n", PID_ptr->pro_gain_Kc);
    printf("3. Setpoint weight factor b = %f\n", PID_ptr->parameter_b);
```

```
printf("4. Integral time variable Ti = %f\n", PID_ptr->integral_Ti);
printf("5.  Sample time in seconds = %f\n", PID_ptr->sample_h);
printf(" Sample time in seconds = %f\n", PID_ptr->sample_h);
printf("6. Noise level variable N = %f\n", PID_ptr->noise_N);
printf("7. Derivative time constant Td = %f\n", PID_ptr->derivative_Td);
printf("8. Step size output value = %f\n", PID_ptr->relay_step);
printf("9. Control output address = %d\n", *PID_ptr->output_addr);
printf("10. Process output address = %d\n", *PID_ptr->process_addr);
printf("11. Setpoint input address = %d\n", *PID_ptr->setpoint_A2D);
printf("12. Exit - finished\n");
scanf("%d", &select);
switch(select)
 {
case 1:
  /* Change setpoint */
      printf("Enter new setpoint between 0 and 10\n");
      scanf("%f", &value);
      PID_ptr->setpoint = value;
      break;
case 2:
  /* Change proportional gain */
      printf("Enter new proportional gain greater than 0\n");
      scanf("%f", &value);
      PID_ptr->pro_gain_Kc = value;
      break;
case 3:
  /* Setpoint weight factor */
```

```
        printf("Enter new weight factor from 0 to 1\n");
        scanf("%f", &value);
        PID_ptr->parameter_b = value;
        break;
case 4:
  /* Integral time variable */
        printf("Enter new integral time variable or 0 for none\n");
        scanf("%f", &value);
        PID_ptr->integral_Ti = value;
        break;
case 5:
  /* Sample period in seconds */
        printf("Enter new sample period in seconds from .020\n");
        scanf("%f", &value);
        PID_ptr->sample_h = value;
        break;
case 6:
  /* Noise level variable */
        printf("Enter new noise level variable N\n");
        scanf("%f", &value);
        PID_ptr->noise_N = value;
        break;
case 7:
  /* Derivative time variable */
        printf("Enter new derivative time variable or 0 for none\n");
        scanf("%f", &value);
        PID_ptr->derivative_Td = value;
```

```
        break;
    case 8:
      /* Step size for step response */
        printf("Enter new step size > 0\n");
        scanf("%f", &value);
        PID_ptr->relay_step = value;
        break;
    case 9:
      /* Control output address */
        printf("Enter new control address\n");
        scanf("%d", &address);
        *PID_ptr->output_addr = address;
        break;
    case 10:
      /* Process output address */
        printf("Enter new process address\n");
        scanf("%d", &address);
        *PID_ptr->process_addr = address;
        break;
    case 11:
      /* Setpoint input address */
        printf("Enter new setpoint input address\n");
        scanf("%d", &address);
        *PID_ptr->setpoint_A2D = address;
        break;
    }
}
```

```
}
void manual(struct PID *PID_ptr) /* run in manual */
{
  int HB, LB, U, a, b, digital_inp;
  int select, address;
  float value;
  while (select != 3)
  {
  printf("Select the desired option:\n\n");
  printf("1. Set control output = %f\n", PID_ptr->control_uk);
  printf("2. Run in manual\n");
  printf("3. Exit - finished\n");
  scanf("%d", &select);
  switch(select)
   {
   case 1:
     /* Change control output */
        printf("Enter new control output value in percent\n");
        scanf("%f", &value);
        PID_ptr->control_uk = value;
        break;
   case 2:
     /* Run in manual */
        D2A_output(PID_ptr); /* control output function */
        A2D_input(PID_ptr);   /* update value of output variable for display */
        display_data(PID_ptr);  /* display variables function */
        break;
```

```
    }
   }
}
void automatic(struct PID *PID_ptr) /* run in automatic */
{
   float integral_gain, derivative_gain1, nd, dd, derivative_gain2, nd1, dd2;
   float Proportional_op, a, b, c, d;
   float temp_op, e, error;
   int done = 1, timer_flag = 0, digital_inp=0, GP_6;
   /* calculate regulator coefficients */
   if(PID_ptr->integral_Ti != 0) /* is integral function on ? */
     integral_gain = PID_ptr->pro_gain_Kc*PID_ptr->sample_h/PID_ptr->integral_Ti;
     else
     {
      integral_gain = 0;
      PID_ptr->integral_Ik = 0;
     }
   if(PID_ptr->derivative_Td != 0) /* is derivative function on ? */
     {
     nd = 2 * PID_ptr->derivative_Td - PID_ptr->sample_h*PID_ptr->noise_N;
     dd = 2 * PID_ptr->derivative_Td + PID_ptr->sample_h*PID_ptr->noise_N;
     derivative_gain1 = nd/dd;
     nd1 = 2 * PID_ptr->pro_gain_Kc * PID_ptr->noise_N * PID_ptr->derivative_Td;
     dd2 = 2 * PID_ptr->derivative_Td + PID_ptr->noise_N * PID_ptr->sample_h;
     derivative_gain2 = nd1/dd2;
      }
     else
```

```
    {
     derivative_gain1 = 0;
     derivative_gain2 = 0;
    }
    /* Check to see if program is to stop running */
        outp(785,0);  /* start GP-6 operate mode */
        GP_6 = inp(786);
        while(GP_6 != 4)
         {
          GP_6 = inp(786);
          GP_6 &=4;
         }
    ini_timer(PID_ptr); /* initialize timer function */
while (digital_inp == 0)
{
        digital_inp = inp(770); /* DAS8 status register */
        digital_inp = (digital_inp & 0x0040)/16; /* isolate IP3 */
        A2D_input(PID_ptr);    /* update value of output variable */
        A2D_setpoint(PID_ptr);  /* update value of setpoint variable */
        timer_flag = ck_timer(PID_ptr);      /* check for timeout function */
        if(timer_flag == done)
        {
          digital_inp = inp(770); /* DAS8 status register */
          digital_inp = (digital_inp & 0x0040)/16; /* isolate IP3 */
          system("cls");
          printf("Setpoint \tControl \tProcess O/P\n");
          printf("%f \t%f \t%f\n", PID_ptr->setpoint, PID_ptr->control_uk,
```

```
PID_ptr->output_yk);
        /* calculate proportional output */
        a = PID_ptr->parameter_b * PID_ptr->setpoint;
        b = PID_ptr->output_yk;
        Proportional_op = PID_ptr->pro_gain_Kc * (a - b);
        /* calculate derivative output */
        if(PID_ptr->derivative_Td != 0)
        {
         c = derivative_gain1 * PID_ptr->derivative_Dk;
         d = derivative_gain2 * (PID_ptr->output_yk -PID_ptr->output_yk1);
         PID_ptr->derivative_Dk = c - d;
        }
         else
         PID_ptr->derivative_Dk = 0;
        temp_op = Proportional_op + PID_ptr->derivative_Dk +
PID_ptr->integral_Ik;
        /* check value for saturation */
        if (temp_op > 100)
              temp_op = 100;
        if (temp_op < -100)
              temp_op = -100;
        PID_ptr->control_uk = temp_op;
        /* calculate new integral control value */
        error = PID_ptr->setpoint - PID_ptr->output_yk;
        PID_ptr->integral_Ik += integral_gain * error;
        /* update old output variable */
        PID_ptr->output_yk1 = PID_ptr->output_yk;
```

```
            D2A_output(PID_ptr); /* control output function */
        }
    }
        printf("\n"); /* new line */
        outp(785,8); /* put GP_6 in IC mode */
        PID_ptr->control_uk = 0; /* set control variable */
        D2A_output(PID_ptr); /* control output function */
}
void D2A_output(struct PID *PID_ptr) /* control output function */
{
    int HB, LB;
    float  U, rndHB;
    /* convert to data for D2A */
    U =  (PID_ptr->control_uk/100) * RANGE;
    rndHB = U/16;
    if(U < 0)
     rndHB -=.5;
    HB = rndHB;
    LB = (U - 16 * HB) * 16;
    /* output to addresses for D2A */
    outp(*PID_ptr->output_addr, LB);
    outp(*PID_ptr->output_addr+1, HB);
}
void display_data(struct PID *PID_ptr)  /* manual ctl display variables function */
{
    int digital_inp, GP_6;
    /* Check to see if program is to stop displaying data */
```

```
        digital_inp = inp(770); /* DAS8 status register */
        digital_inp = (digital_inp & 0x0040)/16; /* isolate IP3 */
        outp(785,0);  /* start GP-6 operate mode */
        GP_6 = inp(786);
        while(GP_6 != 4)
        {
         GP_6 = inp(786);
         GP_6 &=4;
        }
        while (digital_inp == 0)
        {
          system("cls");
          printf("Setpoint \tControl \tProcess O/P\n");
          printf("%f \t%f \t%f\n", PID_ptr->setpoint, PID_ptr->control_uk,
PID_ptr->output_yk);
          digital_inp = inp(770); /* DAS8 status register */
          digital_inp = (digital_inp & 0x0040)/16; /* isolate IP3 */
          A2D_input(PID_ptr);    /* update value of output */
        }
          printf("\n"); /* new line */
          outp(785,8); /* set C3 high for GP_6 IC mode */
}
void A2D_input(struct PID *PID_ptr)  /* update value of output */
{
   int status, HB, LB, count=0;
   outp(784,*PID_ptr->process_addr); /* set mux */
   /* delay for sample & hold amp */
```

```
    while(count < 80)
    {
    count ++;
    }
    outp(786,*PID_ptr->process_addr); /* start A2D */
    status = inp(786); /* get status byte */
    status &=8;        /* isolate EOC bit */
    while(status !=8) /* wait for EOC */
    {
    status = inp(786); /* get status byte */
    status &=8;        /* isolate EOC bit */
    }
    HB = inp(787);
    LB = inp(786);
    PID_ptr->output_yk = HB*16 + LB/16; /* convert to a word of data */
    if(PID_ptr->output_yk > 2047)
       PID_ptr->output_yk -=4096;
       PID_ptr->output_yk = PID_ptr->output_yk * 100 / 2047; /* % of range */
}
A2D_setpoint(struct PID *PID_ptr)  /* update value of setpoint variable */
{
    int status, HB, LB, count=0;
    outp(784,*PID_ptr->setpoint_A2D); /* set mux */
    /* delay for sample & hold amp */
    while(count < 80)
    {
    count ++;
```

```
    }
    outp(786, *PID_ptr->setpoint_A2D); /* start A2D */
    status = inp(786); /* get status byte */
    status &=8;        /* isolate EOC bit */

    while(status !=8) /* wait for EOC */
    {
    status = inp(786); /* get status byte */
    status &=8;        /* isolate EOC bit */
    }
    HB = inp(787);
    LB = inp(786);
    PID_ptr->setpoint = HB*16 + LB/16; /* convert to a word of data */
    if(PID_ptr->setpoint > 2047)
      PID_ptr->setpoint -=4096;
      PID_ptr->setpoint = PID_ptr->setpoint * 100 / 2047; /* % of range */
}
void ini_timer(struct PID *PID_ptr)    /* initialize timer function */
{
  int count0, count1, count2, time_sample;
  unsigned int low_byte, high_byte;
  /* Setup counter 2 in configuration 3: square generator */
  outp(775,182); /* control word for counter 2 */
  /* Load counter 2 */
  count2 = PID_ptr->sample_h * 395 /.200; /* 395 counts for 200 milliseconds */
  low_byte = count2 & 0x00FF;
  high_byte = (count2 & 0xFF00)/256;
```

```
    outp(774,low_byte); /* Low byte for counter 2 */
    outp(774,high_byte); /* High byte for counter2 */
    /* Setup counter 1 in configuration 3: square wave generator */
    outp(775,118); /* control word for counter 1 */
    /* Load counter 1 */
    time_sample = PID_ptr->sample_h * 1000;  /* sample period in milliseconds */
    count1 = 10 * time_sample;
    low_byte = count1 & 0x00FF;
    high_byte = (count1 & 0xFF00)/256;
    outp(773,low_byte); /* Low byte for counter1 */
    outp(773,high_byte); /* High byte for counter1 */
    /* Setup counter 0 in configuration 0 */
    outp(775,48); /* control word for counter 0 */
    /* Load counter 0 */
    count0 = 5 * time_sample;
    low_byte = count0 & 0x00FF;
    high_byte = (count0 & 0xFF00)/256;
    outp(772,low_byte); /* Low byte for counter0 */
    outp(772,high_byte); /* High byte for counter0 */
}
int ck_timer(struct PID *PID_ptr)     /* check for timeout function */
{
    int count0, digital_inp, time_sample;
    unsigned int low_byte, high_byte;
    digital_inp = inp(770); /* DAS8 status register */
    digital_inp = (digital_inp & 0x0070)/16; /* isolate the inputs */
    if (digital_inp == 3)  /* check for counter 0 done */
```

```
    {
     time_sample = PID_ptr->sample_h * 1000;  /* sample period in milliseconds */
     count0 = 5*time_sample;
     low_byte = count0 & 0x00FF;
     high_byte = (count0 & 0xFF00)/256;
     outp(772,low_byte); /* Low byte for counter0 */
     outp(772,high_byte); /* High byte for counter0 */
     return 1; /* timer done and reloaded */
    }
    else
     return 0; /* timer not done */
}
void relay_test(struct PID *PID_ptr)    /* run relay response test */
{
  int peak_count = 0, time_counter = 0, peak_time[12], j;
  int cycle_count = 0, initialize = 0, neg_cycle, done = 1, first_pass = 0;
  int timer_flag, digital_inp, GP_6, period_sum=0;
  float current_op, upper_limit, lower_limit, period_ave=0;
  float neg_peak_ave=0, neg_peak_sum=0, pos_peak_ave, pos_peak_sum;
  float ultimate_gain, peak_value[12];
 /* Initialize test parameters and start test */
  if(initialize == 0)
  {
        A2D_input(PID_ptr);    /* update value of output */
        current_op = PID_ptr->output_yk;   /* store current output */
   /* initialize the peak_value array */
        for(j = 0; j < 13; j++)
```

```
        peak_value[j] = current_op;
    /* establish limits of relay output */
        upper_limit = PID_ptr->relay_step;
        lower_limit = - PID_ptr->relay_step;
        PID_ptr->control_uk = upper_limit; /* set control variable */
        D2A_output(PID_ptr); /* control output function */
    /* wait for output to exceed half the relay step size */
        outp(785,0);  /* start GP-6 operate mode */
        GP_6 = inp(786);
        while(GP_6 != 4)
        {
         GP_6 = inp(786);
         GP_6 &=4;
        }
        while(PID_ptr->output_yk <= current_op + PID_ptr->relay_step / 2 )
          A2D_input(PID_ptr);   /* update value of output */
    /* switch output to lower limit */
        PID_ptr->control_uk = lower_limit;
        D2A_output(PID_ptr); /* control output function */
    /* monitor output for crossover at the original output value */
        while(PID_ptr->output_yk > current_op)
          A2D_input(PID_ptr);   /* update value of output */
    /* start test - initialize timer,  counters */
        ini_timer(PID_ptr);    /* initialize timer function */
        initialize = 1; /* end of initialize routine */
        neg_cycle = 1; /* start negative cycle */
    } /* end of initialize */
```

```
while (cycle_count < 7) /* run 6 cycles and stop */
{
 /* negative peak 1/2 cycle */
    while(neg_cycle == 1)
    {
 /* switch output to upper limit */
      if(first_pass == 0)
       {
        PID_ptr->control_uk = upper_limit; /* set control variable */
        D2A_output(PID_ptr); /* control output function */
        first_pass = 1;
       }
      timer_flag = ck_timer(PID_ptr);     /* check for timeout function */
      if(timer_flag == done)
       {
        A2D_input(PID_ptr);    /* update value of output */
        time_counter +=1;
        if(PID_ptr->output_yk < peak_value[peak_count])
        {
          peak_value[peak_count] = PID_ptr->output_yk; /* store new peak value */
          peak_time[peak_count] = time_counter; /* store time of peak */
        }
        if(PID_ptr->output_yk > current_op) /* check for crossover */
  {
          neg_cycle = 0; /* terminate negative cycle */
          peak_count ++; /* increment peak counter */
          first_pass = 0; /* update flag */
```

```
    }
   }
  }
/* positive peak 1/2 cycle - neg */
  while(neg_cycle == 0)
  {
/* switch output to lower limit */
   if(first_pass == 0)
    {
     PID_ptr->control_uk = lower_limit; /* set control variable */
     D2A_output(PID_ptr); /* control output function */
     first_pass = 1;
    }
   timer_flag = ck_timer(PID_ptr);     /* check for timeout function */
   if(timer_flag == done)
    {
     A2D_input(PID_ptr);   /* update value of output */
     time_counter +=1;
     if(PID_ptr->output_yk > peak_value[peak_count])
      {
    /* store new peak value */
     peak_value[peak_count] = PID_ptr->output_yk;
          peak_time[peak_count] = time_counter; /* store time of peak */
      }
     if(PID_ptr->output_yk < current_op) /* check for crossover */
      {
          neg_cycle = 1; /* terminate positive cycle */
```

```
                peak_count ++; /* increment peak counter */
                first_pass = 0; /* update flag */
                cycle_count ++; /* increment cycle counter */
            }
        }
    }
}
  PID_ptr->control_uk = 0; /* set control variable */
  D2A_output(PID_ptr); /* control output function */
  /* analyze the data and calculate the PID parameters */
  for (j = 0; j < 11; j++)
    period_sum += (peak_time[j+1] - peak_time[j]);
  period_ave = period_sum * 2 / 11;
  period_ave *=PID_ptr->sample_h;
  for (j = 0; j < 11; j+=2)
    neg_peak_sum += peak_value[j]; /* even numbers in array */
  neg_peak_ave = neg_peak_sum / 6;
  for (j = 1; j < 12; j+=2)
    pos_peak_sum += peak_value[j]; /* odd numbers in array */
  pos_peak_ave = pos_peak_sum / 6;
  /* calculate PID parameters */
  ultimate_gain = 4*(PID_ptr->relay_step * 2) / (3.14159 * (pos_peak_ave -        neg_peak_ave));
  PID_ptr->pro_gain_Kc = .35 * ultimate_gain;
  PID_ptr->integral_Ti = 1.13 * period_ave;
  PID_ptr->derivative_Td = .18 * PID_ptr->integral_Ti;
  outp(785,8); /* set C3 high for GP_6 IC mode */
}
```

REFERENCES

[1] Astrom, Karl J., Automatic Tuning of PID Controllers, Instrument Society of America, NC, 1988.

[2] Brogan, William L., Modern Control Theory, Prentice Hall, N. J., 1991.

[3] Carthers, Felix P., Adaptive Control Systems, The Macmillian Company, N.Y., 1963.

[4] Corripio, Armando B., Tuning of Industrial Control Systems, Instrument Society of America, NC, 1990.

[5] Davies, W. D. T., System Identification for Self-Adaptive Control, Wiley-Interscience, N. Y. 1970.

[6] Eveleigh, Virgil W., Adaptive Control and Optimization Techniques, McGraw-Hill Book Co., N. Y. 1967.

[7] Gupta, Madan, Adaptive Methods for Control System Design, IEEE Press, N. Y. 1986.

[8] Hang, Chang C., Adaptive Control, Instrument Society of America, NC, 1993.

REFERENCES

[9] Liptak, Bela G., Instrument Engineer's Handbook, Chilton Book Company, Penn., 1982.

[10] Ogata, Katsuhito, System Dymamics, Prentice Hall, N. J. 1992.

[11] Phillips, Charles L., Feedback Control Systems, Prentice Hall, 1991.

[12] Shinners, Stanley M., Modern Control System Theory and Design, John Wiley and Sons, N. Y., 1992.

[13] Slotine, Jean-Jacques E., Applied Non-Linear Control, Prentice Hall, N. J. 1991.

[14] Truxal, John G., Control Engineer's Handbook, McGraw-Hill Book Co., N. Y., 1958.

[15] Operator's Manual, GP-6 Analog Computer, Comdyna, Inc., IL.

[16] Operator and Maintenance Manual, Model 767 Analog / Digital Position Control Panel, Comdyna Inc, IL.

[17] DAS8 User's Manual, MetraByte Corporation, Ma.