The Design and Implementation of an Advanced Robot Controller

Jonathan Neil Brodkin

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montréal, Québec, Canada

November 1990

ISBN 0-315-90942-0

ABSTRACT

The Design and Implementation of an

Advanced Robot Controller

Jonathan N. Brodkin

This thesis describes an architecture for implementing an Advanced Robot Controller (ARC). The ARC has been designed and built as a replacement for the controllers of conventional industrial robotic systems. The need for such a device arises because most industrial robots are constrained by their controllers' hardware and software to performing in a simple, pre-defined manner. The ARC hardware consists of a hierarchically-designed dual-processor operating in a master-slave configuration and communicating via dual-port RAM, a bank of quadrature decoders/counters, and a bank of digital to analog converters. A software library has been established consisting of various control and path planning algorithms. This library also contains a set of functions to facilitate algorithm coding and testing. The ARC was integrated with an IBM 7545 Manufacturing System and proved extremely successful in achieving tight control over the 7545's four joints.

ACKNOWLEDGEMENTS

In submitting this thesis I must acknowledge Dr. R. V. Patel for the guidance and support that he provided to me throughout the course of my research and writing; but above all I must thank Dr. Patel for his unwavering trust in my abilities which contributed to my motivation and confidence in this endeavor.

There are several others whose assistance was most appreciated at particular junctures, including hardware suggestions, circuit construction, programming, debugging and text review: Guy Gosselin, Harvey Brodkin, Alan Robins, Claude-Marie Lafforgue, Henry Polley, Mike Stashin, Vladimir Zeman, Venkatram Pramod, and Claude Tessier.

Finally, I would like to thank my family and friends for their constant and enthusiastic support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 INDUSTRIAL ROBOTIC SYSTEMS

Since the early 1960s, robotic systems have steadily been replacing traditional automation techniques. Successful applications of robot manipulators have included manufacturing and assembly, materials handling and inspection in fixed automation environments. This type of automation involves the repetition of a specific sequence of operations for the continuous high-volume production of identical or nearly identical parts. Industrial robots intended for use in the fixed automation environment have been designed to perform relatively simple tasks in a preconceived manner. To perform these tasks, the robots require few if any sensors and practically no intelligent decision-making ability. Controllers for these systems, therefore, are based on weak processing hardware and unsophisticated control and trajectory generation software. These types of robots are generally referred to as "dumb" robots since they are inflexible to changes in their workspace and product design.

A characteristic of dumb robots is their high-level, application-oriented user-interface which enables the operator to specify a small variety of "move-type" commands. With this type of user-interface, the operator has little control over the desired trajectory and virtually no control over the servoing of the joint motors. In some cases, this limitation may adversely affect the robot's efficiency. For example, there exist dumb robotic systems whose application languages do not permit the specification of via points; to negotiate around an obstacle, the user must specify a piecewise path and the robot stops at the end of each segment.

Although there will always be applications for dumb robots, there is presently a growing demand for industrial robots which can operate in a flexible automation environment. This type of environment is dynamically complex, often containing moving obstacles and/or other manipulators sharing the same workspace. To perform in this environment, a robot must possess a significant level of intelligence as well as an assortment of advanced sensors. Although there will always be applications for which dumb robots are well suited, many may soon prove to be obsolete as the demand for more efficient and intelligent industrial robotic systems increases. This is unfortunate when one considers that relatively expensive robotic manipulators will go to waste simply because of the shortcomings of their controllers.

In addition to the industrial domain, robots are frequently found in research environments. Here, they are being used for testing modern advanced control strategies for the purpose of both basic research and prototype development. Once again, the limitations associated with the controllers of dumb robotic systems render them inadequate to serve in such a domain. The research presented in this thesis was conceived in order to overcome this problem.

The IBM 7545 Manufacturing System that was used in our research is representative of a large class of industrial robotic systems and can be classified as a dumb robot. Although the 7545 system is an older generation IBM system, its kirematic configuration is essentially identical to that of current IBM robotic systems; the difference being the intelligence of the controllers. Because the intelligence of the 7545 controller is markedly lower, it does not possess some of the features that have been incorporated into latest generation systems, e.g., the ability to specify desired trajectories. What these two generations of systems do share, however, is their inability to evolve as advanced requirements arise. This inability can be attributed to the design philosophy of the 7545

controller which like most robotic controllers makes it inherently difficult to modify the hardware and software.

1.2 SUMMARY OF THE RESEARCH WORK

One solution to overcoming the shortcomings mentioned above is to bypass the manufacturer-supplied controller with an alternate controller possessing sufficient processing power to execute various advanced robot control strategies and an architecture which would support future enhancement. This controller could then be programmed for use in an industrial or research environment. This was the solution adopted for the IBM 7545 Manufacturing System.

The Advanced Robot Controller (ARC) which has been designed and implemented incorporates a master/slave paradigm that assigns to the slave processor the tedious yet essential elements of implementing servo control (e.g., robot state acquisition and limit checking such as joint overruns and excessive torque demands). The slave takes over a significant percentage of the computations required for servo control and thus allows more computationally complex control strategies to be executed on the master. Although the results presented in this thesis reflect work undertaken on a particular commercial robotic system, the approach taken is easily applicable to virtually any robot. It is worth mentioning that an important constraint in the design and implementation of the ARC is that the resulting hardware/software is inexpensive (a small fraction of the cost of the robot) and relatively easy to develop and maintain.

Other researchers have recently reported developments of specialized robotic controllers for the same purpose [1-4]. The systems in [1] and [2] are based on simple single processor architectures. In [1] a robot controller based on an Intel 310 (6 MHz, 80286 processor) running XENIX and with its servo control algorithm linked to the Kernel was introduced. The controller was implemented and evaluated on a six degrees of freedom

PUMA 560 robot and was able to execute PD control for the PUMA's six joints at a servo rate of 100 Hz [5]. In [2] a specialized computer-robot interface was designed. The interface was used to link an IBM AT to a Mitsubishi RM-501 robot for the evaluation of simple and advanced control strategies. A 6.0 millisecond sampling period was achieved for PD control of the robot's first two joints [6]. The methodology and design philosophies which were adopted by us independently of [1] and [2] are in accordance with the suggestions found in the conclusions of these references. It should be noted that the cost of the ARC is of the same order as those for the architectures proposed in [1] and [2]. The systems described in [3] and [4] consist of host SUN workstations directing the efforts of multiple (3-5) VME-based 68000-series processors. These systems exhibit superior performance but at very significantly higher cost.

1.3 THESIS ORGANIZATION

An introduction to the limitations associated with typical commercial industrial robot systems has been given in this chapter. Chapter 2 describes the IBM 7545 Manufacturing System which was used in this research. The required hardware modifications to the 7545 system are also described. Chapter 3 presents a complete ARC hardware description, justifying the ARC's adequacy for controlling the 7545 manipulator. The ARC software description is given in Chapter 4. This covers the high-level master software consisting of control and path planning algorithms as well as the low-level slave algorithms that were developed to increase the performance of the system. Suggestions for future extensions are included in both Chapters 3 and 4. Chapter 5 presents the results of experiments that were designed to show the ARC's effectiveness in executing various control strategies, including proportional-derivative and decentralized adaptive controllers and various path generation schemes. Concluding remarks are given in Chapter 6.

CHAPTER 2

THE IBM 7545 MANUFACTURING SYSTEM

2.1 INTRODUCTION

The IBM 7545 Manufacturing System (Figure 2.1) consists of a control unit, an operator control panel, a manipulator which consists of a four-joint, DC servo-actuated, Selective Compliance Assembly Robot Arm (SCARA) [7], and an application programming device (not shown). The contents of these components are shown in the block diagram in Figure 2.2.

The control unit houses a power supply, drivers for the servo motors, a motor control board (MTCB), and an interface and power distribution board (relay board). The MTCB contains a Z80 microprocessor, memory, communications interface circuits, and robot motion control circuits.

The 7545 system is pre-programmed by the user for a particular application using the AML/E (A Manufacturing Language/Editor) programming language which runs on the application programming device (e.g., IBM PC or compatible). The application program is compiled by AML/E and then downloaded to the MTCB and executed. AML/E supports simplistic high-level commands such as "move" and "grasp". All trajectory planning and joint motor control is performed by the MTCB in a predefined fashion. Inherent in the MTCB design are the shortcomings described in Chapter 1; consequently the MTCB is the sole 7545 component which needs to be bypassed. All other 7545 components can remain operationally intact.

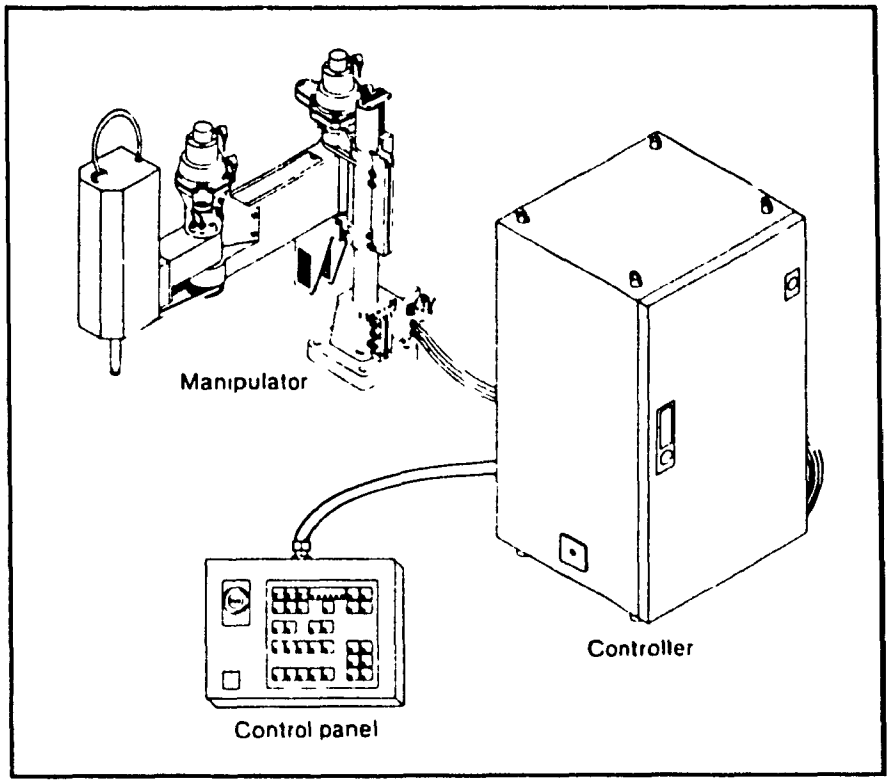


Figure 2.1. IBM 7545 Manufacturing System [7].

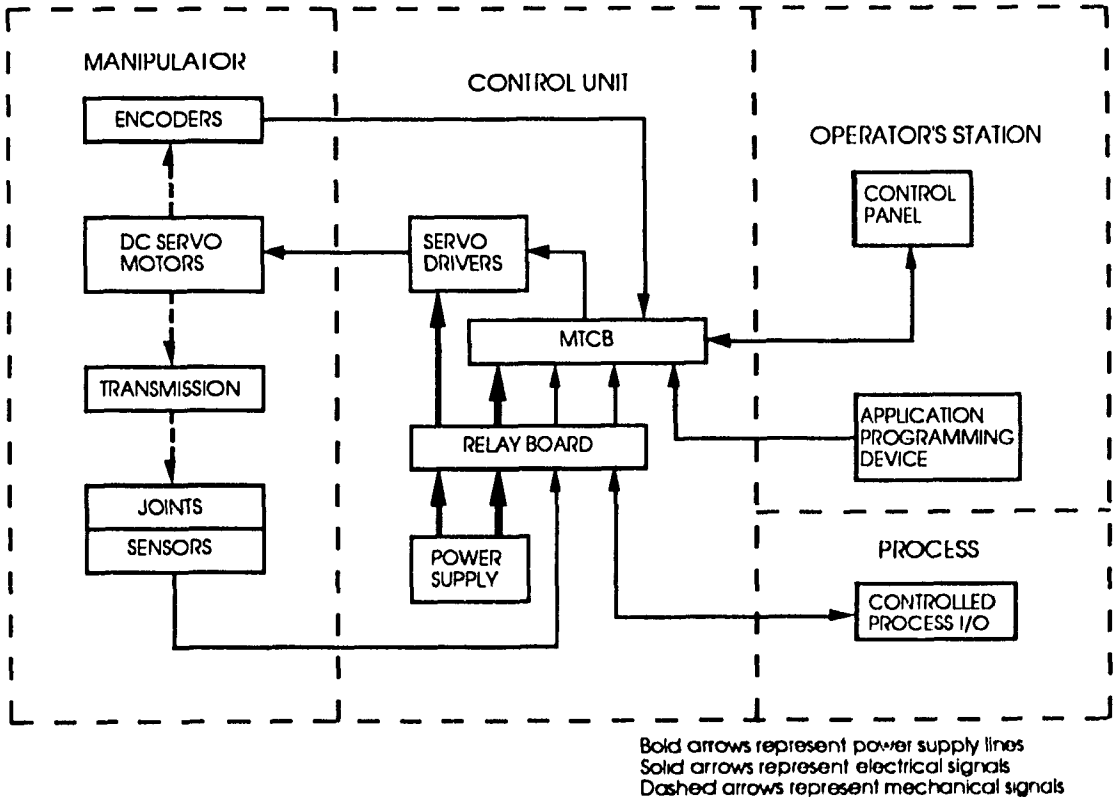


Figure 2.2. Block Diagram of the IBM 7545 Manufacturing System [7].

This chapter provides the necessary information on the 7545 system to effectively bypass the MTCB with the ARC that has been designed. This includes the 7545 specifications as well as the required modifications to various 7545 system components.

2.2 THE IBM 7545 MANIPULATOR

The manipulator of the 7545 Manufacturing System (Figure 2.3) is a four-degrees-of-freedom mechanism with rigid links. The first two revolute joints of the arm, the shoulder (θ_1) and the elbow (θ_2), give two degrees of freedom in the horizontal x-y plane. θ_1 is measured with respect to the x-axis (see Figure 2.3) and its range of motion is 0-200°. θ_2 is measured with respect to the radial axis of link 1 and has a range of motion of 0- 135°.

Located at the end of the arm is a prismatic joint giving one degree of freedom along the vertical z-axis. Attached to the end of this z-axis shaft is a pneumatic gripper. In its uppermost position, the gripper lies in the x-y plane and the joint variable Z (uppercase denotes joint variable) is zero. The shaft can extend 250 mm in the negative z direction. This is the only joint motion affected by gravity.

The Roll joint consists of the revolution of the z-axis shaft (gripper) providing one degree of orientational freedom. The Roll angle, θ_{roll}, is measured with respect to the x-axis and as will be shown in the following section, is independent of θ_1, θ_2, and Z. The range of motion for this joint is from -180° to +180°.

The forward kinematic equations for the manipulator are:

$$\begin{aligned} x &= \ell_2 \cos(\theta_1 + \theta_2) + \ell_1 \cos\theta_1 \\ y &= \ell_2 \sin(\theta_1 + \theta_2) + \ell_1 \sin\theta_1 \\ z &= -h \end{aligned} \qquad (2.1)$$

where ℓ_1 = 400 mm and ℓ_2 = 250 mm are the lengths of links 1 and 2, respectively.

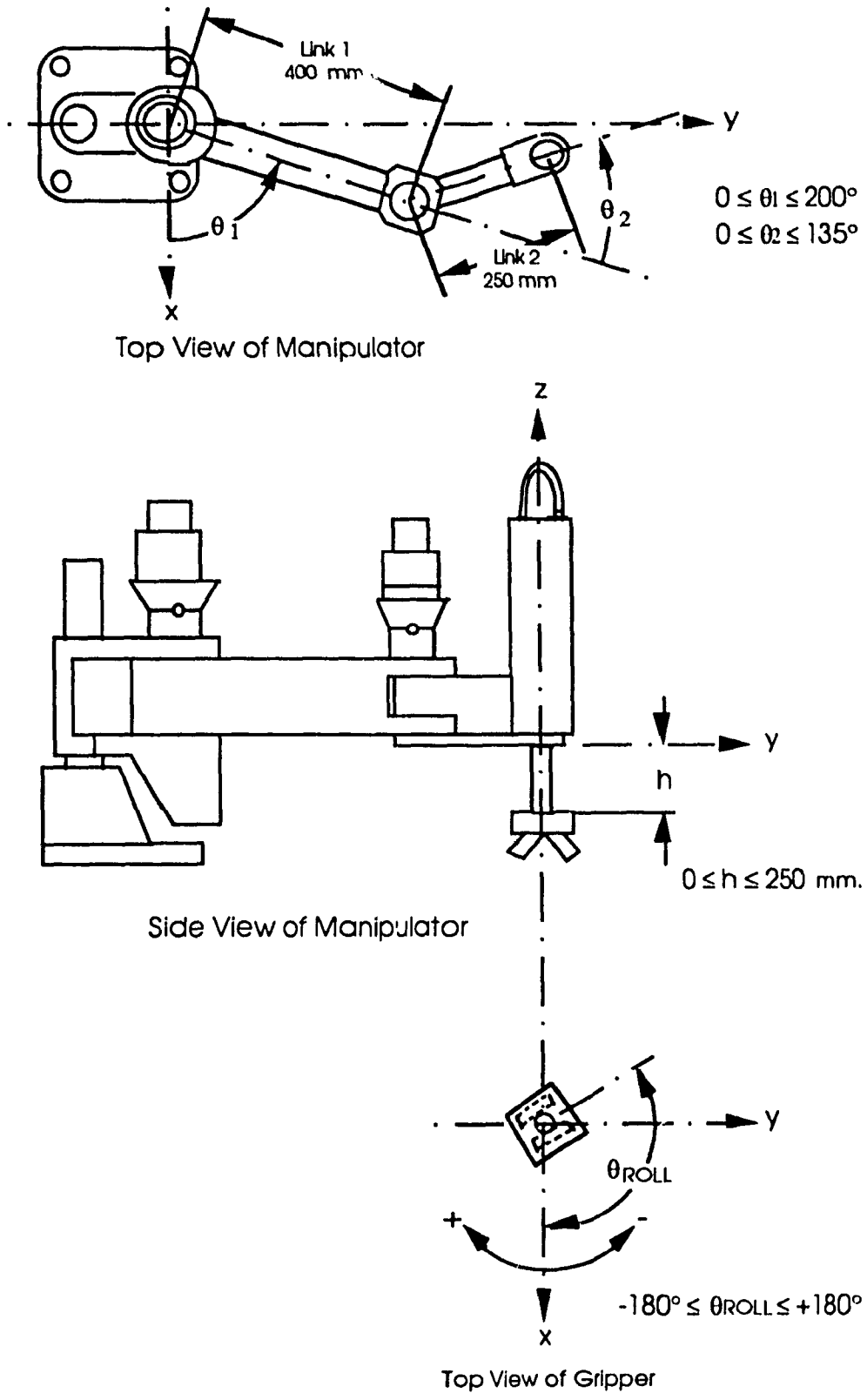


Figure 2.3. IBM 7545 Manipulator [7].

The inverse kinematic equations are:

$$\theta_1 = \text{Atan2}(y, x) + \text{Atan2}(\pm\sqrt{x^2 - y^2 - \omega^2}, \omega) \qquad (2.2)$$

$$\theta_2 = \text{Atan2}(-x \sin\theta_1 + y \cos\theta_1,\ x \cos\theta_1 + y \sin\theta_1 - \ell_1)$$

The dimensions of the manipulator's work envelope are shown in Figure 2.4. The manipulator's HOME (reference) position is located in joint space at $\theta_1 = \theta_2 = \theta_{roll} = 0°$, Z = 0 mm, or in Cartesian space at x = 650 mm, y = z = 0 mm.

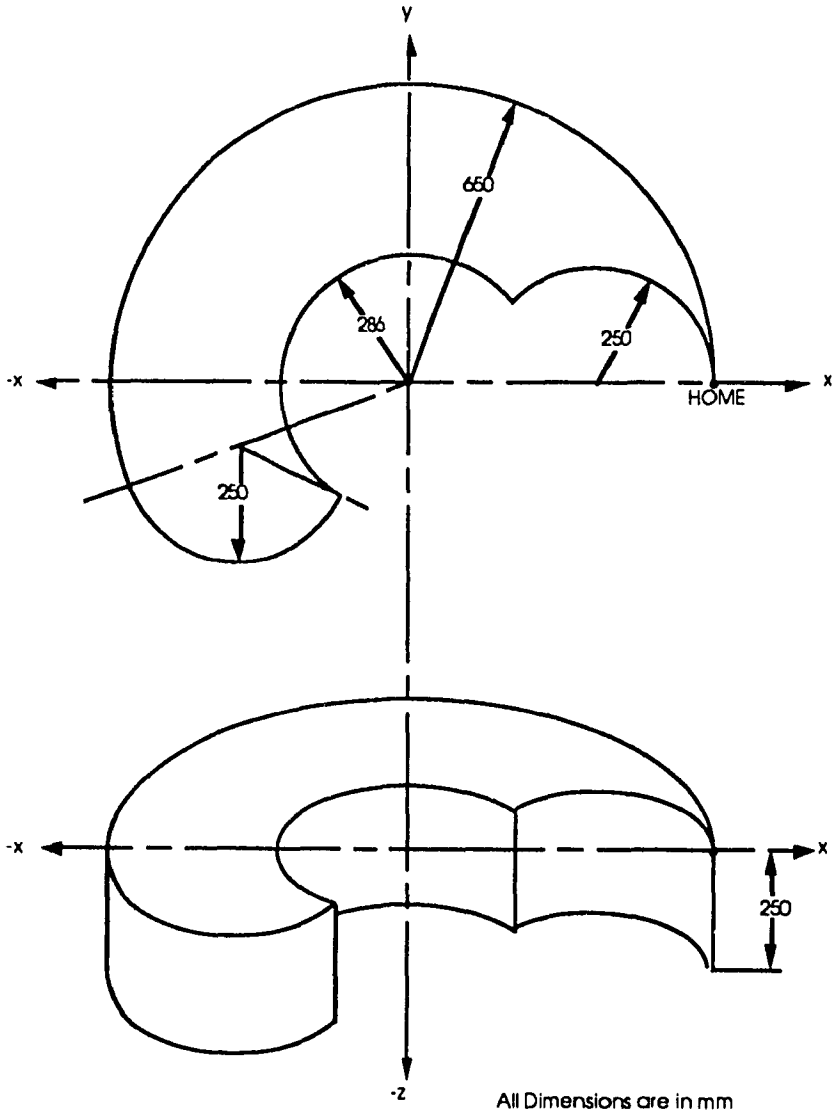


Figure 2.4. IBM 7545 Manipulator Work Envelope [7].

The manipulator has no positional redundancies, i.e., for any point in the work envelope, there exists a unique set of joint values to position the gripper at that point. Orientationally, there is redundancy in that the gripper can rotate a full 360 degrees, i.e., $\theta_{roll} = -180°$ is equivalent to $\theta_{roll} = +180°$.

As indicated in the block diagram in Figure 2.2, each joint of the 7545 manipulator is equipped with a DC servo motor and driver (the latter located in the control unit), a transmission device, an incremental encoder, and various sensors. These components are described in the sections that follow. For reference, the wiring diagrams of the connections between these components and the MTCB are included in Appendix A.

2.2.1 Motor Location and Transmission System

Each link of the 7545 manipulator is driven indirectly by a DC servo motor. For joints 1 and 2, the motors are mounted on their respective joint axes. The transmissions for these joints are harmonic drive assemblies with gear ratios of 157:1 and 80:1, respectively.

The joint Z motor is mounted at the distal end of link 2 with its axis parallel to the z-axis. A belt and a ball-screw mechanism transform the rotational motion of the motor into translational motion along the Z-axis shaft. The transmission ratio is 0.2381 revolutions per mm.

The Roll motor and its associated harmonic drive transmission is located in the base of the manipulator. A drive belt transfers motion to the joint axis. Due to this configuration, the orientation of the gripper is independent of arm position, i.e., joint variable θ_{roll} is independent of θ_1 and θ_2. An interesting consequence of this configuration is that the drive belt dynamically couples the distal end of the arm to the base

of the manipulator. The harmonic drive together with the drive belt provide a reduction ratio of 51.2:1.

2.2.2 DC Servo Motors and Drivers

The DC servo motors are permanent magnet type motors manufactured by Yaskawa Electric Mfg. Co., Ltd. The motors for joints 1 and 2 belong to the Print Motor series [8] while joint Z and Roll motors belong to the Minertia Motor series [9]. Their ratings and specifications are given in Appendix B.

Each motor is driven by a Yaskawa DC Servomotor Controller or Servopack [10]. Power for these units is obtained from the 7545 power supply via the relay board. Servopack power is controlled by the manipulator power key on the control panel which energizes a relay on the relay board. The Servopacks for the Print motors of joints 1 and 2 provide outputs of up to 200 W and for the Minertia motors of joints Z and Roll, up to 100 W.

Figure 2.5 shows a Servopack's internal block diagram. The output (between terminals A and B) comes from a full bridge transistor switching circuit, the driver of which is controlled by the Pulse Width Modulation (PWM) generator. The PWM generator is fed by the output of the Current Amplifier. This amplifier constitutes an armature current controller. The armature current setpoint is located at the C_{ref} testpoint and the control loop is closed by feedback from the Current Detecting Amplifier. C_{ref} is set by the output of the Speed Amplifier which constitutes an armature speed controller. The setpoint of this amplifier is located at the speed reference input terminals of the Servopack and is driven by a digital to analog converter on the MTCB. The feedback for the speed controller is provided by the MTCB which performs a frequency to voltage conversion on the encoder signal. This feedback signal is fed to the TG (tachogenerator) Feedback input of the Servopack.

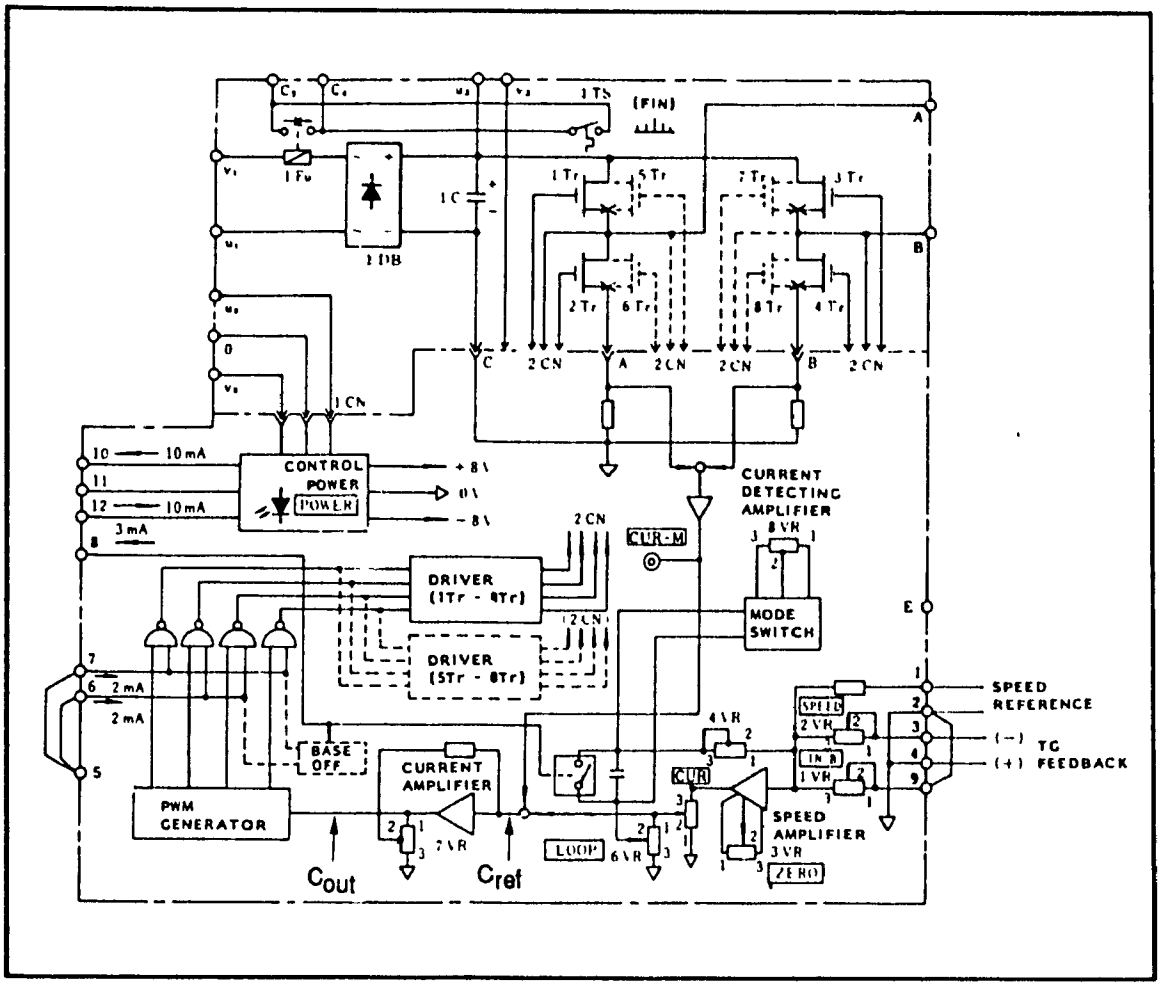


Figure 2.5. Servopack Internal Block Diagram [10].

The Servopack, in its manufacturer-supplied form, is incompatible with typical control strategies whose output driving signals are computed torque values. The Servopack does not support a torque input because of its speed control amplifier. The solution is to disable this speed controller. By doing this, the torque output signal from a robot controller such as the ARC would reach testpoint C_{ref} undistorted. At C_{ref}, the driving signal would be recognized as a desired armature current. (Note that armature current is directly proportional to motor torque.) The Speed Amplifier can be modified (by opening the speed control loop) to serve as an adjustable gain amplifier for calibration purposes. The following is a list of the required modifications.

(a) Transform the Speed Amplifier into a simple inverting amplifier by shorting-out its integrating capacitor (refer to Figure 2.5).

(b) Short the TG feedback input.

(c) Add an external potentiometer at the speed reference input for gain adjustment.

(d) Connect a wire from the internal testpoint C_{ref} to a point outside the Servopack for easy access during calibration. (The calibration procedure is listed in Appendix C.)

A wiring diagram of the modifications for one joint is shown in Figure 2.6. Note that the modifications are implemented using a SPST for (a) and SPDT selector switches for (b) and (c) so that the Servopack can be easily configured (by throwing the three switches) to serve either the MTCB or the ARC.

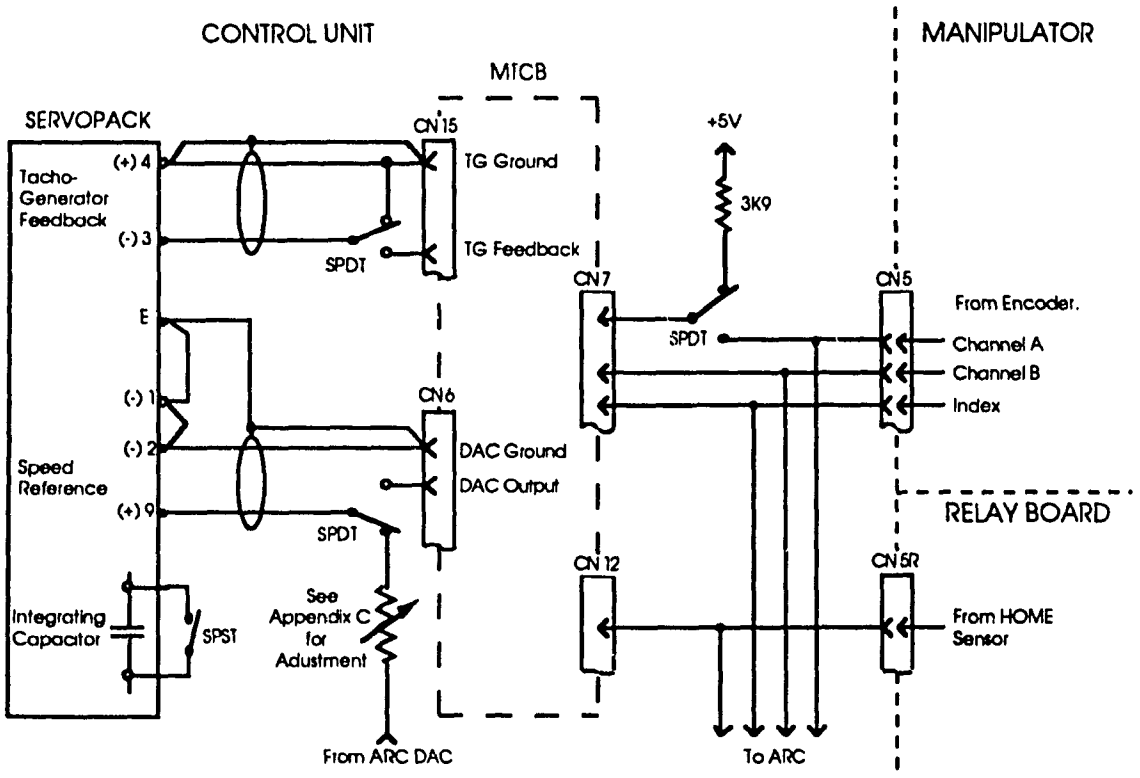


Figure 2.6. Modified Servopack, Encoder and HOME Sensor Connections.

2.2.3 Incremental Encoders

An incremental optical shaft encoder is mounted on the motor shaft of each joint to provide relative joint position information. Channel A and B outputs are in quadrature (90° out of phase) and have resolutions of 500 pulses per revolution for joints 1 and 2 and 400 pulses per revolution for joints Z and Roll. Each encoder also has a one pulse per revolution index output.

The MTCB contains a counter for each encoder. On-board logic decodes the incoming pulse streams to determine the count direction. The counters increment for positive and decrement for negative joint motion. The counters are clocked only by a single transition on a single encoder channel (e.g., the rising edge of channel A).

The MTCB sets the counters to zero when the manipulator is initialized to the HOME position. The counters then give joint positions relative to the HOME position. Joint position resolutions in pulses per degree are calculated by multiplying the encoder resolution by $\frac{1 \text{ rev.}}{360°}$ (not necessary for joint Z since it is translational) and the transmission ratios given in section 2.2.1. Table 2.1 lists the joint position resolutions of the four joints of the 7545 manipulator.

Table 2.1. 7545 Joint Position Resolutions.

Joint	Resolution
1	$218.0\bar{5}$ $\frac{\text{pulses}}{\text{degree}}$
2	$111.\bar{1}$ $\frac{\text{pulses}}{\text{degree}}$
Z	95.24 $\frac{\text{pulses}}{\text{mm}}$
Roll	$56.\bar{8}$ $\frac{\text{pulses}}{\text{degree}}$

Because robot control strategies generally require joint position feedback, the ARC must have access to the outputs of the encoders when it is controlling the manipulator. A potential access point is at a test point on the MTCB (labelled CN16 in [7]) where the two channels and the index of each encoder are conveniently available. Unfortunately, it was discovered that this is not a feasible point for extraction of these signals for the following reason. When manipulator power is switched on, the MTCB's control scheme tries to maintain the manipulator in position. However, while the manipulator is under ARC control, the MTCB's control efforts are futile since its output signals have been prevented (by the SPDT switches mentioned in the previous section) from reaching the Servopacks. The problem arises when the ARC moves a joint to a point where the MTCB's computed tracking error becomes excessive. At this point, the MTCB shuts down manipulator power as a safety feature.

A solution to this problem is to extract the encoder signals before they reach the MTCB and to prevent them from entering the MTCB. Actually, preventing only one channel (channel A for joints 1, 2, and Roll and channel B for joint Z) from entering the MTCB and pulling the MTCB input for that channel to +5 V is sufficient to *fool* the MTCB into thinking that there is no motion.

To accomplish this, a small circuit board was built to extract the encoder signals at MTCB connector CN7 where the signal levels are TTL compatible. SPDT switches were used for selection of either MTCB or ARC mode. Figure 2.6 illustrates the circuit modification for one joint. (Note that for joint Z, channel B is the channel which is to be interrupted.)

An added benefit to the above approach is that the encoder outputs are available at the extraction point whether or not manipulator power is on while at MTCB testpoint CN7 the signals are provided only when manipulator power is on. The benefit is that the ARC can

continually monitor joint position and that reinitialization of position counters is not necessary if manipulator power is switched off and the manipulator is moved manually.

2.2.4 Sensors

Each joint of the manipulator is equipped with overrun and HOME sensors (see Appendix A for wiring details).

The overrun sensors are limit switches that detect joint hyperextension. When an overrun sensor is tripped, manipulator power is turned off. This safety feature protects the links from hitting their mechanical limits and must remain intact. The ARC does not interfere with this safety feature.

The HOME sensors are proximity switches which are activated when a joint is in its HOME or reference position. Referring to Figure 2.3, the respective sensors indicate a HOME condition when link 1 becomes collinear with the x-axis, link 2 becomes collinear with the radial axis of link 1, the gripper point lies in the x-y plane, and the Roll angle is anywhere in its negative region. The HOME sensors are used in initialization of the joint position counters. The MTCB has a *find-HOME* routine that consists of moving a joint away from its HOME position if the sensor is already active and then moving it back towards the HOME sensor. When the HOME sensor becomes activated, the position counter for the joint is set to zero and the joint motion is discontinued.

The HOME sensor signals are extracted for use with the ARC from the ribbon cable connecting CN5R to CN12 as illustrated in Figure 2.6. HOME condition is indicated by 0V and not HOME condition by 22.5V.

2.3 SUMMARY

This chapter described the IBM 7545 Manufacturing System and identified the MTCB component as the source of some of its limitations. It was indicated that to overcome these limitations, the MTCB must be bypassed with a more powerful processing system such as the ARC which is presented in subsequent chapters. The chapter described important details of the 7545 system and outlined the modifications to be made to the system in order to bypass the MTCB.

CHAPTER 3

HARDWARE DESCRIPTION

FOR THE ADVANCED ROBOT CONTROLLER

3.1 INTRODUCTION

This chapter discusses the design philosophy of the ARC and describes the hardware that was developed for its implementation.

The ARC was conceived primarily as a replacement for the MTCB in the IBM 7545 Manufacturing System for the purpose of converting the 7545 into a testbed for robotic research. For this purpose the following ARC design objectives were adopted:

- The ability to execute both simple and sophisticated robot control algorithms at respectable servo rates (baseline of 1000 Hz for PD control).
- Basic functions to facilitate the development and implementation of robot control strategies.
- An environment conducive to robotic research.
- The cost of the overall system a fraction of the cost of the 7545 system.

To achieve these objectives, a hierarchical dual-processor architecture was chosen (see Figure 3.1) whereby the master processor is used for developing and executing control strategies and the slave serves as an intelligent interface between the master and the robot. Good performance from this type of architecture is achieved since the slave relieves a significant portion of the computational and I/O burden from the master. The slave handles some of the tedious yet essential functions associated with servo control. For example, the slave can be programmed to keep track of joint positions by continuously reading the joint encoder signals emanating from the robot and test for overrun conditions. It can also verify that the output motor torques as computed by the master are

within the limitations of the robot. All the functions handled by the slave are invisible to the user and thus facilitate the development of the master software.

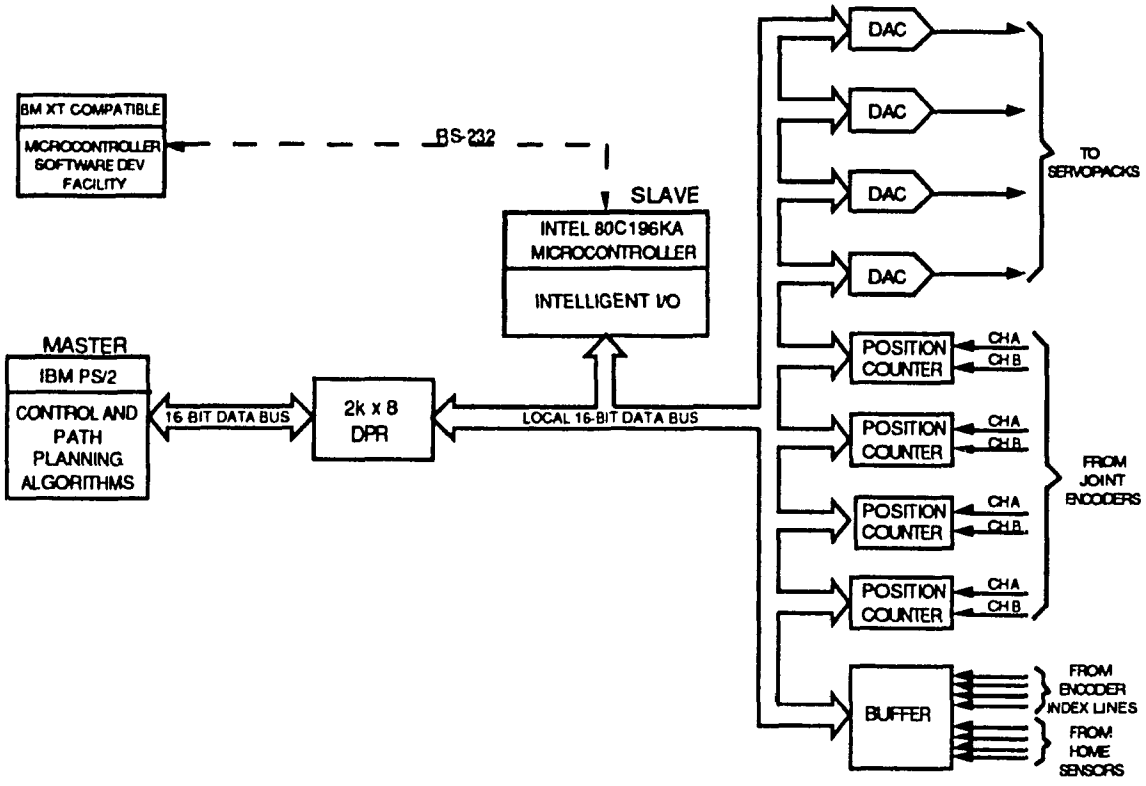


Figure 3.1. Block Diagram of the ARC Hardware.

The IBM Personal System /2 (PS/2) model 50 computer and the Intel 80C196KA microcontroller were chosen as the master and slave, respectively. The PS/2 model 50 is adequate for initial needs and can easily be substituted by the more powerful model 70 or model 80 for future needs. The Intel 80C196KA has many features which suit the ARC's needs. These will be discussed in Section 3.3.

The block diagram of Figure 3.1 also shows the remaining ARC hardware components: a dual-port RAM (DPR) for communication between the master and slave processors; four digital to analog converters (DACs) to drive the 7545's Servopacks; four

position counters to read the 7545's joint encoders; and an input buffer for reading the 7545's encoder index and HOME sensor lines.

The following sections of this chapter describe each of the components mentioned above and the manner in which they interconnect.

3.2 MASTER PROCESSOR AND DUAL-PORT RAM (DPR)

The master processor of the ARC is an 80286-based IBM Personal System /2 (PS/2) model 50 [11] which features the Micro Channel architecture and an 80287 math co-processor. As indicated in Figure 3.1 the master communicates with the slave via a dual-port RAM (DPR).

The dual-port RAM, as its name implies, is RAM that can be accessed by two processors, one at each port. The DPR was chosen over other types of data communication methods (e.g., FIFO) as it can store control program variables required by both the master and the slave processors and it can hold predefined semaphores for timing purposes thus eliminating the need for hardware interrupts. Variable that are passed between the two processors via the RAM include the robot state from slave to master and computed torques from master to slave.

For implementation, an Integrated Device Technology's IDT7132/IDT7142, master/slave pair [12] was chosen. In parallel, these two 2k x 8 dual-port static RAMs form a 2k x 16 memory that interfaces directly to the 16-bit word width of the PS/2. The PS/2-DPR interface circuit that was designed is shown in Figure 3.2. This circuit was constructed on a Micro Channel prototype adapter card which can be inserted into an expansion slot inside the PS/2. The critical timing parameters of the Micro Channel's memory cycles are shown in Appendix D while the timing waveforms of the DPR are given in Appendix E. A short description of the circuit is given next.

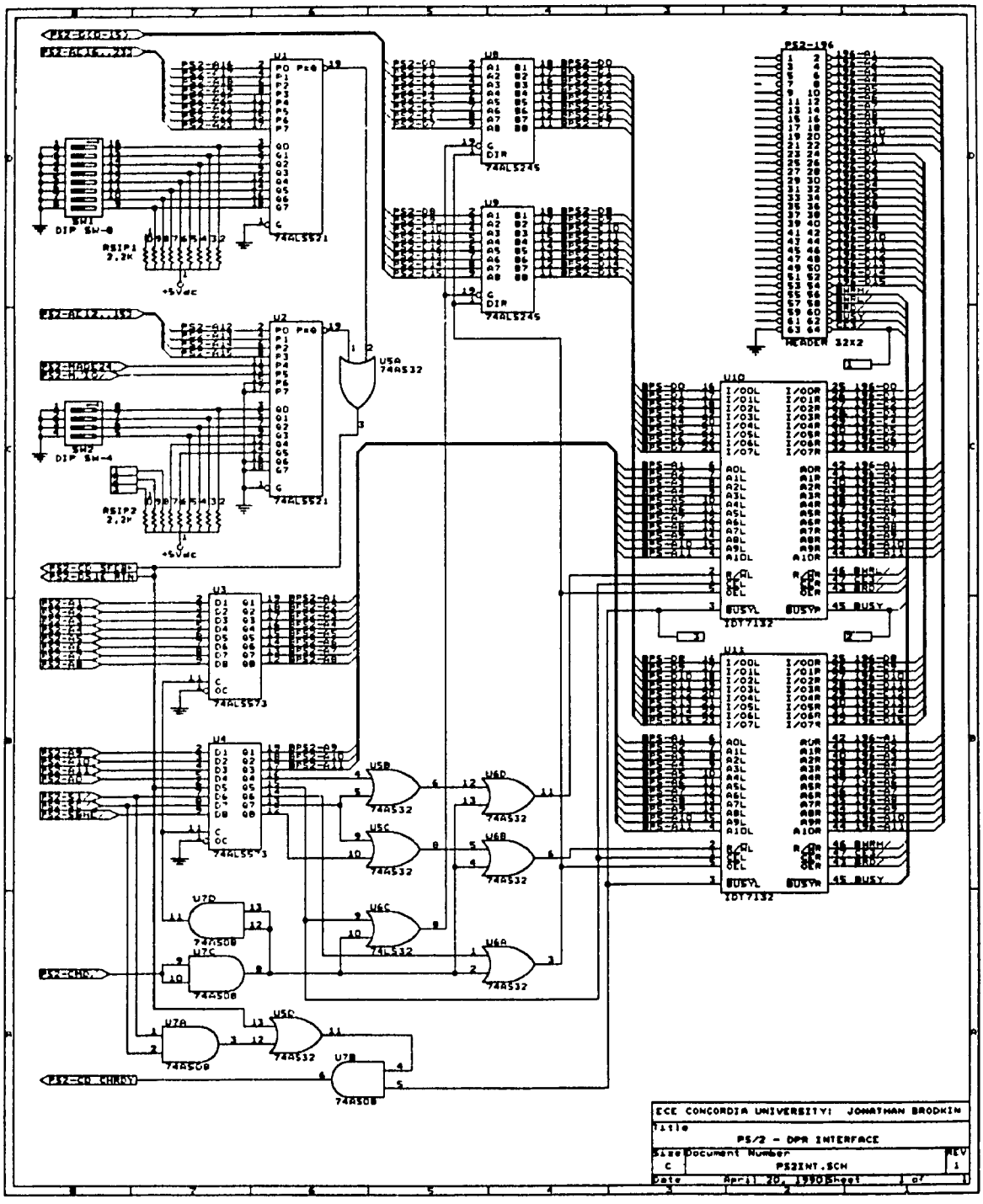


Figure 3.2. PS/2 - Dual-Port Ram Interface Schematic.

Referring to Figure 3.2, the twelve DIP switches enable initialization of the starting address of the DPR. ICs U1 and U2 decode the upper 12 address lines A(12-23) along with the M/$\overline{I/O}$ and MADE 24 signals to produce the unlatched address decode signal. This signal provides the required feedback to the master on lines $\overline{CD\ DS\ 16}$ and $\overline{CD\ SFDBK}$. The unlatched address decode signal, along with the lower 12 address lines A(0-11) and the control signals $\overline{S0}$, $\overline{S1}$, and $\overline{SBHE}$, are latched by U3 and U4 on the leading edge of $\overline{CMD}$. The latched address decode signal then enables the left port of the dual-port RAM and, together with $\overline{CMD}$, enables the data bus buffers, U8 and U9. $\overline{S0}$ and $\overline{S1}$ are the $\overline{write}$ and $\overline{read}$ enable signals, respectively.

The IDT7132 has on-chip port arbitration logic to resolve the situation in which both ports simultaneously address the same memory location. When this situation occurs, the IDT7132 determines which processor has access and holds the operation of the other processor through the use of a $\overline{BUSY}$ flag. For example, the IDT7132 sets $\overline{BUSYL}$ to indicate right port priority. This signal is then used to delay the PS/2 by setting its $\overline{CD\ CHRDY(n)}$ (channel ready) line.

Unfortunately, implementation of the above is complicated by the fact that the propagation delays of the IDT7132 and the other interface circuit components cannot guarantee generation of the $\overline{BUSYL}$ within the time required by the PS/2. Referring to Figure 3.2, the solution is to automatically set the $\overline{CD\ CHRDY(n)}$ signal at the beginning of the memory cycle and reset it on the leading edge of $\overline{CMD}$ using gates U7A, U5D and U7B. This affords the circuit extra time to set the $\overline{BUSYL}$ signal. In this configuration, the hardware extends the Micro Channel's default cycle to the 300 nanosecond synchronous cycle given in Appendix D. It should be noted that the PS/2 model 50 automatically extends the default cycle to the 300 nanosecond synchronous extended cycle. It is performed by the interface hardware, however, to assist portability to other systems.

The starting address of the left port of the DPR is set by the DIP switches to 0C0000H. An 18 inch twisted pair ribbon cable is connected to the header of the circuit board and links the right port of the DPR to the microcontroller which is described in the next section.

3.3 SLAVE PROCESSOR

The slave processor chosen for the ARC system is the Intel 80C196KA Microcontroller [13]. To simplify the design of the slave system the Intel EV80C196KA Microcontroller Evaluation Board [14] was employed. This evaluation board consists of the 80C196KA 16-bit embedded microcontroller, 16k x 16 static RAM, 16k x 16 EPROM and a UART for communications with a host IBM or compatible. Edge connectors on the board provide easy access to the microcontroller's system (data, address, control) bus. For reference, the EV80C196KA schematic diagram is given in Appendix F.

The EV80C196KA also facilitates software development as it contains a system debug monitor (SDM) for loading, executing and debugging code. The SDM is actually composed of two separate programs. One resides in the EPROM on the evaluation board and executes on the EV80C196KA and the other runs on the host computer, in our case an IBM compatible. The two programs communicate via an RS-232 channel.

3.3.1 The EV80C196KA Interface

As illustrated in the ARC block diagram of Figure 3.1, the microcontroller communicates with the DPR, the digital to analog converters, the position counters, and the robot overrun and HOME sensors. Each one of these components is viewed by the microcontroller as external memory residing somewhere in the upper half of the 80C196KA's 64k address space (8000H-FFFFH). Referring to Appendix F, the EV80C196KA's main interface is located at the JP2 memory expansion connector where the system bus is available. The timing diagram of the bus is given in Appendix G. The

schematic diagram of the first stage of the interface circuit which was designed to connect directly to JP2 is shown in Figure 3.3. A short explanation of the circuit follows.

U5 buffers all output control lines and is always enabled. U1 and U2 latch the 16-bit address from the multiplexed address/data bus (AD0-AD15) on the trailing edge of BALE. U3, U4 and U7 are bidirectional data bus buffers. Their direction is controlled by the $\overline{BRD}$ signal. D0 has its own buffer so that two position counters can be accessed simultaneously (this will be explained in more detail in the section 3.5). U8, a 3-8 decoder, generates the chip enable signals for the components of the ARC and is enabled for addresses in the range 8000H-FFFFH (A15 high). The outputs $\overline{CE1}$, $\overline{CE2}$, $\overline{CE3}$, and $\overline{CE4}$ enable the position counters, the digital to analog converters, the HOME and overrun sensors buffer, and the DPR, respectively. Since the chip enable signals are generated by a latched address decode, they are active for the entire memory cycle of the 80C196KA. In light of this, data transmission timing is controlled by the pulse width of the 80C196KA's read ($\overline{RD}$) and write ($\overline{WRH}$,$\overline{WRL}$) lines. For ARC devices that require longer pulse widths than these lines provide, an antedated read/write signal AR/W is generated. The AR/W pulse begins earlier, on the trailing edge of ALE, and ends on the trailing edge of either $\overline{RD}$ or $\overline{WRL}$.

U21B, the D flip-flop, is set and reset by $\overline{RD}$ and $\overline{WRH}$, respectively. The output of the flip-flop, LR/W (latched read/write), drives the R/$\overline{W}$ line of the position counters. This will be explained in greater detail in section 3.5. An additional requirement of the position counters is a 2 MHz clock. This clock is generated by dividing the 80C196KA's 6 MHz CLOCKOUT signal by three using the dual J-K flip-flops of U10. The clock signal, as shown in Figure 3.3, is termed DBCLK.

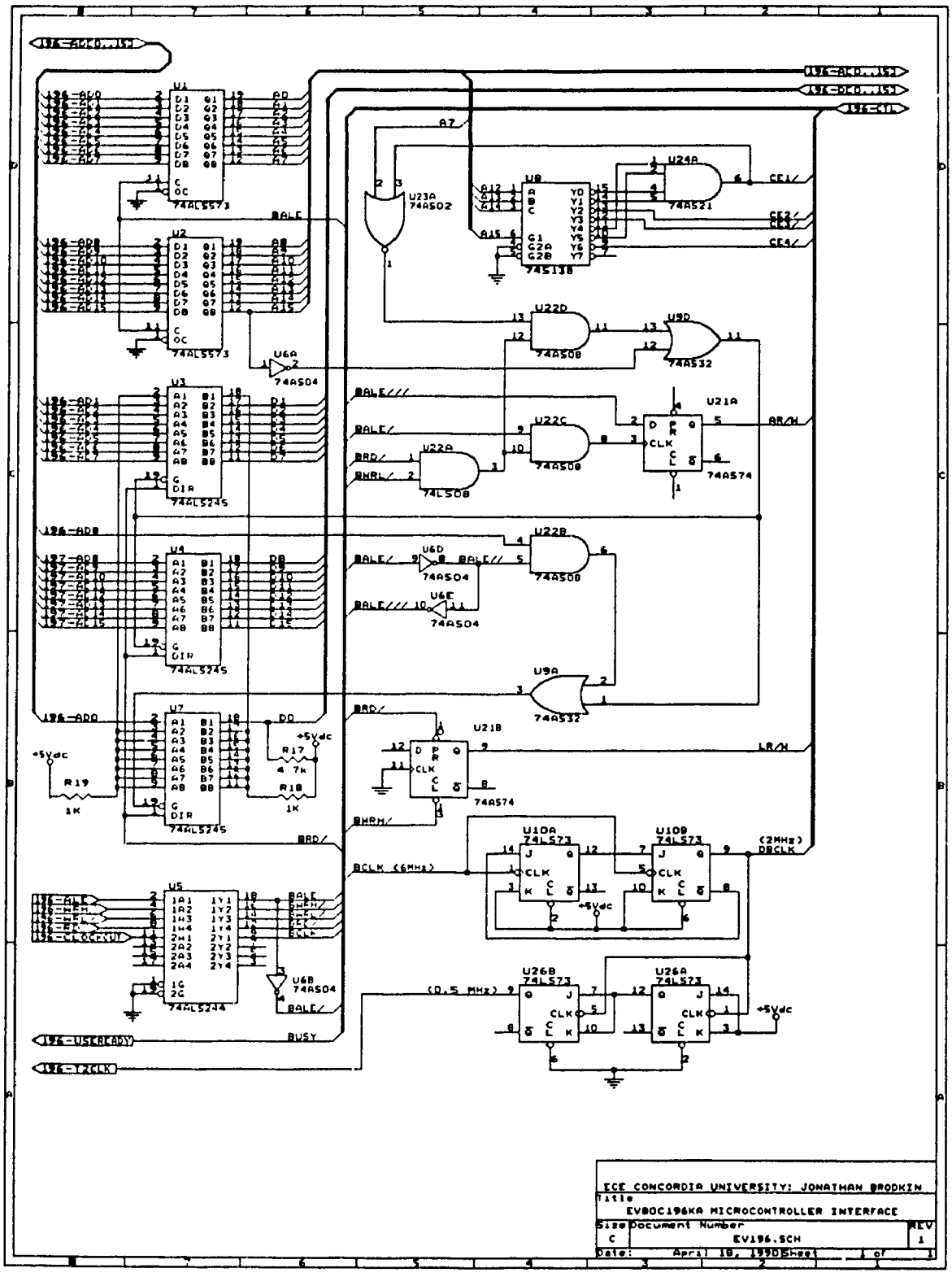


Figure 3.3. 80C196KA Interface Schematic.

TIMER2 is a 16-bit counter on the 80C196KA which is used by the ARC as a sample period timer. TIMER2 increments on any transition at the T2CLK terminal. The desired TIMER2 counting frequency is 1 MHz and the required 500 KHz clock for the T2CLK input is generated by dividing the 2 MHz clock by four using the dual J-K flip-flops of U26.

3.4 DIGITAL TO ANALOG CONVERSION

Digital to analog converters (DACs) are the means by which the computed motor torque values are converted into analog voltage signals to drive the Servopacks. In its present configuration, the ARC employs four Analog Devices AD667 12-bit double-buffered DACs [15], one for each joint of the 7545 manipulator. The double-buffered digital inputs enable all DACs to be simultaneously updated without the need for external latches and the 12-bits provides adequate resolution. The AD667's digital and analog circuits are discussed below.

3.4.1 Digital Circuit Details

The AD667 functional block diagram and write cycle timing diagrams are given in Appendix H. The AD667 bus interface logic consists of four independently addressable registers in two ranks. The first rank consists of three four-bit registers controlled by address inputs A0-A2. The second rank register holds all 12-bits and is controlled by address input A3.

The AD667-80C196KA interface is shown in Figure 3.4. The first rank register address lines (A0-A2) on each of the four DACs are tied together and connected to 80C196KA address lines A1-A4, respectively. The second rank register address line (A3) on all DACs are tied together and connected to 80C196KA address line A5. In this configuration, the first rank registers of each DAC can be loaded separately with their respective torque data as per Write Cycle #1. The transfer from first to second rank is then performed

simultaneously on all DACs as per Write Cycle #2. This double buffered organization eliminates spurious analog outputs originating from data bus activity while the DACs' chip select inputs ($\overline{cs}$) are low.

The $\overline{cs}$ inputs are driven by $\overline{CE2}$ whose base address is A000H. The $\overline{CE2}$ signal is gated by the $\overline{AR/W}$ signal to provide the longer pulse width required by the DACs. The DAC addresses are listed in Table 3.1.

Table 3.1. Digital to Analog Converter Addresses.

DAC #	RANK REGISTER	80C196KA ADDRESS
1	1	AFFCH
2	1	AFFAH
3	1	AFF6H
4	1	AFEEH
ALL	2	AFDEH

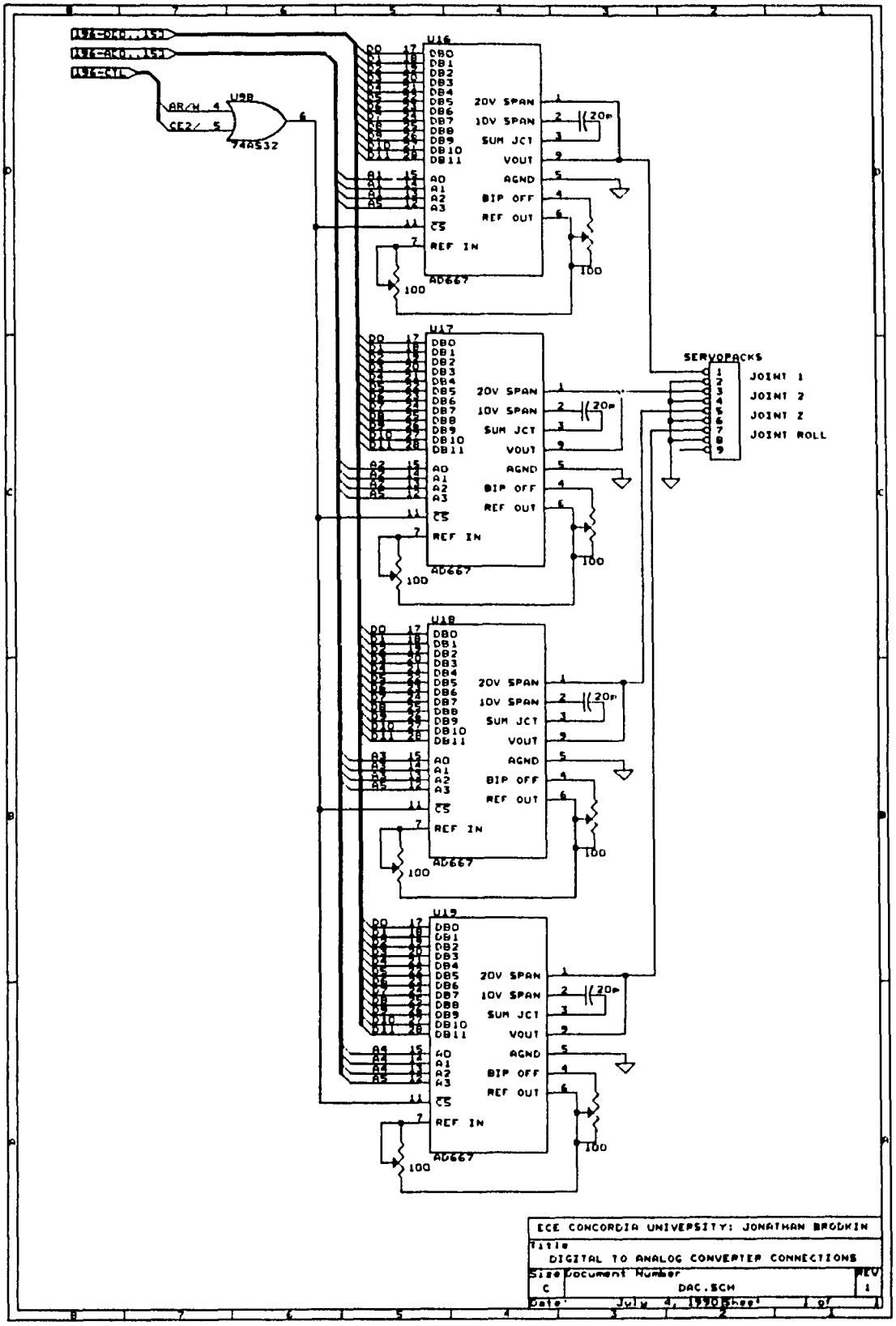


Figure 3.4. Digital to Analog Converter Schematic.

3.4.2 Analog Circuit Details

The DAC's internal output amplifier supplies up to 40 mA of current to drive the 7545's Servopack. Each AD667 is configured to produce bipolar output voltage ranges of ±10V. Two 100Ω potentiometers are used in calibrating the outputs. The calibration process is described in Appendix C. A 20pF capacitor is connected across the feedback resistor to optimize dynamic performance.

3.5 POSITION COUNTERS

As mentioned in the first chapter, the joints of the 7545 manipulator are equipped with position feedback devices in the form of optical incremental encoders (see Section 2.2.3). To recover the encoded position information, the ARC employs four Hewlett Packard HCTL-1000 General Purpose Motion Control ICs [16]. These ICs were chosen as they contain quadrature decoders and 24-bit counters for reading the 7545's joint encoders as well as 3-bit state delay digital filters to remove noise spikes on the encoder lines. Each HCTL-1000 takes the place of a multitude of ICs used in [1] and [2] for the same purpose. Another feature of the HCTL-1000 is that a count is produced for both transitions (low→high and high→low) on Channel A and Channel B. Therefore, the 500-count encoders on the first two 7545 manipulator joints are decoded into 2000 quadrature counts per revolution. As a result, the HCTL-1000s give four times better resolution than the counters on the MTCB as well as the counter described in [1] and [2] which count on one transition of one encoder channel only. In addition, the 24-bit capacity of the HCTL-1000s is sufficient to hold the pulse counts corresponding to the maximum range of motion of the joints. Thus no special software is required to keep track of the total pulse count. The schematic diagram of the HCTL-1000 circuit is shown in Figure 3.5. A description of the circuit is given below.

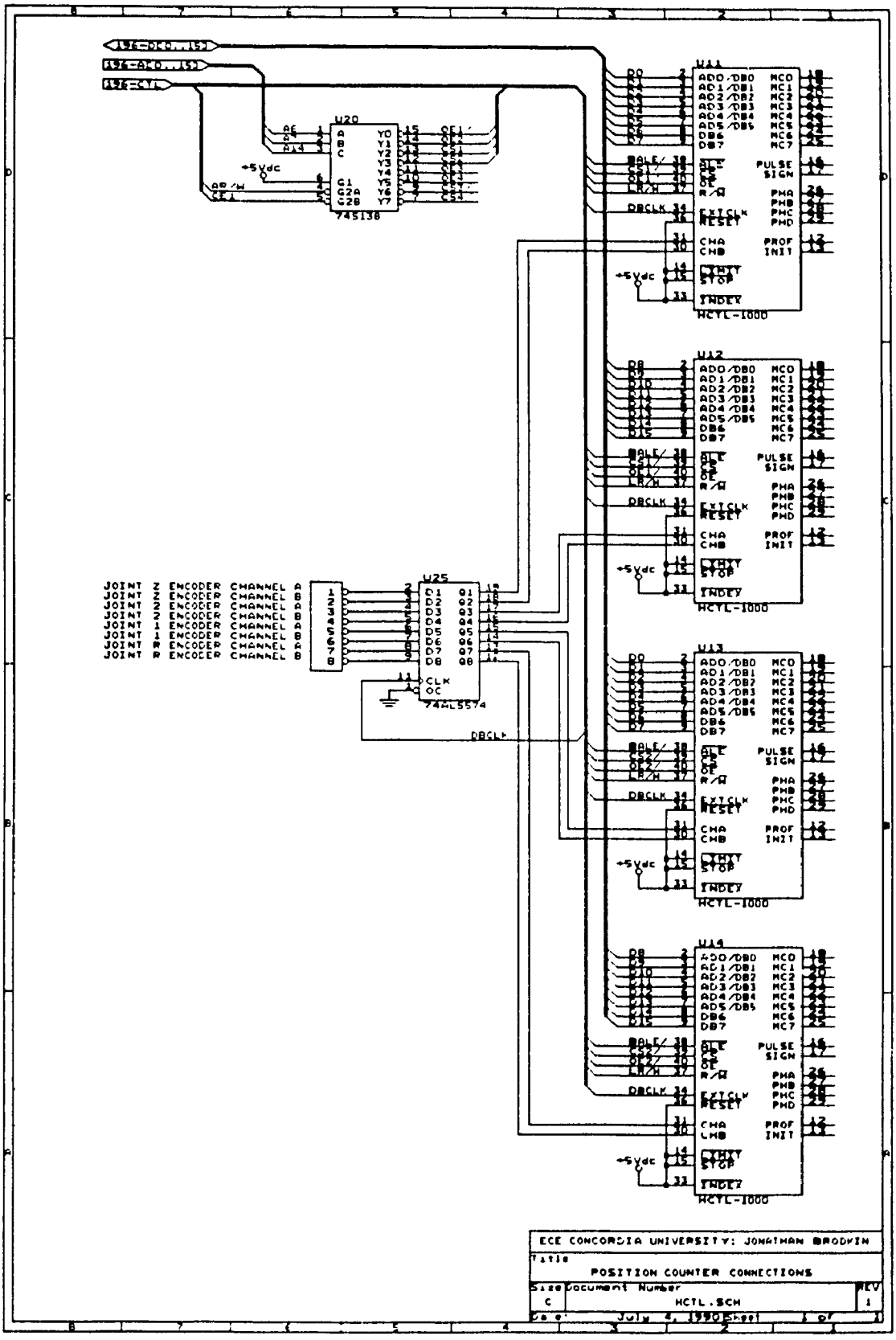


Figure 3.5. HCTL-1000 Schematic.

The HCTL-1000s are clocked by the 2 MHz DBCLK line as are the D flip-flops in U25. The flip-flops ensure that all encoder pulse transitions occur at the HCTL-1000 encoder inputs coincident with the clock, thus preventing any possibility of erroneous encoder counts as explained in [16].

The 24-bit pulse count is located in three of the HCTL-1000s 64 8-bit internal registers. The registers are accessed over the HCTL-1000s multiplexed 6-bit address/8-bit data bus. The bus timing diagram for the HCTL-1000 is included in Appendix I and a description of the I/O operation follows.

On the leading edge of $\overline{\text{BALE}}$ the HCTL-1000 begins sampling the bus into an internal address latch. This bus information, which represents the 6-bit address of one of the HCTL-1000's 64 8-bit registers, gets latched on the trailing edge of $\overline{\text{BALE}}$. Next, on the leading edge of $\overline{\text{CS}}$, the HCTL-1000 begins sampling the bus into an internal data latch. On the trailing edge of $\overline{\text{CS}}$ the HCTL-1000 checks the LR/W line and performs one of two operations. In the case of a write operation, the data in the data latch is written into the addressed location. In the case of a read operation, the data in the addressed location is sent to an internal output latch which can then be enabled onto the bus by setting $\overline{\text{OE}}$. Because the HCTL-1000 takes a relatively long time (minimum 1.8 microseconds) to transfer the data to the output latch, the read operation is performed by two microcontroller read instructions. The first read instruction sets the LR/W line high, selects the desired register and begins the transfer process by asserting the appropriate $\overline{\text{CS}}$ line. The second read instruction asserts the $\overline{\text{OE}}$ line to enable the data onto the bus to be read by the microcontroller.

The $\overline{\text{CS}}$ and $\overline{\text{OE}}$ signals are generated by U20, a 3-8 decoder, which decodes address lines A6, A14, and A7. U20 is enabled by the $\overline{\text{CE1}}$ and by the AR/W which provides the longer $\overline{\text{CS}}$ and $\overline{\text{OE}}$ pulse widths required by the HCTL-1000s.

To increase the ARC performance, the 8-bit HCTL-1000s are connected in pairs to the 16-bit bus of the EV80C196KA microcontroller. The upper and lower HCTL-1000s that make up a pair are connected to bus lines D8-D15 and D0-D7, respectively. The first pair is controlled by $\overline{CE1}$ and $\overline{OE1}$ and the second pair is controlled by $\overline{CE2}$ and $\overline{OE2}$. Two other sets of control lines are generated by U20, $\overline{CE3}$, $\overline{OE3}$ and $\overline{CE4}$ $\overline{OE4}$, for future expansion. The memory map for these control signals is given in Table 3.2.

Table 3.2. Memory Map for the HCTL-1000s.

CNTRL	ADDRESS															
SIGNAL	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
$\overline{OE1}$	1	0	0	d	d	d	d	d	0	0	d	d	d	d	d	0
$\overline{OE2}$	1	0	0	d	d	d	d	d	0	1	d	d	d	d	d	0
$\overline{CS1}$	1	0	0	x	x	x	x	x	1	0	x	x	x	x	x	0
$\overline{CS2}$	1	0	0	x	x	x	x	x	1	1	x	x	x	x	x	0
$\overline{OE3}$	1	1	0	d	d	d	d	d	0	0	d	d	d	d	d	0
$\overline{OE4}$	1	1	0	d	d	d	d	d	0	1	d	d	d	d	d	0
$\overline{CS3}$	1	1	0	x	x	x	x	x	1	0	x	x	x	x	x	0
$\overline{CS4}$	1	1	0	x	x	x	x	x	1	1	x	x	x	x	x	0

d-don't care

x-HCTL-1000 register address

Since address line A13 is always 0 only the first 32 registers on the HCTL-1000s can be accessed (i.e. registers R00H - R1FH). This does not pose a problem as the 24-bit actual position count is held in the three 8-bit registers R12H (MSB), R13H, and R14H (LSB) and the only other register of interest is R05H. Writing 01H to R05H commands the HCTL-1000 to enter the Initialization/Idle mode where it simply keeps track of joint position. Upon writing 00H to R05H, the position counter registers (R12H, R13H, R14H) are cleared and the Initialization/Idle mode is entered.

As seen in Table 3.2, address line A0 is always 0. This is due to the fact that the microcontroller requires an even operand in all instructions which refer to word (16-bit) memory locations. This would dictate that only even registers on the lower HCTL-1000 can be addressed. This limitation, however, is overcome by providing bus line D0 with its

own buffer (U7) as shown in the schematic diagram in Figure 3.3. When the buffer is enabled, D0 takes the level of AD0 and when the buffer is disabled, D0 is pulled high by the pull-up resistor R17. It is presumed that the microcontroller addresses the same location in both the upper and lower HCTL-1000s. Therefore, the D0 input of the lower HCTL-1000 should always be the same as D8 on the upper HCTL-1000 during addressing. Hence, during the addressing portion on the multiplexed bus (i.e., if AD8 is high during $\overline{\text{BALE}}$ the buffer will be disabled by U22B and D0 will be high even though AD0 is low. If AD8 is low, the buffer will be enabled and D0 will take on the value of AD0.

As was mentioned previously, the $\overline{\text{OE}}$ line which enables the output of the HCTL-1000s onto the data bus begins with the antedated AR/W line on the rising edge of $\overline{\text{BALE}}$. Bus contention could occur at the beginning of this period when the microcontroller is asserting an address on AD[0..15]. The contention would last until the $\overline{\text{BRD}}$ line goes low and the direction of the data buffers changes for a read instruction. The contention is avoided by means of U23A, U22D, U6A, U9A, and U9D, which act to disable the data buffers if the HCTL-1000s are being accessed ($\overline{\text{CE1}}$ is low), if it is a read instruction (A7 and $\overline{\text{OE}}$ are low), and if the $\overline{\text{BRD}}$ line is high. This buffer disabling period begins on the address decode and ends with the start of the read pulse.

The final timing requirement of the HCTL-1000 is that its $\overline{\text{R/W}}$ input remain asserted for a short period after the $\overline{\text{CS}}$ pulse. This is accomplished by latching the $\overline{\text{BRD}}$ and $\overline{\text{BWRH}}$ lines in the D flip-flop, U21B. The output of the flip-flop, labelled LR/W (latched read/write), drives the $\overline{\text{R/W}}$ inputs of the HCTL-1000s.

3.6 HOME SENSORS, ENCODER INDEX AND DUAL-PORT RAM CONNECTION

As described in section 2.2.4 the extracted HOME sensor signals have a voltage of approximately 22.5 V when the joint is not HOME and 0 V when the joint is HOME. These

voltages are converted to TTL levels by the circuit in Figure 3.6 which consists of a voltage divider, a Darlington transistor and pull-up resistor, and for buffering an opto-isolator. The HOME signals along with the index pulses generated by the optical encoders (TTL compatible) are made available to the microcontroller through buffer U15. The buffer is enabled by $\overline{CE2}$ and $\overline{BRD}$ and its address is B000H.

Also included in Figure 3.6 is the edge connector for the dual-port RAM. Because the dual-port RAM has a very fast access time it can be connected directly to the basic interface circuit in Figure 3.3. The dual-port RAM is chip-enabled by $\overline{CE4}$ and has a base address of E000H.

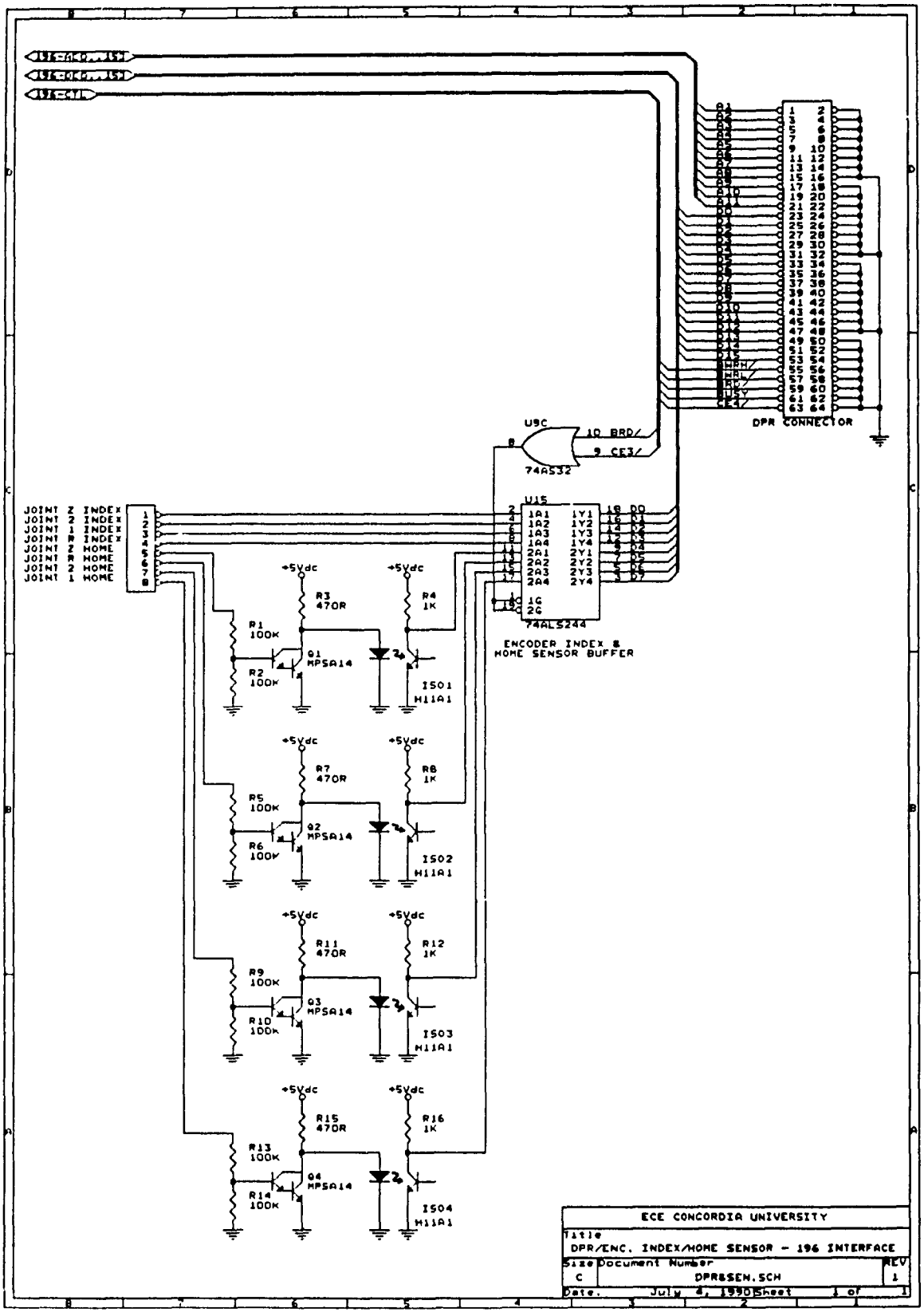


Figure 3.6. Index and HOME Sensor Schematic.

3.7 HARDWARE IMPLEMENTATION

The circuits of sections 3.3 - 3.6 were wire-wrapped on a prototype board. The EV80C196KA microcontroller evaluation board was attached along side the prototype board and was electrically connected via a ribbon cable. The assembly was mounted on a Plexiglass base for attachment inside the 7545's control unit housing. As mentioned previously, connection to the dual-port RAM (residing in the PS/2) was via an 18 inch, 64 conductor, twisted-pair ribbon cable.

3.8 SUMMARY AND FUTURE WORK

This chapter described the hardware that was designed specifically for the ARC. The dual-processing nature of the ARC provides the processing power needed for advanced control strategies and also enables some of the basic functions executing on the slave to be invisible to the user. The system also serves as a basis for future hardware expansion. For example, supplemental hardware such as force sensor feedback circuits (required for impedance control) and circuitry for additional degrees of freedom can be easily interfaced to the slave system bus. As for the master processor, greater processing power can be easily achieved in two ways. The first is to replace the PS /2 model 50 by the model 70 or the model 80. The second is to add one or more transputer (processor) boards inside the PS/2 .

CHAPTER 4

SOFTWARE DESCRIPTION

FOR THE ADVANCED ROBOT CONTROLLER

4.1 INTRODUCTION

As mentioned in Chapter 1, the ARC was conceived to enhance the operation of the IBM 7545 Manufacturing System and, in the process, transform it into a testbed for robotic research. The previous chapter described the hardware that was designed to achieve this objective. This chapter presents the master and slave software that has been developed for the purpose of creating the research environment.

Figure 4.1 shows a block diagram illustrating the role of each processor in the overall system. Note that the user interfaces only with the master processor; the slave is completely invisible. The slave software performs the function of I/O handler for the master and some basic safety features such as limit checking of variables. The slave also contributes by maintaining the sample period timer. Perhaps the most valuable function of the slave, however, is the reading of the joint position counters (HCTL-1000s). This function, as detailed in Section 3.5, is both complicated and time consuming. The slave, therefore, performs some of the tedious yet essential tasks associated with robotic control and enables the master (and user) to concentrate on higher level issues. The slave software is described in Section 4.3.

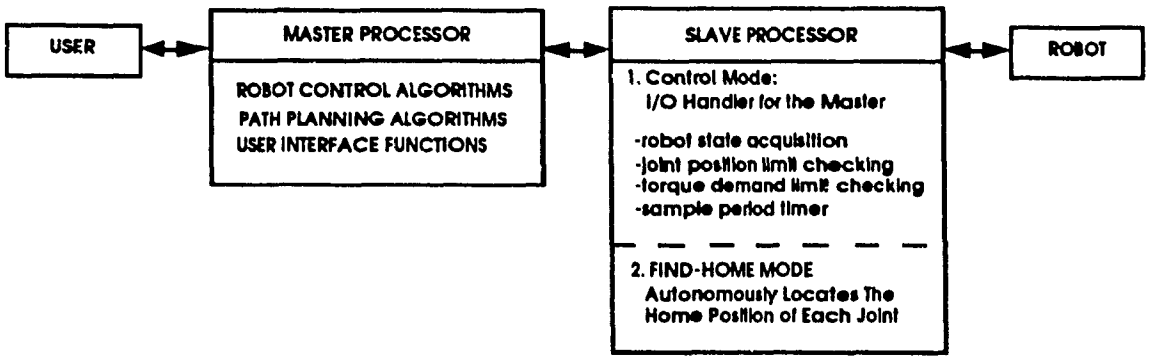


Figure 4.1. Block Diagram of the ARC Software.

The master processor is used for developing, storing and executing path planning and robot control algorithms. So far, a library consisting of two path planning algorithms and two control algorithms has been coded in the C language. Each of the four functions in the library resides in its own separate file for developmental purposes. Once debugged, a function is linked to a main program for execution. The main program, also residing in a separate file, is composed of menu, display, and user-interaction functions. These functions help to create a user-friendly environment conducive to robotic research. Details of the master processor software are included in Section 4.4.

The following section describes how the dual-port RAM is set up to handle the flow of data between the master and slave processors.

4.2 DUAL-PORT RAM UTILIZATION

The DPR of the ARC constitutes the ARC's shared memory system and is used for the passing parameters and variables between the master and slave processors. Each parameter and variable is assigned a specific location (register) in the DPR. The address and name of each register in the DPR is listed in Table 4.1.

Table 4.1. Dual-Port RAM Register Reference Table.

ADDRESS		SIZE	NAME	USER ACCESS	
Master	Slave			Master	Slave
C0000H	E000H	word	*sample_period*	w	r
C0002H	E002H	word	*timing_A*	r/w	r/w
C0004H	E004H	word	*timing_B*	r/w	r/w
C0006H	E006H	word	*command*	r/w	r/w
C0008H	E008H	word	*error*	r/w	r/w
C0010H	E010H	double word	*actual_position_joint_1*	r	w
C0014H	E014H	double word	*actual_position_joint_2*	r	w
C0018H	E018H	double word	*actual_position_joint_Z*	r	w
C001CH	E01CH	double word	*actual_position_joint_R*	r	w
C0020H	E020H	word	*torque_joint_1*	w	r
C0022H	E022H	word	*torque_joint_2*	w	r
C0024H	E024H	word	*torque_joint_Z*	w	r
C0026H	E026H	word	*torque_joint_R*	w	r

The function of each of the registers listed in the table is described below.

4.2.1 *Sample_Period* Register

The *sample_period* register is loaded by the master with a value corresponding to the controller sampling period. As mentioned in Section 3.3.1, sample period timing is handled by TIMER2 on the 80C196KA which counts up at a frequency of 1 MHz. Therefore, multiplying the value in the *sample_period* register by 10^{-6} yields the actual sampling period in seconds. The slave maintains the sample period timer by resetting TIMER2 when it reaches or exceeds the value in the *sample_period* register. The reset is accomplished by subtracting the value in the *sample_period* register from TIMER2.

4.2.2 *Timing_A* Register

The *timing_A* register contains two semaphores which are set by the slave to initiate the master's sampling period and to indicate to the master that joint positions have been read and stored in the DPR. The semaphore indications are specified in the table below.

Bit Number	Indication
0	Begin new servo loop
1	Joint positions available

The master resets both semaphores after completing the computations for the sampling period and then waits for bit 0 to become set indicating the start of the next period.

4.2.3 *Timing_B* Register

The *timing_B* register contains one semaphore (bit 0) which is set by the master to indicate to the slave that the joint motor torques have been computed and stored in the DPR. The slave resets the semaphore after reading the torques.

4.2.4 *Command* Register

As will become clear in Section 4.3, the slave processor can execute in one of two modes of operation: the *find-HOME* mode, and the *control* mode. The master indicates the desired mode by setting a semaphore in the *command* register. The slave reads the *command* register while in a waiting loop and performs one of the actions specified below.

Bit Number Set	Indication
0	Enter the Control Mode
7	Enter the Find Home Mode
None	Continue Waiting

4.2.5 *Error* Register

The slave processor informs the master of any error conditions that have been detected by setting flags in the *error* register. The flag indications are specified below. The errors will be described in their proper context in Section 4.3.

Bit Number	Indication
0	Joint 1 in positive overrun
1	Joint 1 in negative overrun
2	Joint 2 in positive overrun
3	Joint 2 in negative overrun
4	Joint Z in positive overrun
5	Joint Z in negative overrun
6	Joint Roll in positive overrun
7	Joint Roll in negative overrun
8	Excessive torque for joint 1
9	Excessive torque for joint 2
10	Excessive torque for joint Z
11	Excessive torque for joint Roll
12	Motor torques arriving too late
13	Master processor too slow

4.2.6 *Actual_Position* Registers

The slave processor reads the HCTL-1000 position counters and stores the count values in the long integer or double word (32 bit) *actual_position* registers.

4.2.7 *Torque* Registers

The master computes the joint motor torques and stores them in the *torque* registers.

4.3 SLAVE PROCESSOR SOFTWARE

The tasks of the slave are shown in Figure 4.1. These tasks are fundamental to virtually all robot control strategies and therefore do not require modification. By assigning these tasks to the slave, they in effect become invisible to the user. As shown in Figure 4.1, the slave software is designed to execute in one of two modes of operation: the *find-HOME* mode and the *control* mode. The first mode is used for locating the manipulator's HOME (reference) position, the second mode contains the tasks which assist the master's control program. An operating mode is selected while the processor is looping in a main program. The main program, the two operating modes and the common procedures are described in the following sections. Flow charts are included in the text and source code listings can be found in Appendix J. The slave processor software is written in 80C196KA assembly language and resides in a single module.

4.3.1 Main Program

The flow chart for the slave's main program is shown in Figure 4.2. The initialization sequence consists of program variable and pointer initialization, resetting of the HCTL-1000 position counters and putting the 80C196KA's TIMER2 into fast increment mode to enable it to count at the desired 1 MHz frequency. The slave then clears the *command* register in the DPR and enters the waiting loop.

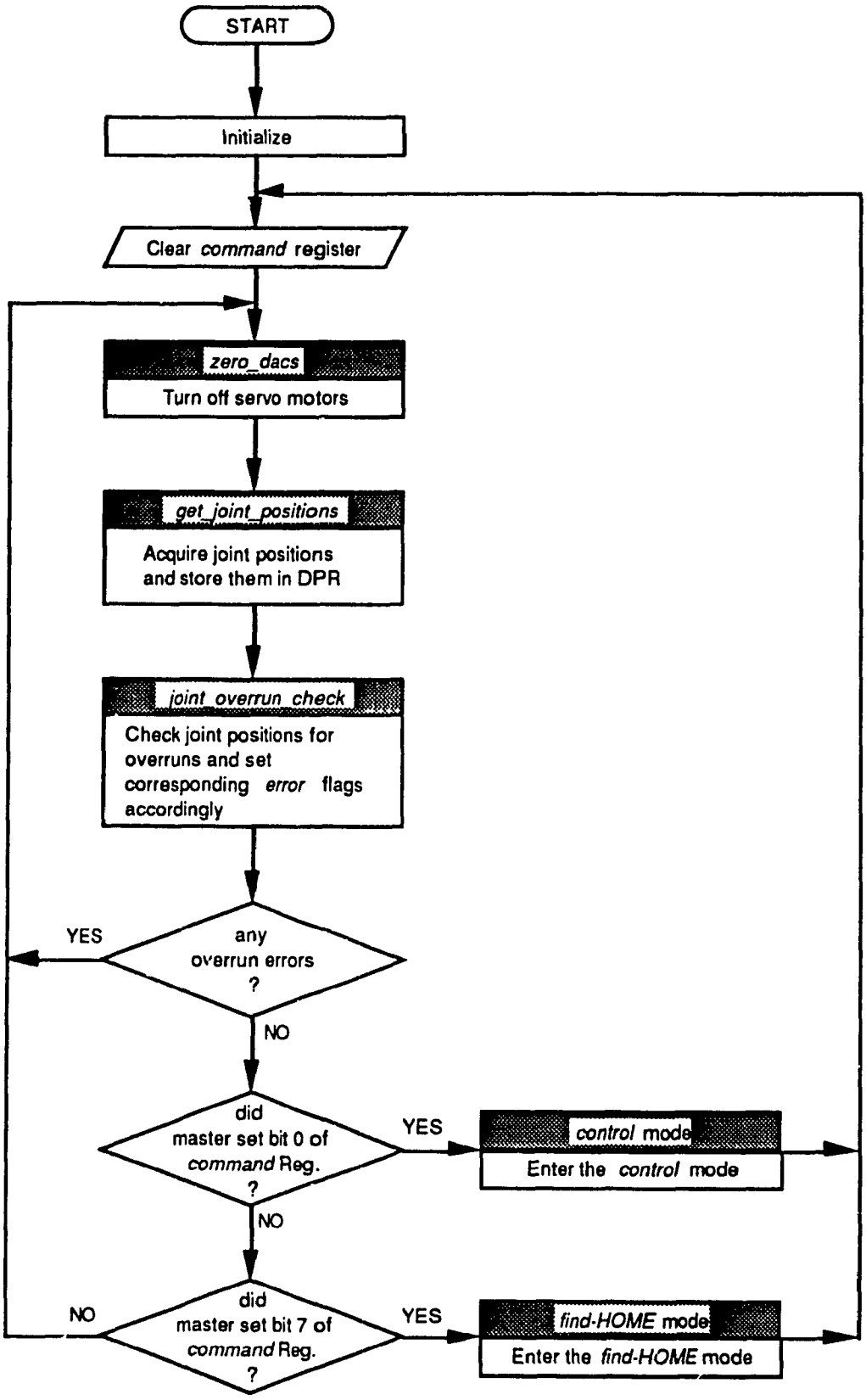


Figure 4.2. Flow Chart for the Slave's Main Program.

While in the waiting loop, the slave calls procedures to zero the DACs so that no torques are fed to the Servopacks, to acquire the joint positions, and to check joint positions for overruns. If no joint overrun conditions exist, the *command* register is polled. Depending on the value in this register (see Section 4.2.4), the slave either enters the *control* mode, enters the *find-HOME* mode, or restarts the waiting loop. If, however, a joint overrun is detected, *command* is not polled and the waiting loop is restarted.

4.3.2 *Find-HOME* Mode

The *find-HOME* mode is entered if, while in the waiting loop, the slave detects a value of 0080H (bit 7 set) in the *command* register. In the *find-HOME* mode, the slave autonomously moves the 7545's joints in order to locate the reference position of each joint. As mentioned in Section 2.2.4, the HOME sensor signals are generated by proximity switches mounted on the 7545 manipulator. Because of the imprecise nature of proximity switches, the ARC recognizes a joint's HOME position as being located at the first encoder index pulse following activation of the proximity switch. The sequence of *find-HOME* is depicted in the flow chart of Figure 4.3 and is described below.

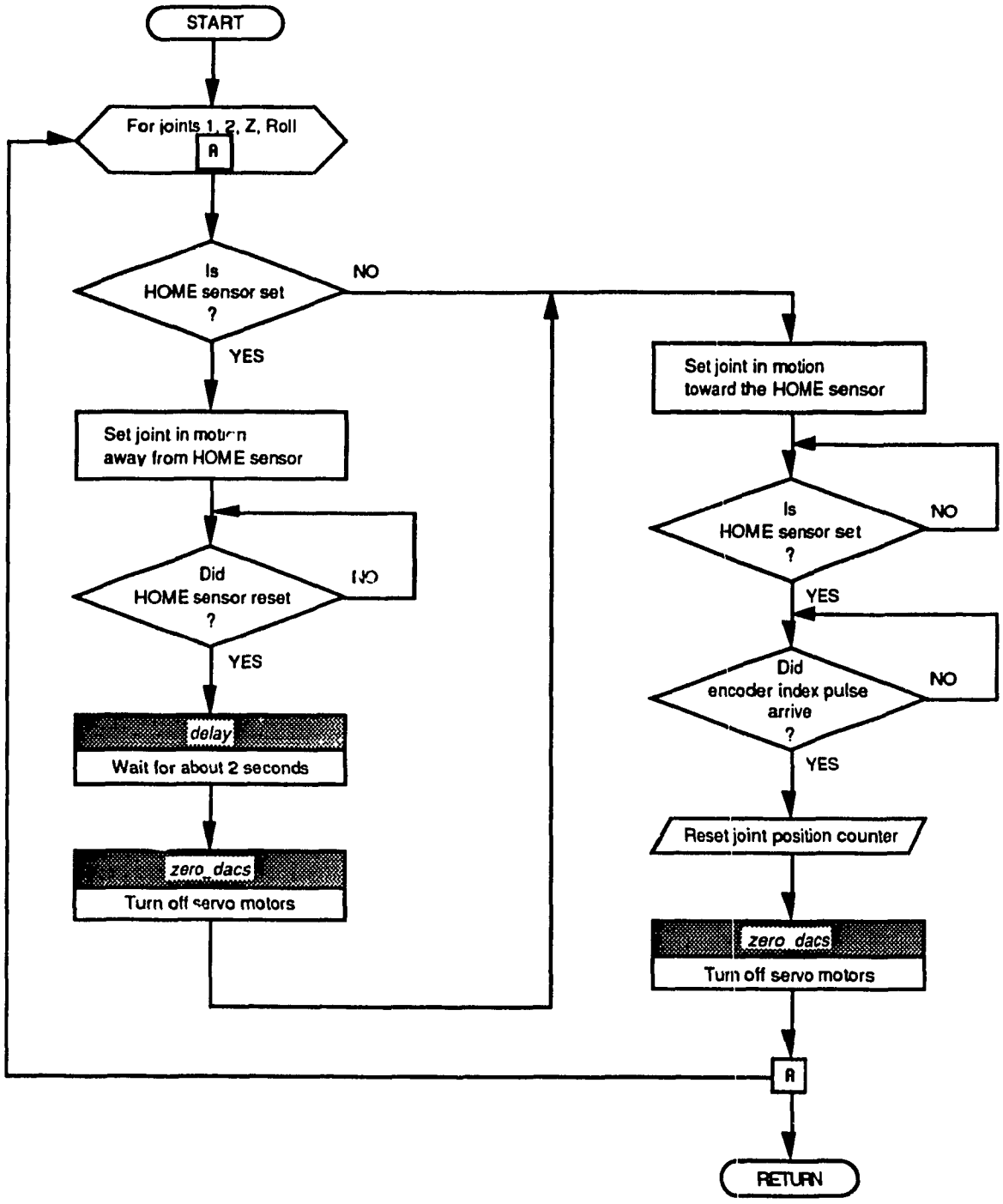


Figure 4.3. Flow Chart for the *Find-HOME* Mode.

The *find-HOME* sequence commences with the reading of the joint's HOME sensor. If the HOME sensor is activated (active low) the slave applies a positive constant torque to the motor in order to move the joint away from the HOME sensor (open-loop

control). The torque continues to be applied for a period of approximately 2 seconds (governed by the *delay* procedure) following the deactivation of the HOME sensor.

Hence or otherwise, the slave applies a negative torque to the motor in order to move the joint toward the HOME sensor. The slave polls the HOME sensor during this motion. When the slave detects activation of the HOME sensor, it begins polling the encoder index line. When the index pulse is detected, the slave processor first instructs the HCTL-1000 to perform an on-chip software reset which among other things resets the 24-bit position counter to zero, and then instructs the joint motor to stop. In the first instruction, the second HCTL-1000 of the pair (recall from Section 3.5 that the HCTL-1000s are accessed in pairs) is instructed to remain in its Idle/Initialization mode.

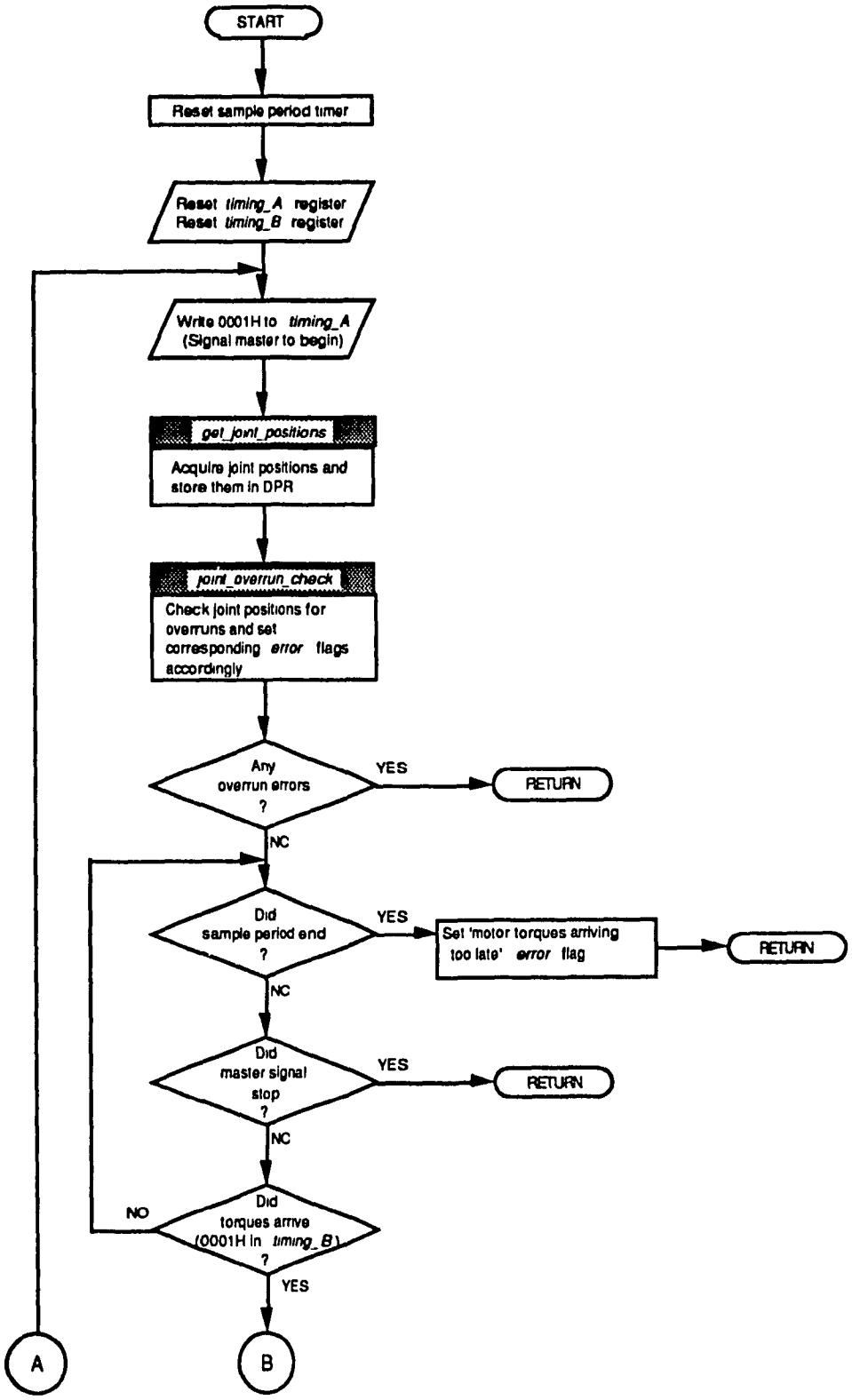
Joint motion during the *find-HOME* sequence is the result of a constant torque applied in an open-loop configuration. The torque is of sufficient magnitude to move the joint at a constant slow speed. Because of inertia, motion does not cease instantaneously and there is always slight overshoot of the HOME position. The *find-HOME* procedure, therefore, does not leave the manipulator in the HOME position; it simply locates HOME and initializes the position counters at that point.

Upon completion of the *find-HOME* procedure, the slave processor reenters the main program at the point where *command* is cleared. A clear *command* register indicates to the master that HOME was found.

4.3.3 *Control* Mode

The *control* mode is entered if, while in the waiting loop, the slave finds a value of 0001H (bit 0 set) in the *command* register. In the *control* mode, the slave processor serves as an intelligent interface between the 7545 manipulator and the master processor which is

executing the robot control algorithm. The *control* mode procedure is shown in the flow chart in Figure 4.4 and is described below.



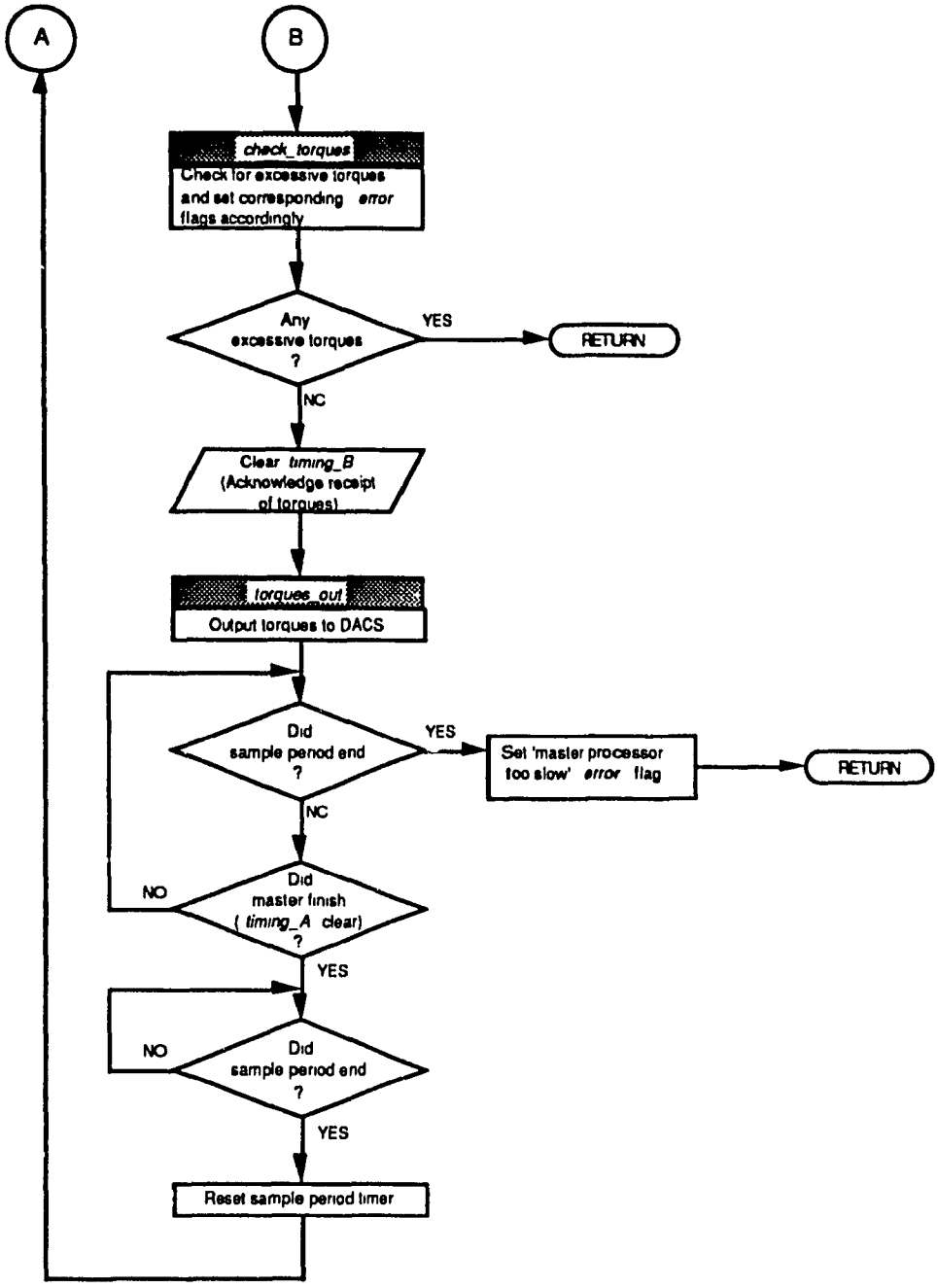


Figure 4.4. Flow Chart for the *Control* Mode.

Upon entering the *control* mode, the slave resets the sample period timer (80C196KA special purpose register TIMER2) and the *timing_A* and *timing_B* registers. Following this, the slave begins its servo loop and tells the master to do the same by setting the 'begin servo loop' semaphore (bit 0) in *timing_A*. The slave then calls the *get_joint_positions* procedure whereby the position counters are read and stored in the *actual_position* registers

and a semaphore set in the *timing A* register to indicate to the master that the joint positions are available (the procedure *get_joint_positions* is described in detail in Section 4.3.4.1). The slave then calls the *joint_overrun_check* procedure (see Section 4.3.4.4) where the pulse count of each joint is checked for excessive values and the *error* register updated. The slave returns from this procedure, checks the *error* register, and exits the *control* mode if any semaphores are set. This immediately halts manipulator motion by turning off the servo motors (see Figure 4.2).

Continuing in the *control* mode, the slave tests for the end of the sampling period and for the motor torques to arrive from the master processor. The end of the sampling period is when the sample period timer, TIMER2, reaches the value in the *sample_period* register and the arrival of the torques is indicated by a semaphore in *timing_B* . If the servo loop ends before the torques arrive, the slave sets the 'torques arriving too late' flag in the *error* register and exits the *control* mode. If the master commands the slave to stop during this process, the slave exits the *control* mode.

The slave proceeds in the *control* mode by calling the *check_torques* procedure (see Section 4.3.4.5 and Figure 4.4) which checks for excessive torque values and sets the corresponding flag in the *error* register.

Returning from this procedure, the slave checks the *error* register and exits the *control* mode if it is not clear. If all torques are within range, the slave continues by clearing *timing_B* and outputting the torques to the DACs.

The slave then tests for the end of the sampling period and for the master to indicate completion of its control loop (clearing of *timing_A*). If the sampling period ends before the master completes its loop, the slave sets the 'master processor too slow' flag in the *error* register and exits the *control* mode. If the master finishes its loop in time, the slave enters another loop where it waits for the end of the sample period.

When the sampling period ends, the sampling period timer value will equal or exceed the value in the *sample_period* register. The slave resets the timer by subtracting from it the value in the *sample_period* register.

4.3.4 Additional Procedures

This section describes in more detail the procedures mentioned in the previous sections. Flow charts are provided for most of the procedures and the source code is included in Appendix J.

4.3.4.1 *Get_Joint_Positions*

This procedure reads the three 8-bit actual position registers of each HCTL-1000, sign-extends the 24-bit position data to 32 bits, and stores them in the DPR *actual_position* registers.

As mentioned in section 3.5, reading of an HCTL-1000 8-bit register actually requires two read instructions (the first to generate the chip select signal and the second to generate the output enable signal 1.8 microseconds later). In addition, it was mentioned that the HCTL-1000s are read in pairs. Because of this, the 16-bits of data obtained by the second read instruction must be sorted. In light of these requirements, the two read instructions are interlaced with instructions that sort the data and sign-extend them to 32 bits. The exact instruction sequence can be found in the source code listing in Appendix J.

Sign-extension enables the 24-bit position data to occupy the double word *actual_position* registers and still retain their positive or negative sense. To accomplish sign-extension, we recognize that each joint can move a finite distance into its negative region and even in their most negative positions, the most significant byte of their 24-bit

position count will be FFH. Therefore, sign extension is easily accomplished by first clearing the most significant byte of the 32-bit register and then setting it to FFH providing the second most significant byte (i.e., the most significant byte of the 24-bit count) is FFH.

The final step in the procedure is the storage of the 32-bit joint positions in the *actual_position* registers and setting the semaphore in *timing_A*.

4.3.4.2 *Delay*

This procedure generates a delay of roughly 2 seconds for use in the *find-HOME* mode. The delay is generated by instructing the processor to count down from FFFH to 0 eight times. The flow chart is given in Figure 4.5.

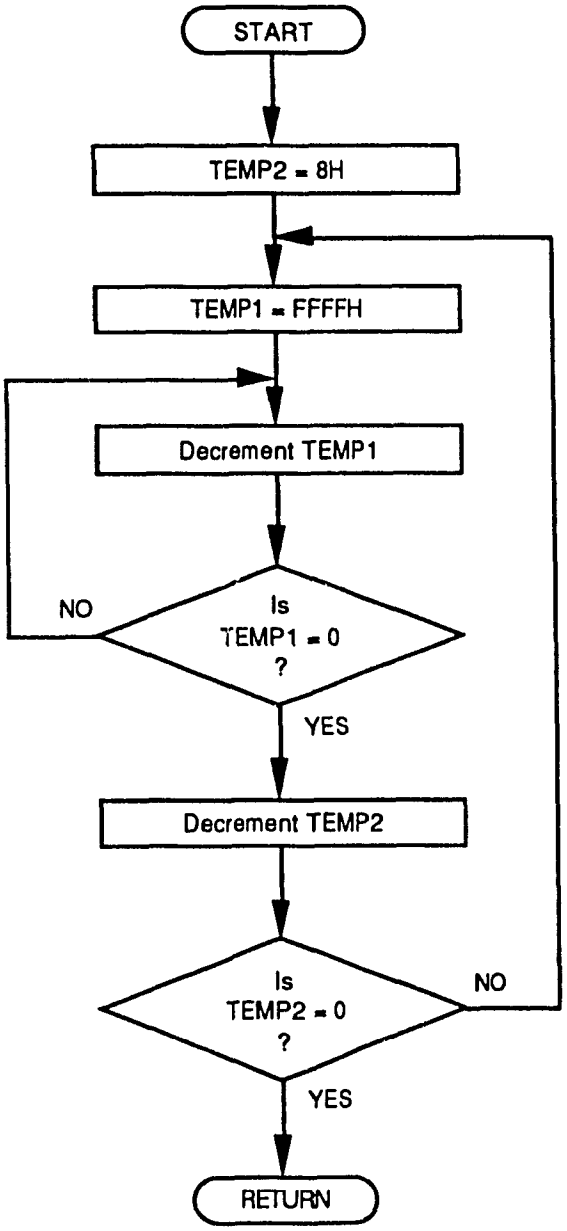


Figure 4.5. Flow Chart for Delay Procedure.

4.3.4.3 *Zero_DACs*

This procedure simply sets all the DAC outputs to zero. The slave begins by writing 800H (the bipolar offset value) to the first rank register of each DAC. Then the second rank register of each DAC is loaded simultaneously with one write instruction. The flow chart is shown in Figure 4.6.

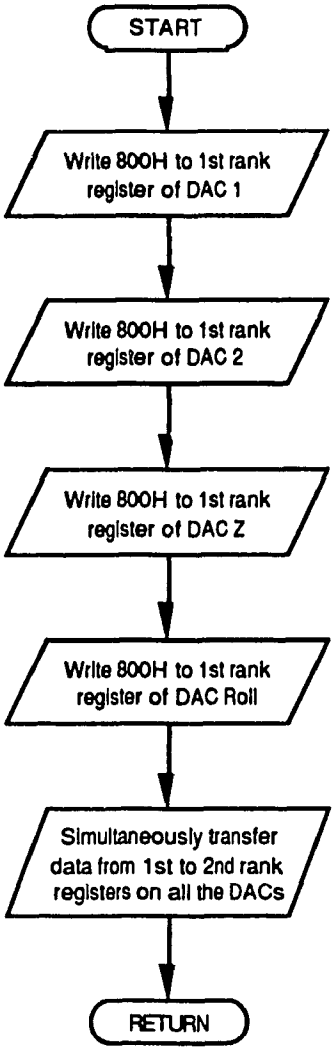


Figure 4.6. Flow Chart for *Zero_DACs* Procedure.

4.3.4.4 *Joint_Overrun_Check*

This procedure constitutes software limit switches for the detection of joint overrun conditions. A negative joint position is compared with *MIN*, a positive joint position is compared with *MAX*. If a joint position is more negative than *MIN* or more positive than *MAX*, a flag is set in the *error* register. The values for *MIN* and *MAX* are chosen so that the software limit switches activate before the 7545's hardware limit switches. This prevents unnecessary shut-down of manipulator power by the 7545's control unit. Figure 4.7 shows the flow chart for this procedure.

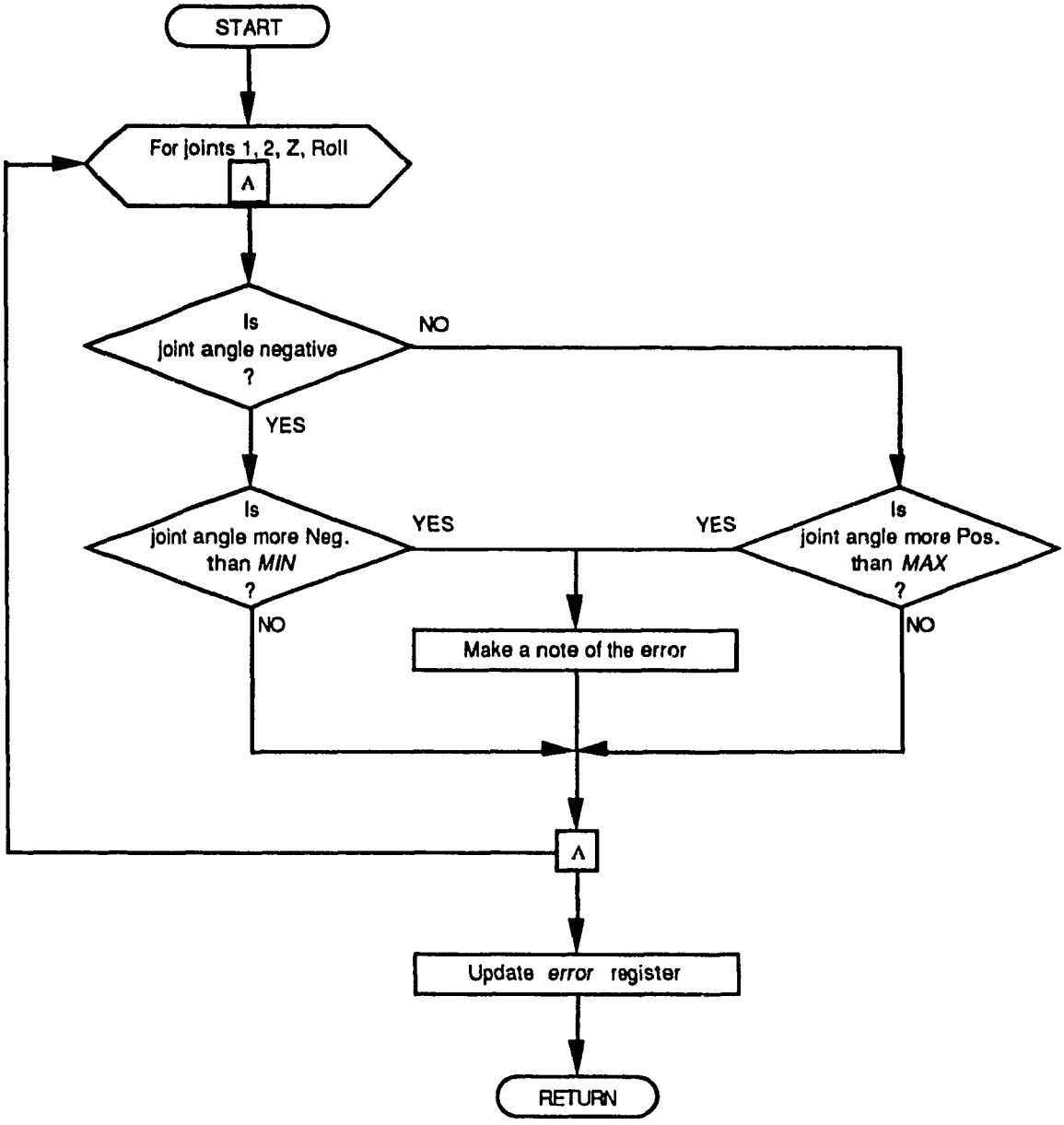


Figure 4.7. Flow Chart for *Joint_Overrun_Check* Procedure.

4.3.4.5 *Check_Torques*

This procedure reads the torques from the DPR and verifies that they are within the 12-bit range (0-FFFH) of the DACs. Each torque is logically ANDed with F000H. If the result is non-zero, the torque is excessive and the corresponding flag is set in the *error* register. The flow chart is given in Figure 4.8.

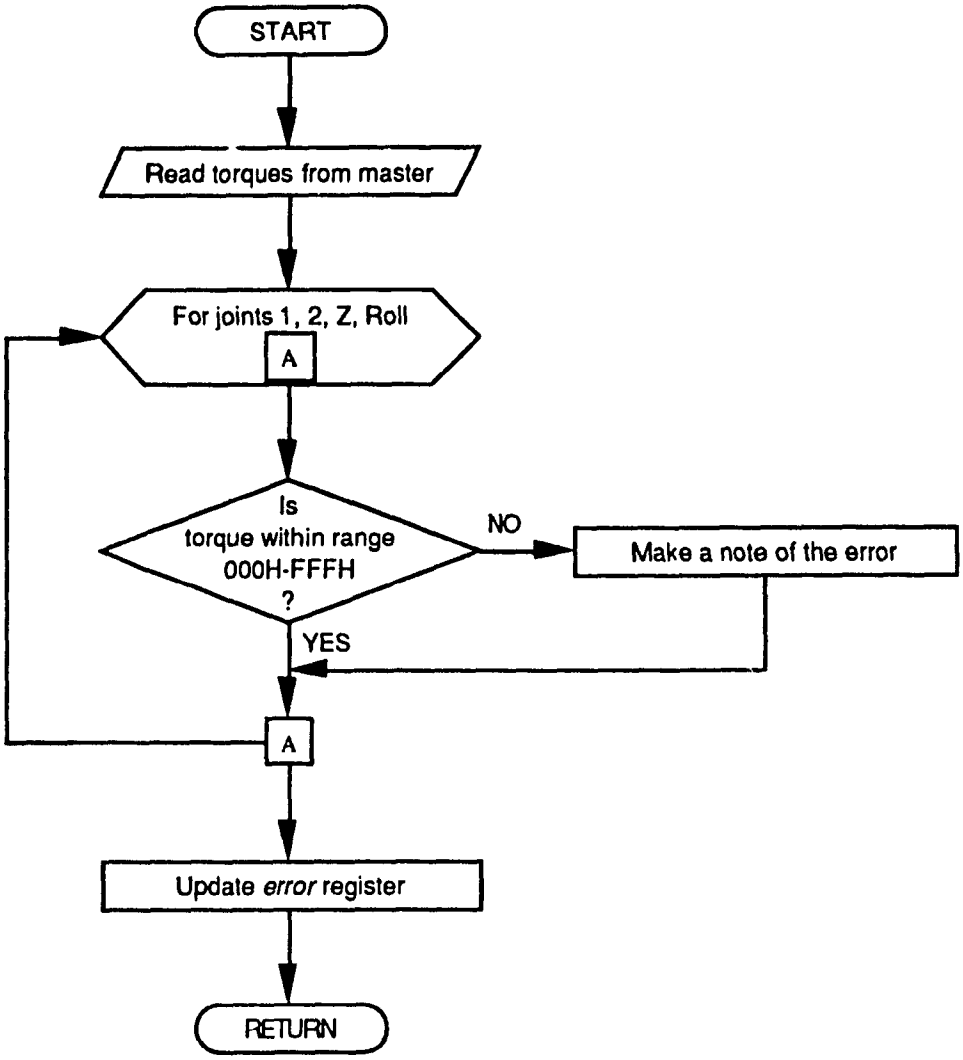


Figure 4.8. Flow Chart for *Check_Torques* Procedure.

4.3.4.6 *Torques_Out*

This short procedure sequentially writes the torques to the first rank of each DAC and then simultaneously writes them to the second rank registers. The flow chart is given in Figure 4.9.

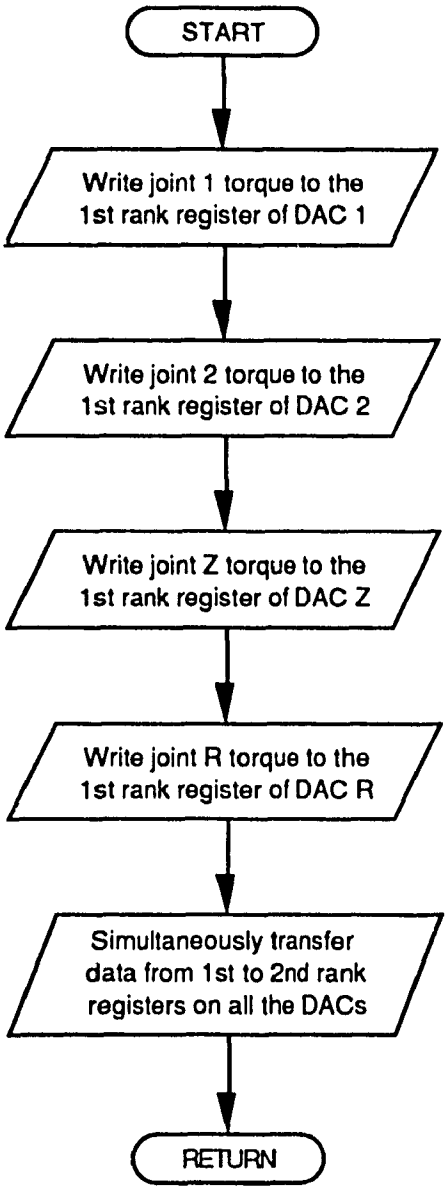


Figure 4.9. Flow Chart for *Torques_Out* Procedure.

4.4 MASTER PROCESSOR SOFTWARE

As indicated in Figure 4.1, the master software consists of path planning algorithms and robot control algorithms. So far, two control and two path planning algorithms have been implemented. These algorithms have been coded as functions in the C language. Each of the functions resides in its own separate file for developmental purposes and once debugged, is linked to a main function for execution. Together, these files form a growing library of robotic control strategies. The library also contains a utility file which consists of the main function and various user-interface functions. These functions were designed to create an environment conducive to the development, integration, execution and evaluation of path planning and control algorithms. The library source code can be found in Appendix K.

The following sections outline the operation of the master processor. The first section describes the sequence of events that lead up to the execution of a move and hence the manner in which the library files are tied together. The second section lists the operations that are performed by the master during the move. This is followed by a description of the various control and path planning algorithms currently in the library.

4.4.1 Master Sequence of Operation

The sequence of operation of the master software is shown in Figure 4.10. The events are listed chronologically and are displayed in their respective files.

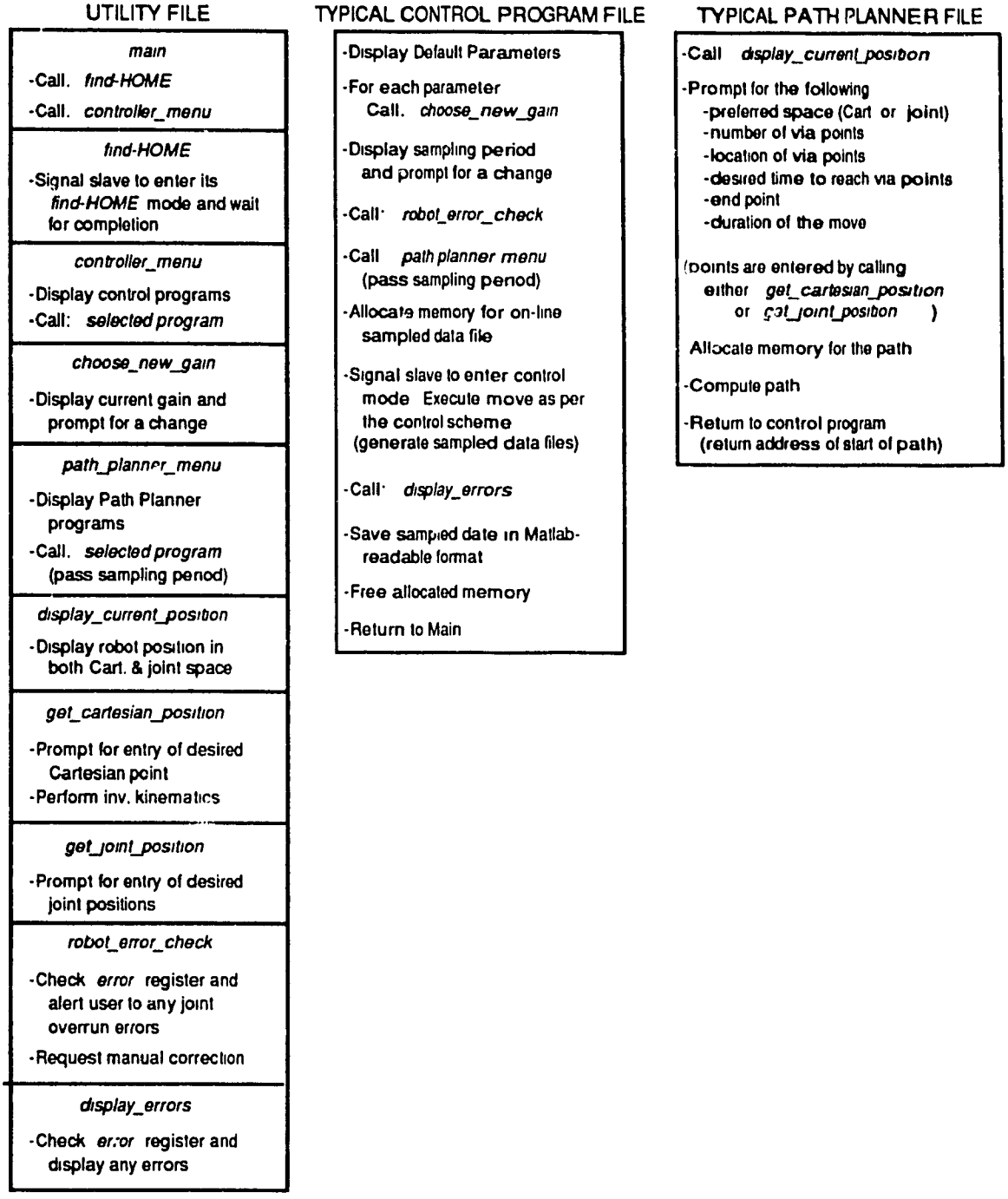


Figure 4.10. Sequence of Operation of the Master Software.

The sequence begins in the *main* function with a call to the *find-HOME* function. This function clears the DPR *error* register, commands the slave to enter its *find-HOME* procedure by writing 0080H to the *command* register, and waits for the slave to clear the register indicating that HOME was found.

Following the *find-HOME* function the *controller_menu* function is called whereby the user is prompted for the selection of a control algorithm. The menu also includes the *find-HOME* option

The control algorithms typically begin by displaying the parameter (gain) default values and then prompting the user for changes. Parameter changes are handled by the *choose_new_gain* function. The control algorithm proceeds by displaying the sampling period (calculated from the value in the *sample_period* register) and prompting the user for a change. Next, the *robot_error_check* function is called whereby the *error* register is checked and the user alerted to any joint overruns. The function requires that the overrun joints be manually moved to correct the error. The control algorithm then calls the *path_planner_menu* function where the user is prompted for the selection of a path planning algorithm.

The path planning algorithms typically begin by calling the *display_current_position* function which displays the manipulator's current point in both Cartesian and joint space (the term "point" refers to both position and orientation). The user is then prompted for the preferred space (Cartesian or joint) in which to enter desired path points. The user is then prompted for entry of the path end point, the duration of the move, and if the planner has via point specification capability, the number of via points, their values, and the desired time between successive points.

The input of a path point is handled by one of two functions: *get_joint_position* or *get_Cartesian_position*. The latter function solves the inverse kinematic equations. Both functions warn when the desired path point lies outside the manipulator's workspace.

Once all the necessary information has been entered, the entire path (the position of each joint at every sampling period) is computed off-line and stored contiguously in

memory. A flag indicates the path end-point. The arrangement of the path in memory is shown below.

sample period 1, joint 1 pulse count
sample period 1, joint 2 pulse count
sample period 1, joint Z pulse count
sample period 1, joint Roll pulse count
sample period 2, joint 1 pulse count
. . .
last sample period, joint Roll pulse count
end-of-path flag

Lastly, the path planner algorithm returns to the control algorithm, a pointer to the first path point. In the control algorithm, the master allocates space in its memory to hold the data that will be obtained during the move for future analysis. The control algorithm then checks to make sure there are no joint overruns and prompts the user to begin or abort the move. If the user aborts the move, control passes to the *controller_menu* function. If the user specifies to begin the move, the master signals the slave to enter its *control* mode (sets bit 0 in the *command* register). The succeeding sequence of operations is described in the following section.

4.4.2 Servo Control Loop

The master waits for the slave to signal the start of a new sampling period (0001H in the *timing_A* register). When the signal arrives, the master begins by storing certain variables from the previous period for use in the present sampling period (e.g., joint positions from the previous period are employed to compute joint velocities using the difference method). When the actual joint positions become available (indicated by 0003H in *timing_A*), the master computes the required tracking errors. The master then

computes the torques using one of the two control algorithms (see Section 4.4.3). It stores them in the *torque* registers and sets the 'torques available' semaphore (bit 0) in *timing_B*. At this point, the slave verifies the desired torques and the master saves the relevant data (e.g., actual and desired joint positions, torques) in data files for future analysis. If the slave finds the torques are acceptable, it clears *timing_B*; and the master subsequently clears *timing_A*. The master then waits for the start of the next servo loop. If the slave indicates an error, the master exits the control mode before the start of the next sampling period.

If the entire move is successfully executed (no errors), the control program saves the sampled data files on hard disk in MATLAB readable format for later analysis using PC- or PRO-MATLAB. If the move ends prematurely due to an error, the *display_errors* function is called which alerts the user as to the nature of the error.

4.4.3 Robot Control Programs

The library at present contains two robot control algorithms: Proportional-Derivative (PD) control, and decentralized adaptive control.

4.4.3.1 PD Control

The PD control algorithm provides error-driven, independent joint control whereby each joint is controlled separately by a simple position-velocity servo-loop with predefined constant gains. The joint torque at a sampling instant N is given by

$$\tau(N) = k_p\, e_p(N) + k_v\, e_v(N) \qquad (4.1)$$

where $e_p(N)$ =desired joint position(N)-actual joint position(N) and $e_v(N) = \frac{e_p(N) - e_p(N-1)}{T_s}$ are the position and velocity tracking errors, and $k_p > 0$ and $k_v > 0$ are the position and velocity constant scalar feedback gains.

The PD controller's main attribute is that it requires few mathematical computations and thus can be implemented at high servo rates.

4.4.3.2 Decentralized Adaptive Control

The decentralized adaptive controller is based on an algorithm developed at the Jet Propulsion Laboratory by Seraji [17]. The control scheme is based on the independent joint control concept, does not use the complex manipulator dynamic model, and each joint is controlled simply by a PID feedback controller with adjustable gains. The reader should note that the control scheme as presented in [17] features a position-velocity-acceleration feedforward controller (with adjustable gain). However, in our implementation (as in Seraji's practical implementation), the feedforward controller is not included to reduce the on-line computation time.

The control algorithm for each joint is as follows. The motor torque τ computed at a sampling instant N is given by

$$\tau(N) = f(N) + k_p\, e_p(N) + k_v\, e_v(N) \qquad (4.2)$$

where f(n) is an auxiliary signal produced by the adaptation scheme to improve tracking performance and partly compensate for disturbances, k_p is the proportional feedback gain, k_v is the velocity feedback gain, $e_p(N)$ = desired joint position(N) - actual joint position(N) is the position error, and $e_v(N) = \frac{e_p(N) - e_p(N-1)}{T_s}$ is the velocity error formed by the software. The auxiliary signal and the controller gains are obtained by the following recursive adaptation laws:

$$f(N) = f(N-1) + \delta\frac{T_s}{2}[r(N)+r(N-1)] + \rho[r(N)-r(N-1)] \qquad (4.3)$$

$$k_p(N) = k_p(N-1) + \alpha_p\frac{T_s}{2}[r(N)e_p(N) + r(N-1)e_p(N-1)] + \beta_p[r(N)e_p(N) - r(N-1)e_p(N-1)]$$

$$k_v(N) = k_v(N-1) + \alpha_v\frac{T_s}{2}[r(N)e_v(N) + r(N-1)e_v(N-1)] + \beta_v[r(N)e_v(N) - r(N-1)e_v(N-1)]$$

where $r(N) = \omega_p e_p(N) + \omega_v e_v(N)$ is a weighted error with $\{\omega_p, \omega_v\}$ being positive constant scalar weighting factors which reflect the relative significance of the position and velocity errors, $\{\delta, \alpha_p, \alpha_v\}$ are positive constant scalar integral adaptation gains, and $\{\rho, \beta_p, \beta_v\}$ are non-negative proportional adaptation gains.

4.4.4 Path Planning Programs

The path planning algorithms are called by the control programs and compute the entire path off-line prior to the execution of the move. This affords the master processor more time during the move to execute computationally intensive control algorithms. Two path planning algorithms have been coded for the library: a cycloid function-based generator and a cubic spline generator. The latter features via point specification capability.

4.4.4.1 The Cycloid Function

This path generator plans a path of duration T seconds between the initial point $\theta(0)$ and the goal point according to the function

$$\theta(t) = \theta(0) + \frac{\Delta}{2\pi}[\omega t - \sin \omega t], \qquad 0 \le t \le T \qquad (4.4)$$

where $\omega = 2\pi / T$ and $\Delta = \theta(T) - \theta(0)$. The cycloid is a smooth trajectory with a bell-shaped velocity profile and a sinusoidal acceleration profile given by

$$\dot{\theta}(t) = \frac{\Delta}{t}[1 - \cos \omega t]$$
$$\ddot{\theta}(t) = \frac{2\pi\Delta}{T^2} \sin \omega t, \quad 0 \le t \le T \tag{4.5}$$
$$\dot{\theta}(t) = \ddot{\theta}(t) = 0, \qquad \tau \ge T.$$

4.4.4.2 Cubic Spline

The algorithm for this path generator enables the specification of via point locations and also provides automatic selection of joint velocities at the via points. The velocities at the via points are computed according to the following simple heuristic scheme. First, the slopes (average joint velocities) between adjacent via points is calculated. The velocity at a via point is then chosen to be zero if the two slopes on either side are of opposite sign. If the slopes are of the same sign, the velocity is computed as the average of the two slopes.

Now that both position and velocities for the joints at the via points are known, the algorithm determines the cubic splines linking adjacent via points. The cubic splines are governed by the following four constraints

$$\begin{aligned} \theta(0) &= \theta_0, \\ \theta(t_f) &= \theta_f, \\ \dot{\theta}(0) &= \dot{\theta}_0, \\ \dot{\theta}(t_f) &= \dot{\theta}_f, \end{aligned} \tag{4.6}$$

where θ_0 and θ_f are the joint positions at the beginning and end of the spline, $\dot{\theta}_0$ and $\dot{\theta}_f$ are the joint velocities at the beginning and end of the spline, and t_f is the time duration of the spline. The four constraints can be satisfied by the third degree (cubic) polynomial

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3. \tag{4.7}$$

The joint velocity and acceleration are given by the first and second derivatives

$$\dot{\theta}(t) = a_1 + 2a_2 t + 3a_3 t^2 \qquad (4.8)$$
$$\ddot{\theta}(t) = 2a_2 + 6a_3 t.$$

Substituting the four constraints into equations (4.7) and (4.8) gives four equations in four unknowns:

$$\begin{aligned} \theta_0 &= a_0 \\ \theta_f &= a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 \\ \dot{\theta}_0 &= a_1, \\ \dot{\theta}_f &= a_1 + 2 a_2 t_f + 3 a_3 t_f^2. \end{aligned} \qquad (4.9)$$

The solution of these equation for the coefficients a_i is

$$\begin{aligned} a_0 &= \theta_0, \\ a_1 &= \dot{\theta}_0, \\ a_2 &= \frac{3}{t_f^2}(\theta_f - \theta_0) - \frac{2}{t_f}\dot{\theta}_0 - \frac{1}{t_f}\dot{\theta}_f, \\ a_3 &= -\frac{2}{t_f^3}(\theta_f - \theta_0) + \frac{1}{t_f^2}(\dot{\theta}_f + \dot{\theta}_0). \end{aligned} \qquad (4.10)$$

Expression (4.10) is sequentially evaluated at all adjacent points to derive the cubic splines. The end result is a smooth continuous path linking the initial point, all the via points and the goal point together.

4.5 SUMMARY AND FUTURE WORK

This chapter presented the software that was developed to implement a testbed for robotic control strategies. The slave processor software was designed to support a variety of control algorithms and therefore does not require any immediate modifications. Of course, implementation of some of the hardware suggested in Section 3.8 would demand

modification to the slave software. This would also be the case if the ARC were to be used to control a robotic system other than the IBM 7545.

With regard to the master, future work may consist of expanding the library through the implementation of other advanced control strategies. An interesting project here is the provision for on-line path planning. This would allow the researcher to address the problem of collision avoidance (this, of course, would require hardware for workcell operation). Other areas for future work include the enhancement of the user-interface software in the utility file. This may be accomplished through the use of specialized programs such as C-Scape.

With suitable development, the ARC could serve in the industrial domain. This would require a compiler or interpreter for the translation of an existing (eg., AML) or custom-designed high-level application language.

CHAPTER 5

EXPERIMENTAL RESULTS

This chapter presents the results of three experiments which were conducted to verify the operation of the ARC and to judge its effectiveness and performance in robotic control applications. The experiments involved the use of both the cubic spline and cycloid function path generators, as well as the PD and adaptive control schemes. The servo rates for these latter controllers was set to 1 KHz and 250 Hz, respectively. It should be noted that although the controller gains were tuned, no effort was made to optimize the gains for the experiments. Furthermore, comparisons between the control schemes are not given. It is not the purpose of this project to make value judgements regarding the control schemes but simply to show the effectiveness of the ARC for implementing such schemes. A comparison between the performance of the ARC and the MTCB would have been desirable. Unfortunately, the MTCB does not provide the means for acquiring data during execution of a move. The mere fact that the ARC provides this feature makes it a valuable research tool for the evaluation of robotic control strategies.

In first experiment, the ARC was used to evaluate a controller's ability to compensate for unexpected disturbances. A path was planned by the cycloidal function generator for joints 1 and 2 to move from 0° to 90° in 2.5 seconds and for joints Z and Roll to maintain their position. A disturbance was applied at approximately mid-path when the end-effector came into contact with a movable object which was constrained to move only along the x direction. The end-effector was required to push the object as it proceeded toward its goal position. The experiment was performed under both PD and adaptive control. Figures 5.1, 5.2, and 5.3 show the joint 1 tracking errors, the joint 2 tracking errors, and the torque outputs for the PD controller. Figures 5.4, 5.5, and 5.6 show the

corresponding results for the adaptive controller. The effect of the disturbance is clearly shown in each graph. However, despite the disturbance, both controllers are able to compensate and reduce the tracking error within a finite period of time.

In the second experiment, the ARC was used to demonstrate a controller's ability to track a closed path. The cubic spline generator with its via point capability was used to plan a closed path for all four of the manipulator's joints. The duration of the move was 8 seconds and three via points were specified at 2 second intervals. The set of joint variables {q1, q2, Z, Roll} for the start point, the three via points and the end point were {0°, 90°, 0 mm, 0°}, {90°, 0°, -70 mm, 135°}, {180°, 90°, -210 mm, -30°}, {90°, 135°, -140 mm, -150°}, {0°, 90°, 0 mm, 0°}, respectively. Figure 5.7 shows the desired path in the x-y plane (Cartesian space). Figures 5.8, 5.9, 5.10, 5.11, and 5.12 show the joint tracking errors for joints 1 2, Z, and Roll and the torques for the PD controller, respectively. Corresponding results for the adaptive controller are shown in Figures 5.13, 5.14, 5.15, 5.16 and 5.17. Despite the fact the gains were not optimized, tight control over all four joints was achieved.

The third experiment demonstrated the ARC's effectiveness in handling slow and fast trajectories. In this experiment, the starting point was {90°, 90°, -70 mm, -35°} and the goal point was {10°, 30°, 0 mm, 0°}. The cubic spline generator was used to generate a 3 sec., a 2 sec., and a 1 sec. trajectory between these two points for use by the adaptive controller. The results are shown in Figures 5.18 - 5.32. As expected, the joint tracking errors and the torque demands were highest for the 1 sec. trajectory. Attempts at trajectories faster than 1 sec. were met with excessive torque demand errors.

The finite processing power of the ARC imposes a limitation on the maximum servo rate of the controller (e.g., 1000 Hz for PD control, 250 Hz for adaptive control). This limitation reflects on the maximum trajectory that can be executed without changing the controller gains. A faster servo rate will generally produce smaller tracking errors and

thus smaller torque demands for a particular sampling period. This in turn will allow for faster trajectories to be executed. For a fixed servo rate, faster trajectories may be achieved by detuning the controller gains. The detuned gains prevent the demanded torques from reaching the limits for the robot. The faster trajectories, however, come at the expense of accuracy since the detuned gains will degrade performance by increasing the tracking errors.

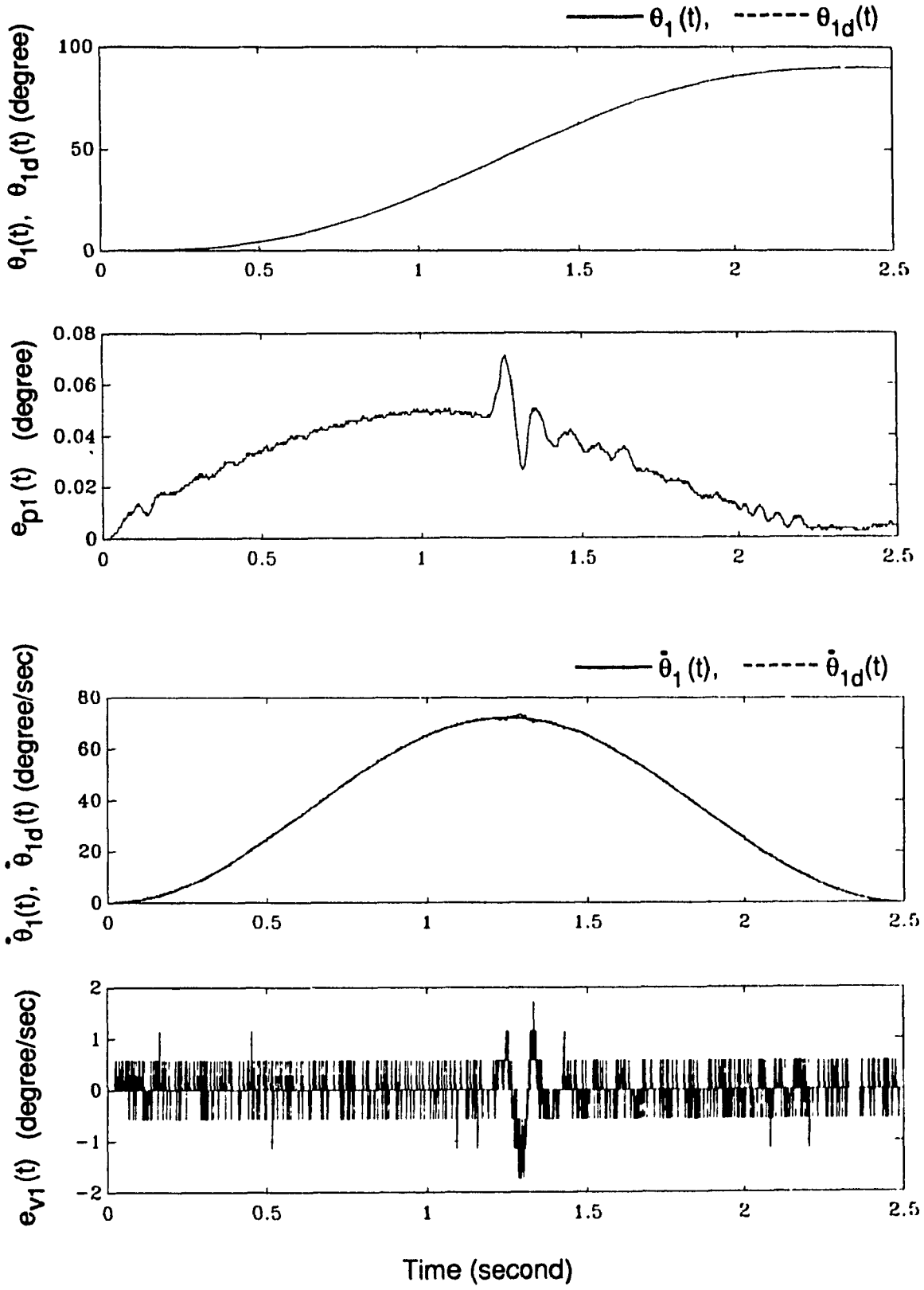


Figure 5.1. Joint 1 Tracking Errors, PD Control, Experiment 1.

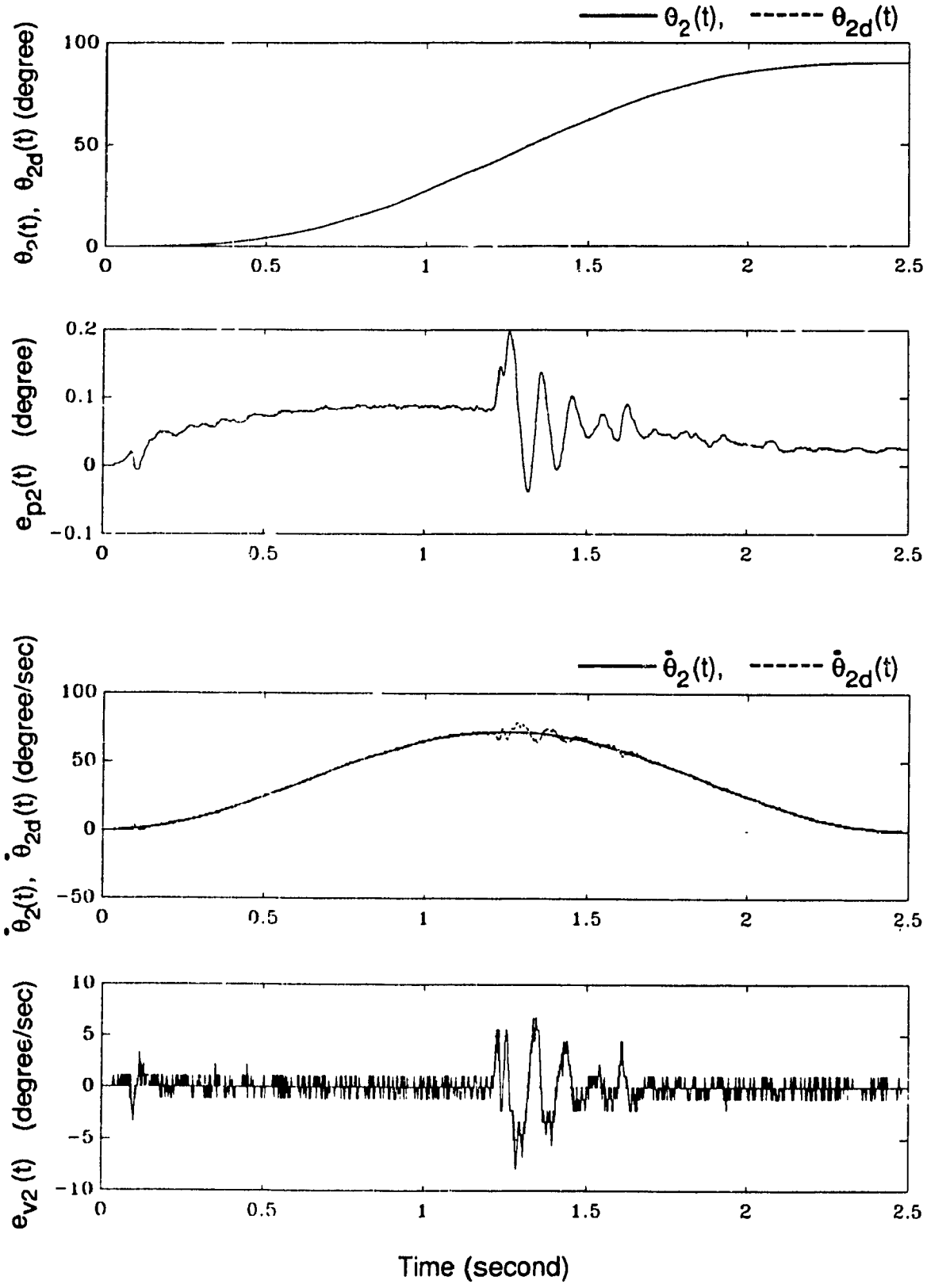


Figure 5.2. Joint 2 Tracking Errors, PD Control, Experiment 1.

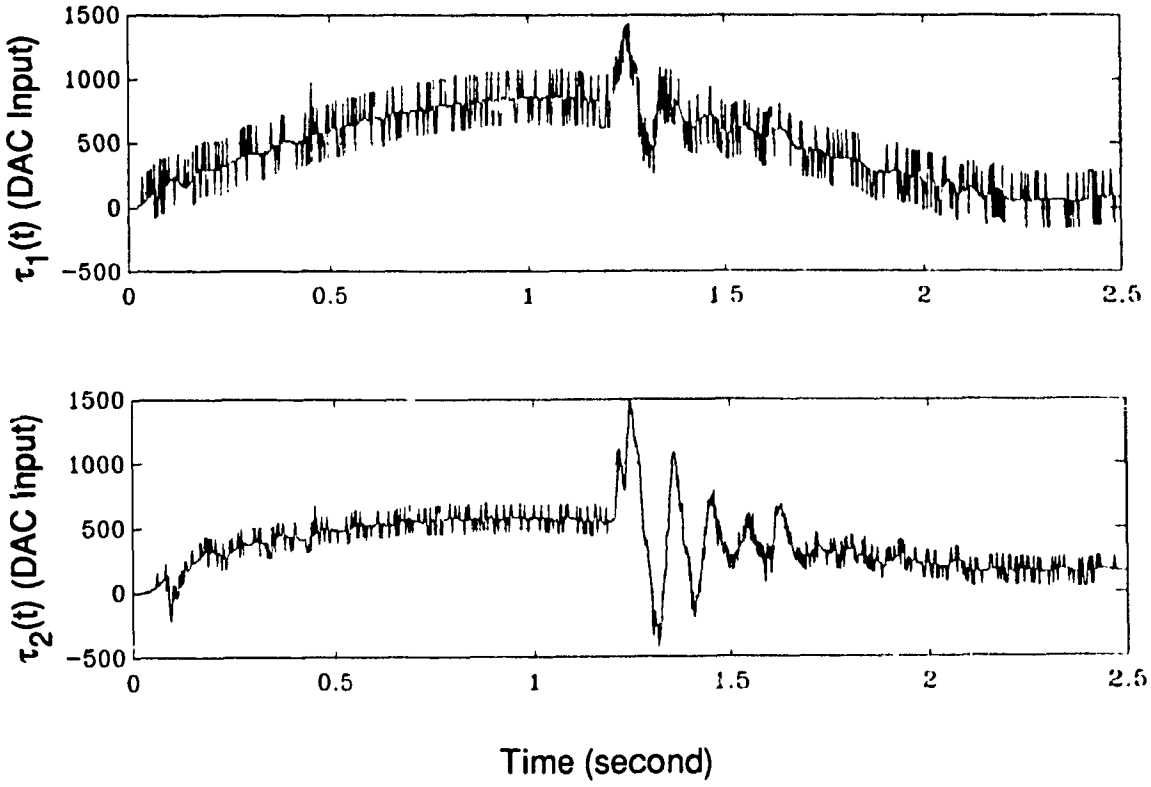


Figure 5.3. Torques, PD Control, Experiment 1.

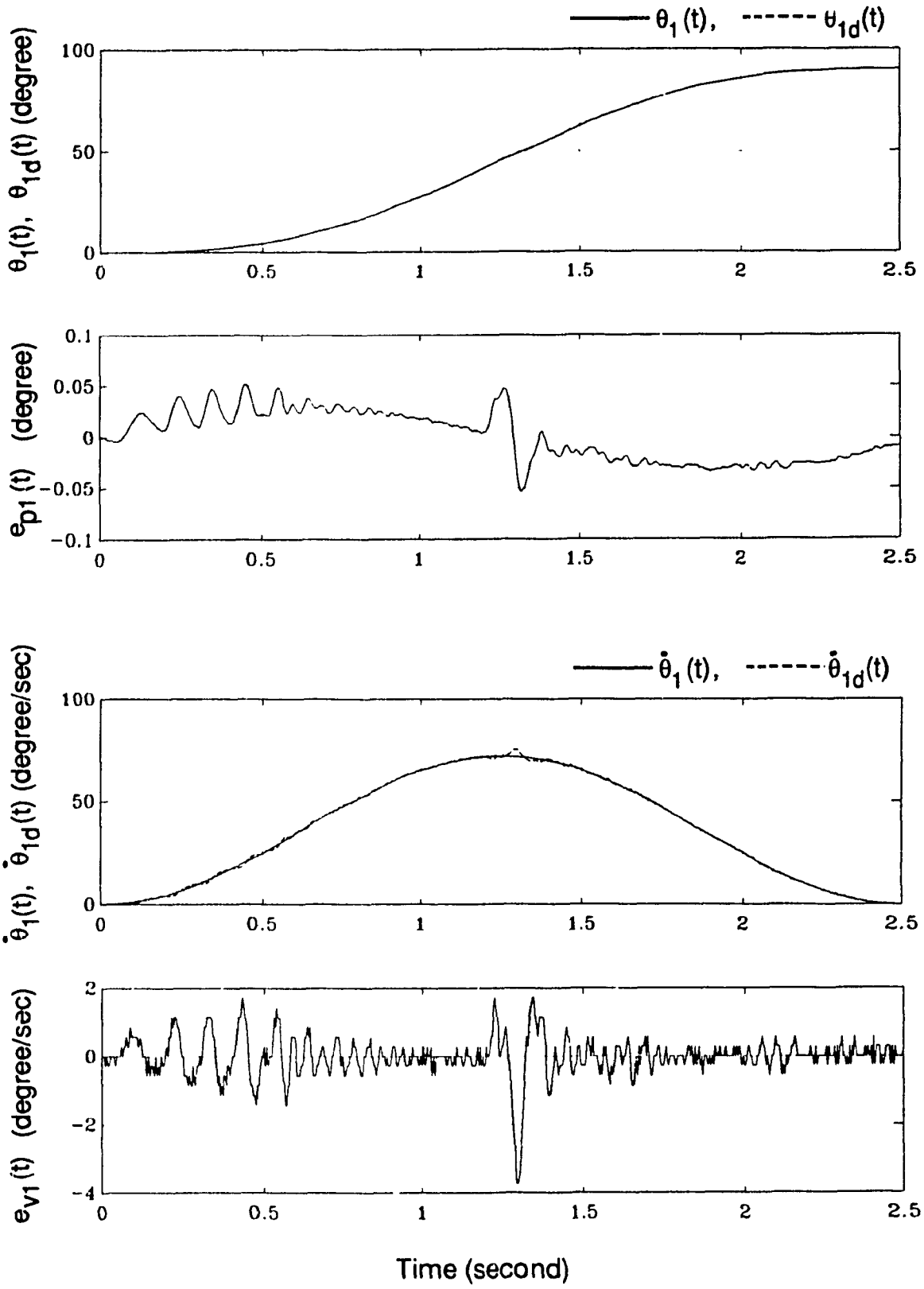


Figure 5.4. Joint 1 Tracking Errors, Adaptive Control, Experiment 1.

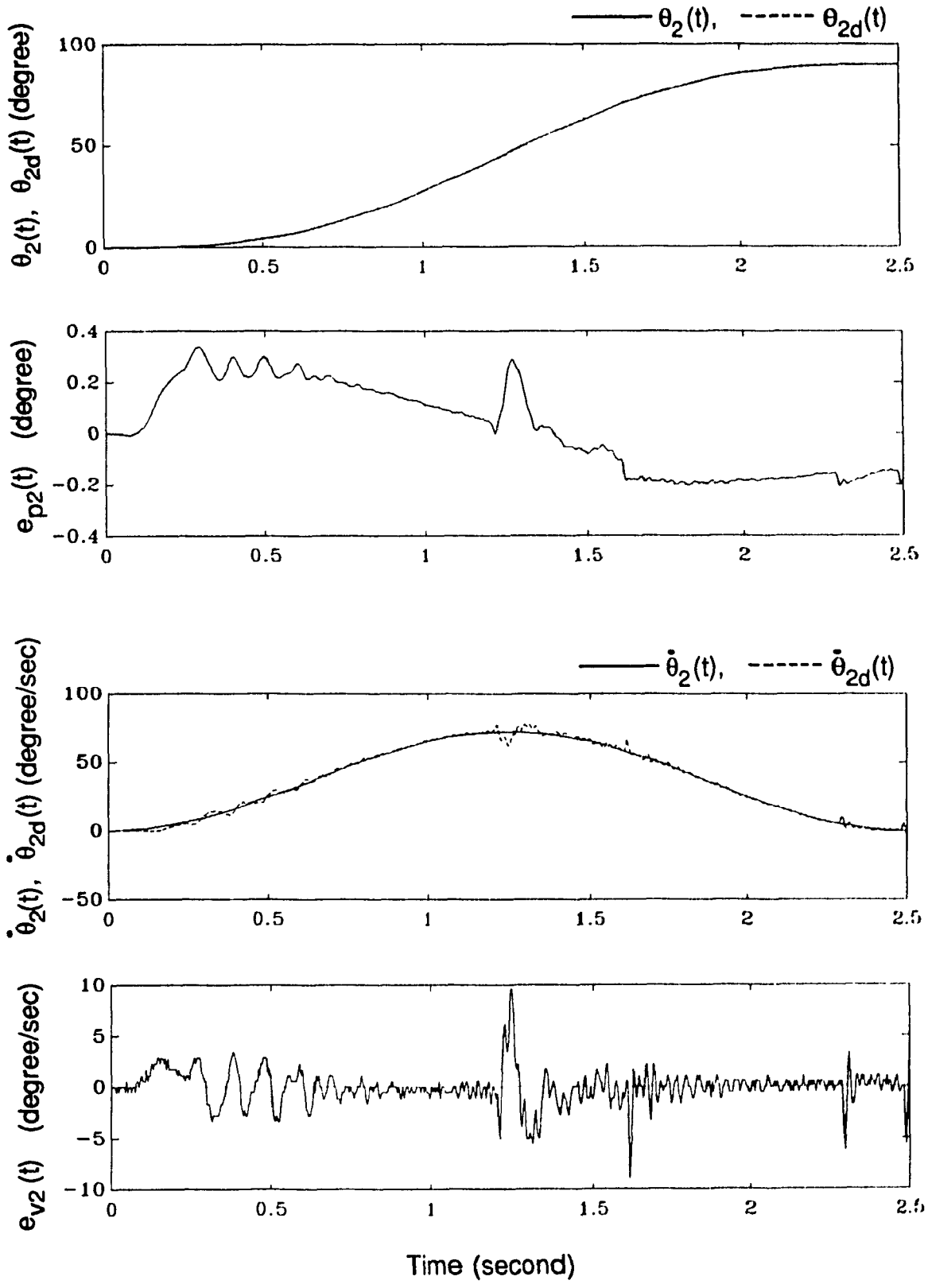


Figure 5.5. Joint 2 Tracking Errors, Adaptive Control, Experiment 1.

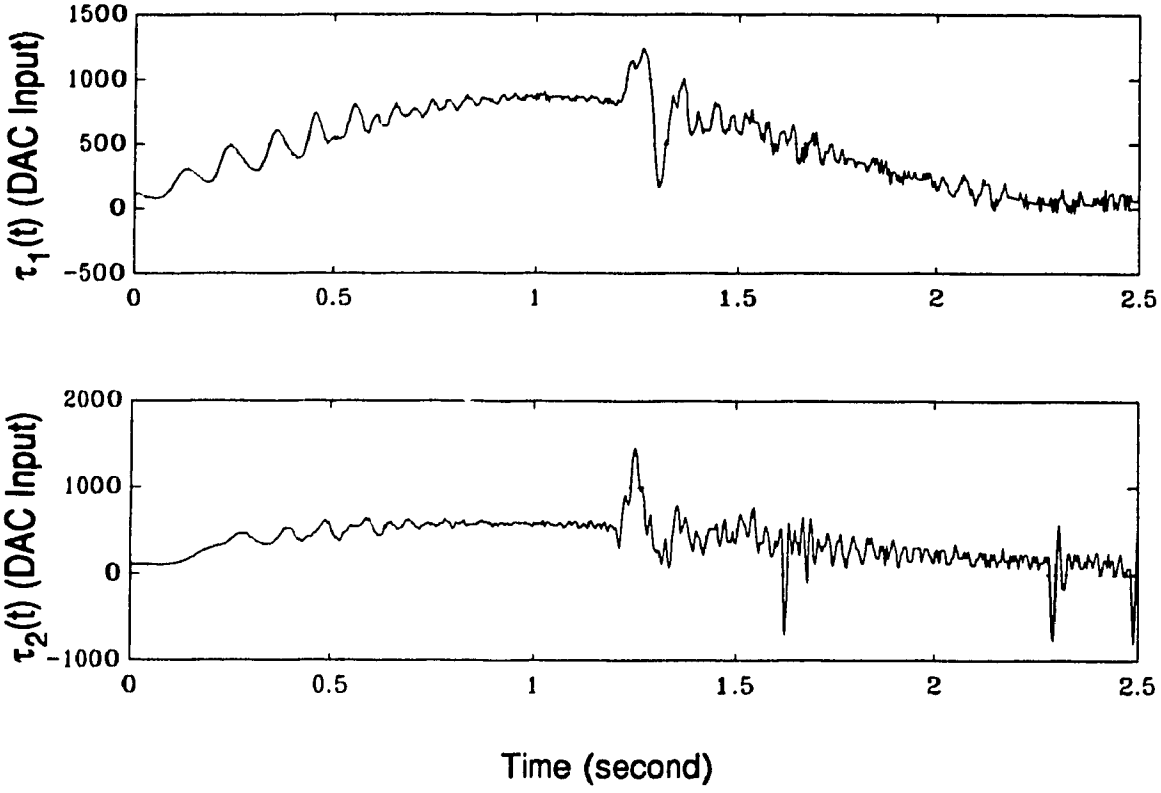


Figure 5.6. Torques, Adaptive Control, Experiment 1.

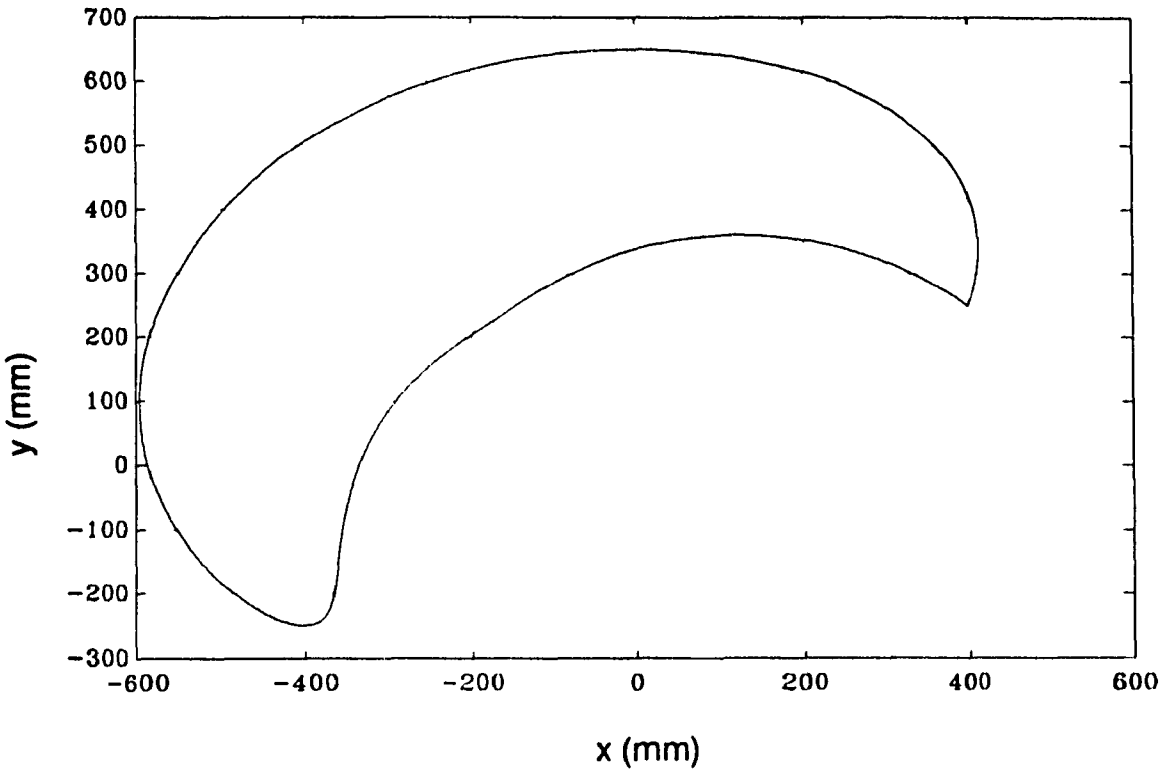


Figure 5.7. Desired Path of Joints 1 and 2 for Experiment 2 Shown in Cartesian Space.

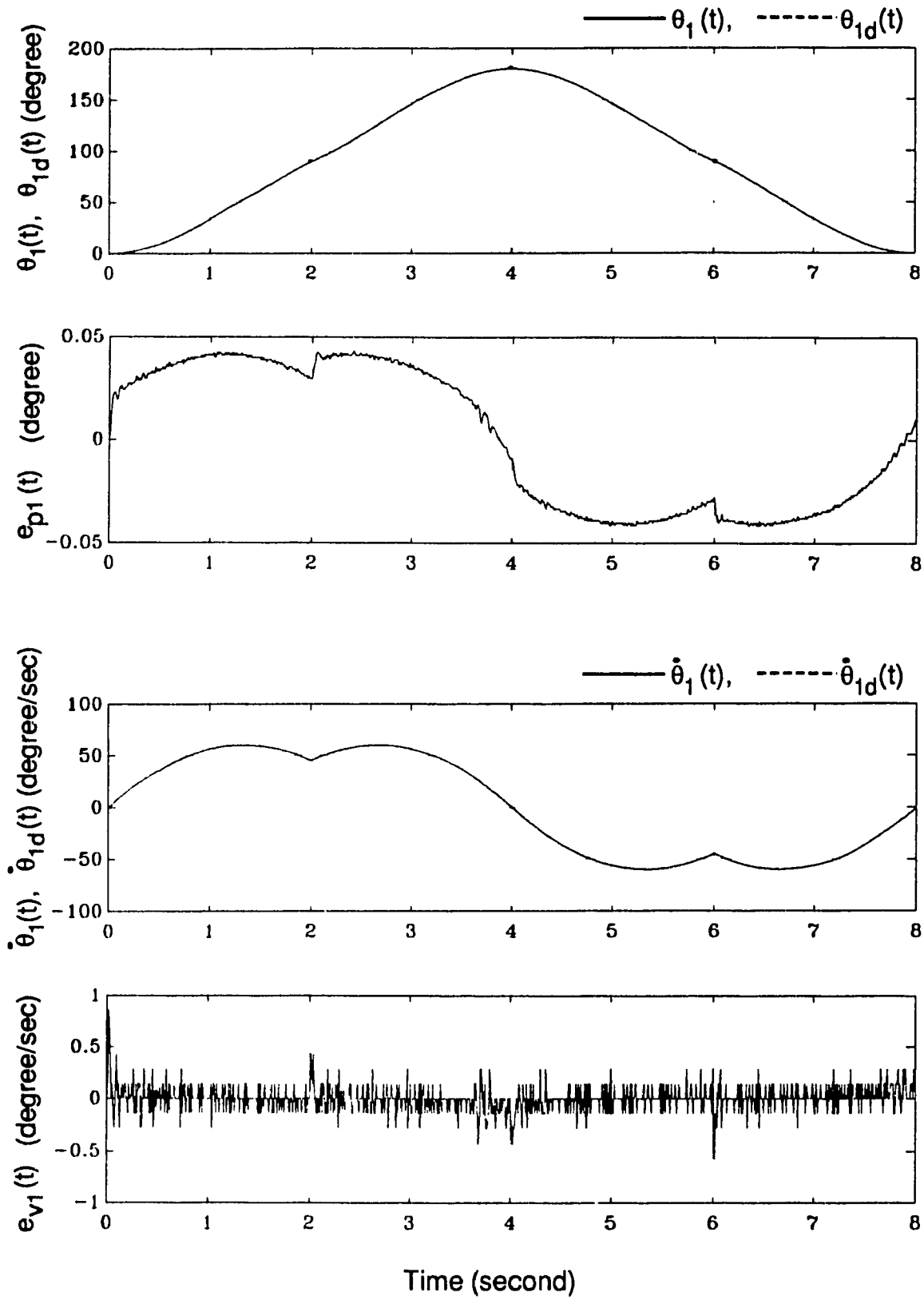


Figure 5.8. Joint 1 Tracking Errors, PD Control, Experiment 2.

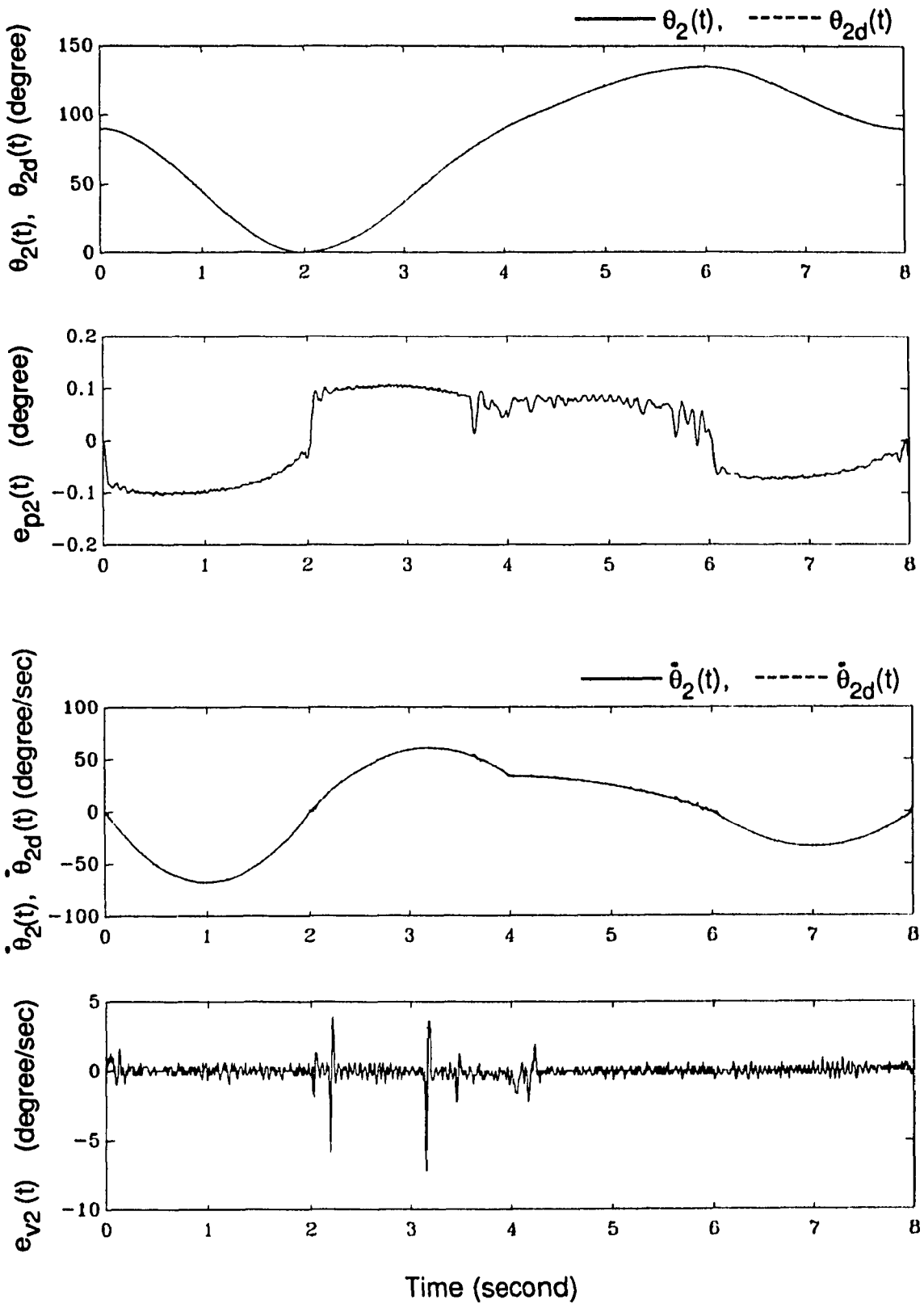


Figure 5.9. Joint 2 Tracking Errors, PD Control, Experiment 2.

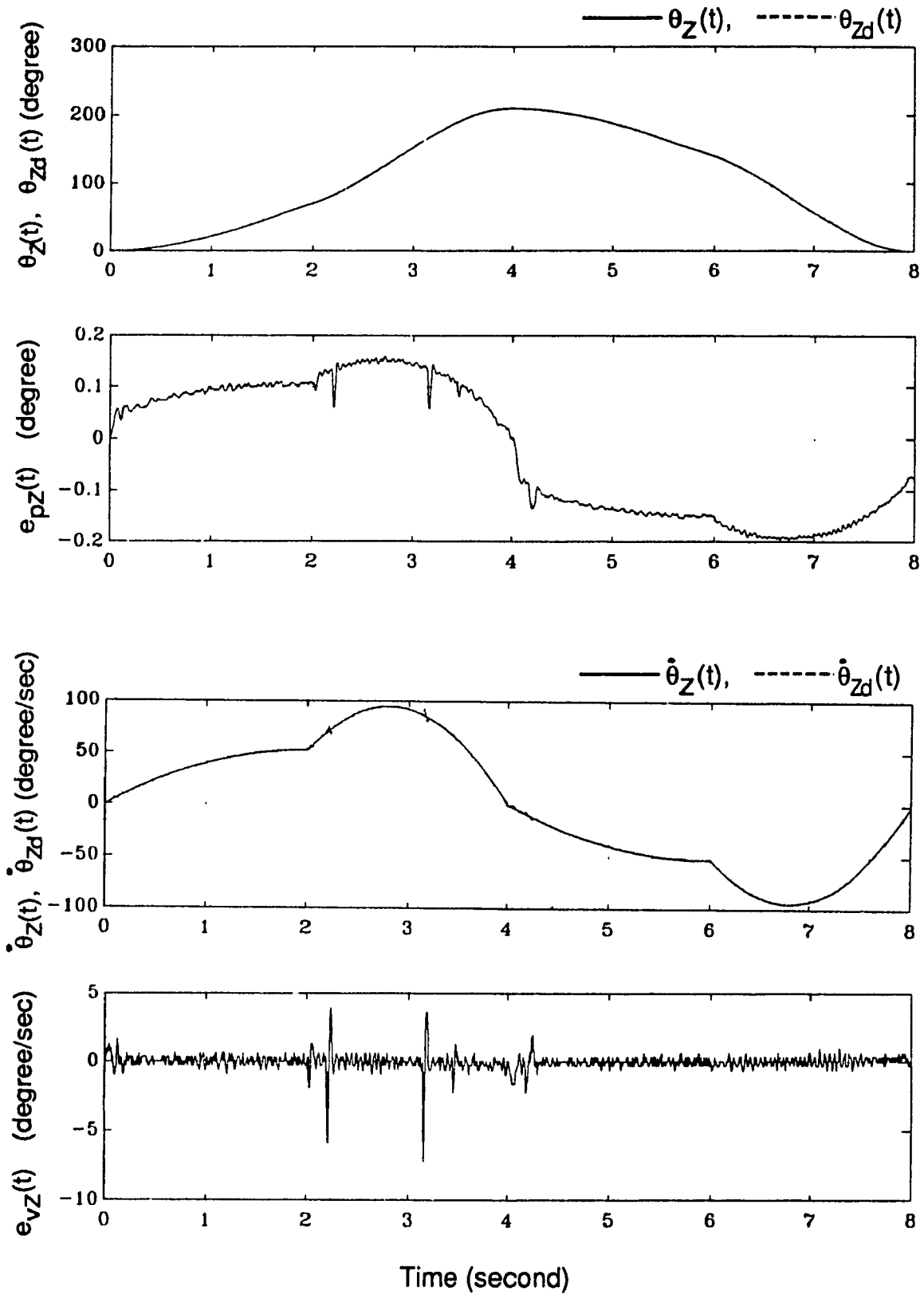


Figure 5.10. Joint Z Tracking Errors, PD Control, Experiment 2.

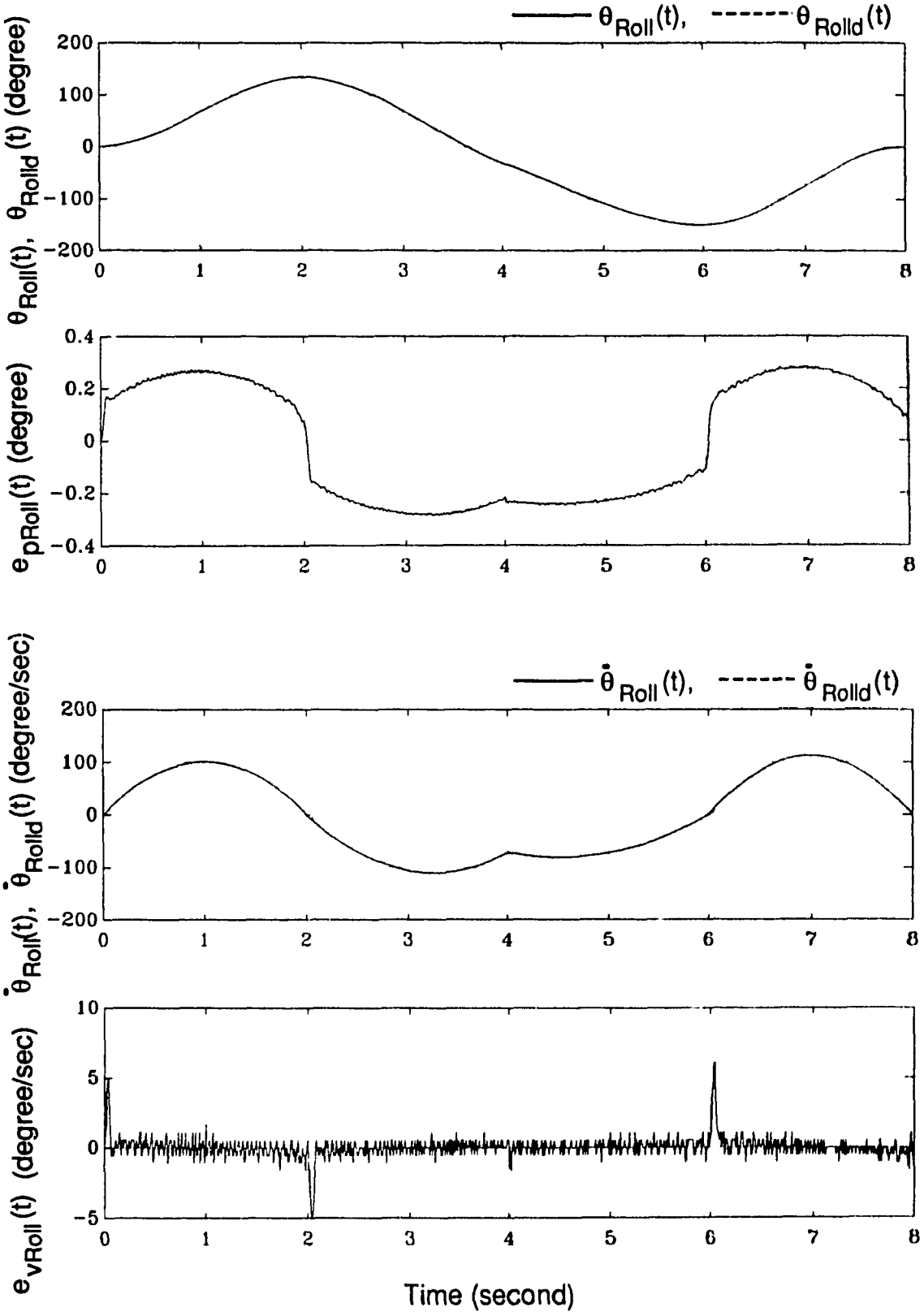


Figure 5.11. Joint Roll Tracking Errors, PD Control, Experiment 2.

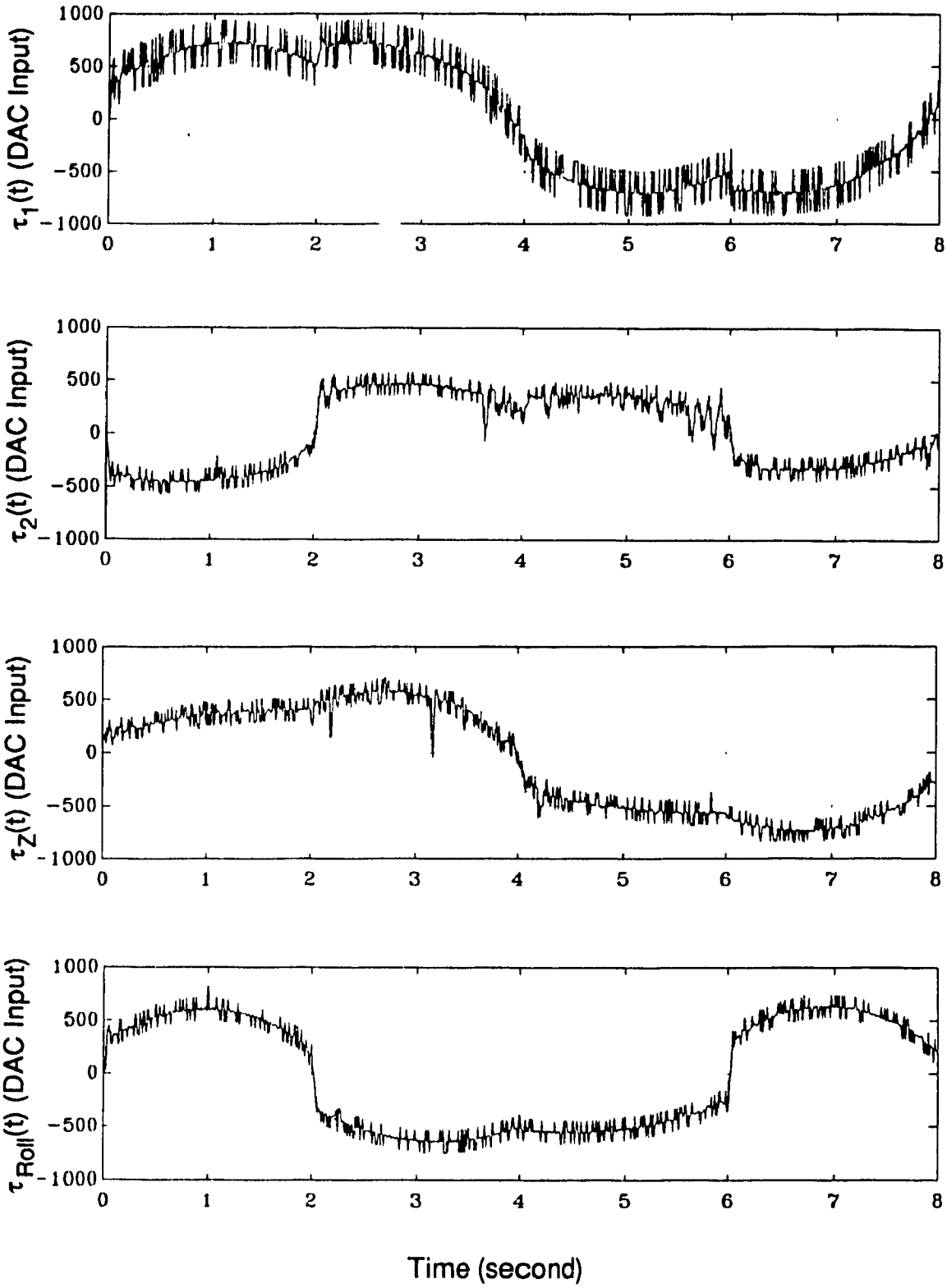


Figure 5.12. Torques, PD Control, Experiment 2.

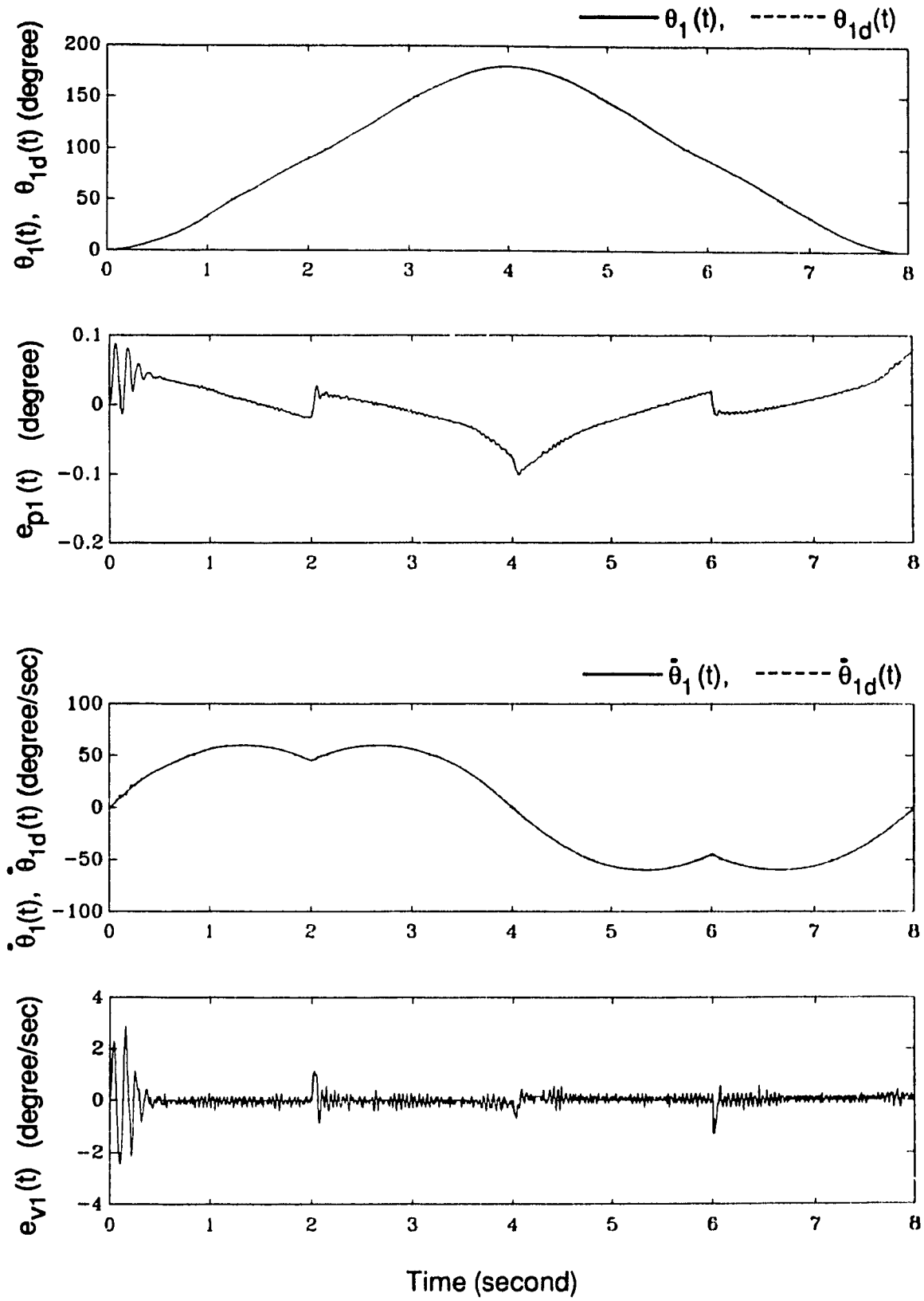


Figure 5.13. Joint 1 Tracking Errors, Adaptive Control, Experiment 2.

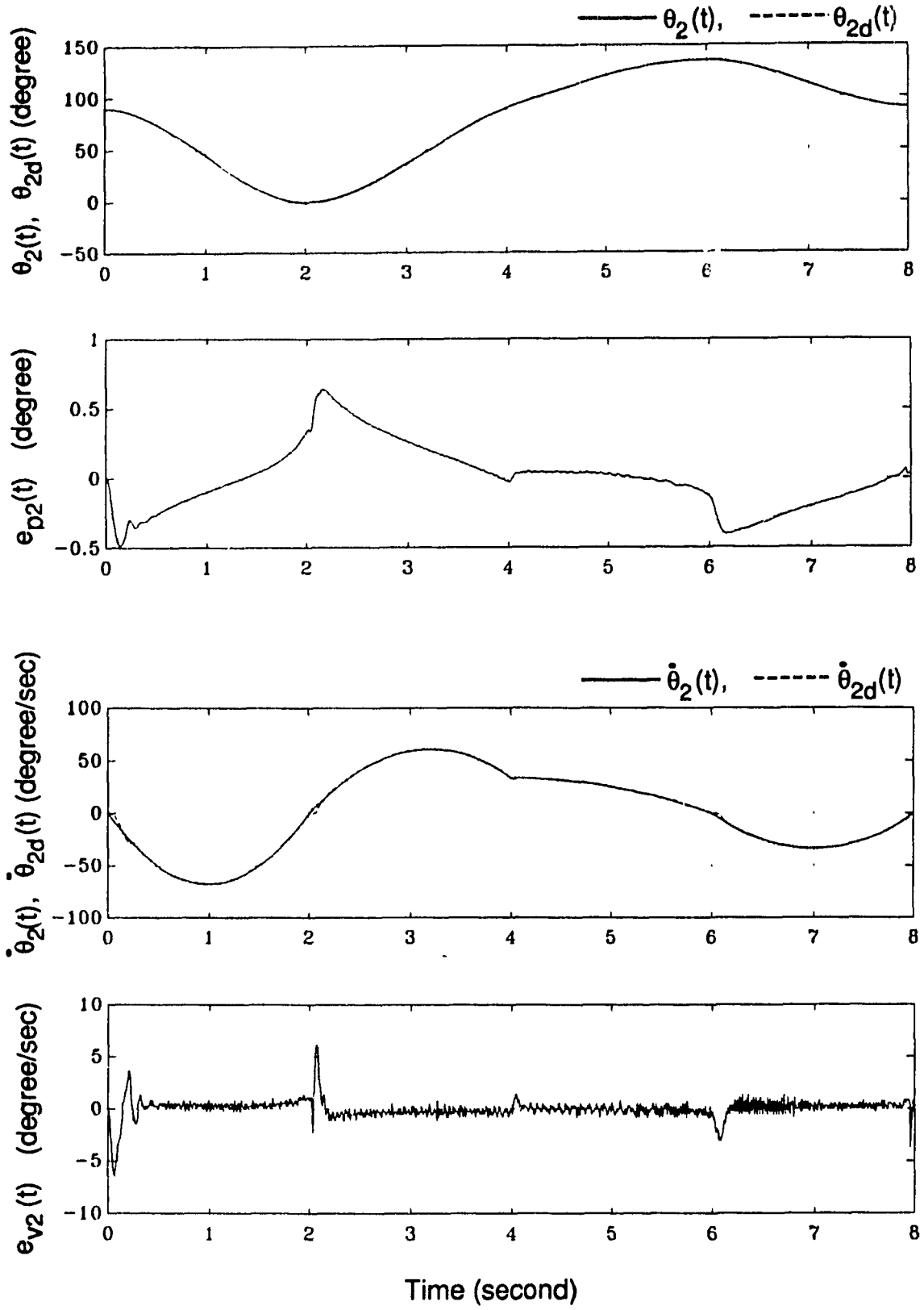


Figure 5.14. Joint 2 Tracking Errors, Adaptive Control, Experiment 2.

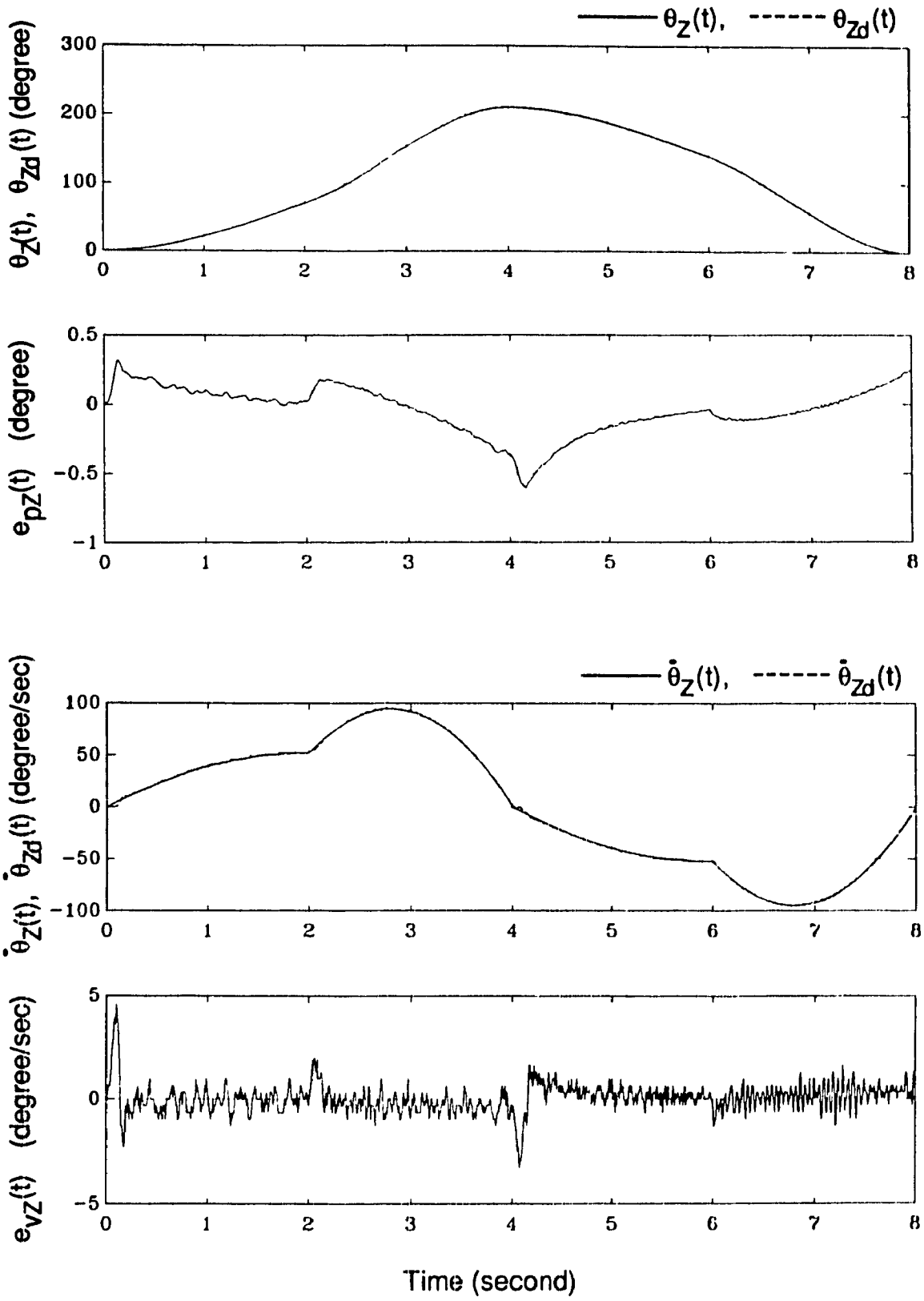


Figure 5.15. Joint Z Tracking Errors, Adaptive Control, Experiment 2.

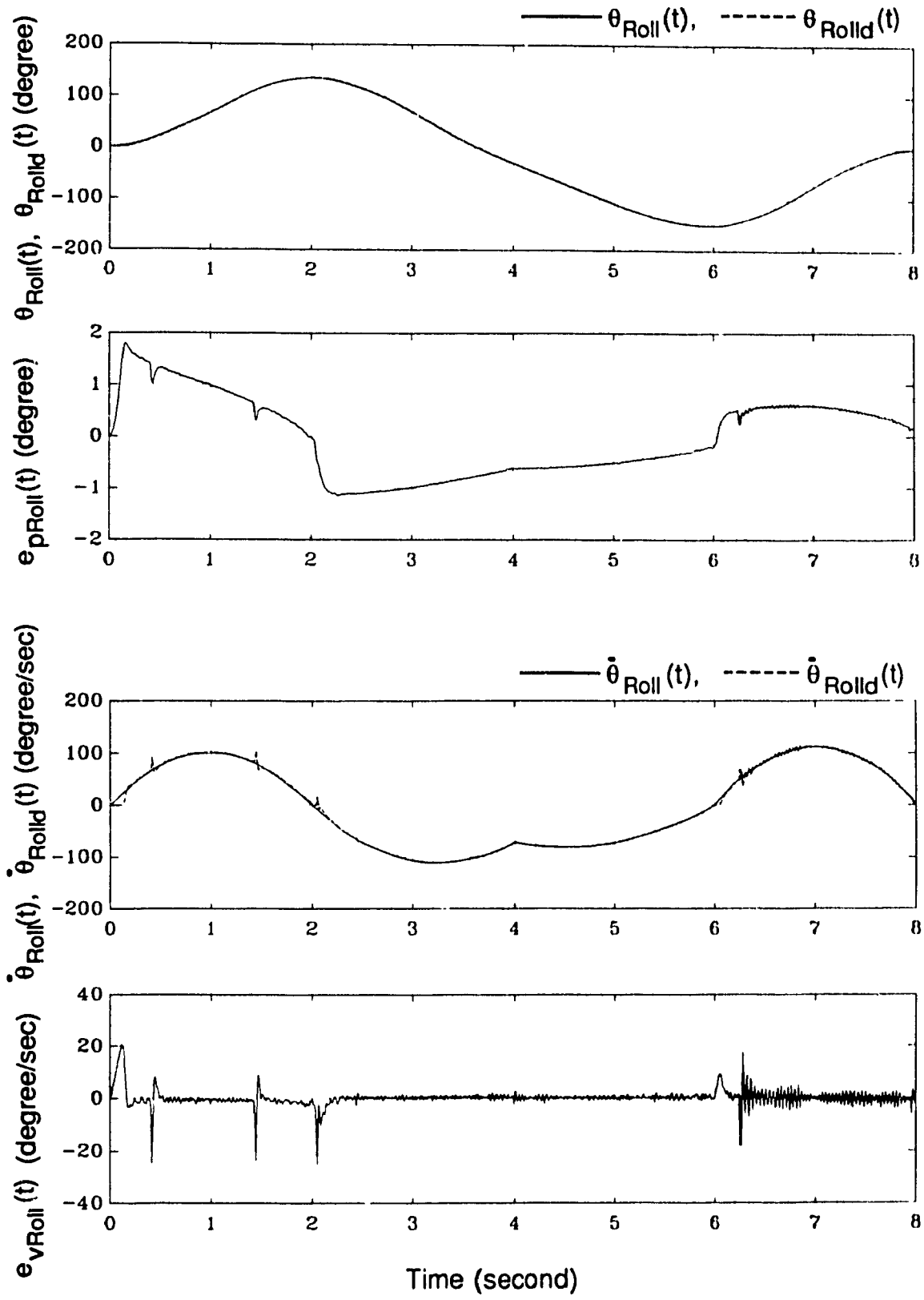


Figure 5.16. Joint Roll Tracking Errors, Adaptive Control, Experiment 2.

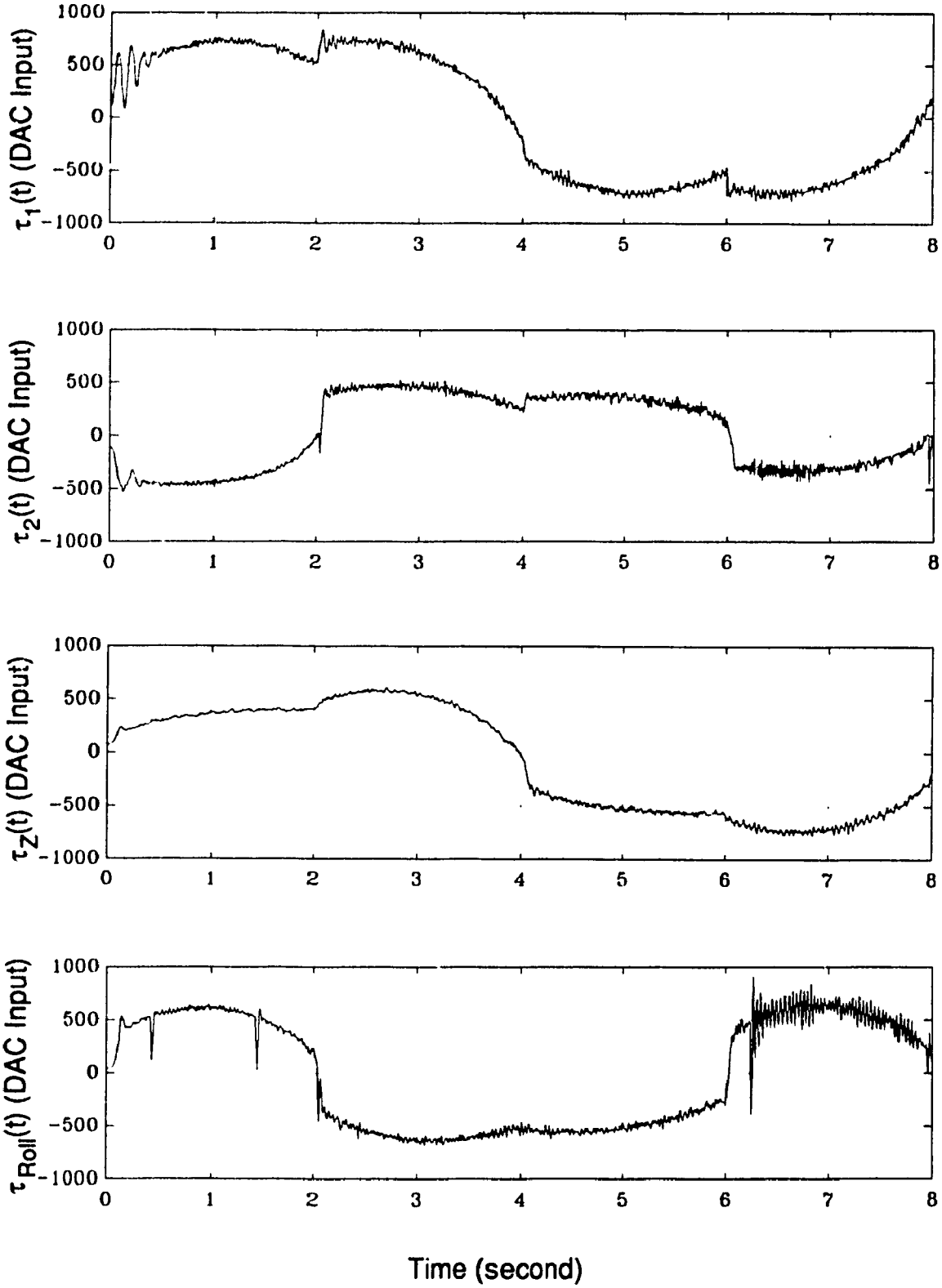


Figure 5.17. Torques, Adaptive Control, Experiment 2.

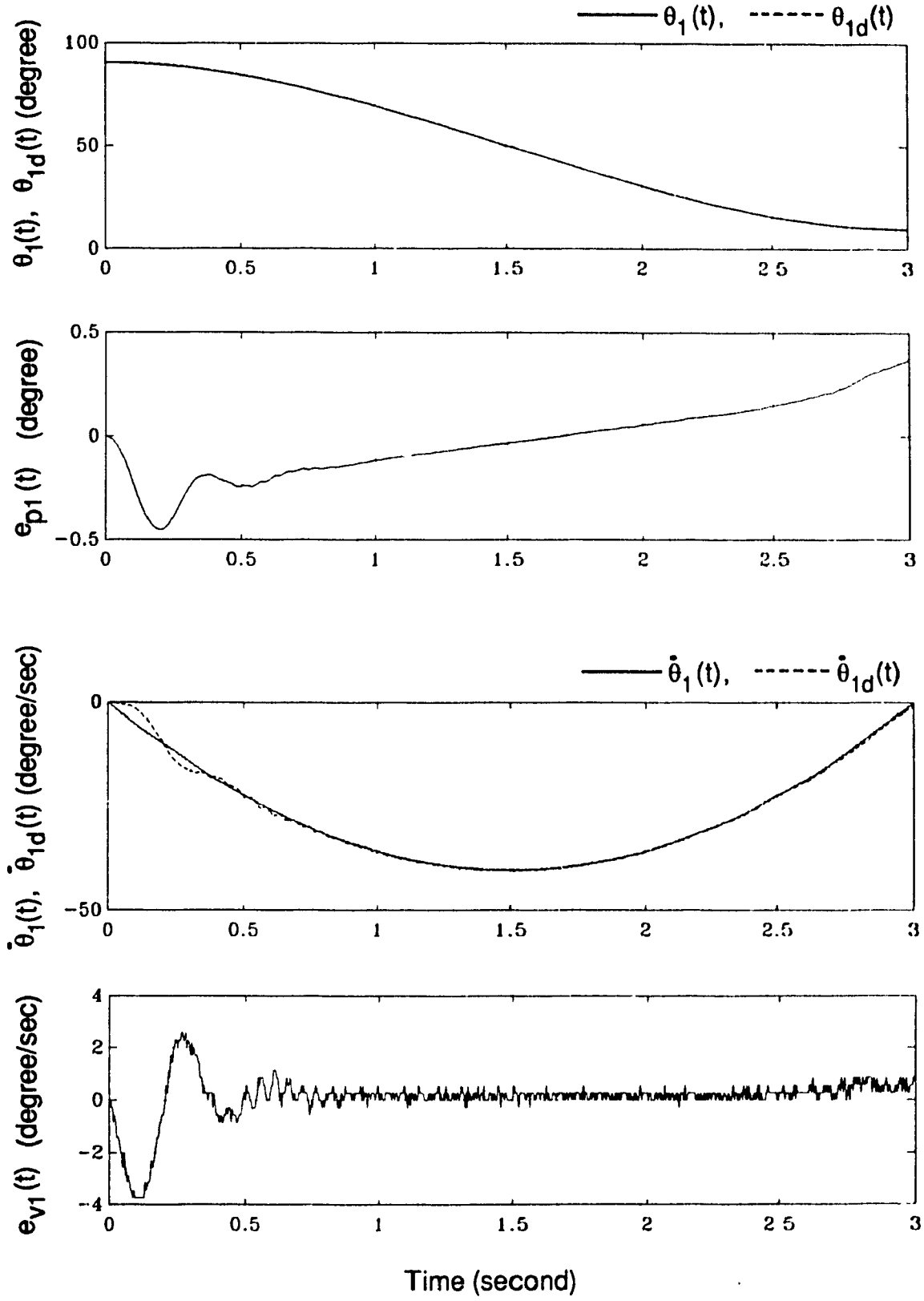


Figure 5.18. Joint 1 Tracking Errors, Adaptive Control, Experiment 3, Three Sec. Move.

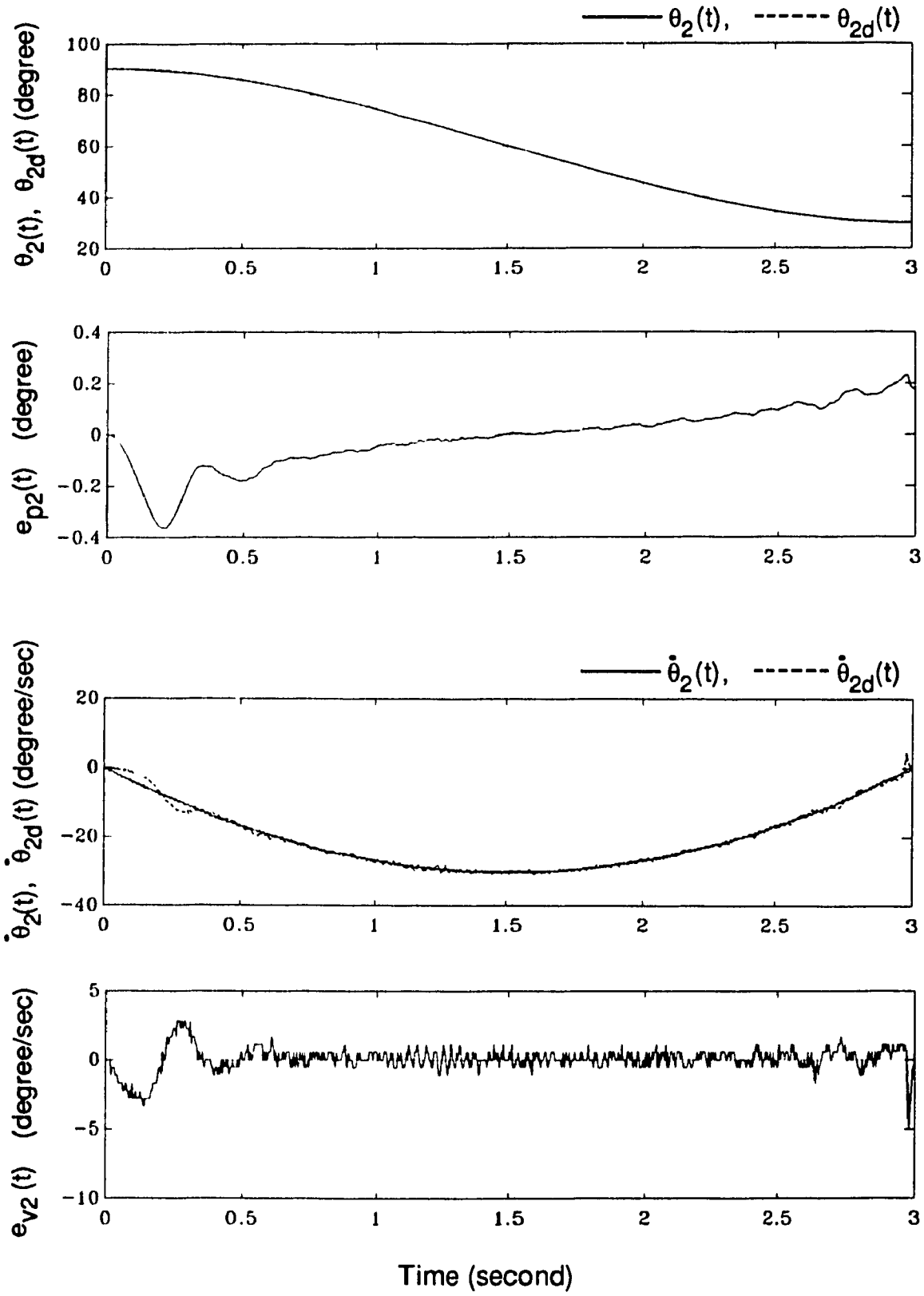


Figure 5.19. Joint 2 Tracking Errors, Adaptive Control, Experiment 3, Three Sec. Move.

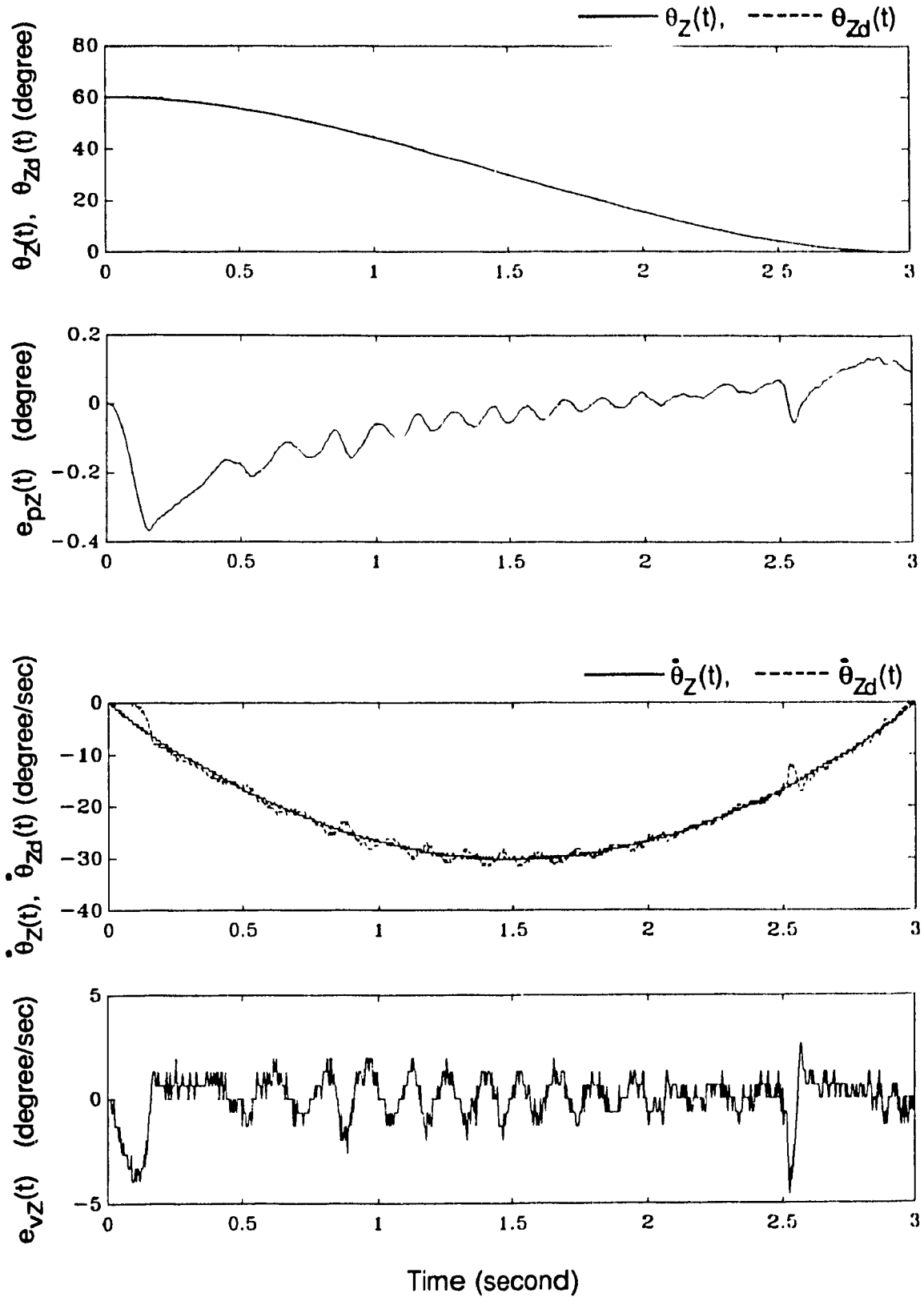


Figure 5.20. Joint Z Tracking Errors, Adaptive Control, Experiment 3, Three Sec. Move.

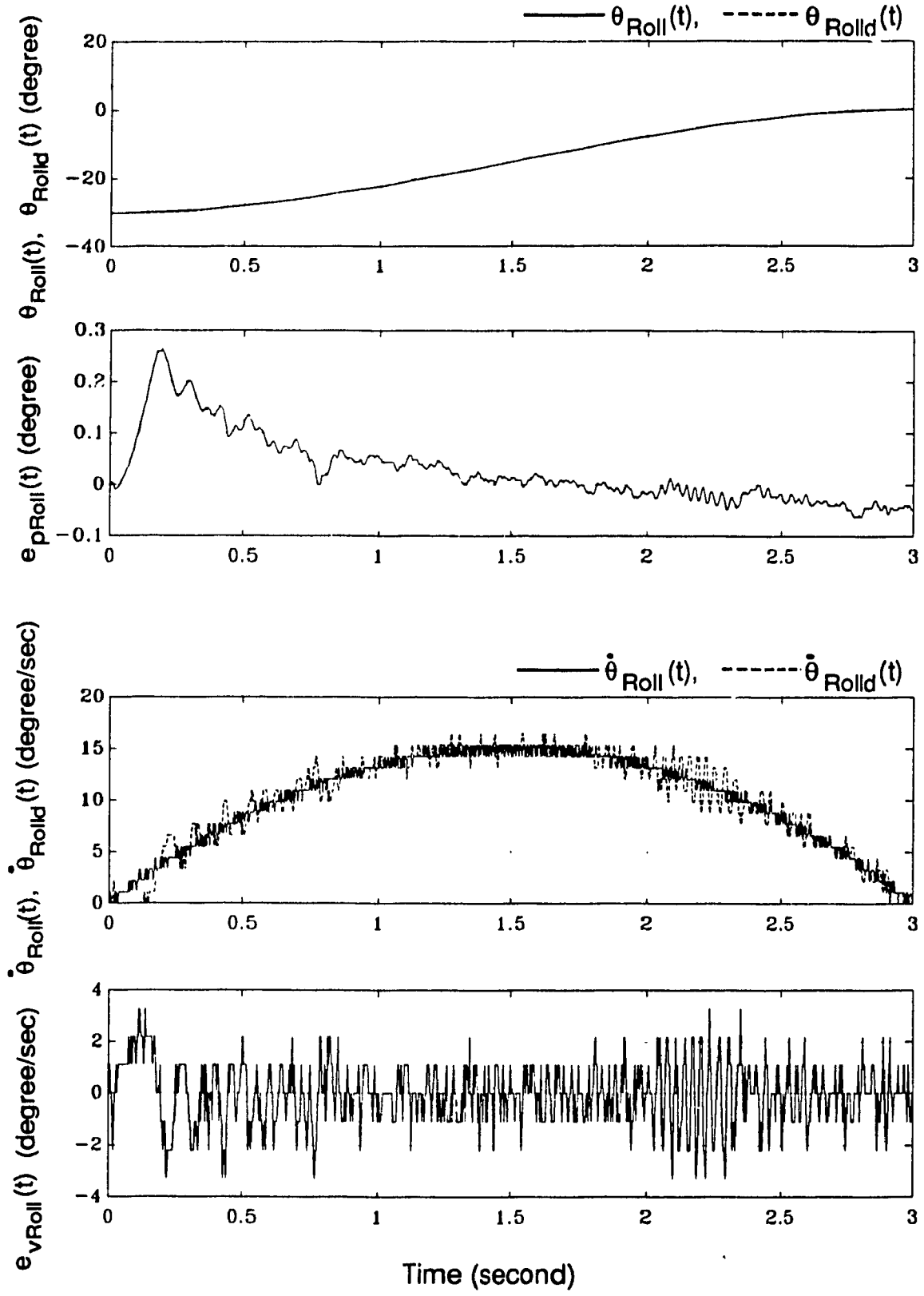


Figure 5.21. Joint Roll Tracking Errors, Adaptive Control, Experiment 3, Three Sec. Move.

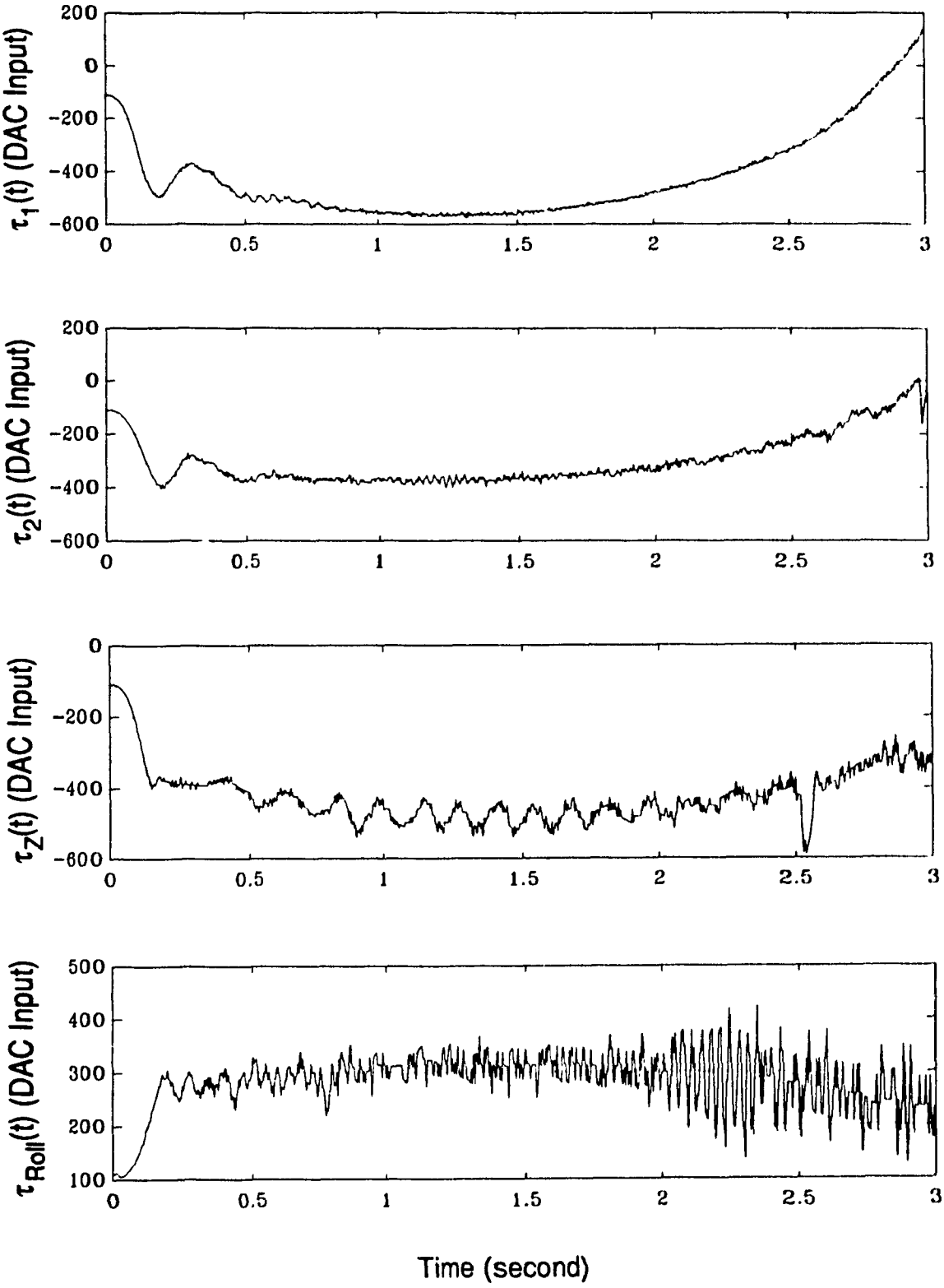


Figure 5.22. Torques, Adaptive Control, Experiment 3, Three Sec. Move.

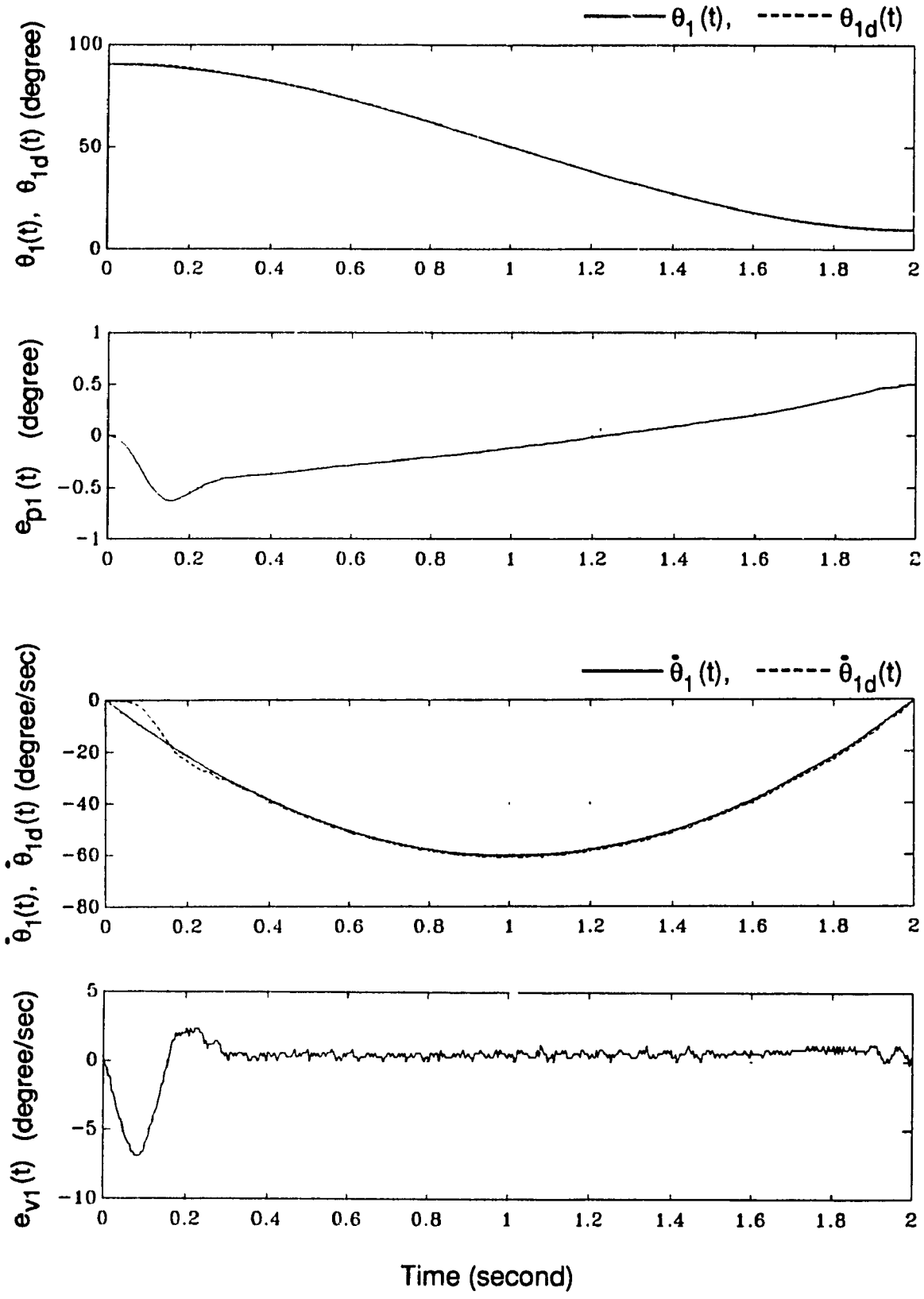


Figure 5.23. Joint 1 Tracking Errors, Adaptive Control, Experiment 3, Two Sec. Move.

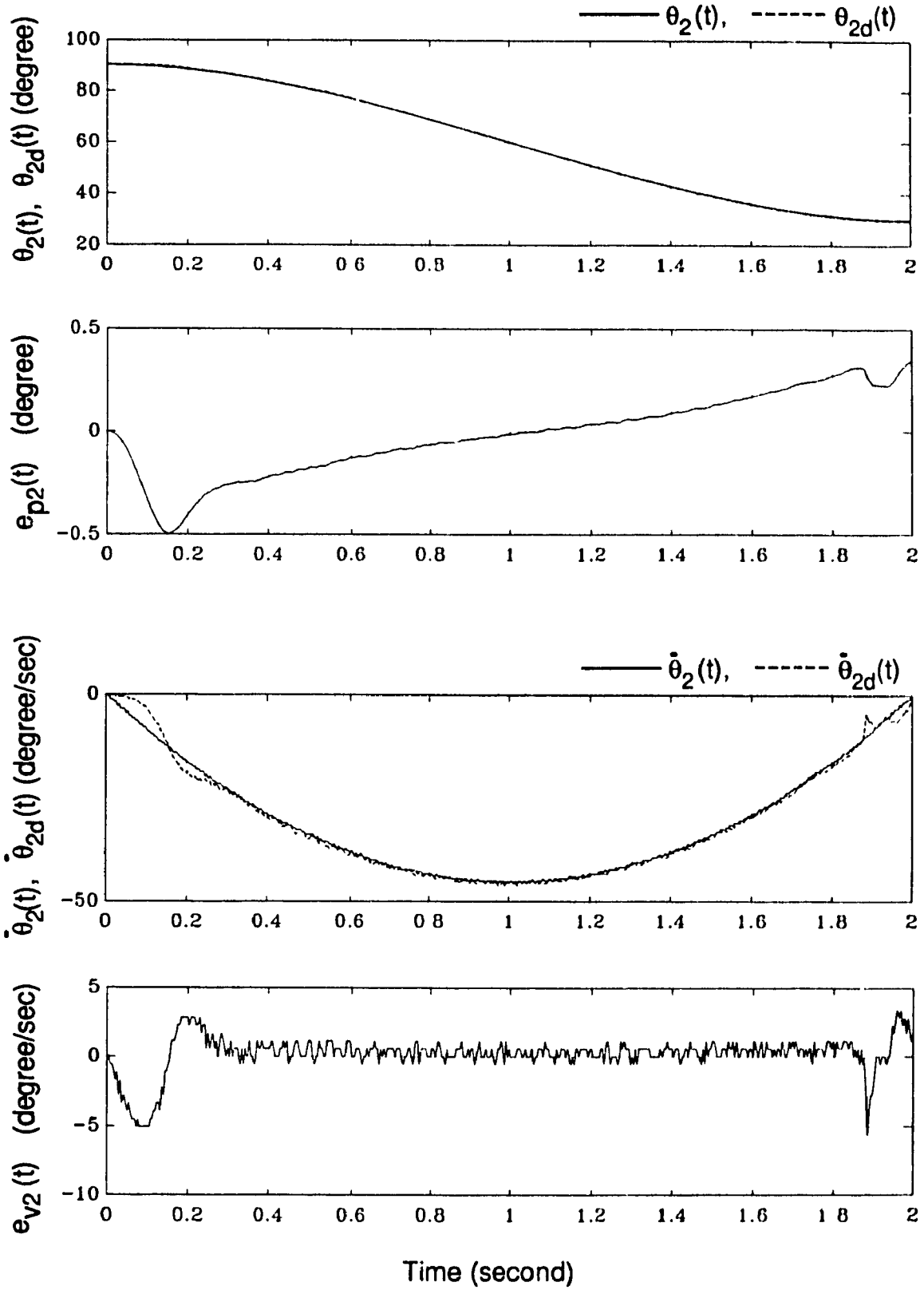


Figure 5.24. Joint 2 Tracking Errors, Adaptive Control, Experiment 3, Two Sec. Move.

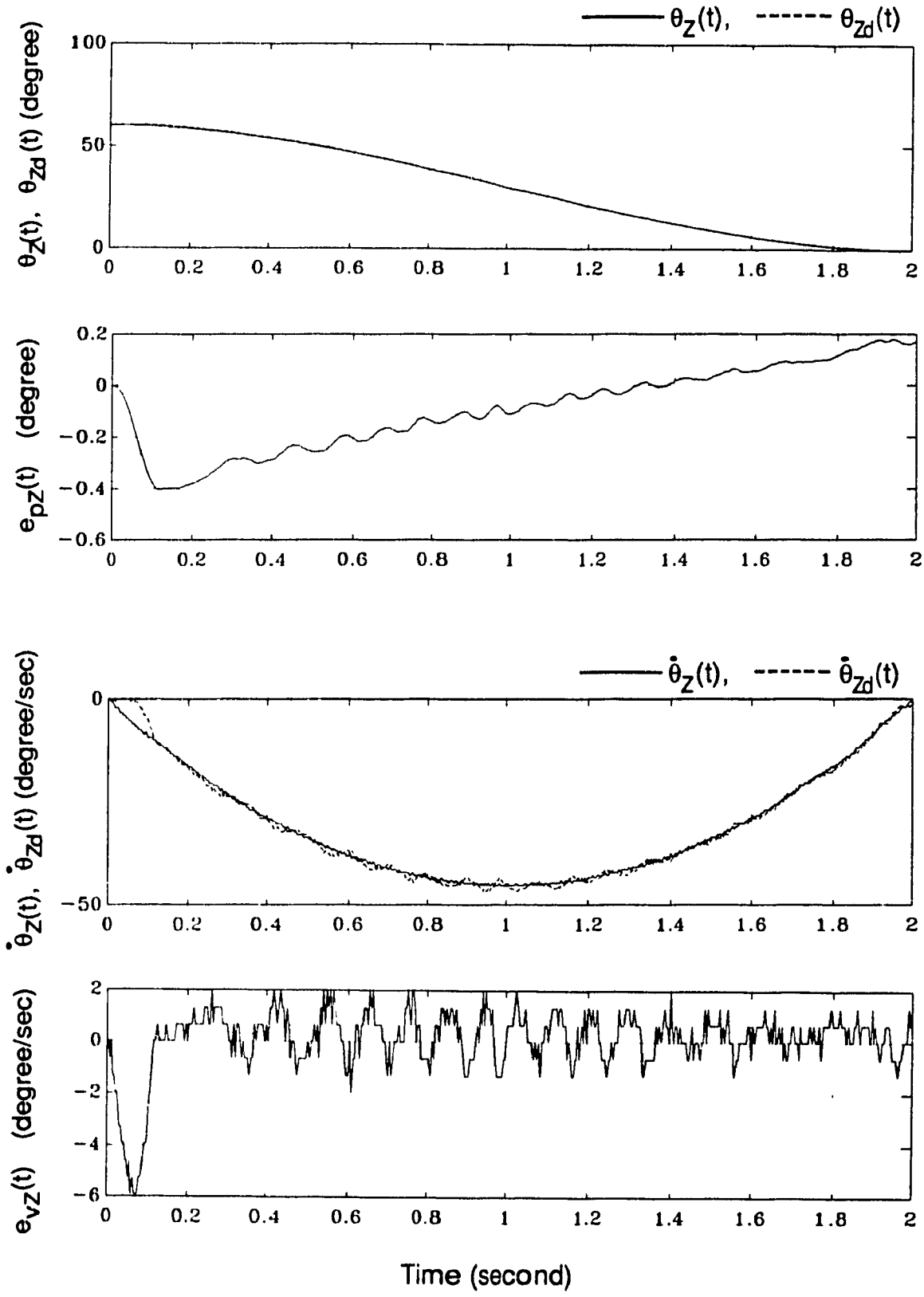


Figure 5.25. Joint Z Tracking Errors, Adaptive Control, Experiment 3, Two Sec. Move.

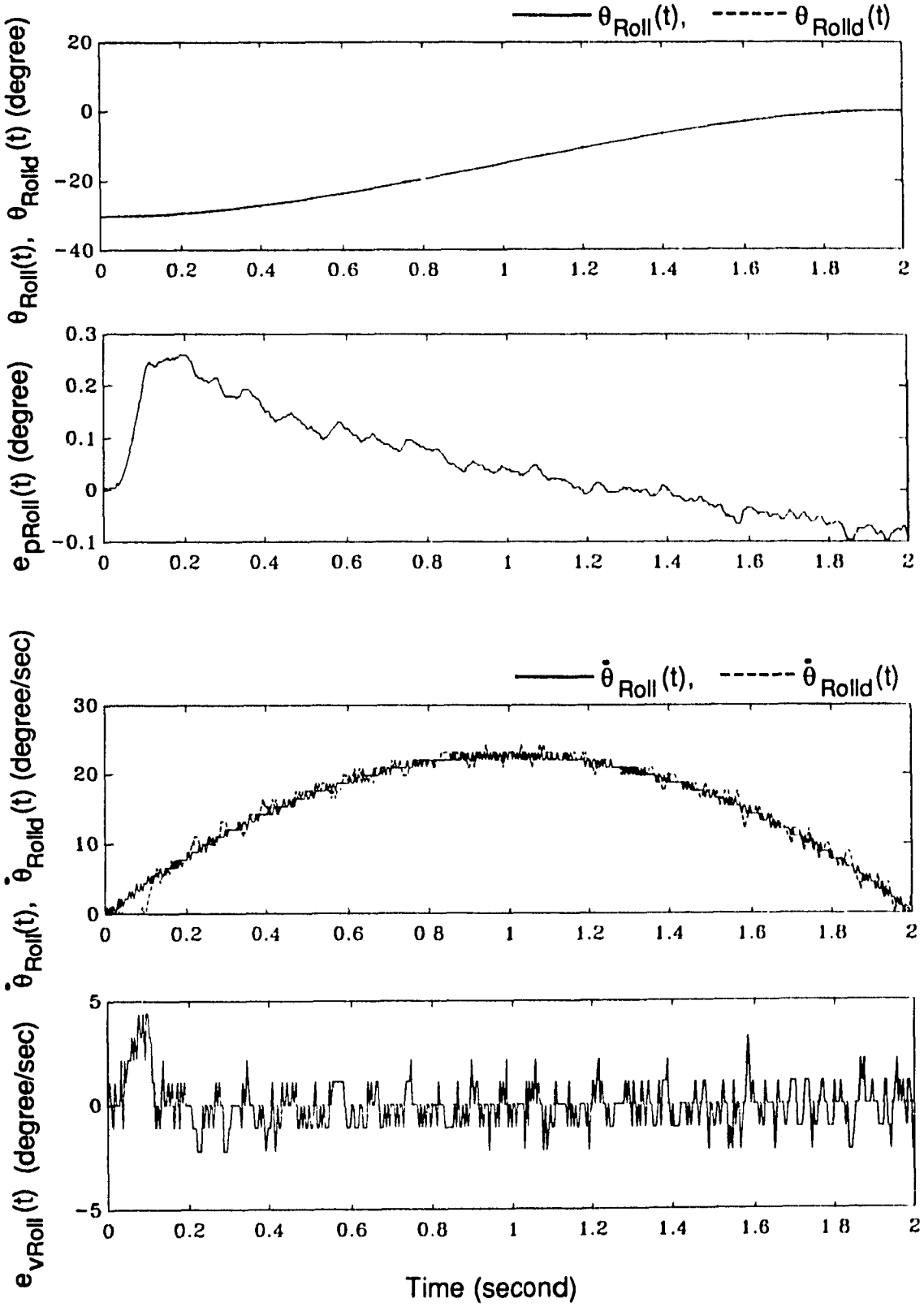


Figure 5.26. Joint Roll Tracking Errors, Adaptive Control, Experiment 3, Two Sec. Move.

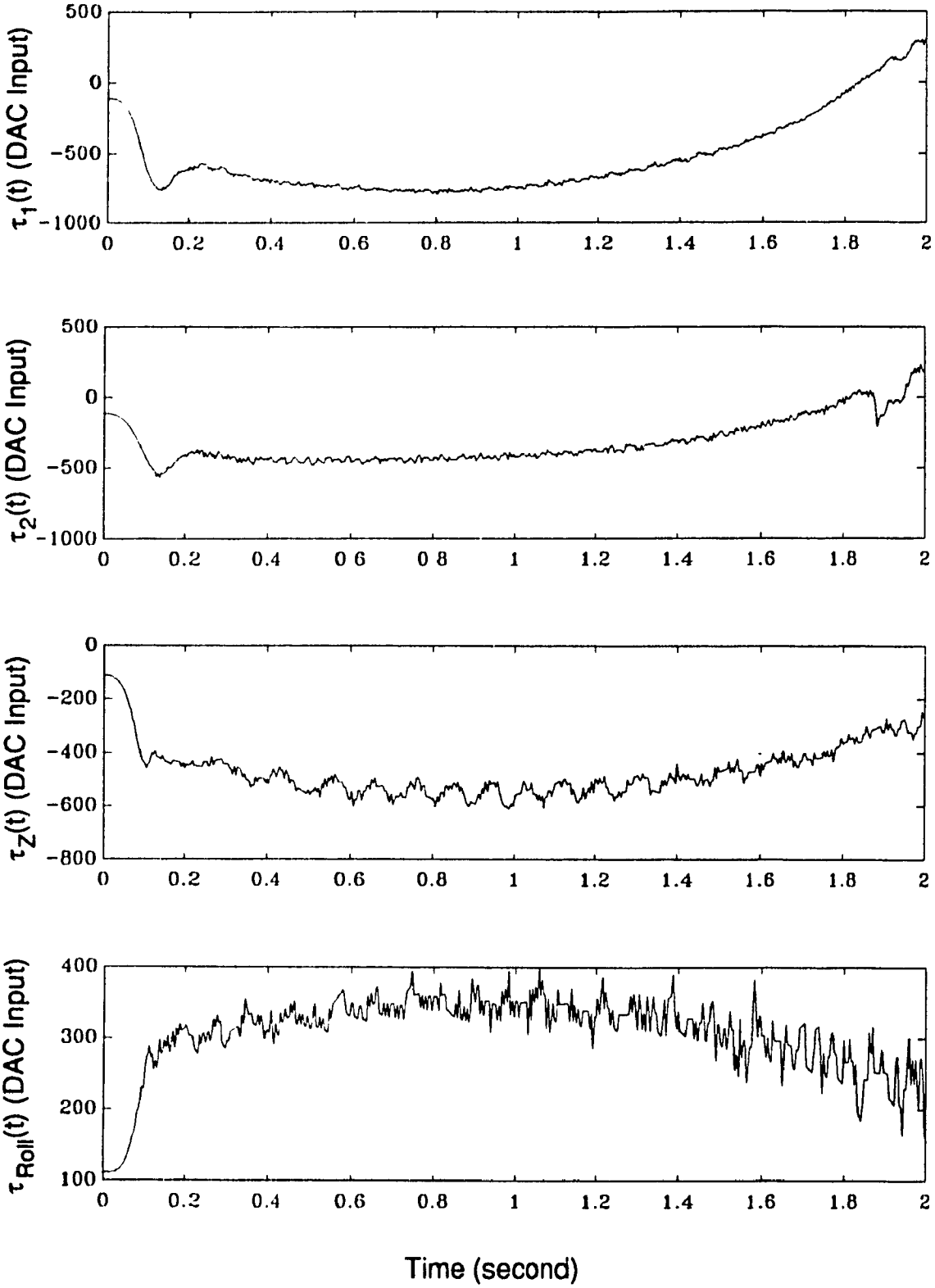


Figure 5.27. Torques, Adaptive Control, Experiment 3, Two Sec. Move.

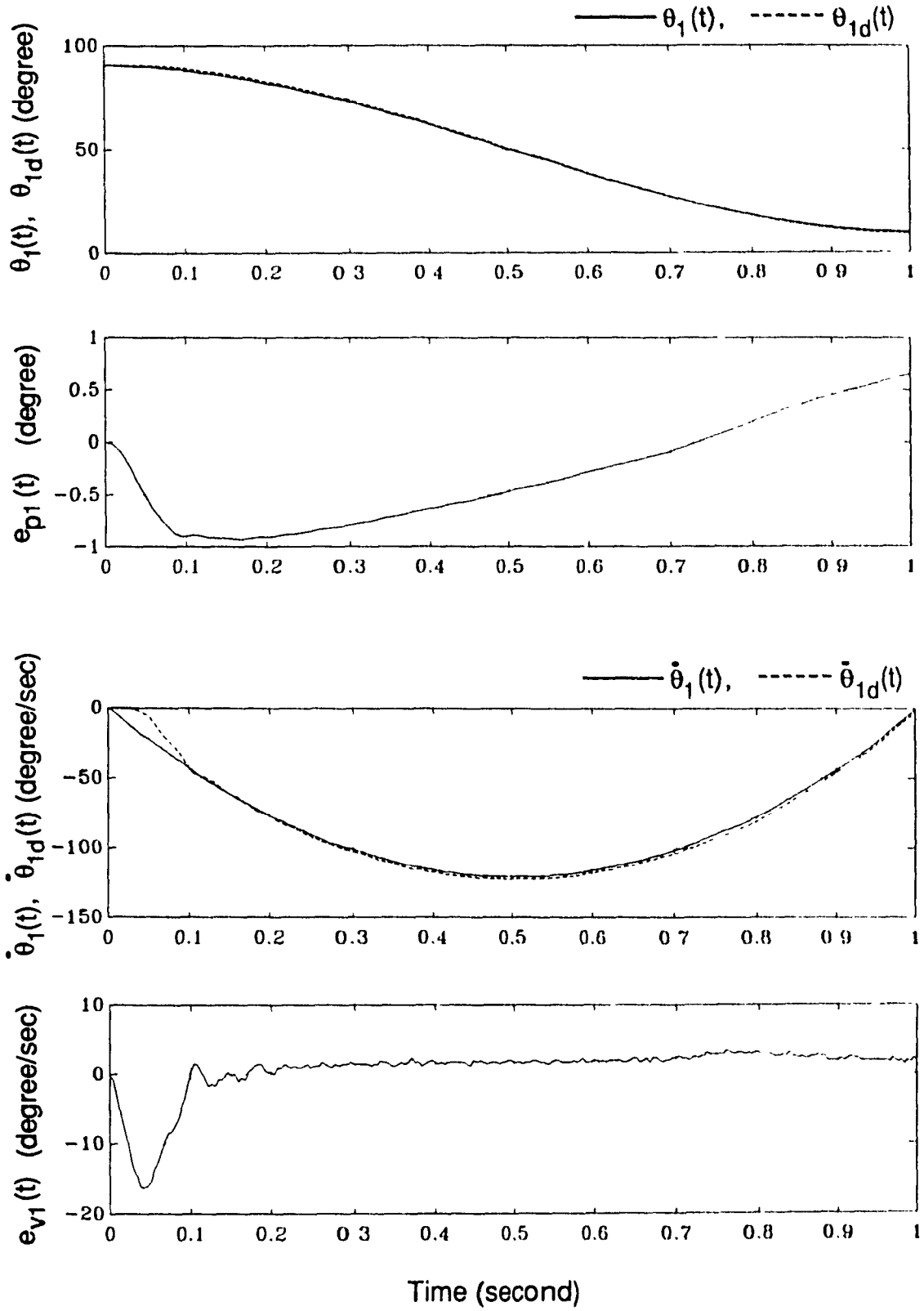


Figure 5.28. Joint 1 Tracking Errors, Adaptive Control, Experiment 3, One Sec. Move.

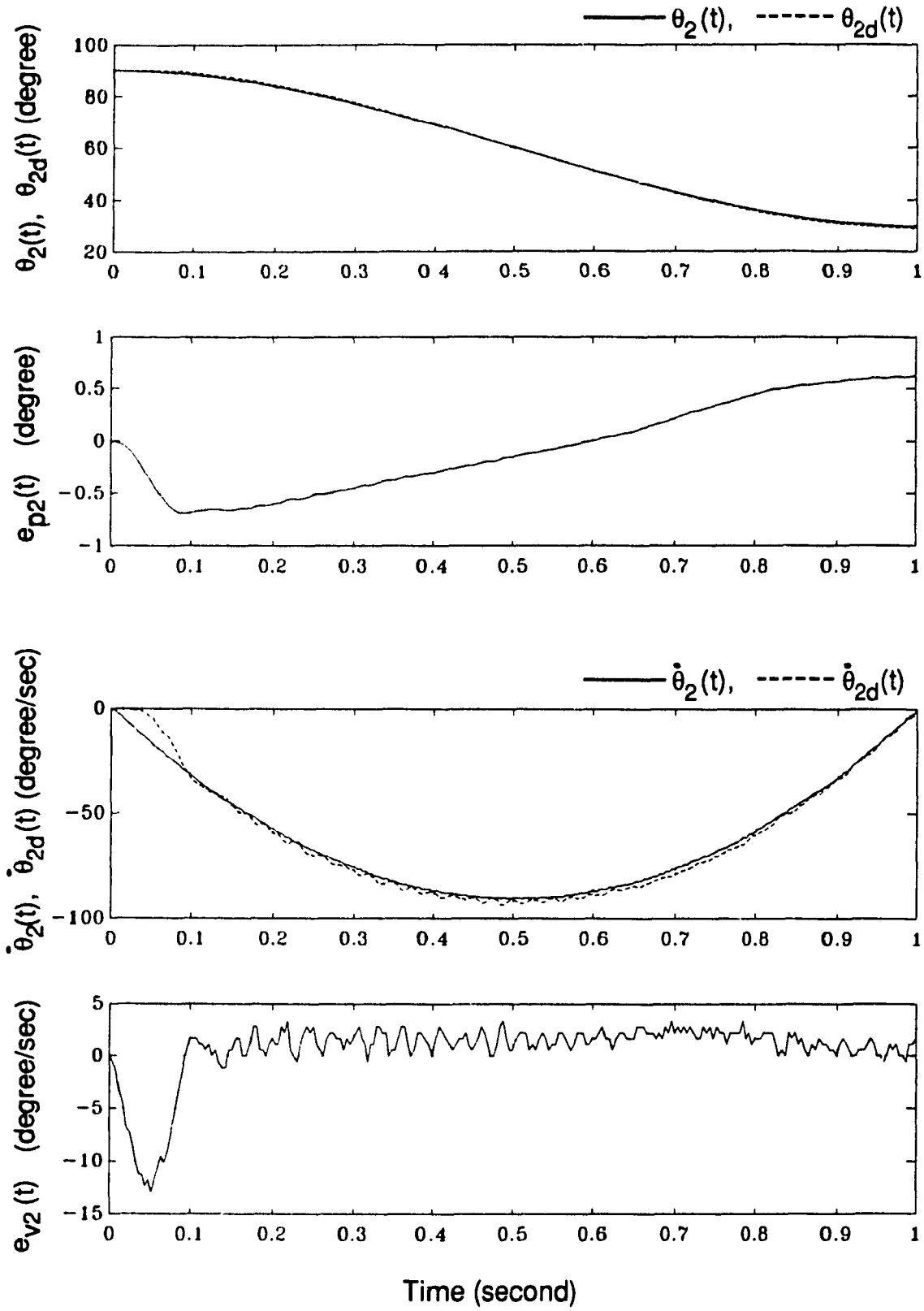


Figure 5.29. Joint 2 Tracking Errors, Adaptive Control, Experiment 3, One Sec. Move.

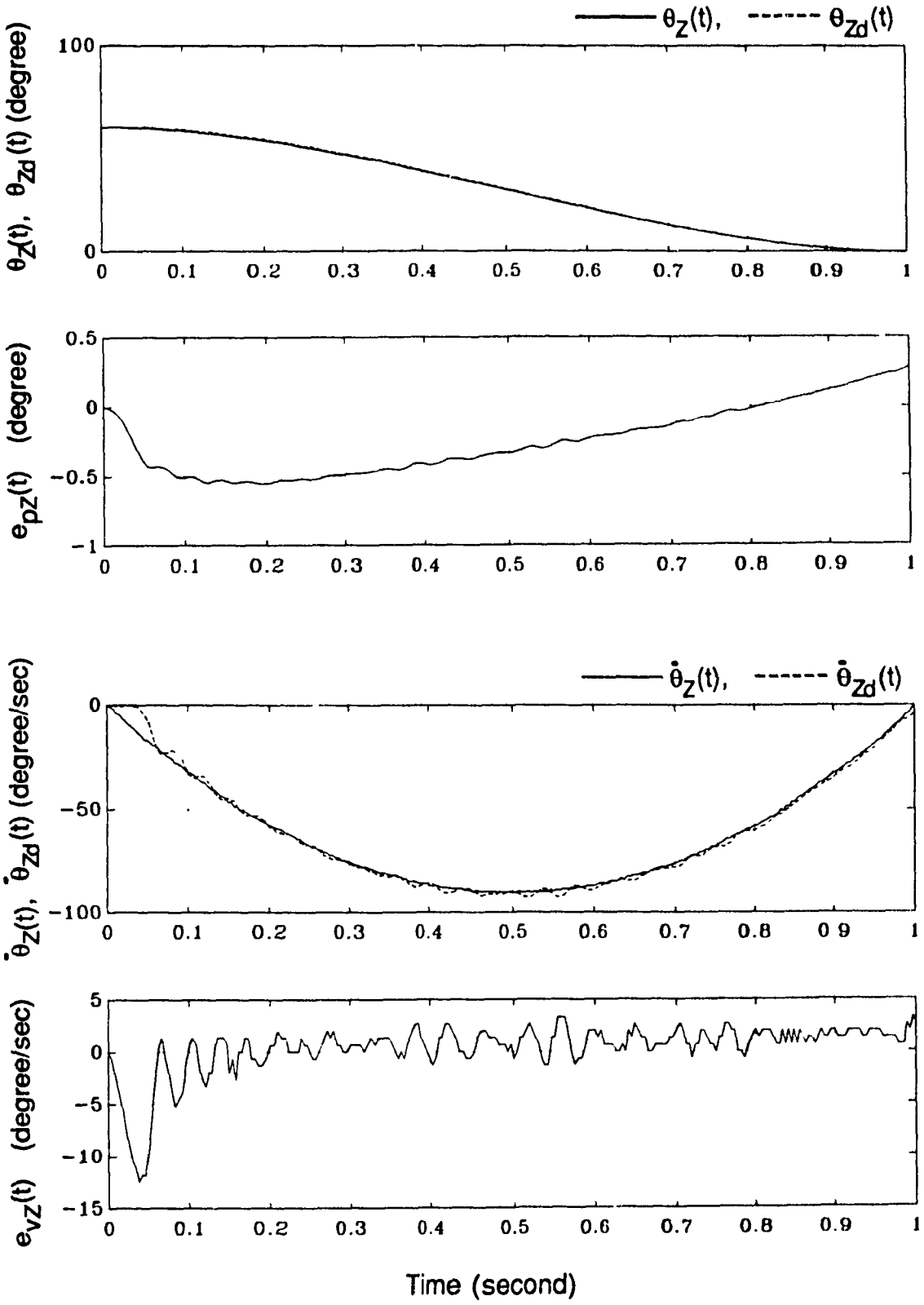


Figure 5.30. Joint Z Tracking Errors, Adaptive Control, Experiment 3, One Sec. Move.

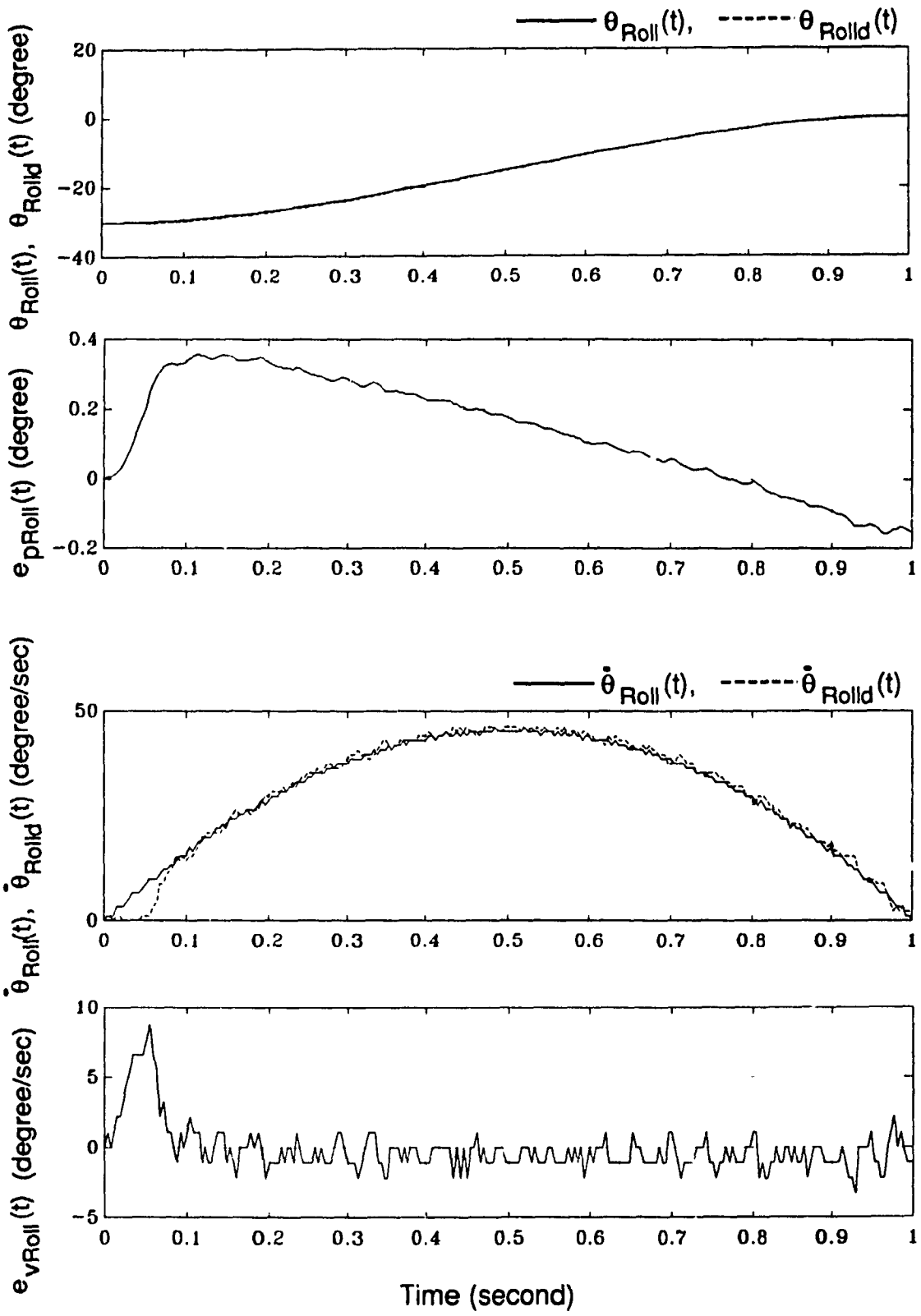


Figure 5.31. Joint Roll Tracking Errors, Adaptive Control, Experiment 3, One Sec. Move.

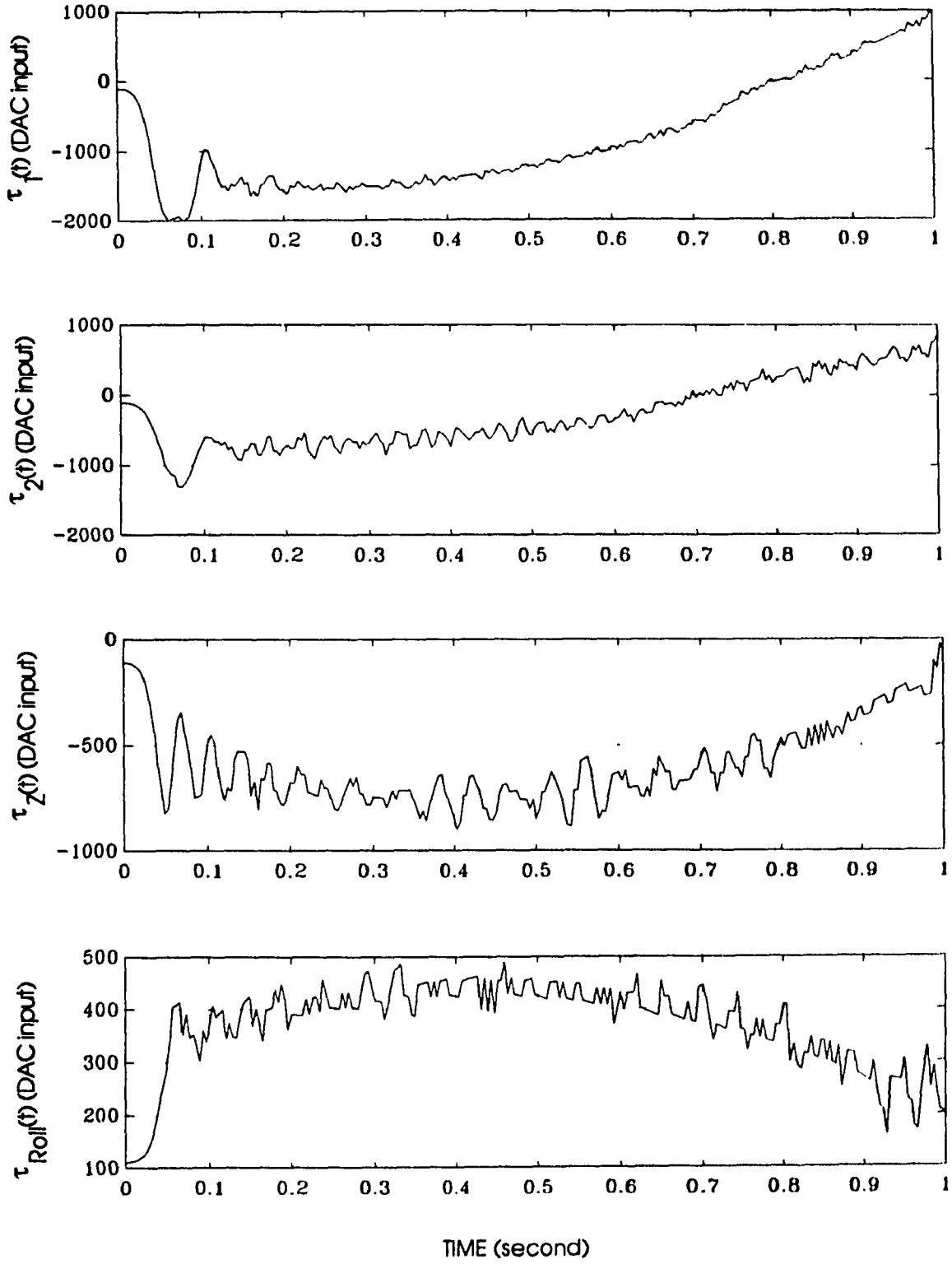


Figure 5.32. Torques, Adaptive Control, Experiment 3, One Sec. Move.

CHAPTER 6

CONCLUSIONS

An architecture for implementing advanced robot control strategies has been presented. The work that was done is a contribution to the effort that is being made to enhance the operation of conventional industrial robotic systems. The primary goal of the work has been to create an inexpensive (relative to the cost of the robot) environment for implementing robot control systems which is easy-to-develop and maintain and applicable to a variety of robotic systems. The resulting ARC system has been designed as a replacement for manufacturer-supplied controllers to provide greater flexibility. In the design process, we have considered the following aspects of the controller:

1) the design of the hardware component;
2) the design of the software component; and
3) the application of the above-mentioned components to the IBM 7545 Manufacturing System.

From the hardware standpoint, we have created a system possessing sufficient processing power to execute advanced robot control strategies at respectable servo rates. From the software standpoint, we have constructed the building blocks to facilitate the development and implementation of these control algorithms.

In the system design process, we have recognized the fact that there exist common low-level tasks associated with all robot control algorithms, such as robot state acquisition, torque verification, joint overrun checking, and sample period timing. As a result, the controller has been designed to incorporate a slave processor which executes the procedures associated with these tasks. The slave unloads a significant percentage of the

computational burden and thus allows the implementation of more computationally complex control schemes on the master processor.

Through experimentation, we have demonstrated the effectiveness of the ARC to control the 7545 manipulator. We have shown that the ARC is capable of replacing the IBM-supplied controller (MTCB) and is capable of executing both simple and sophisticated control and path planning strategies at respectable servo rates, e.g., 1000 Hz for PD control and 250 Hz for decentralized adaptive control. The experimental results given in Chapter 5 show that the ARC delivers good performance for a variety of control strategies for various types of trajectories.

In its present form, the ARC is well suited for use in research environments for the implementation and evaluation of new control strategies. It was for this purpose that the project was undertaken, and the above conclusions show that this intention has been fully realized using a typical industrial robot.

REFERENCES

[1] D. G. Bihn and T. C. Steve Hsia, "Universal Six-Joint Robot Controller," *IEEE Control Systems Magazine*, vol. 8, no. 1, pp. 31-36, 1988.

[2] M. Erlic and W. -S. Lu, "Computer Independent Driver-Sensor Interface (DSI) for Closed Loop Control of a Mechanical Manipulator," in *Proc. of IEEE Canadian Conf. on Elec. & Comp. Eng.*, Montréal, Québec, 1989, pp. 498-501.

[3] D. B. Stewart, D. E. Schmitz, and P.K. Khosla, "CHIMERA II: A Real-Time Multiprocessing Environment for Sensor-Based Robot Control," in *Proc. of IEEE International Symposium on Intelligent Control,* Albany, NewYork, 1989, pp. 265-271.

[4] V. Hayward, L Daneshmend, and S Hayati, "An Overview of KALI: A System to Program and Control Cooperative Manipulators," in *Advanced Robotics*, ed. K. J. Waldron, Springer Verlag, New York, 1989.

[5] D. G. Bihn, *A Universal Six Joint Robot Controller.* M.S. Thesis, Department of Electrical and Computer Engineering, University of California, Davis, 1986.

[6] M. Erlic, *A Study in Unconstrained Motion Control of a Robotic Manipulator.* Master's Thesis, Dept. of Electrical and Computer Engineering, University of Victoria, 1989.

[7] *IBM 7545 Manufacturing System Hardware Library,* IBM Corporation, Boca Raton, Florida, 1984.

[8] *Print Motor*, Yaskawa Electric Mfg. Co., Ltd., Fairfield, New Jersey, 1982.

[9] *Minertia Motor for Industrial Robot Drives*, Yaskawa Electric Mfg. Co., Ltd., Fairfield, New Jersey, 1989.

[10] *DC Servomotor Controller Servopack*, Yaskawa Electric Mfg. Co., Ltd., Fairfield, New Jersey, 1985.

[11] IBM Personal System/2 Model 50 and 60 Technical Reference, International Business Machines Corporation, 1987.

[12] *High Performance CMOS Data Book*, Integrated Device Technology, Santa Clara, California, 1988.

[13] *Embedded Controller Handbook*, Intel Corporation, Santa Clara, California, Volume II, 16-bit, 1988.

[14] *EV80C196KA Microcontroller Evaluation Board User's Manual*, Intel Corporation, Santa Clara, California, Release 001, March 20, 1988.

[15] *Data Conversion Products Databook*, Analog Devices, Norwood, Ma., 1988.

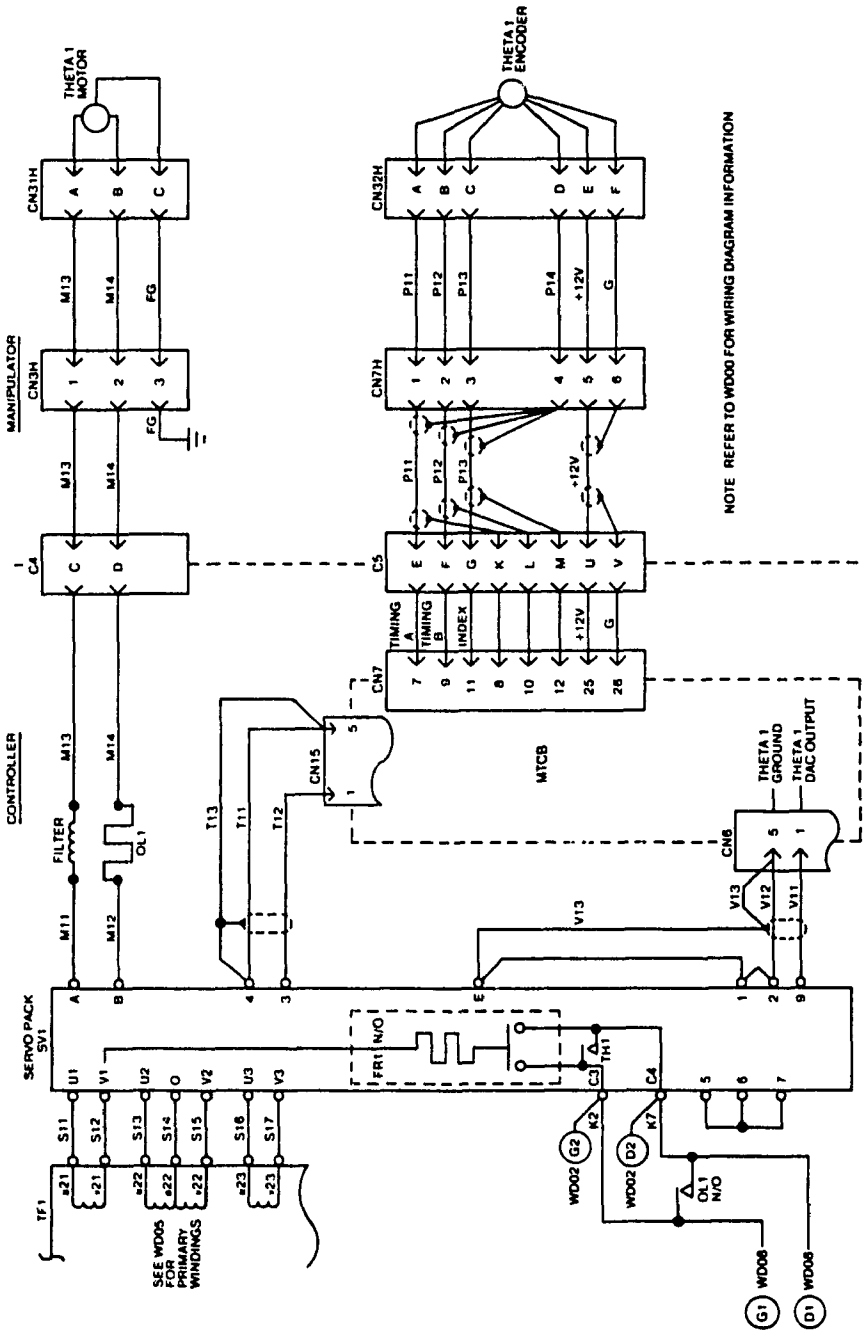
[16] *General Purpose Motion Control IC, HCTL-1000*, Hewlett Packard, Palo Alto, California, 1987.

[17] H. Seraji, "Decentralized Adaptive Control of Manipulators: Theory, simulation and Experimentation," *IEEE Trans. on Robotics and Automation*, vol. RA-5, no. 2, pp. 183-201.

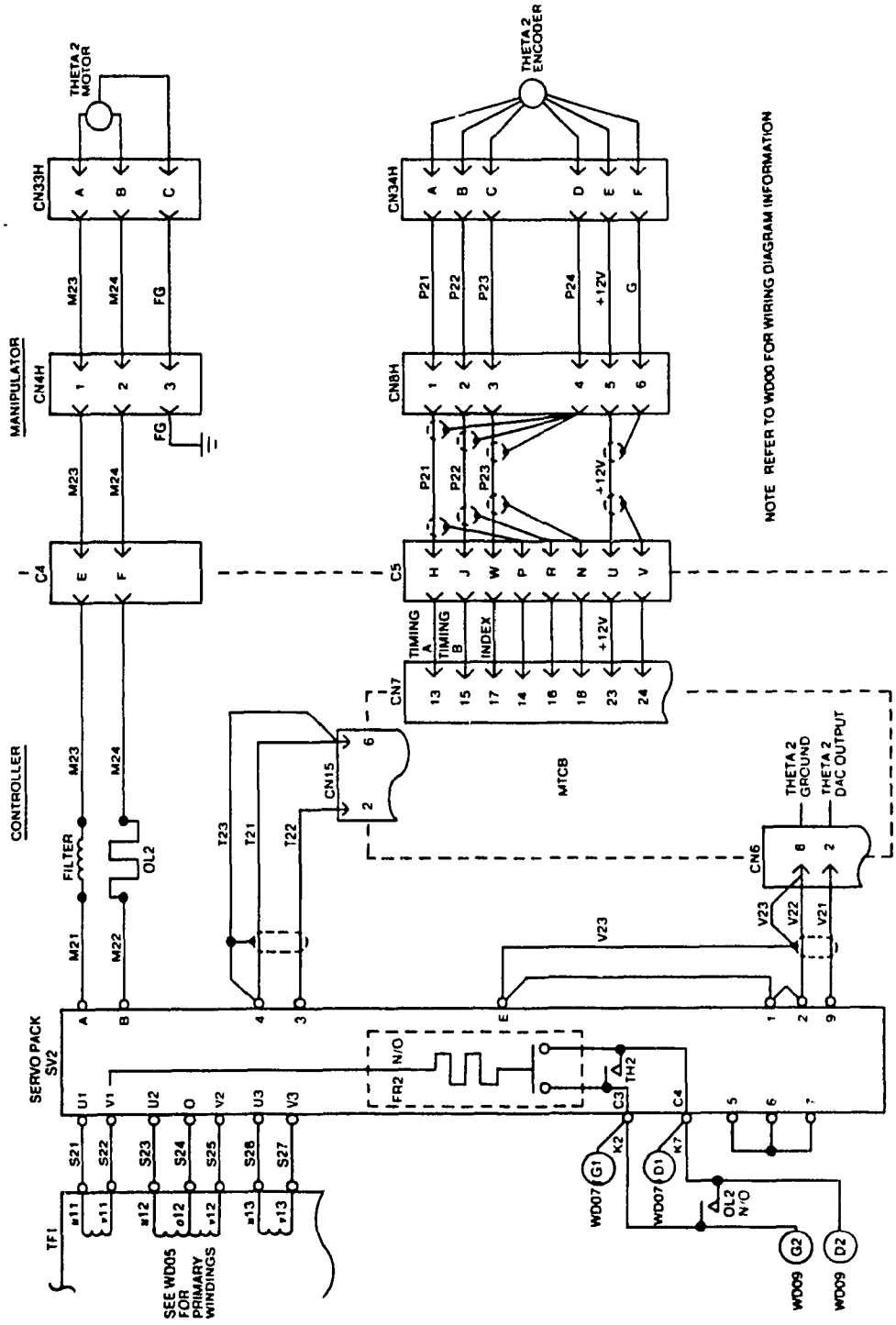
APPENDIX A

IBM 7545 WIRING DIAGRAMS

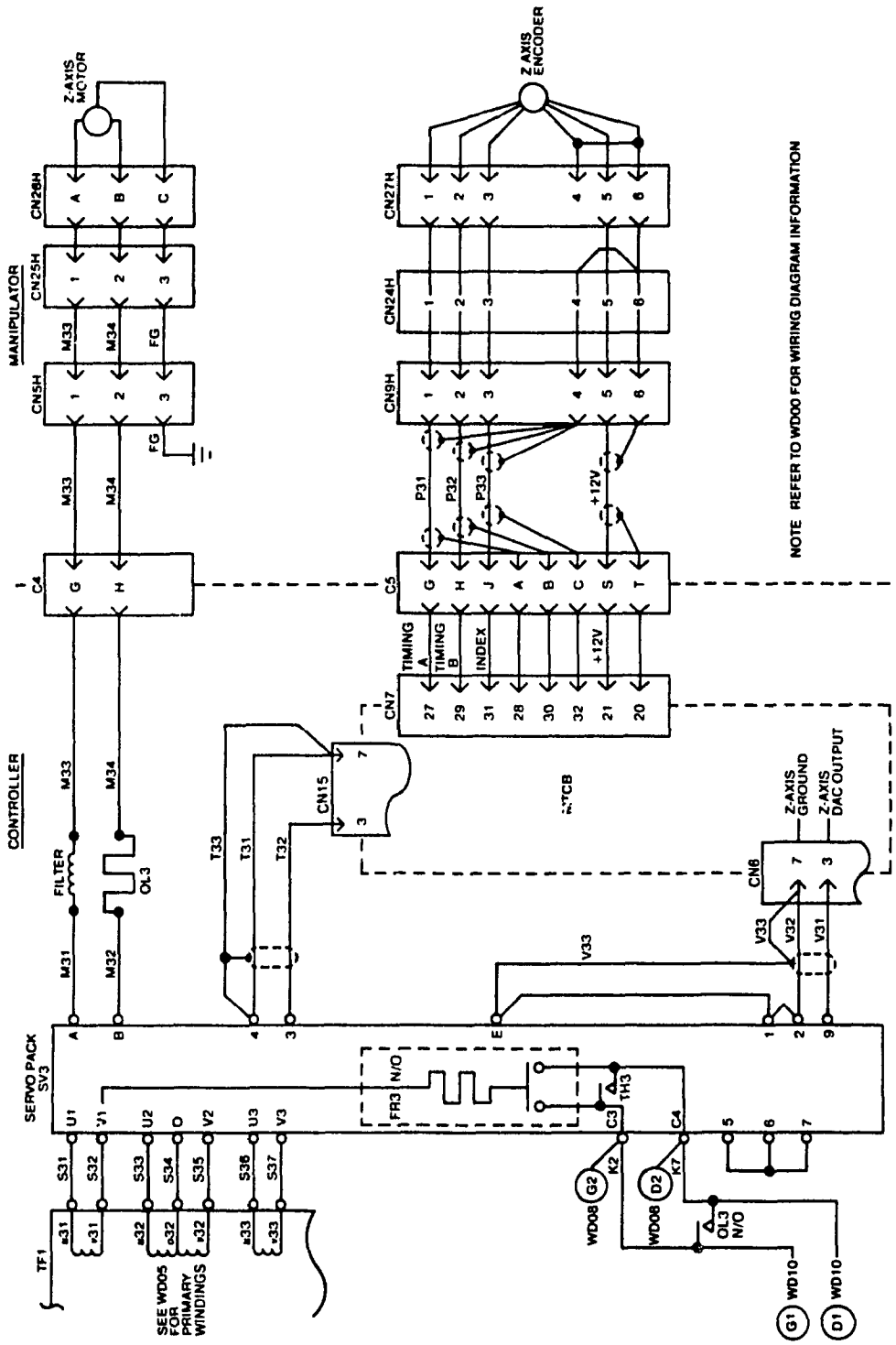
The wiring diagrams in this appendix, taken from [7], provide the necessary details to bypass the MTCB component of the IBM 7545 Manufacturing system.



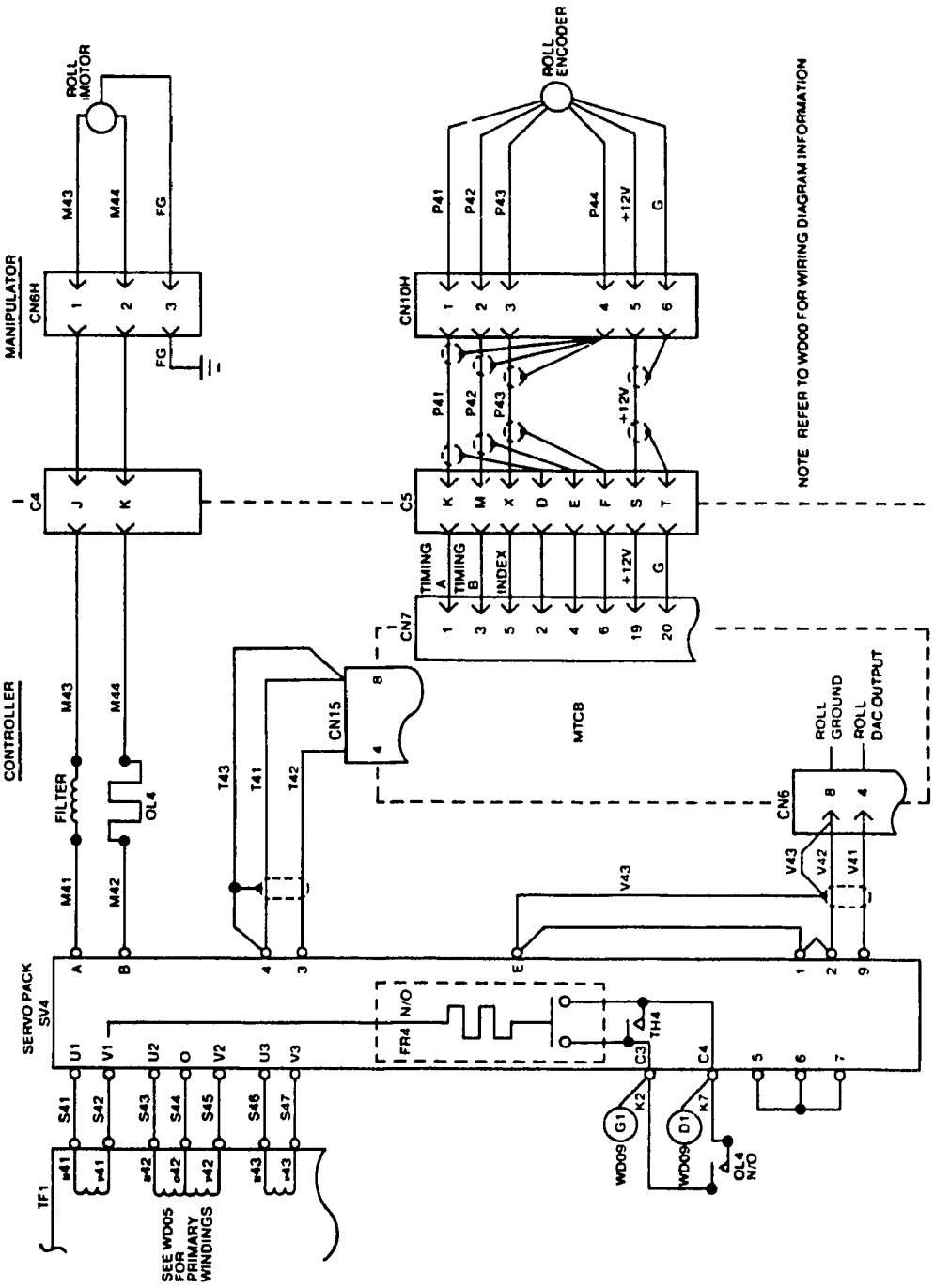
THETA 1 MOTOR AND FEEDBACK



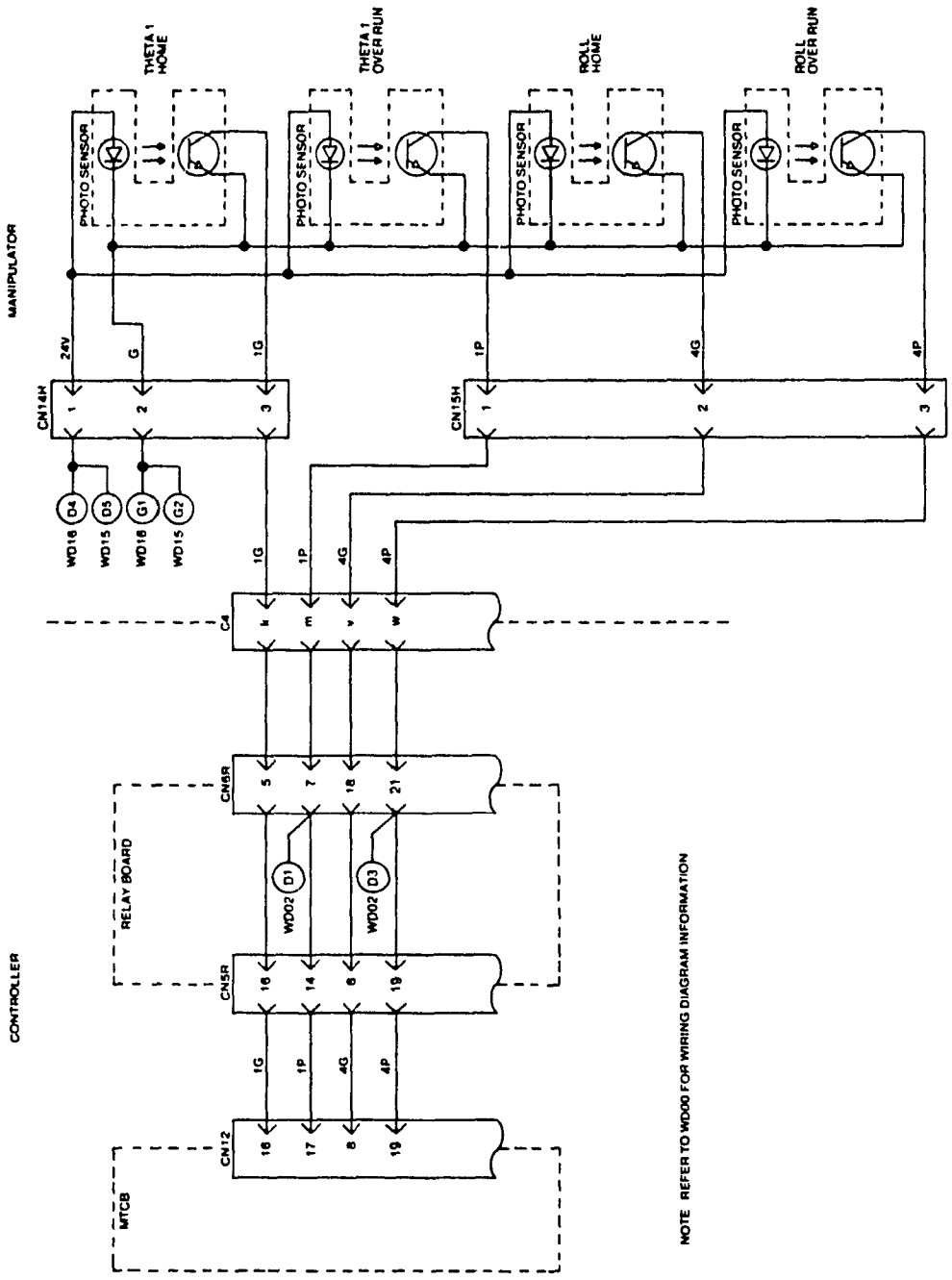
THETA 2 MOTOR AND FEEDBACK



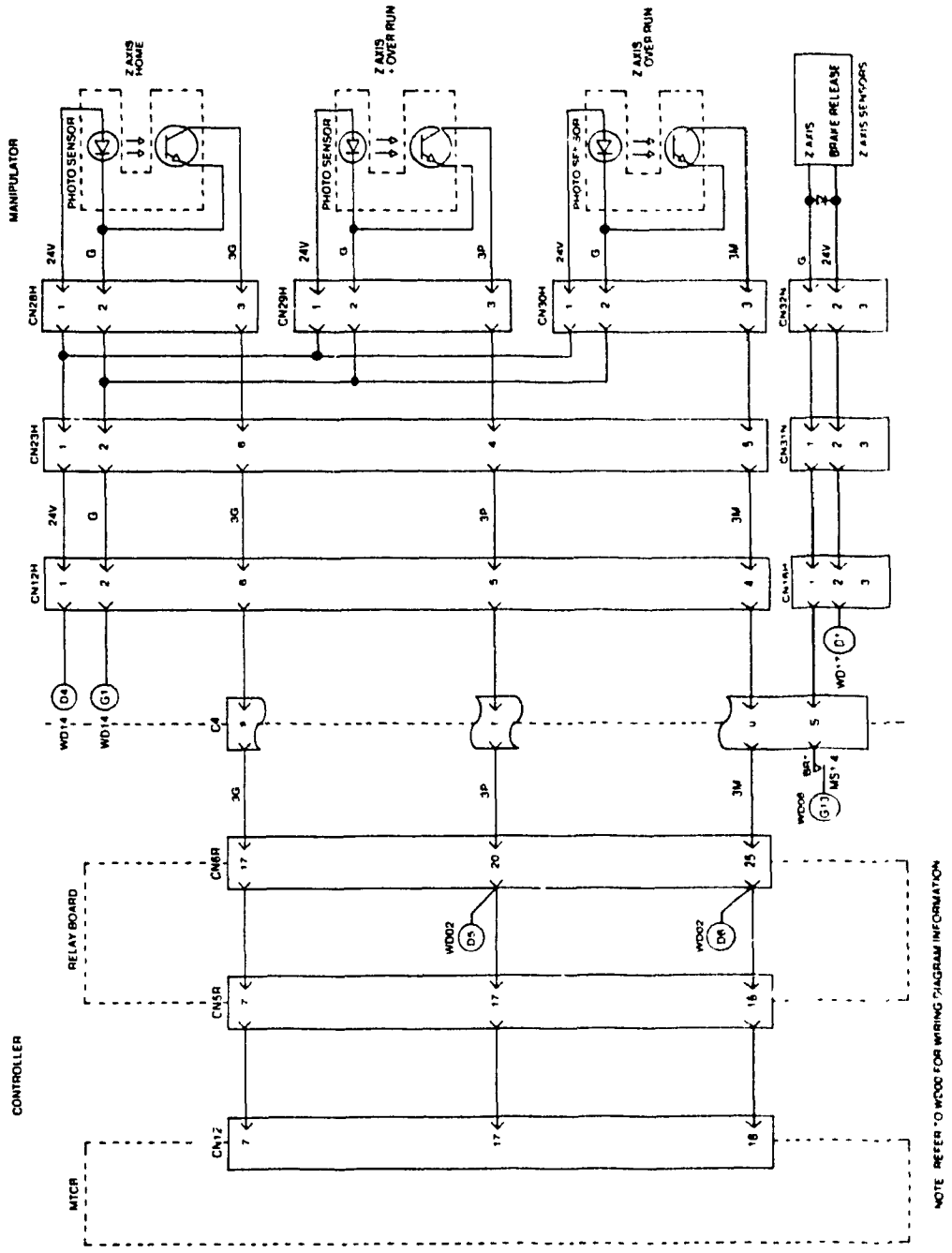
Z-AXIS MOTOR AND FEEDBACK



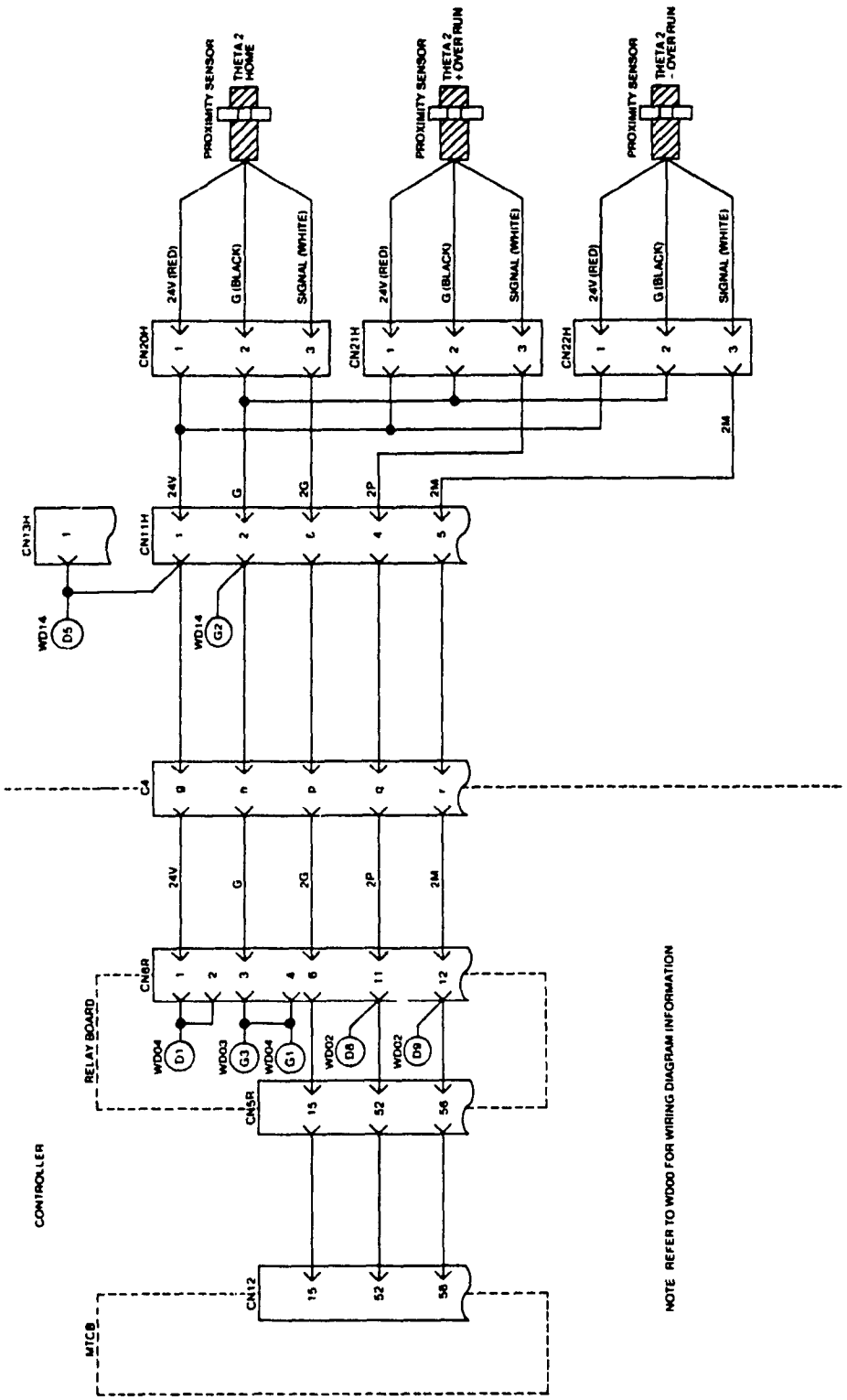
ROLL MOTOR AND FEEDBACK



THETA 1 AND ROLL SENSORS



Z-AXIS SENSORS



NOTE REFER TO WD00 FOR WIRING DIAGRAM INFORMATION

THETA 2 SENSORS

APPENDIX B

JOINT MOTOR RATINGS AND SPECIFICATIONS

The following table contains the ratings and specifications for the Yaskawa Print motors [8] used in joints 1 and 2 and Minertia motors [9] used in joints Z and Roll.

Parameter		Joint 1	Joint 2	Joints Z and Roll
Rated Output	(W)	200	100	n/a
Rated Torque	(kg•cm)	6.5	2.43	57.3
Rated Speed	(rpm)	3000	4000	3000
Rated Armature Voltage	(V)	42	26	n/a
Rated Armature Current	(A)	6.4	5.5	n/a
Power Rate	(kW/sec)	2.4	1.3	4.45
Torque Inertia Ratio	($rads/sec^2$)	4300	5200	12600
Max. Torque / 1 sec	(kg•cm)	36.4	14.4	n/a
Max. Armature Current/1 sec	(A)	33	29	22.1
Max. Safe Speed/1 sec	(rpm)	4950	6600	4000
Armature Inertia	($kg•cm^2$)	1.5	0.46	n/a
Armature Resistance	(Ω)	0.68	0.54	0.94
Armature Inductance	(mH)	0.06	0.02	0.9
Voltage Constant	(mV/rpm)	11.5	5.2	8.5
Torque Constant	(kg•cm/A)	0.23	0.11	0.83
Mech. Time Constant	(msec)	8.5	10	4.0
Elec. Time Constant	(msec)	0.09	0.04	0.96

n/a - not available

APPENDIX C

CALIBRATION

Calibration of the ARC system is a two part process. The first part involves the calibration of the DACs. The second part involves the calibration of the gain of the Speed Amplifier of the Servopacks. The calibration steps listed below should be performed for all four joints.

DAC Calibration

1. Disconnect the DAC from the Servopack input.
2. Connect a digital voltmeter (DVM) to the DAC output.
3. Write 0H to the DAC and adjust the potentiometer at the DAC's REF OUT terminal to obtain a -10.0000 V dc output.
4. Write 0FFFH to the DAC and adjust the potentiometer at the DAC's REF IN terminal to obtain a +9.9951 V dc output.
5. Reconnect the DAC to the Servopack input.

Speed Amplifier Calibration

1. Disconnect the motor from Servopack. This can be done by unplugging connectors CN3H, CN4H, CN5H, and CN6H for joint motors 1, 2, Z, and Roll, respectively.
2. Determine the saturation point of the Servopack's Speed Amplifier as follows:
    2.1. Disconnect the ARC from the Servopack.
    2.2. Ground the TG Feedback input on the Servopack.
    2.3. Connect a DVM to the Servopack's C_{ref} testpoint. This point is at the output of the Speed Amplifier.
    2.4. Connect a variable DC power supply in series with a 220K (1/4 W) resistor to the Servopack's Speed Reference input.
    2.5. Starting at 0 V dc, increase the power supply voltage and note the voltage at C_{ref} just before the Speed Amplifier saturates. The voltage

should be approximately -4.4, -4.0, -3.2, and -3.0 for joints 1, 2, Z, and Roll, respectively.

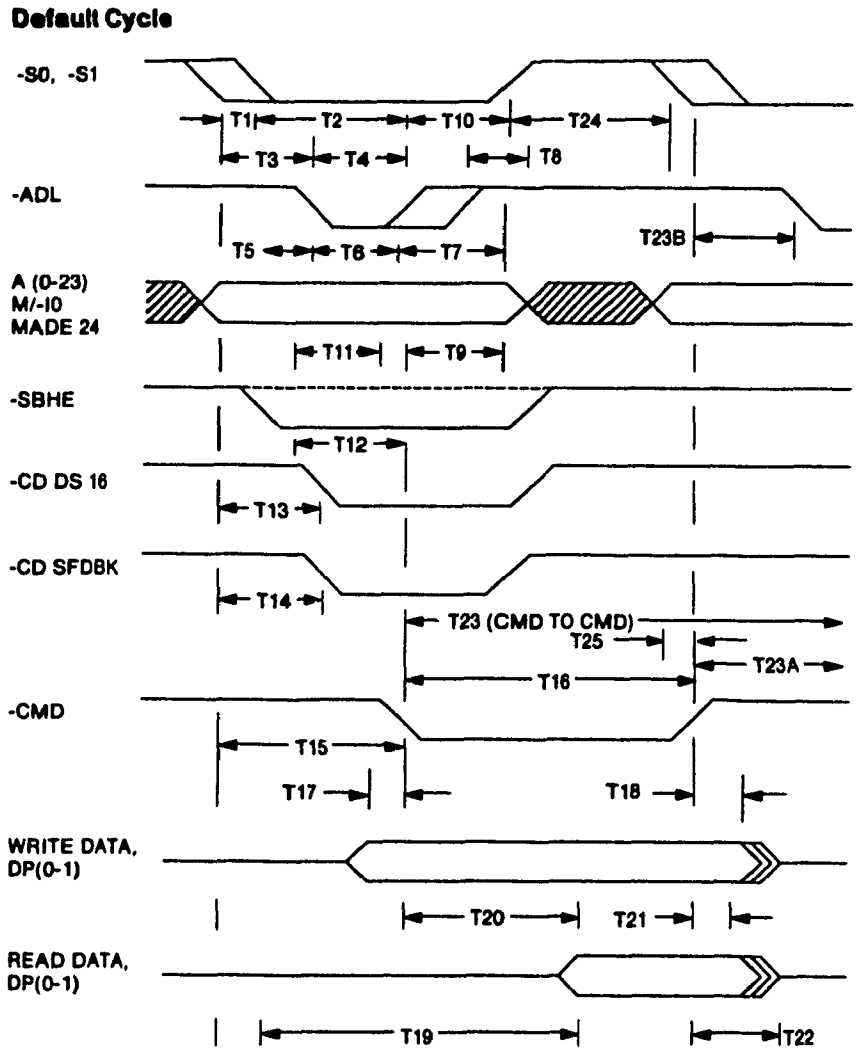
2.6. Disconnect the power supply and resistor and reconnect the ARC. The DAC should now be connected through the potentiometer to the Servopack Speed Reference input and all selector switches should be set for ARC control (see Figure 2.6 for details).

3. Write 0FFFH to the DAC and adjust the potentiometer until C_{ref} reaches the voltage noted in step 2. For better resolution, a resistor can be inserted in series with the potentiometer. The potentiometer can then be substituted by one of a lower value.

APPENDIX D

PS/2 TIMING DIAGRAMS

This appendix gives the I/O and Memory cycle timing diagrams for the IBM PS/2 model 50 computer as they appear in [11].



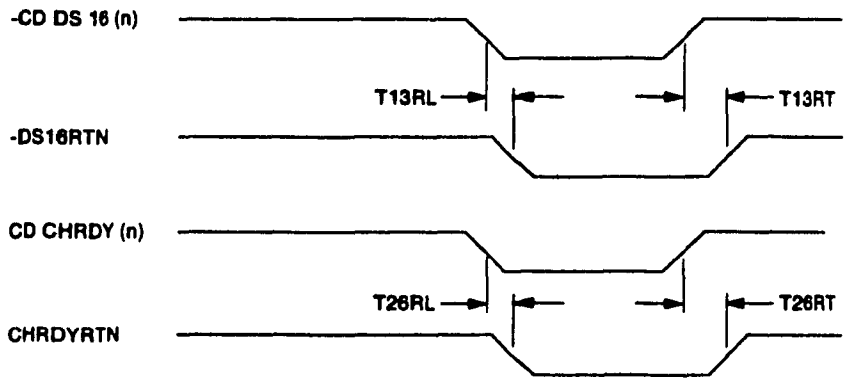
	Timing Parameter	Min/Max	Note
T1	Status active (low) from ADDRESS,M/-IO,-REFRESH valid	10 / - ns	
T2	-CMD active (low) from Status active (low)	55 / - ns	2
T3	-ADL active (low) from ADDRESS,M/IO,-REFRESH valid	45 / - ns	
T4	-ADL active (low) to -CMD active (low)	40 / - ns	
T5	-ADL active (low) from Status active (low)	12 / - ns	
T6	-ADL pulse width	40 / - ns	
T7	Status hold from -ADL inactive (high)	25 / - ns	2
T8	ADDRESS,M/-IO,-REFRESH,-SBHE hold from -ADL inactive	25 / - ns	2
T9	ADDRESS,M/-IO,-REFRESH,-SBHE hold from -CMD active (low)	30 / - ns	3
T10	Status hold from -CMD active	30 / - ns	2
T11	-SBHE setup to -ADL inactive	40 / - ns	2
T12	-SBHE setup to -CMD active	40 / - ns	2
T13	-CD DS 16 active (n) (low) from ADDRESS,M/-IO,-REFRESH valid	- / 55 ns	3
T14	-CD SFDBK active (low) from ADDRESS,M/-IO,-REFRESH valid	- / 60 ns	1
T15	-CMD active (low) from Address valid	85 / - ns	2
T16	-CMD pulse width	90 / - ns	
T17	Write data setup to -CMD active (low)	0 / - ns	
T18	Write data hold from -CMD inactive (high)	30 / - ns	
T19	Status to Read Data valid (Access Time)	- / 125 ns	
T20	Read data valid from -CMD active (low)	- / 60 ns	
T21	Read data hold from -CMD inactive (high)	0 / - ns	
T22	Read data bus tri-state from -CMD inactive (high)	- / 40 ns	
T23	-CMD active to next -CMD active	190 / - ns	4
T23A	-CMD inactive to next -CMD active	80 / - ns	
T23B	-CMD inactive to next -ADL active	40 / - ns	
T24	Next Status active (low) from Status inactive	30 / - ns	
T25	Next Status active (low) to -CMD inactive	- / 20 ns	

Figure 2-34. I/O and Memory Default Cycle (200 nanoseconds minimum)

Notes:

1. All slaves must drive -CD SFDBK whenever selected either by the system microprocessor or the DMA Controller. The slaves do not drive -CD SFDBK when they are selected by the 'setup' signal.
2. It is recommended that slaves use transparent latches to latch information with the leading or trailing edge of -ADL or with the leading edge of -CMD.
3. -CD DS 16 and -CD SFDBK must be driven by *unlatched address decodes* because the next address may come early into the current cycle.
4. Any master in any system, including the system microprocessor or DMA controller, can operate at a performance less than the level specified. Designers should not design to a given performance level as this level can be reduced by CD CHRDY, a lower microprocessor rate, a lower DMA controller rate, or system contention.
5. Model 50 and Model 60 automatically extend all default cycles to synchronous extended cycles. Adapter designs should support the 200 nanosecond default cycle to assist portability to other systems or drive CD CHRDY regardless of the system synchronous extension cycle.

Default Cycle Return Signals



Timing Parameter		Min/Max	Note
T13RL	-CD DS 16 (n) low to -DS 16 RTN low	- / 20 ns	1
T13RT	-CD DS 16 (n) high to -DS 16 RTN high	- / 20 ns	1
T26RL	CD CHRDY (n) low to CHRDYRTN low	- / 20 ns	2
T26RT	CD CHRDY (n) high to CHRDYRTN high	- / 20 ns	3

Figure 2-35. Default Cycle Return Signals (200 nanoseconds minimum)

Notes:

1. This signal is developed from a negative OR of signals received from each channel connector.
2. CHRDYRTN becomes active 40 nanoseconds maximum after -ADL becomes active.
3. This signal is developed from a positive AND of signals received from each channel connector.

Synchronous Special Case of Extended Cycle

A Synchronous Extended cycle occurs when a slave releases CD CHRDY synchronously within the specified time after the leading edge of -CMD. The slave provides the Read data within a specified time from -CMD. The timing sequence is illustrated by the following figure.

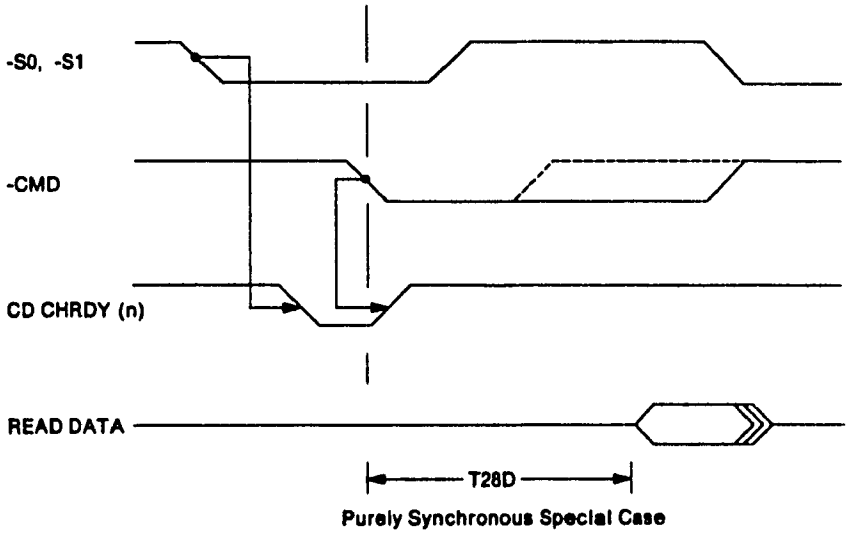
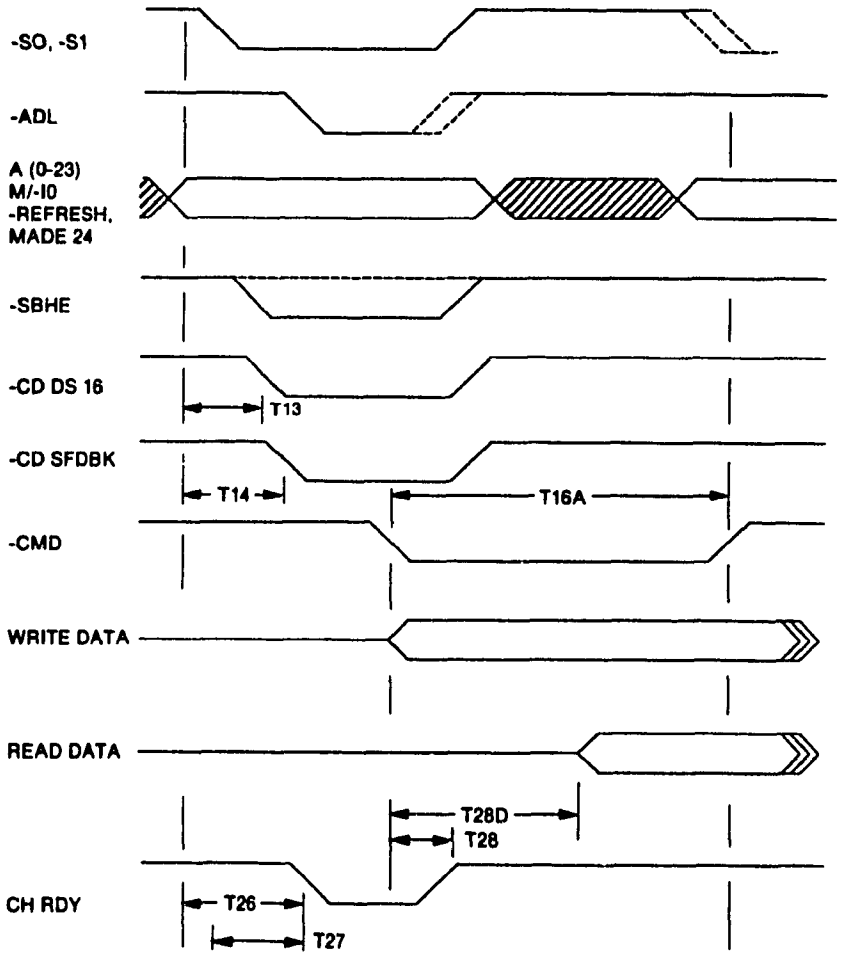


Figure 2-36. Timing Sequence for the Synchronous Special Case of Extended Cycle

Synchronous Extended Cycle (300 nanoseconds minimum - Special Case)



	Timing Parameter	Min/Max	Note
T13	-CD DS 16 (n) active (low) from ADDRESS,M/-IO,-REFRESH valid	- / 55 ns	2
T14	-CD SFDBK (n) active (low) ADDRESS,M/-IO,-REFRESH valid	- / 60 ns	2
T13A	-CMD pulse width	190 / - ns	
T26	CD CHRDY (n) inactive (low) from ADDRESS valid	- / 60 ns	3, See T27
T27	CD CHRDY (n) inactive (low) from Status active	0 / 30 ns	3
T28	CD CHRDY (n) release (high) from -CMD active (low)	0 / 30 ns	1
T28D	Read Data valid from -CMD active (when used along with T28)	0 / 160 ns	1

This figure shows only the parameters additional to the default cycle. All other parameters are the same as the default cycle

Figure 2-37. Synchronous Extended Cycle (300 nanoseconds minimum - Special Case)

Notes:

1. CD CHRDY is released by a slave performing a 300 nanoseconds extended cycle synchronous with the leading edge of -CMD. Since CD CHRDY is generally an asynchronous signal, this is referred to as a purely synchronous special case.
2. This is the same as default cycle timing (listed here for emphasis).
3. T27 is valid only when Status becomes active 30 nanoseconds or more after the address is valid.
4. If Status overlaps with previous -CMD, then CD CHRDY state is not valid during the overlapped period.
5. Slaves must not hold CD CHRDY inactive (low) in excess of 3.0 microseconds.

APPENDIX E

DUAL-PORT RAM TIMING WAVEFORMS

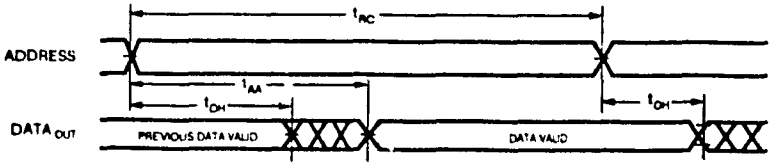
This appendix gives the timing information for the IDT7132/IDT7142 dual-port RAMs as found in [12].

AC ELECTRICAL CHARACTERISTICS OVER THE OPERATING TEMPERATURE AND SUPPLY VOLTAGE RANGE

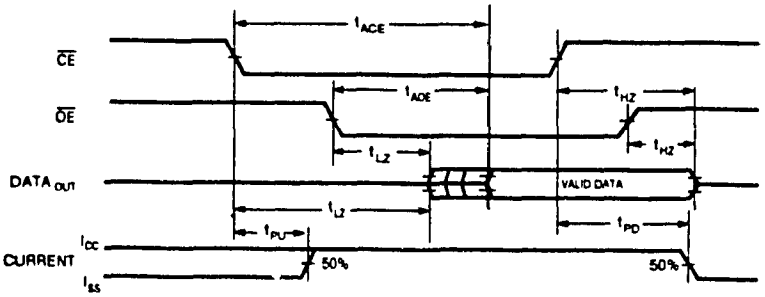
SYMBOL	PARAMETER	71321SA/LA35[2] 71421SA/LA35[2] MIN	MAX.	71321SA/LA45 71421SA/LA45 MIN	MAX.	71321SA/LA55 71421SA/LA55 MIN.	MAX.	71321SA/LA70[3] 71421SA/LA70[3] MIN	MAX.	UNIT
READ CYCLE										
t_{RC}	Read Cycle Time	35	–	45	–	55	–	70	–	ns
t_{AA}	Address Access Time	–	35	–	45	–	55	–	70	ns
t_{ACE}	Chip Enable Access Time	–	35	–	45	–	55	–	70	ns
t_{AOE}	Output Enable Access Time	–	25	–	30	–	35	–	40	ns
t_{OH}	Output Hold From Address Change	0	–	0	–	0	–	0	–	ns
t_{LZ}	Output Low Z Time [1, 4]	5	–	5	–	5	–	5	–	ns
t_{HZ}	Output High Z Time [1, 4]	–	15	–	20	–	30	–	35	ns
t_{PU}	Chip Enable to Power Up Time [4]	0	–	0	–	0	-	0	–	ns
t_{PD}	Chip Disable to Power Down Time [4]	–	50	–	50	–	50	–	50	ns

NOTES
1 Transition is measured ±500mV from low or high impedance voltage with load (Figures 1, 2 and 3)
2 0°C to +70°C temperature range only
3 -55°C to +125°C temperature range only
4 This parameter guaranteed but not tested

TIMING WAVEFORM OF READ CYCLE NO. 1, EITHER SIDE [1, 2, 4]

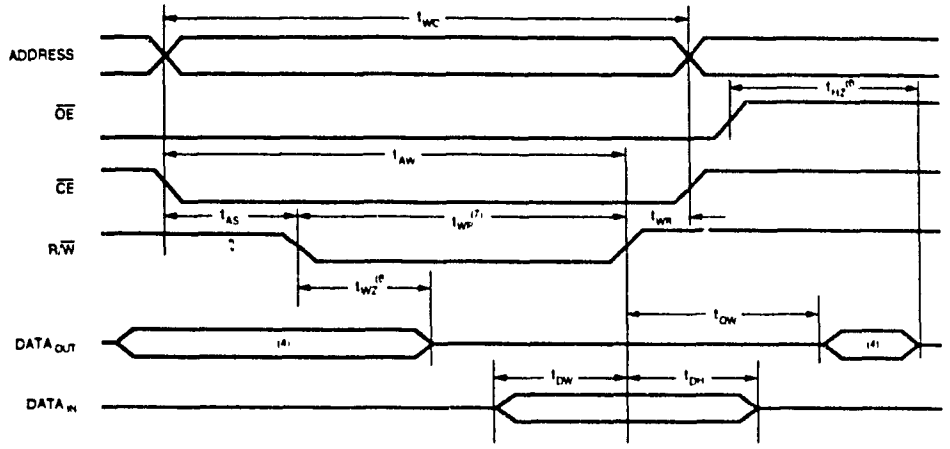


TIMING WAVEFORM OF READ CYCLE NO. 2, EITHER SIDE [1, 3]

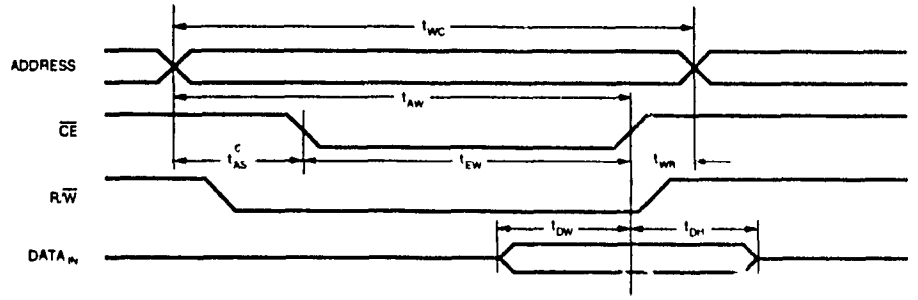


NOTES
1 R/$\overline{W}$ is high for Read Cycles
2 Device is continuously enabled $\overline{CE}$ = V_{IL}
3 Addresses valid prior to or coincident with $\overline{CE}$ transition low
4 $\overline{OE}$ = V_{IL}

TIMING WAVEFORM OF WRITE CYCLE NO. 1, R/$\overline{W}$ CONTROLLED TIMING [1, 2, 5, 7]



TIMING WAVEFORM OF WRITE CYCLE NO. 2, $\overline{CE}$ CONTROLLED TIMING [1, 2, 5, 6]



NOTES.

1 $\overline{WE}$ must be high during all address transitions
2 A write occurs during the overlap (t_{EW} or t_{WP}) of a low $\overline{CE}$ and a low R/$\overline{W}$
3 t_{WR} is measured from the earlier of $\overline{CE}$ or R/$\overline{W}$ going high to the end of write cycle
4 During this period the I/O pins are in the output state and input signals must not be applied
5 If the $\overline{CE}$ low transition occurs simultaneously with or after the R/$\overline{W}$ low transition the outputs remain in the high impedance state
6 Transition is measured ±500mV from steady state with a 5pF load (including scope and jig) This parameter is sampled and not 100% tested
7 If $\overline{OE}$ is low during a R/$\overline{W}$ controlled write cycle the write pulse must be the larger of t_{WP} or (t_{WZ} + t_{DW}) to allow the I/O drivers to turn off data to be placed on the bus for the required t_{DW} If $\overline{OE}$ is high during an R/$\overline{W}$ controlled write cycle this requirement does not apply and the write pulse can be as short as the specified t_{WP}

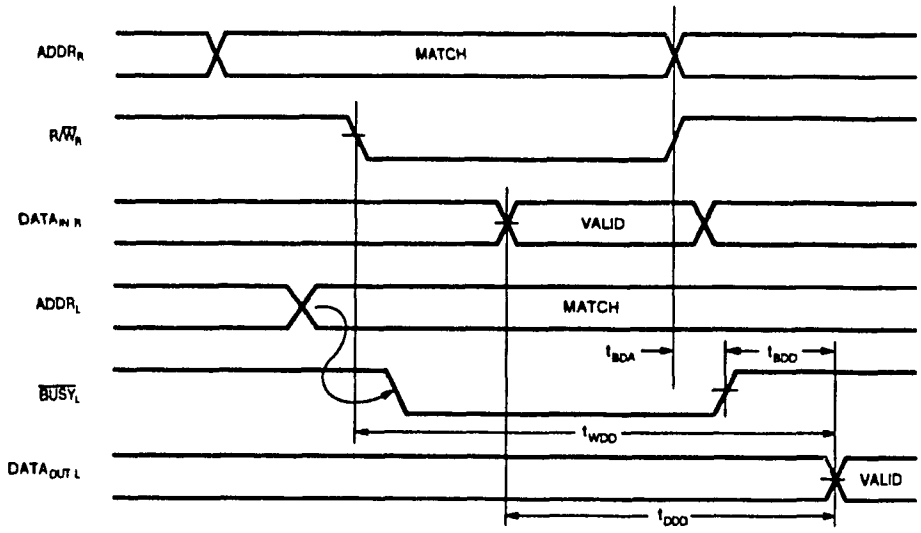
AC ELECTRICAL CHARACTERISTICS OVER THE OPERATING TEMPERATURE AND SUPPLY VOLTAGE RANGE

SYMBOL	PARAMETER	71321SA/LA35[2] 71421SA/LA35[3] MIN	MAX.	71321SA/LA45 71421SA/LA45 MIN	MAX.	71321SA/LA55 71421SA/LA55 MIN	MAX.	71321SA/LA70[3] 71421SA/LA70[3] MIN.	MAX	UNIT
WRITE CYCLE										
t_{WC}	Write Cycle Time [5]	35	–	45	–	55	–	70	–	ns
t_{EW}	Chip Enable to End of Write	30	–	35	–	40	–	50	–	ns
t_{AW}	Address Valid to End of Write	30	–	35	–	40	–	50	–	ns
t_{AS}	Address Set-up Time	0	–	0	–	0	–	0		ns
t_{WP}	Write Pulse Width	30	–	35	–	40	–	50	–	ns
t_{WR}	Write Recovery Time	0	–	0	–	0	–	0	–	ns
t_{DW}	Data Valid to End of Write	20	–	20	–	20	–	30		ns
t_{HZ}	Output High Z Time [1,4]	–	15	–	20	–	30	–	35	ns
t_{DH}	Data Hold Time	0	–	0	–	0	–	0	–	ns
t_{WZ}	Write Enabled to Output in High Z [1,4]	–	15	–	20	–	30	–	35	ns
t_{OW}	Output Active From End of Write [1,4]	0	–	0	–	0	–	0	–	ns

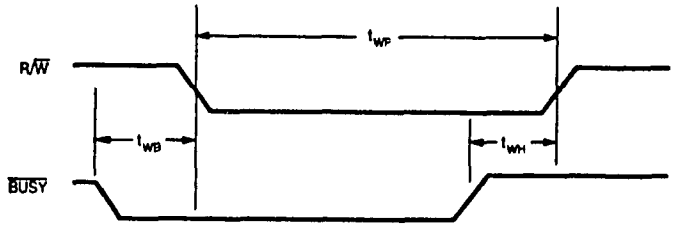
NOTES:

1 Transition is measured ±500mV from low or high voltage with load (Figures 1 2 and 3)
2 0°C to +70°C temperature range only
3 -55°C to +125°C temperature range only
4 This parameter guaranteed but not tested
5 For MASTER/SLAVE combination t_{WC} = t_{BAA} + t_{WP}

TIMING WAVEFORM OF READ WITH $\overline{BUSY}$



TIMING WAVEFORM OF WRITE WITH $\overline{BUSY}$



AC ELECTRICAL CHARACTERISTICS OVER THE OPERATING TEMPERATURE AND SUPPLY VOLTAGE RANGE

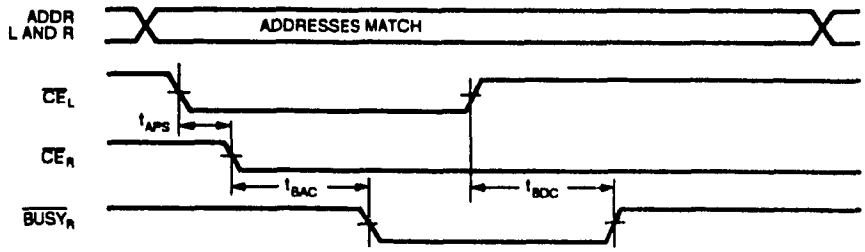
SYMBOL	PARAMETER	71321SA/LA35[1] 71421SA/LA35[1] MIN.	MAX.	71321SA/LA45 71421SA/LA45 MIN.	MAX.	71321SA/LA55 71421SA/LA55 MIN.	MAX.	71321SA/LA70[2] 71421SA/LA70[2] MIN.	MAX.	UNIT
BUSY TIMING										
t_{WB}	Write to $\overline{BUSY}$ [3,6]	0	–	0	–	-10	–	-10	–	ns
t_{WH}	Write Hold After $\overline{BUSY}$ [7]	20	–	20	–	20		20	–	ns
t_{BAA}	$\overline{BUSY}$ Access Time to Address	–	35	–	35	–	45	–	45	ns
t_{BDA}	$\overline{BUSY}$ Disable Time to Address	–	30	–	35	–	40	–	40	ns
t_{BAC}	$\overline{BUSY}$ Access Time to Chip Enable	–	30	–	30	–	35	–	35	ns
t_{BDC}	$\overline{BUSY}$ Disable Time to Chip Enable	–	25	–	25	–	30	–	30	ns
t_{WDD}	Write Pulse to Data Delay [4]	–	60	–	70	–	80	–	90	ns
t_{DDD}	Write Data Valid to Read Data Delay [4]	–	35	–	45	–	55	–	70	ns
t_{APS}	Arbitration Priority Set up Time	5	–	5	–	5	–	5	–	ns
t_{BDD}	$\overline{BUSY}$ Disable to Valid Data [5]	–	Note 5	–	Note 5	–	Note 5	–	Note 5	ns

NOTES.

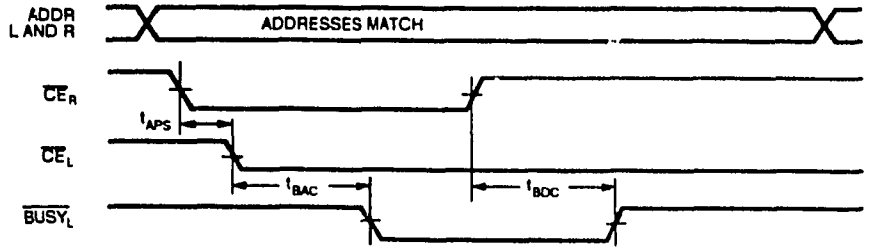
1 0°C to +70°C temperature range only
2 -55°C to +125°C temperature range only
3 For SLAVE part (IDT71421) only
4 Port to-port delay through RAM cells from writing port to reading port
5 t_{BDD} is a calculated parameter and is the greater of 0 $t_{WDD} - t_{WP}$ (actual) or $t_{DDD} - t_{DW}$ (actual)
6 To ensure that the write cycle is inhibited during contention
7 To ensure that a write cycle is completed after contention

TIMING WAVEFORM OF CONTENTION CYCLE NO. 1, $\overline{CE}$ ARBITRATION

$\overline{CE}_L$ VALID FIRST:

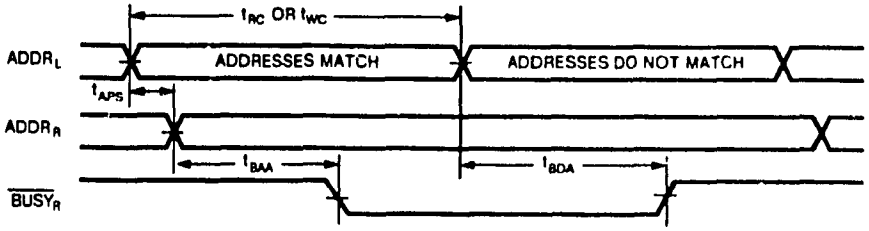


$\overline{CE}_R$ VALID FIRST:

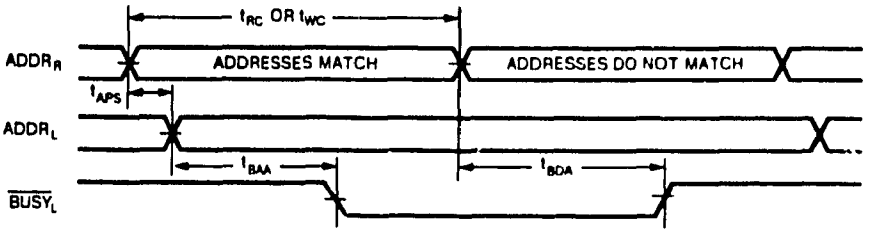


TIMING WAVEFORM OF CONTENTION CYCLE NO. 2, ADDRESS VALID ARBITRATION [1]

LEFT ADDRESS VALID FIRST:



RIGHT ADDRESS VALID FIRST:

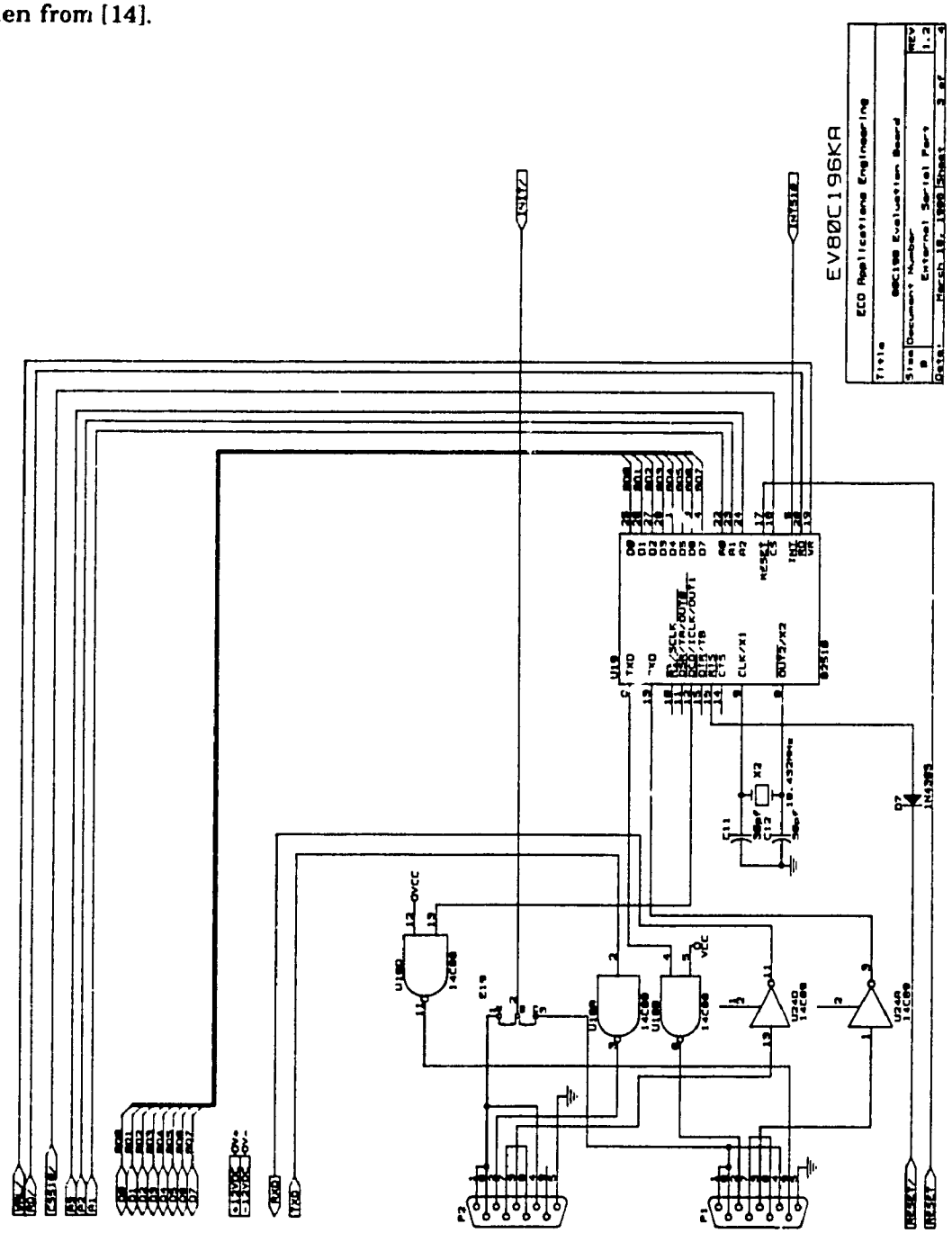


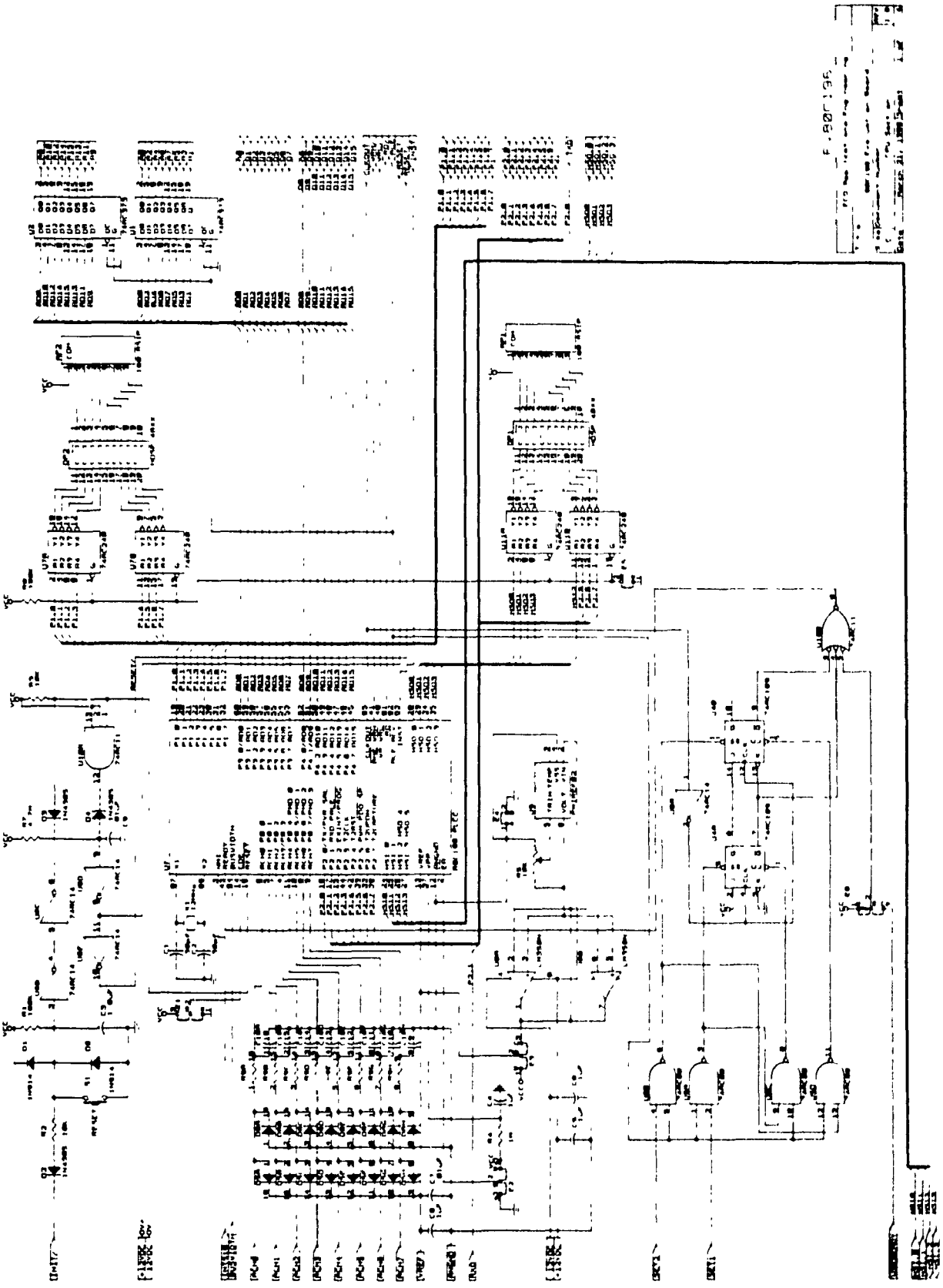
NOTE:
1 $\overline{CE}_L = \overline{CE}_R = V_{IL}$

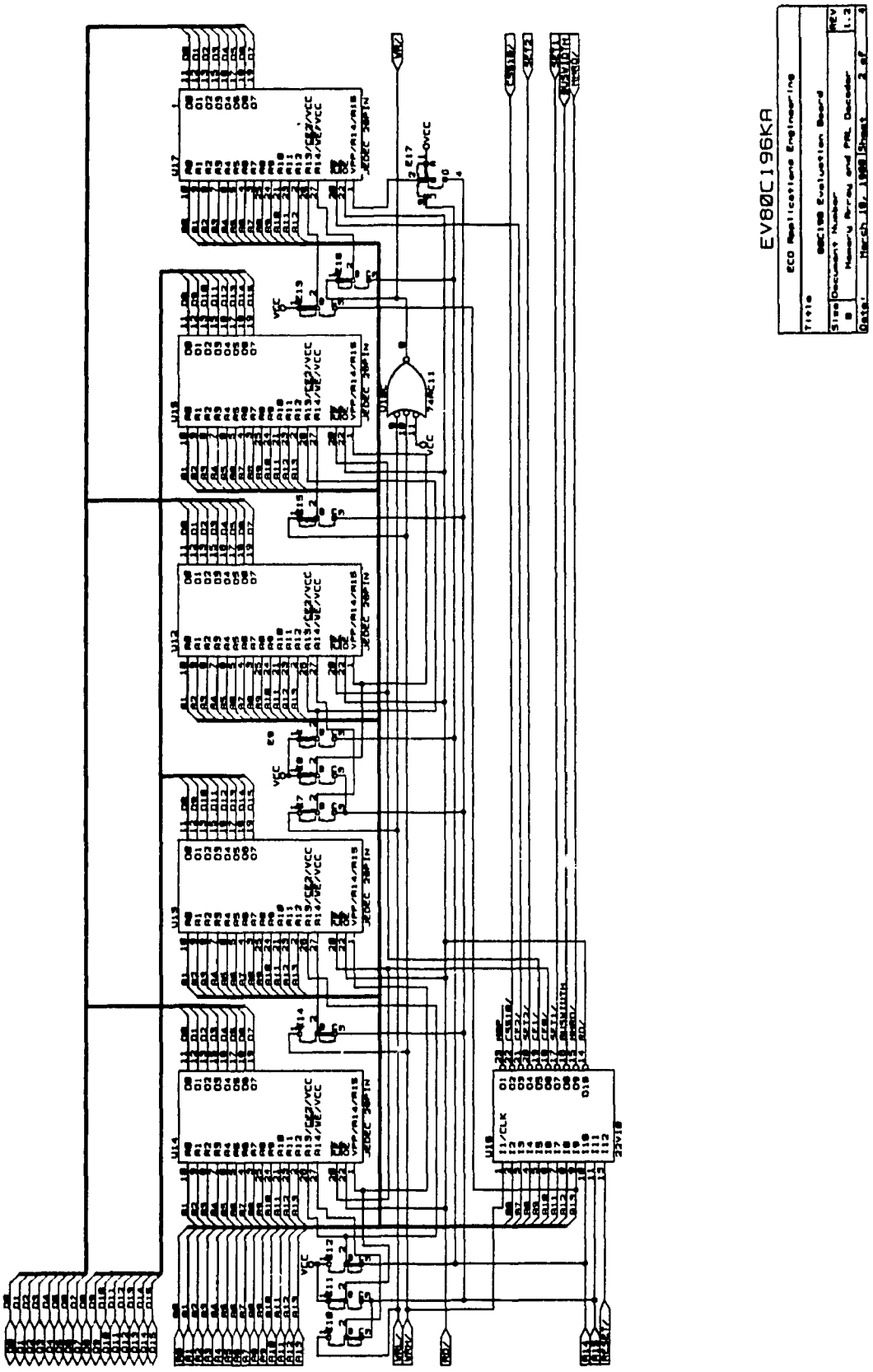
APPENDIX F

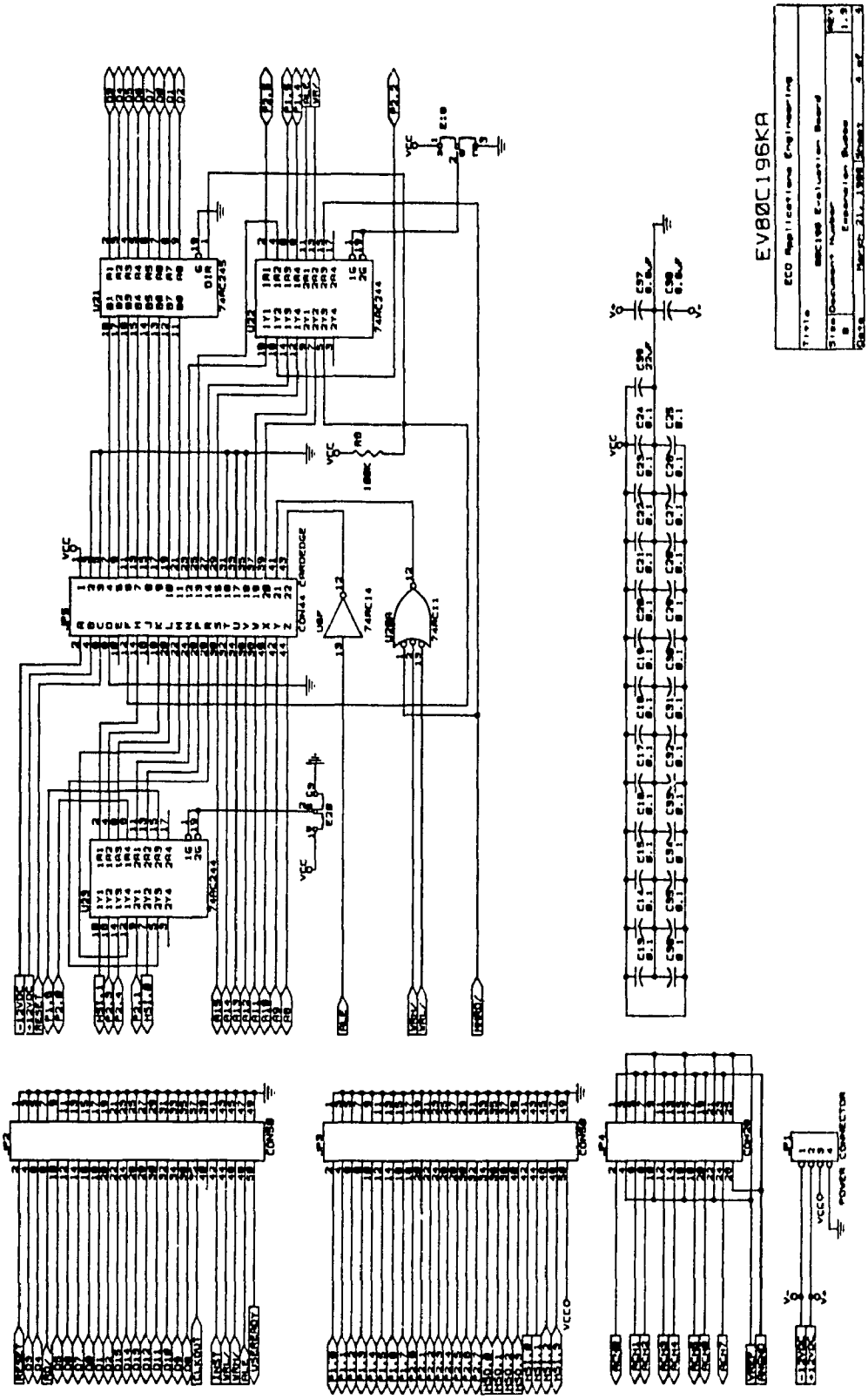
EV80C196KA SCHEMATIC DIAGRAM

The following schematic diagram for the INTEL EV80C196KA evaluation board is taken from [14].






EV80C196KA
ECO Applications Engineering
80C196 Evaluation Board
Memory Array and PAL Decoder

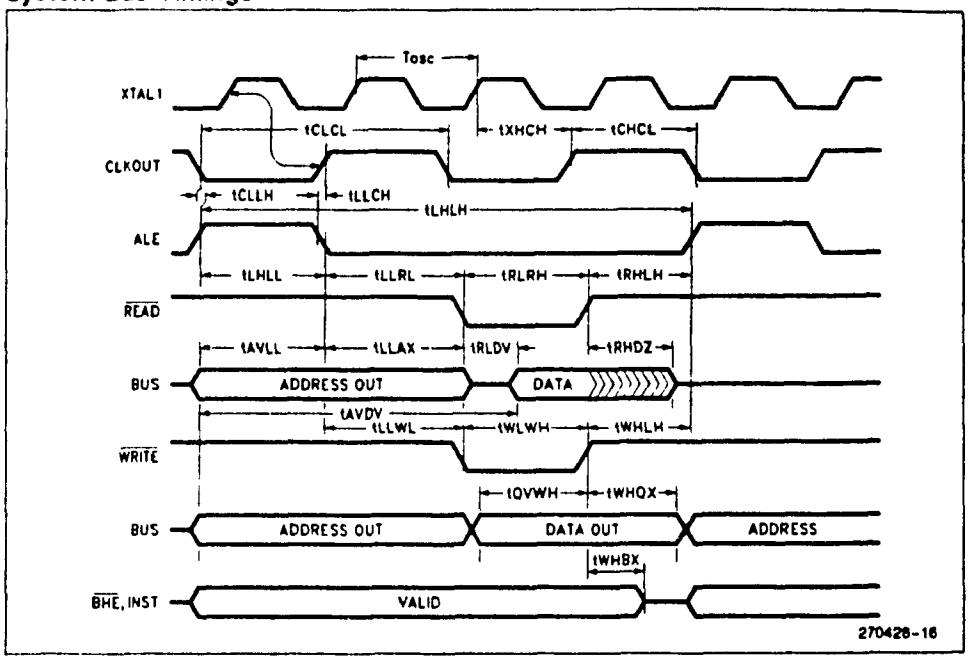

EV80C196KA
ECO Applications Engineering
80C196 Evaluation Board
POWER CONNECTOR

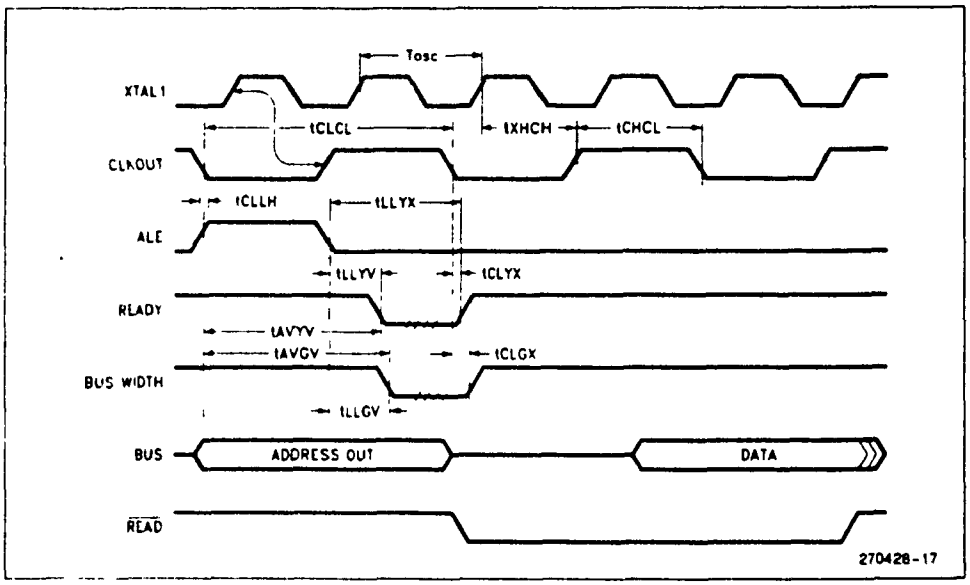
APPENDIX G

80C196KA BUS TIMING

The following information regarding the timing of the INTEL 80C196KA Microcontroller IC is taken from [13].

System Bus Timings





A.C. Characteristics (Over specified operating conditions)

These are ADVANCED specifications, the parameters may change before Intel releases the product for sale.

Test Conditions Capacitive load on all pins = 100 pF, Rise and fall times = 10 ns, f_{OSC} = 12 MHz

The system must meet these specifications to work with the 80C196:

Symbol	Description	Min	Max	Units	Notes
T_{AVYV}	Address Valid to READY Setup		$2T_{OSC} - 55$	ns	
T_{LLYV}	ALE Low to READY Setup		$T_{OSC} - 55$	ns	
T_{YLYH}	NonREADY Time	No upper limit		ns	
T_{CLYX}	READY Hold after CLKOUT Low	0	$T_{OSC} - 30$	ns	(Note 2)
T_{LLYX}	READY Hold after ALE Low	$T_{OSC} + 5$	$2T_{OSC} - 40$	ns	(Note 2)
T_{AVGV}	Address Valid to Buswidth Setup		$2T_{OSC} - 55$	ns	
T_{LLGV}	ALE Low to Buswidth Setup		$T_{OSC} - 55$	ns	
T_{CLGX}	Buswidth Hold after CLKOUT Low	0		ns	
T_{AVDV}	Address Valid to Input Data Valid		$3T_{OSC} - 60$	ns	
T_{RLDV}	RD# Active to Input Data Valid		$T_{OSC} - 25$	ns	
T_{CLDV}	CLKOUT Low to Input Data Valid		$T_{OSC} - 55$	ns	
T_{RHDZ}	End of RD# to Input Data Float		$T_{OSC} - 20$	ns	
T_{RXDX}	Data Hold after RD# Inactive	0		ns	

NOTES:
1 Typical specification not guaranteed
2 If max is exceeded additional wait states will occur

The 80C196KA will meet these specifications:

Symbol	Description	Min	Max	Units	Notes
F_{XTAL}	Frequency on XTAL1	3 5	12 0	MHz	
T_{OSC}	$1/F_{XTAL}$	83	286	ns	
T_{XHCH}	XTAL1 High to CLKOUT High or Low	40	110	ns	(Note 1)
T_{CLCL}	CLKOUT Cycle Time	$2T_{OSC}$		ns	
T_{CHCL}	CLKOUT High Period	$T_{OSC} - 10$	$T_{OSC} + 10$	ns	
T_{CLLH}	CLKOUT Falling Edge to ALE Rising	−10	10	ns	
T_{LLCH}	ALE Falling Edge to CLKOUT Rising	−10	10	ns	
T_{LHLH}	ALE Cycle Time	$4T_{OSC}$		ns	
T_{LHLL}	ALE High Period	$T_{OSC} - 10$	$T_{OSC} + 10$	ns	
T_{AVLL}	Address Setup to ALE Falling Edge	$T_{OSC} - 25$		ns	
T_{LLAX}	Address Hold after ALE Falling Edge	$T_{OSC} - 15$		ns	
T_{LLRL}	ALE Falling Edge to $\overline{RD}$ Falling Edge	$T_{OSC} - 25$		ns	
T_{RLCL}	$\overline{RD}$ Falling Edge to CLKOUT Falling Edge	0	20	ns	
T_{RLRH}	$\overline{RD}$ Low Period	$T_{OSC} - 5$		ns	
T_{RHLH}	$\overline{RD}$ Rising Edge to ALE Rising Edge	$T_{OSC} - 15$	$T_{OSC} + 15$	ns	(Note 2)
T_{LLWL}	ALE Falling Edge to $\overline{WR}$ Falling Edge	$T_{OSC} - 10$		ns	
T_{CLWL}	CLKOUT Low to $\overline{WR}$ Falling Edge	−5	15	ns	
T_{QVWH}	Data Stable to $\overline{WR}$ Rising Edge	$T_{OSC} - 20$		ns	
T_{CHWH}	CLKOUT Rising Edge to $\overline{WR}$ Rising Edge	−10	10	ns	
T_{WLWH}	$\overline{WR}$ Low Period	$T_{OSC} - 20$		ns	
T_{WHQX}	Data Hold after $\overline{WR}$ Rising Edge	$T_{OSC} - 20$		ns	
T_{WHLH}	$\overline{WR}$ Rising Edge to ALE Rising Edge	$T_{OSC} - 20$	$T_{OSC} + 20$	ns	(Note 2)
T_{WHBX}	$\overline{BHE}$, INST HOLD after $\overline{WR}$ Rising Edge	$T_{OSC} - 30$		ns	

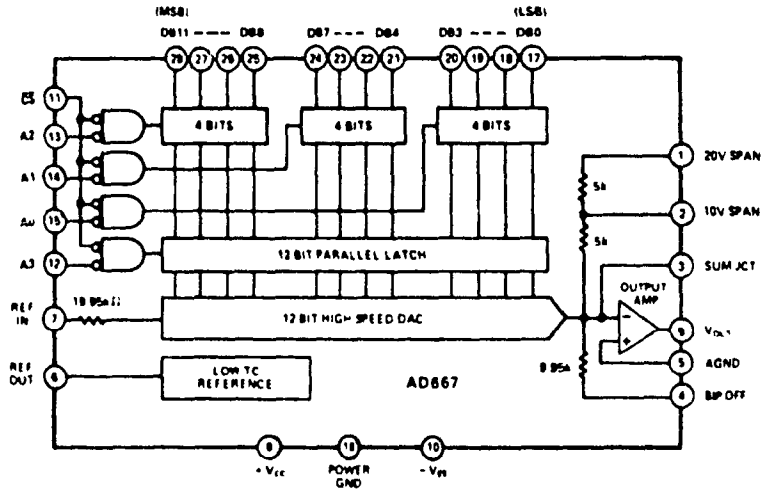
NOTES:
T_{OSC} = 83 3 ns at 12 MHz, T_{OSC} = 125 ns at 8 MHz
1 Typical specification not guaranteed
2 Assuming back to-back bus cycles

APPENDIX H

AD667 BLOCK DIAGRAM AND TIMING DIAGRAMS

The information in this appendix regarding the AD667 has been taken from [15].

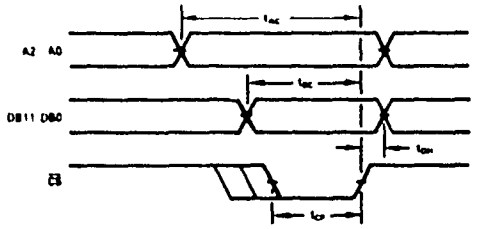
AD667 FUNCTIONAL BLOCK DIAGRAM



TIMING DIAGRAMS

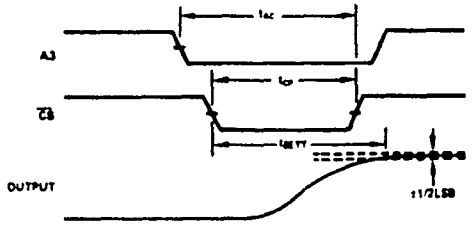
WRITE CYCLE #1

(Load First Rank from Data Bus; A3 = 1)



WRITE CYCLE #2

(Load Second Rank from First Rank; A2, A1, A0 = 1)



TIMING SPECIFICATIONS

(All Models, T_A = 25°C, V_{CC} = + 12V or + 15V, V_{EE} = − 12V or − 15V)

Symbol	Parameter	Min	Typ	Max	
t_{DC}	Data Valid to End of $\overline{CS}$	50	–	–	ns
t_{AC}	Address Valid to End of $\overline{CS}$	100	–	–	ns
t_{CP}	$\overline{CS}$ Pulse Width	100	–	–	ns
t_{DH}	Data Hold Time	0	–	–	ns
t_{SETT}	Output Voltage Settling Time	–	2	4	µs

APPENDIX I

HCTL-1000 BLOCK AND TIMING DIAGRAMS

The information in this appendix regarding the HCTL-1000 is taken from [16].

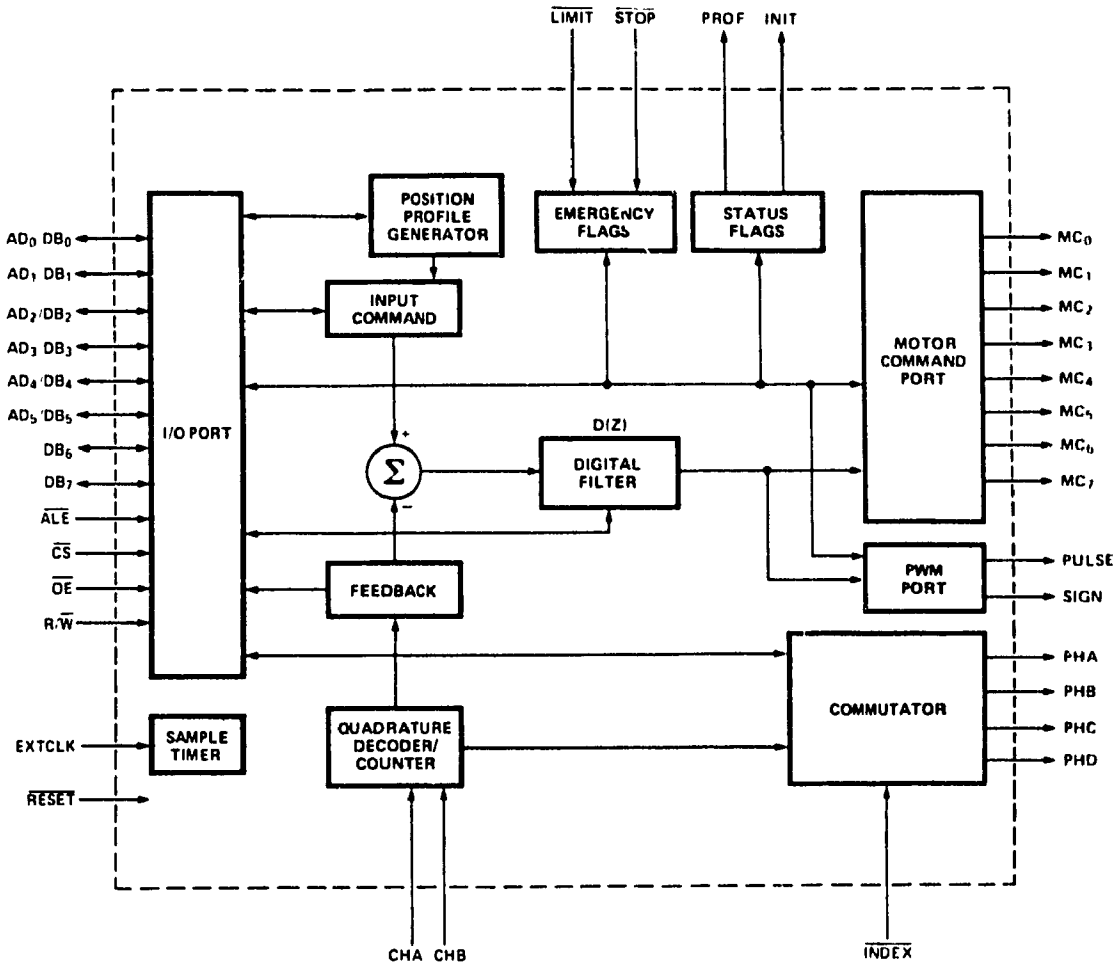


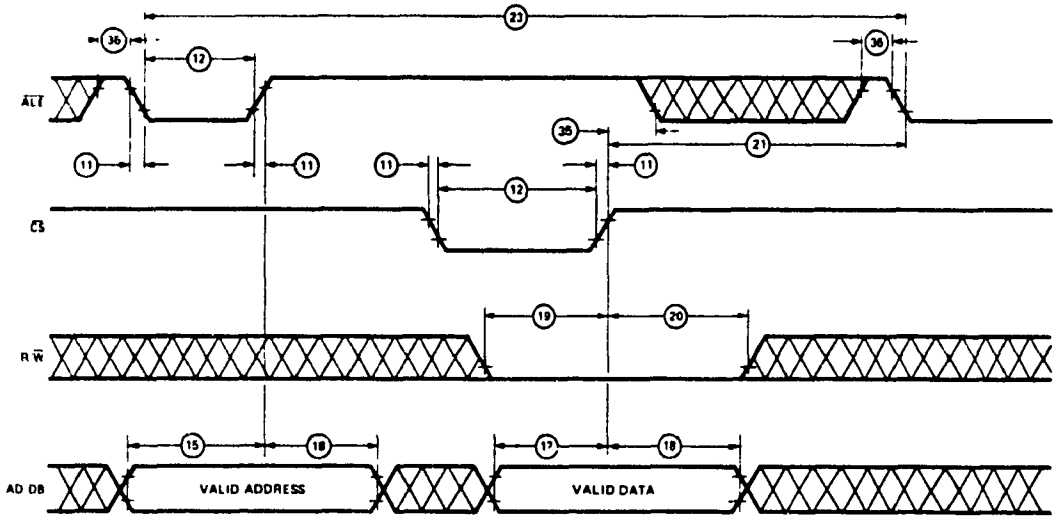
Figure 2. Internal Block Diagram

HCTL-1000 I/O Timing Diagrams

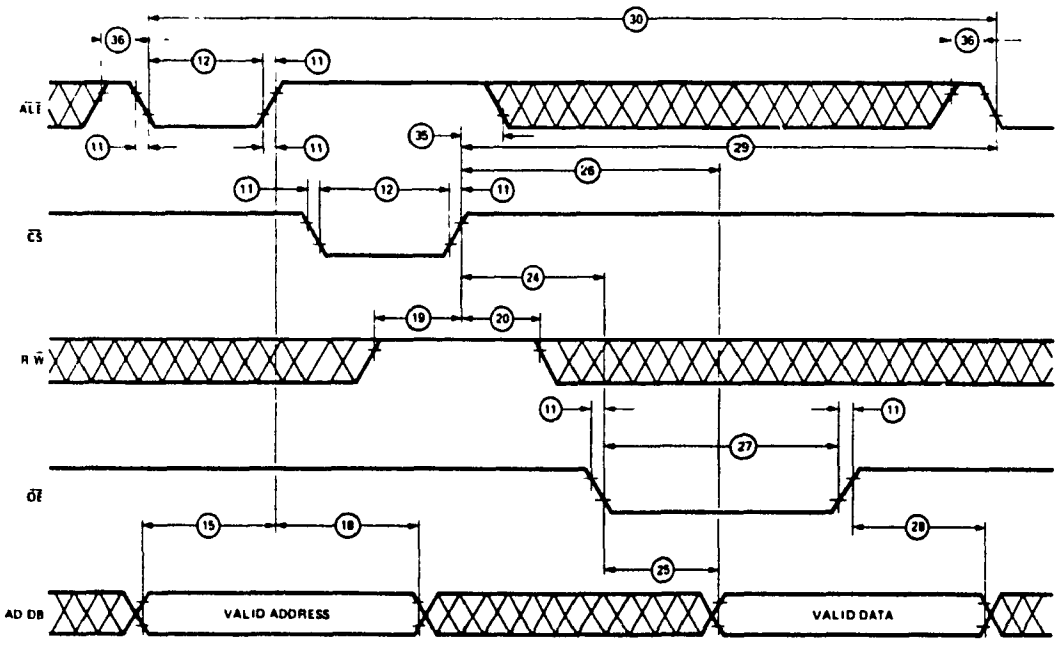
There are three different timing configurations which can be used to give the user flexibility to interface the HCTL-1000 to most microprocessors See the I/O interface section for more details

$\overline{ALE}/\overline{CS}$ NON OVERLAPPED

Write Cycle



Read Cycle



A.C. Electrical Characteristics T_A = 0°C to 70°C V_{CC} = 5 V ± 5%, Units = nsec

ID#	Signal	Symbol	Clock Frequency			
			2 MHz		1 MHz	
			Min.	Max.	Min.	Max.
1	Clock Period	t_{CPER}	500		1000	
2	Pulse Width, Clock High	t_{CPWH}	230		300	
3	Pulse Width, Clock Low	t_{CPWL}	200		200	
4	Clock Rise and Fall Time	t_{CR}		50		50
5	Input Pulse Width $\overline{\text{Reset}}$	t_{IRST}	2500		5000	
6	Input Pulse Width $\overline{\text{Stop}}$, $\overline{\text{Limit}}$	t_{IP}	600		1100	
7	Input Pulse Width $\overline{\text{Index}}$, Index	t_{IX}	1600		3100	
8	Input Pulse Width CHA, CHB	t_{IAB}	1600		3100	
9	Delay CHA to CHB Transition	t_{AB}	600		1100	
10	Input Rise/Fall Time CHA, CHB, Index	t_{IABR}		450		900
11	Input Rise/Fall Time $\overline{\text{Reset}}$, $\overline{\text{ALE}}$, $\overline{\text{CS}}$, $\overline{\text{OE}}$, $\overline{\text{Stop}}$, $\overline{\text{Limit}}$	t_{IR}		50		50
12	Input Pulse Width $\overline{\text{ALE}}$, $\overline{\text{CS}}$	t_{IPW}	80		80	
13	Delay Time, $\overline{\text{ALE}}$ Fall to $\overline{\text{CS}}$ Fall	t_{AC}	50		50	
14	Delay Time, $\overline{\text{ALE}}$ Rise to $\overline{\text{CS}}$ Rise	t_{CA}	50		50	
15	Address Set Up Time Before $\overline{\text{ALE}}$ Rise	t_{ASR1}	20		20	
16	Address Set Up Time Before $\overline{\text{CS}}$ Fall	t_{ASR}	20		20	
17	Write Data Set Up Time Before $\overline{\text{CS}}$ Rise	t_{DSR}	20		20	
18	Address/Data Hold Time	t_H	20		20	
19	Set Up Time, R/$\overline{\text{W}}$ Before $\overline{\text{CS}}$ Rise	t_{WCS}	20		20	
20	Hold Time, R/$\overline{\text{W}}$ After $\overline{\text{CS}}$ Rise	t_{WH}	20		20	
21	Delay Time, Write Cycle, $\overline{\text{CS}}$ Rise to $\overline{\text{ALE}}$ Fall	t_{CSAL}	1700		3400	
22	Delay Time, Read/Write, $\overline{\text{CS}}$ Rise to $\overline{\text{CS}}$ Fall	t_{CSCS}	1500		3000	
23	Write Cycle, $\overline{\text{ALE}}$ Fall to $\overline{\text{ALE}}$ Fall For Next Write	t_{WC}	1830		3530	
24	Delay time, $\overline{\text{CS}}$ Rise to $\overline{\text{OE}}$ Fall	t_{CSOE}	1700		3200	
25	Delay Time, $\overline{\text{OE}}$ Fall to Data Bus Valid	t_{OEDB}	100		100	
26	Delay Time, $\overline{\text{CS}}$ Rise to Data Bus Valid	t_{CSDB}	1800		3300	
27	Input Pulse Width $\overline{\text{OE}}$	t_{IPWOE}	100		100	
28	Hold Time, Data Held After $\overline{\text{OE}}$ Rise	t_{DOEH}	20		20	
29	Delay Time, Read Cycle, $\overline{\text{CS}}$ Rise to $\overline{\text{ALE}}$ Fall	t_{CSALR}	1820		3320	
30	Read Cycle, $\overline{\text{ALE}}$ Fall to $\overline{\text{ALE}}$ Fall For Next Read	t_{RC}	1950		3450	
31	Output Pulse Width, PROF, INIT, Pulse, Sign, PHA-PHD, MC Port	t_{OF}	500		1000	
32	Output Rise/Fall Time, PROF, INIT, Pulse, Sign PHA-PHD, MC Port	t_{OR}	20	150	20	150
33	Delay Time, Clock Rise to Output Rise	t_{EP}	20	300	20	300
34	Delay Time, $\overline{\text{CS}}$ Rising to MC Port Valid	t_{CSMC}		1600		3200
35	Hold Time, $\overline{\text{ALE}}$ High After $\overline{\text{CS}}$ Rise	t_{ALH}	100		100	
36	Pulse Width, $\overline{\text{ALE}}$ High	t_{ALPWH}	100		100	

APPENDIX J

SLAVE SOURCE CODE

```
;*******************************************************************************
;
;       SLAVE.A96 - 80C196KA ASSEMBLY CODE FOR THE ARC SLAVE PROCESSOR.

;*******************************************************************************

$GE                     ; Expand all MACROs in slave.lst listing
$include(8096.inc)      ; Include symbolic definitions from file 8096.inc.
                        ;   (listing of 8096.inc follows this listing)

;;
;; Storage Reservation for Program Variables and Pointers in the 80C196KA's
;; 232-byte Register File.  These registers reside on the 80C196KA chip and
;; should not be confused with the so-called dual-port RAM registers.
;;

rseg    at 40h          ; Registers are allocated beginning at 40H

        TEMP1:          dsw     1       ; Temporary registers.
        TEMP2:          dsw     1

        N0001:          dsw     1       ; Registers to hold frequently used
        N0003:          dsw     1       ; numbers.

        SAMPLE_PERIOD:  dsw     1       ; Registers to hold pointers to
        TIMING_A:       dsw     1       ; dual-port RAM registers.
        TIMING_B:       dsw     1
        COMMAND:        dsw     1
        ERROR:          dsw     1
        POS:            dsw     1
        TOR:            dsw     1

        SENSORS:        dsw     1       ; Register to hold pointer to the buffer
                                        ; for HOME and encoder index signals

        POS_1:          dsl     1       ; Registers to hold actual joint
        POS_2:          dsl     1       ; position information.
        POS_Z:          dsl     1
        POS_R:          dsl     1
        POS_1L:         dsl     1       ; Registers to hold the actual joint
        POS_2L:         dsl     1       ; positions from the previous (last)
        POS_ZL:         dsl     1       ; sampling period
        POS_RL:         dsl     1
        TORQUE_1:       dsw     1       ; Registers to hold joint torques
        TORQUE_2:       dsw     1
        TORQUE_Z:       dsw     1
        TORQUE_R:       dsw     1

        PCZ2_R14_CS:    dsw     1       ; Pointers to HCTL-1000 position counters
        PCZ2_R13_CS:    dsw     1       ; Z2 or 1R => joints Z & 2 or 1 & R
        PCZ2_R12_CS:    dsw     1       ; R12, R13, R14 => register 12, 13, 14
        PCZ2_R14_OE:    dsw     1       ; CS => chip select
        PCZ2_R13_OE:    dsw     1       ; OE => output enable
        PCZ2_R12_OE:    dsw     1
        PC1R_R14_CS:    dsw     1
        PC1R_R13_CS:    dsw     1
        PC1R_R12_CS:    dsw     1
        PC1R_R14_OE:    dsw     1
        PC1R_R13_OE:    dsw     1
        PC1R_R12_OE:    dsw     1
```

```
PC_1_PC:        dsw     1       ; Registers to hold pointers to the
PC_2_PC:        dsw     1       ; Position counters' program counters
PC_Z_PC:        dsw     1
PC_R_PC:        dsw     1
PC_1_RESET:     dsw     1       ; Registers to hold values to be to
PC_2_RESET:     dsw     1       ; sent to position counters'
PC_Z_RESET:     dsw     1       ; program counters
PC_R_RESET:     dsw     1

DAC_1:          dsw     1       ; Registers to hold pointers to the
DAC_2:          dsw     1       ; first rank registers of the DACS
DAC_Z:          dsw     1
DAC_R:          dsw     1
DACS_OUT:       dsw     1       ; Register to hold pointer to the
                                ; second rank register of each DAC

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;                        INITIALIZATION                          ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

cseg    at 2080h        ; Code segment begins at 2080H

;; initialization of Program Variables and Pointers

        LD      DAC_1, #0AFFCH          ; DAC first rank addresses
        LD      DAC_2, #0AFFAH
        LD      DAC_Z, #0AFF6H
        LD      DAC_R, #0AFEEH
        LD      DACS_OUT, #0AFDEH       ; All DACS second rank address

        LD      PCZ2_R14_CS, #09494H    ; Position counter addresses
        LD      PCZ2_R13_CS, #09392H
        LD      PCZ2_R12_CS, #09292H
        LD      PCZ2_R14_OE, #09414H
        LD      PCZ2_R13_OE, #09312H
        LD      PCZ2_R12_OE, #09212H
        LD      PC1R_R14_CS, #094D4H
        LD      PC1R_R13_CS, #093D2H
        LD      PC1R_R12_CS, #092D2H
        LD      PC1R_R14_OE, #09454H
        LD      PC1R_R13_OE, #09352H
        LD      PC1R_R12_OE, #09252H
        LD      PC_1_PC, #085C4H        ; Address of program counter register
        LD      PC_2_PC, #08584H        ;   on the position counters.
        LD      PC_Z_PC, #08584H        ;   1 & R, 2 & Z are equal
        LD      PC_R_PC, #085C4H        ;   since they are accessed in pairs.
        LD      PC_1_RESET, #0100H      ; 0100 resets 1 but leaves R idling.
        LD      PC_2_RESET, #0001H      ; 0100 resets 2 but leaves Z idling.
        LD      PC_Z_RESET, #0100H      ; 0100 resets Z but leaves 2 idling.
        LD      PC_R_RESET, #0001H      ; 0001 resets R but leaves 1 idling.

        LD      N0001, #0001H
        LD      N0003, #0003H

        LD      SP, #100H               ; Init stack at top of reg. file
        LDB     IOC2, #00000001B        ; Put TIMER2 into fast increment mode

        LD      SAMPLE_PERIOD, #0E000H ; Addresses of dual-port RAM registers
        LD      TIMING_A, #0E002H
        LD      TIMING_B, #0E004H
        LD      COMMAND, #0E006H
        LD      ERROR, #0E008H
        LD      POS, #0E010H
        LD      TOR, #0E020H

        LD      SENSORS, #0B000H        ; Address of buffer for HOME and
                                        ; encoder index signals

        ST      0, [ERROR]      ; Clear error register
        ST      0, [PC_1_PC]    ; Software reset all position counters by
        ST      0, [PC_2_PC]    ;   writing 0 to their program counters.
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;                       MAIN PROGRAM                              ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

main:
        ST      0, [COMMAND]            ; Clear Command register
wait_for_command:
        CALL    zero_dacs               ; Turn off all servo motors.
        CALL    get_joint_positions     ; get state of robot.
        CALL    joint_overrun_check
        CMP     0, [ERROR]              ; If there is a joint overrun error
        JNE     wait_for_command        ;   keep waiting until corrected.

        LD      TEMP1, [COMMAND]        ; Read Command register
        BBS     TEMP1, 0, control       ; If bit 0 set, enter control_mode.
        BBS     TEMP1, 7, find_home     ; If bit 7 set, enter find_home mode.
        BR      wait_for_command        ; Otherwise keep waiting

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;                       FIND HOME MODE                            ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

FH      MACRO   joint, positive_torque, negative_torque, home_bit, index_bit
        LOCAL   nh, fh_1, fh_2, fh_3
        LD      TEMP1,[SENSORS]
        JBS     TEMP1, home_bit, fh_1           ;; If HOME sensor not set, goto fh_1.
        LD      TEMP2, #positive_torque         ;;   else move joint away from HOME.
        ST      TEMP2, [DAC_&joint]
        ST      0, [DACS_OUT]
nh:     LD      TEMP1, [SENSORS]                ;; If HOME sensor still set, goto nh
        JBC     TEMP1, home_bit, nh             ;;   else keep moving away from HOME
        CALL    delay                           ;;   during 'delay' period (~2 secs).
        CALL    zero_dacs                       ;; Stop the motor.
fh_1:   LD      TEMP2, #negative_torque         ;; Move joint toward home,
        ST      TEMP2, [DAC_&joint]
        ST      0, [DACS_OUT]
fh_2:   LD      TEMP1, [SENSORS]                        ;; Wait for home sensor
        JBS     TEMP1, home_bit, fh_2                   ;;   to set.
fh_3:   LD      TEMP1, [SENSORS]                        ;; Wait for index pulse.
        JBC     TEMP1, index_bit, fh_3                  ;;
        ST      PC_&joint&_RESET,[PC_&joint&_PC]        ;; Set pos. counter to zero.
        CALL    zero_dacs                               ;; Remove the torque.
        ENDM

find_home:
        FH      Z, 900H, 640H, 4, 0
        FH      1, 900H, 700H, 7, 2
        FH      2, 900H, 700H, 6, 1
        FH      R, 920H, 6A0H, 5, 3

        BR      main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;                       CONTROL MODE                              ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

control:
        CLR     TIMER2                  ; Clear the sample period timer
        ST      0, [TIMING_A]           ; Clear the timing registers
        ST      0, [TIMING_B]

control_loop:
        ST      N0001, [TIMING_A]       ; Signal master to begin its loop
        SCALL   get_joint_positions     ; get actual joint positions

        SCALL   joint_overrun_check     ; Check positions for overruns.
        CMP     0,[ERROR]               ; End if an overrun error was found
        BNE     main

wait_for_torque:
        CMP     TIMER2, [SAMPLE_PERIOD]         ; Wait for torques but end (with
```

```
        JGT     torques_too_late            ;   error) if they arrive too late.
        CMP     0, [COMMAND]                ; End (without error) if master
        BE      main                        ;   clears command register.
        AND     TEMP1, N0001,[TIMING_B]     ; Check for arrival of torques.
        JE      wait_for_torques

        SCALL   check_torques               ; Get and check torques.
        CMP     0, [ERROR]                  ; End if excess torque found.
        BNE     main

        ST      0, [TIMING_B]               ; Tell master torques were accepted.
        SCALL   torques_out                 ; Output torques.

wait_for_master:
        CMP     TIMER2, [SAMPLE_PERIOD]     ; Wait for master to finish loop.
        JGT     master_too_slow             ; End if master is too slow.
        CMP     0, [TIMING_A]
        JNE     wait_for_master
wait_next_period:
        CMP     TIMER2, [SAMPLE_PERIOD]     ; Wait for next sample period
        JLT     wait_next_period
        SUB     TIMER2, [SAMPLE_PERIOD]     ; Reset the sample period timer
        BR      control_loop

torques_too_late:
        LD      TEMP1, [ERROR]
        OR      TEMP1, #1000H
        ST      TEMP1, [ERROR]
        BR      main

master_too_slow:
        LD      TEMP1, [ERROR]
        OR      TEMP1, #2000H;
        ST      TEMP1, [ERROR]
        BR      main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;                  GET_JOINT_POSITIONS PROCEDURE                    ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
get_joint_positions:

        LD      TEMP1, [PC1R_R14_CS]    ; 1st read of R14, pos. counters 1&R
        LD      TEMP1, [PC22_R14_CS]    ; 1st read of R14, pos. counters 2&2
        ST      POS_1, POS_1L           ; Store 'last' positions - low word
        ST      POS_2, POS_2L           ;   and kill > than 1.8 usec.
        ST      POS_Z, POS_ZL
        ST      POS_R, POS_RL
        LD      POS_1, [PC1R_R14_OE]    ; 2nd read of R14, pos. counters 1&R
        LD      POS_Z, [PC22_R14_OE]    ; 2nd read of R14, pos. counters 2&2
        ST      POS_1+2, POS_1L+2       ; Store 'last' positions - high word
        ST      POS_2+2, POS_2L+2

        LD      TEMP1, [PC1R_R13_CS]    ; 1st read of R13, pos. counters 1&R
        LD      TEMP1, [PC22_R13_CS]    ; 1st read of R13, pos. counters 2&2
        STB     POS_1+1, POS_R          ; Sort data and kill > 1.8 usec.
        STB     POS_Z+1, POS_2
        LD      TEMP1, [PC1R_R13_OE]    ; 2nd read of R13, pos. counters 1&R
        STB     TEMP1, POS_1+1          ; Sort data
        STB     TEMP1+1, POS_R+1
        LD      TEMP1, [PC22_R13_OE]    ; 2nd read of R13, pos. counters 2&2
        ST      POS_Z+2, POS_ZL+2       ; Store 'last' positions - high word
        ST      POS_R+2, POS_RL+2

        LD      TEMP2, [PC1R_R12_CS]    ; 1st read of R12, pos. counters 1&R
        LD      TEMP2, [PC22_R12_CS]    ; 1st read of R12, pos. counters 2&2
        STB     TEMP1, POS_Z+1          ; Sort data and kill > 1.8 usec.
        STB     TEMP1+1, POS_2+1
        LD      TEMP1, [PC1R_R12_OE]    ; 2nd read of R12, pos. counters 1&R
        STB     TEMP1, POS_1+2          ; Sort the data
        STB     TEMP1+1, POS_R+2
        LD      TEMP1, [PC22_R12_OE]    ; 2nd read of R12, pos. counters 2&2
        STB     TEMP1, POS_Z+2          ; Sort data
```

```
        STB     TEMP1+1, POS_2+2

        CLRB    POS_1+3                 ; Clear the most significant byte
        CLRB    POS_2+3
        CLRB    POS_Z+3
        CLRB    POS_R+3
        CMPB    POS_1+2, #0FFH          ; Set most sig. byte to FF if 2nd most
        JNE     pos2                    ;   significant byte is FF, i.e.,
        LDB     POS_1+3, #0FFH          ;   sign-extend to 32 bits.
pos2:   CMPB    POS_2+2, #0FFH
        JNE     posZ
        LDB     POS_2+3, #0FFH
posZ:   CMPB    POS_Z+2, #0FFH
        JNE     posR
        LDB     POS_Z+3, #0FFH
posR:   CMPB    POS_R+2, #0FFH
        JNE     pos_end
        LDB     POS_R+3, #0FFH
pos_end:
        ST      POS_1, [POS]            ; Save positions in dual-port RAM
        ST      POS_1+2, 2[POS]         ;   actual_position registers
        ST      POS_2, 4[POS]
        ST      POS_2+2, 6[POS]
        ST      POS_Z, 8[POS]
        ST      POS_Z+2, 0AH[POS]
        ST      POS_R, 0CH[POS]
        ST      POS_R+2, 0EH[POS]

        ST      N0003, [TIMING_A]       ; Signal to masterthat joint positions
                                        ;   are available
        RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;                          DELAY PROCEDURE                          ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

delay:  LD      TEMP2, #8H              ; Generate ~2 sec. delay
delay0: LD      TEMP1, #0FFFFH          ;   by counting down from FFFFH
delay1: DEC     TEMP1                   ;   8H times.
        CMP     TEMP1, 0
        JNE     delay1
        DEC     TEMP2
        CMP     TEMP2, 0
        JNE     delay0
        RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;                        ZERO_DACS PROCEDURE                        ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

zero_dacs:                      ; Reset all DACS to 0 Vdc (offset = 800H)
        LD      TEMP1, #800H
        ST      TEMP1, [DAC_1]  ; Load first rank registers of all DACS
        ST      TEMP1, [DAC_2]
        ST      TEMP1, [DAC_Z]
        ST      TEMP1, [DAC_R]
        ST      0, [DACS_OUT]   ; load 2nd rank registers of each DAC
        RET                     ;   simultaneously

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;                JOINT_OVERRUN_CHECK PROCEDURE                  ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

OVERRUN         MACRO   JOINT, MIN, MAX, NEG_OVR_FLAG, POS_OVR_FLAG, NEXT_JOINT, MSB
        LOCAL   or_a, or_b
        CMP     POS_&JOINT+2, #0FFFFH  ;; If joint is not in negative region
        JNE     or_a                   ;;   goto or_a
        CMP     POS_&JOINT, #MIN       ;; Else compare its position against
        JC      or_&NEXT_JOINT         ;;   the specified MIN.
        OR      TEMP1, #NEG_OVR_FLAG   ;;   If position is < MIN, make note of
        BR      or_&NEXT_JOINT         ;;     neg ovrrun error, then next joint
or_a:   CMP     POS_&JOINT+2, #MSB     ;; Compare 3rd byte against MSB:
```

```
        JLT     or_&NEXT_JOINT          ;;   IF position < MSB, check next joint
        JGT     or_b                    ;;   IF position > MSB, goto or_b
        CMP     POS_&JOINT, #MAX        ;;   Else compare position against MAX.
        JNH     or_&NEXT_JOINT          ;;     IF < MAX, check next joint
or_b:   OR      TEMP1, #POS_OVR_FLAG    ;; Make note of positive overrun error.
        ENDM

overrun_check:
        CLR     TEMP1
or_1:   OVERRUN 1, 0EEF7H, 0B03CH, 2H, 1H. 2, 2H
or_2:   OVERRUN 2, 0F752H, 0EB3EH, 8H, 4H, Z, 0H
or_Z:   OVERRUN Z, 0FF83H, 07585H, 20H, 10H, R, 1H
or_R:   OVERRUN R, 05F1DH, 0A0E3H, 80H, 40H, end, 0H
or_end: LD      TEMP2, [ERROR]          ; Update Error Register
        STB     TEMP1, TEMP2
        ST      TEMP2, [ERROR]
        RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;                    CHECK_TORQUES PROCEDURE                          ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

CT      MACRO   JOINT, OFFSET, NEXT_JOINT, ERROR_FLAG
        LD      TORQUE_&JOINT, OFFSET[TOR]      ;; Get torque from DPR register
        AND     TEMP1, TORQUE_&JOINT, #0F000H   ;; Mask-out all but most sig. byte
        JE      ct_&NEXT_JOINT                  ;; If result is not zero
        OR      TEMP2, #ERROR_FLAG              ;;   make a note of the error
        ENDM

check_torques:
        CLR     TEMP2
ct_1:   CT      1, 0, 2, 100H
ct_2:   CT      2, 2, Z, 200H
ct_Z:   CT      Z, 4, R, 400H
ct_R:   CT      R, 6, end, 800H
ct_end: OR      TEMP2, [ERROR]          ; Update Error Register
        ST      TEMP2, [ERROR]
        RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;                       TORQUES_OUT PROCEDURE                         ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

torques_out:
        ST      TORQUE_1, [DAC_1]       ; Load first rank registers of DACs.
        ST      TORQUE_2, [DAC_2]
        ST      TORQUE_Z, [DAC_Z]
        ST      TORQUE_R, [DAC_R]
        ST      0, [DACS_OUT]           ; Load second rank register of DACs.
        RET

        END
```

```
;*****************************************************************
; 8096.INC - DEFINITION OF SYMBOLIC NAMES FOR THE I/O REGISTERS OF THE
;       8096 AND THE 80C196
;       (C) INTEL CORPORATION 1983
;*****************************************************************
;
;/*
; *     8096 & 80C196KA SPECIAL FUNCTION REGISTERS
; */
R0              EQU   00H:WORD      ; R    CONTAINS THE VALUE 0000H
AD_COMMAND      EQU   02H:BYTE      ;   W  NOT USED BY ARC
AD_RESULT_LO    EQU   02H:BYTE      ; R     "   "   "   "
AD_RESULT_HI    EQU   03H:BYTE      ; R     "   "   "   "
HSI_MODE        EQU   03H:BYTE      ;   W   "   "   "   "
HSO_TIME        EQU   04H:WORD      ;   W   "   "   "   "
HSI_TIME        EQU   04H:WORD      ; R     "   "   "   "
HSO_COMMAND     EQU   06H:BYTE      ;   W   "   "   "   "
HSI_STATUS      EQU   06H:BYTE      ; R     "   "   "   "
SBUF            EQU   07H:BYTE      ; R/W   "   "   "   "
INT_MASK        EQU   08H:BYTE      ; R/W   "   "   "   "
INT_PENDING     EQU   09H:BYTE      ; R/W   "   "   "   "
WATCHDOG        EQU   0AH:BYTE      ;   W  WATCHDOG TIMER
TIMER1          EQU   0AH:WORD      ; R    NOT USED BY ARC
TIMER2          EQU   0CH:WORD      ; R    USED AS SAMPLE PERIOD TIMER
BAUD_RATE       EQU   0EH:BYTE      ;   W  NOT USED BY ARC
IOPORT0         EQU   0EH:BYTE      ; R     "   "   "   "
IOPORT1         EQU   0FH:BYTE      ; R/W   "   "   "   "
IOPORT2         EQU   10H:BYTE      ; R/W   "   "   "   "
SP_CON          EQU   11H:BYTE      ;   W   "   "   "   "
SP_STAT         EQU   11H:BYTE      ; R     "   "   "   "
IOC0            EQU   15H:BYTE      ;   W   "   "   "   "
IOS0            EQU   15H:BYTE      ; R     "   "   "   "
IOC1            EQU   16H:BYTE      ;   W   "   "   "   "
IOS1            EQU   16H:BYTE      ; R     "   "   "   "
PWM_CONTROL     EQU   17H:BYTE      ;   W   "   "   "   "
SP              EQU   18H:WORD      ; R/   STACK POINTER
;
;/*
; *     SPECIAL FUNCTION REGISTERS FOR 80C196 ONLY
; */
IOC2            EQU   0BH:BYTE      ;   W  PUTS TIMER2 IN FAST INCR. MODE
IPEND1          EQU   12H:BYTE      ; R/W  NOT USED BY ARC
IMASK1          EQU   13H:BYTE      ; R/W   "   "   "   "
WSR             EQU   14H:BYTE      ; R/W   "   "   "   "
IOS2            EQU   17H:BYTE      ; R     "   "   "   "
```

APPENDIX K

MASTER SOURCE CODE

```
/*****************************************************************************
 *
 * UTILITY.ROB - C PROGRAM CONTAINING ARC MAIN AND OTHER FUNCTIONS
 *
 *****************************************************************************/

#include "stdlib.h"
#include "math.h"
#include "alloc.h"
#include "stdio.h"

/* GLOBAL VARIABLES (begin with a capital letter) */
    double Duration;       /* Duration (in sec.) of the move */
    int    Home_found = 0; /* A flag which is set to 1 when home is found */

/*****************************************************************************/
main()
{
    for(;;) {
        clrscr();
        controller_menu();
    }
}

/*****************************************************************************/
int check_home_found_flag();   /* function prototypes */
prop_control();
prop_deriv_control();
seraji_decen_adapt_ctrl();
find_home();

controller_menu()              /* Display menu of available control algorithms */
{
    char ch;

    printf("Choose controller\n\n");
    printf("[1]  FIND HOME\n");
    printf("[2]  PROPORTIONAL AND DERIVATIVE CONTROL\n");
    printf("[3]  SERAJI-DECENTRALIZED ADAPTIVE CONTROL\n");
    printf("[0]  QUIT\n\n");
    printf("ENTER YOUR CHOICE: ");
    do {
        switch(ch = getch()) {
            case '1': find_home();
                      return;
            case '2': check_home_found_flag();
                      prop_deriv_control();
                      return;
            case '3': check_home_found_flag();
                      seraji_decen_adapt_ctrl();
                      return;
            case '0': clrscr();
                      exit(0);
            default : delline();
                      printf("\rINVALID CHOICE. TRY AGAIN: ");
        }
    } while(ch != '1' && ch != '2' && ch != '3' && ch != '0');
}
```

```
/*****************************************************************************/
find_home()                    /* tells slave to enter find-HOME mode */
{
    int far *command = 0xC0000006;
    unsigned int far *error = 0xC0000008;

    *error = 0x0000;   /* clear the error register */
    *command = 0x0080; /* give 'find-HOME' command */
    clrscr();
    printf("\t\t\tFINDING HOME, PLEASE WAIT");
    while(*command);   /* wait for 196 to finish the move */
    Home_found = 1;
    delline();
    printf("\r\tHome was found.  Press any key to continue ");
    while('getch());
}

/*****************************************************************************/
/* check home_found_flag and warn if home was not previously found */

check_home_found_flag()
{
    if(!Home_found) {
        printf("\r\t\t    Robot will first FIND HOME.\n\n");
        printf("\t\t       (P)roceed or (Q)uit? ");
        if(tolower(getch()) == 'p') find_home();
        else exit(0);
    }
}

/*****************************************************************************/
/* Display menu of available path planning algorithms */

long int huge *cubic_spline(double);   /* function prototypes */
long int huge *cycloid(double);
long int huge *step_function(double);

long int huge *choose_path_planner(double sample_per)
{
    char ch;

    clrscr();
    printf("\n\nChoose path planner\n\n");
    printf("[1]  CUBIC SPLINE\n");
    printf("[2]  CYCLOID\n");
    printf("[3]  STEP FUNCTION\n");
    printf("[0]  QUIT\n\n");
    printf("ENTER YOUR CHOICE: ");
    do {
        switch(ch = getche()) {
            case '1': return(cubic_spline(sample_per));
            case '2': return(cycloid(sample_per));
            case '3': return(step_function(sample_per));
            case '0': clrscr();
                      exit(0);
            default : delline();
                      printf("\rINVALID CHOICE. TRY AGAIN: ");
        }
    } while(ch != '1' && ch != '2' && ch != '3' && ch != '0');
}

/*****************************************************************************/
/* display current position and orientation of robot */

current_position()
{
    int out_of_workspace;
    long int huge *p = 0xC0010000;
    long int i_1p = *p;
    long int i_2p = *(p+1);
    long int i_zp = - *(p+2);
    long int i_rp = *(p+3);
```

```
    double i_1 = i_1p / 872.2222;
    double i_2 = i_2p / 444.4444;
    double i_z = i_zp / 380.96;
    double i_r = i_rp / 227.5556;
    double i_1r= i_1 * M_PI / 180;
    double i_2r= i_2 * M_PI / 180;

    printf("Current position:\n\n");
    printf("Cartesian space  \tJoint space\n\n");
    printf("X = %8.4lf mm.\t", 250*cos(i_1r + i_2r) + 400*cos(i_1r));
    printf("Joint 1 = %06li (%08lxH) pulses or ", i_1p, i_1p);
    printf("%10.6f deg.\n", i_1);
    printf("Y = %8.4lf mm.\t", 250*sin(i_1r + i_2r) + 400*sin(i_1r));
    printf("Joint 2 = %06li (%08lxH) pulses or ", i_2p, i_2p);
    printf("%10.6f deg.\n", i_2);
    printf("Z = %8.4lf mm.\t", i_z);
    printf("Joint z = %06li (%08lxH) pulses or ", i_zp, i_zp);
    printf("%10.6f mm.\n", i_z);
    printf("R = %8.4lf deg.\t", i_r);
    printf("Joint R = %06li (%08lxH) pulses or ", i_rp, i_rp);
    printf("%10.6f de  .\n\n", i_r);
}

/*****************************************************************************/
/* prompt for the entry of a cartesian point. Verify that points are
 * within robot workspace. perform inverse kinematics to find joint angles */

get_cartesian_position(p)
long int *p;
{
    int out_of_workspace;
    double w, px, py, pz, pr, j1, j2;
    do {
        do {
            printf("\n\tEnter x (in mm.): ");
            scanf("%lf", &px);
            if(px<-650 || px>650) printf("\trange of x: -650 to 650 mm.\n");
        } while( px < -650 || px > 650 );
        do {
            printf("\n\tEnter y (in mm.): ");
            scanf("%lf", &py);
            if(py<-386 || py>650) printf("\trange of y: -386 to 650 mm.\n");
        } while( py < -386 || py > 650 );
        w = (px * px + py * py + 97500.0)/800.0;
        if(px*px+py*py >= w*w) {
            j1 = atan2(py,px)+atan2( -sqrt(px*px+py*py-w*w),w); /* inv. kin. */
            if(j1 < 0)  j1 = j1 + 2 * M_PI;
            j2 = atan2(-px*sin(j1)+py*cos(j1), px*cos(j1)+py*sin(j1)-400);
            *p = (long int)(j1 * 180 / M_PI * 872.2222 + 0.5);
            *(p+1)    .long int)(j2 * 180 / M_PI * 444.4444 + 0.5);
        }
        else {          /* out of workspace */
            *p = -1;
            *(p+1) = -1;
        }
        out_of_workspace = (*p<0 || *p>174445 || *(p+1)<0 || *(p+1) >60001);
        if(out_of_workspace) printf("\n\tOUT OF WORKSPACE IN X-Y PLANE\n");
    } while(out_of_workspace);
        do {
            printf("\n\tEnter z (in mm.): ");
            scanf("%lf", &pz);
            if(pz < -250 || pz > 0) printf("\trange of z: -250 to 0 mm.\n");
        } while( pz < -250 || pz > 0 );
        *(p+2) = (long int)(-pz * 380.96 + 0.5);
        do {
            printf("\n\tEnter r (in deg.): ");
            scanf("%lf", &pr);
            if(pr < -180 || pr > 180) printf("\trange of r: -180 to 180 deg.\n");
        } while( pr < -180 || pr > 180 );
        *(p+3) = (long int)(pr * 227.5556 + 0.5);
        printf("\n\t Equivalent joint-space destination:\n");
        printf("\tjoint 1 = %li\n", *p);
```

```
        printf("\tjoint 2 = %li\n", *(p+1));
        printf("\tjoint z = %li\n", *(p+2));
        printf("\tjoint r = %li\n", *(p+3));
}

/*****************************************************************************/
/* prompt for the entry of joint positions */

get_joint_position(p)
long int *p;
{
    double jv;

    do {
        printf("\n\tEnter joint 1 (in degrees): ");
        scanf("%lf", &jv);
        if(jv<0||jv>200) printf("\tjoint 1 range: 0 to 200 degrees\n");
    } while(jv<0||jv>200);
    *p = (long int)(jv * 872.2222 + 0.5);
    do {
        printf("\n\tEnter joint 2 (in degrees): ");
        scanf("%lf", &jv);
        if(jv<0||jv>135) printf("\tjoint 2 range: 0 to 135 degrees\n");
    } while(jv<0||jv>135);
    *(p+1) = (long int)(jv * 444.4444 + 0.5);
    do {
        printf("\n\tEnter joint z (in mm.): ");
        scanf("%lf", &jv);
        if(jv<-250||jv>0) printf("\tjoint z range: -250 to 0 mm.\n");
    } while(jv<-250||jv>0);
    *(p+2) = (long int)(-jv * 380.96);
    do {
        printf("\n\tEnter joint r (in degrees): ");
        scanf("%lf", &jv);
        if(jv<-180||jv>180) printf("\tjoint r range: -180 to 180 deg.\n");
    } while(jv<-180||jv>180);
    *(p+3) = (long int)(jv * 227.5556);
}

/*****************************************************************************/
extern double Duration;
get_time(p)
double *p;
{
    printf("\n\tLength of time (in sec.) to reach this point: ");
    do {
        scanf("%lf",p);
        if(*p <= *(p-1) || *p > Duration) {
            printf("Must be greater than that of the ");
            printf("previous point and less than the ");
            printf("Duration.\nReenter: ");
        }
    } while(*p <= *(p-1) || *p > Duration);
}

/*****************************************************************************/
/* check for joint overuns */

robot_error_check()
{
    int far *error_pointer = 0xC0000008;
    int error;

    *error_pointer = (*error_pointer) & (0x00ff);
    if(*error_pointer) {
        do {
            error = *error_pointer;
            clrscr();
            printf("\t\t\tROBOT ERRORS\n\n");
            if(error & 0x0001) printf("Joint 1 in positive overun\n");
            if(error & 0x0002) printf("Joint 1 in negative overun\n");
            if(error & 0x0004) printf("Joint 2 in positive overun\n");
```

```
            if(error & 0x0008) printf("Joint 2 in negative overun\n");
            if(error & 0x0010) printf("Joint z in positive overun\n");
            if(error & 0x0020) printf("Joint z in negative overun\n");
            if(error & 0x0040) printf("Joint r in positive overun\n");
            if(error & 0x0080) printf("Joint r in negative overun\n");
            printf("\nMANUALLY MOVE THESE JOINTS BACK INTO THE WORKSPACE");
            while(error == *error_pointer);
        } while(*error_pointer);
        printf("\n\nAll overun errors fixed.  Press any key to continue ");
        while(!getch());
        return;
    }
}

/*****************************************************************************/
/* display errors */

display_errors()
{
    unsigned int far *error = 0xC0000008;
    unsigned int temp;

    temp = *error;
    if(temp & 0x0001) printf("\tjoint 1 entered positive overun\n");
    if(temp & 0x0002) printf("\tjoint 1 entered negative overun\n");
    if(temp & 0x0004) printf("\tjoint 2 entered positive overun\n");
    if(temp & 0x0008) printf("\tjoint 2 entered negative overun\n");
    if(temp & 0x0010) printf("\tjoint z entered positive overun\n");
    if(temp & 0x0020) printf("\tjoint z entered negative overun\n");
    if(temp & 0x0040) printf("\tjoint r entered positive overun\n");
    if(temp & 0x0080) printf("\tjoint r entered negative overun\n");
    if(temp & 0x0100) printf("\tToo much torque was applied to joint 1\n");
    if(temp & 0x0200) printf("\tToo much torque was applied to joint 2\n");
    if(temp & 0x0400) printf("\tToo much torque was applied to joint z\n");
    if(temp & 0x0800) printf("\tToo much torque was applied to joint r\n");
    if(temp & 0x1000) printf("\tPS/2 did not provide torques in time\n");
    if(temp & 0x2000) printf("\tPS/2 did not finish its loop in time\n");
}

/*****************************************************************************/
/* compare two numbers (double precision).
 * return 0 if their signs are the same.
 * return 1 if their signs are different or if at least one number is zero. */

cmp_sign(double num_1, double num_2)
{
    if(num_1 && num_2){  /* if neither number is zero */
        if( (num_1 > 0 && num_2 > 0) || (num_1 < 0 && num_2 < 0) ) return 0;
    }
    else return 1;
}

/*****************************************************************************/
/* change the value of a gain or parameter */

double choose_new_gain(char *name, double gain)
{
        printf("Change %s? [y/n]: ", name);
        if(getch() == 'y') {
                printf("\tFrom %lf to: ", gain);
                scanf("%lf", &gain);
        }
        else printf("\n");

        return(gain);
}
```

```
/*****************************************************************************
 *
 * Proportional, Derivative (PD) Control Algorithm
 *
 *****************************************************************************/

#include "alloc.h"
#include "stdio.h"

long int huge *choose_path_planner(double);  /* function prototypes */
double choose_new_gain(char *name, double gain);

extern double Duration; /* Global variable declared in utility.rob */

prop_deriv_control()
{
    static double kp_1 = 5, kp_2 = 7, kp_z = 5, kp_r = 5;  /* controller gains */
    static double kv_1 = .02, kv_2 = .02, kv_z = .02, kv_r = .02;
    double kv1, kv2, kvz, kvr;
    double e_1=0, e_2=0, e_z=0, e_r=0;        /* joint position error */
    double el_1, el_2, el_z, el_r;     /* previous (last) joint position error */
    static double sample_per;          /* sample_period */
    char ch;
    int not_enough_memory;
    long int huge *pp;  /* pointer to array of desired path points */
    long int huge *s_pp; /* a place to save this pointer */
    unsigned int far *sample_timer = 0xC0000000; /* pointer to DPR sample_period Reg. */
    double num_samples;
    unsigned int num_captures;
    unsigned int capture_counter = 0;
    unsigned int capture_intrvl;
    int far *error = 0xC0000008;         /* pointer to DPR error Reg. */
    long int far *act_pos_1 = 0xC0000010; /* pointer to DPR actual pos. Regs. */
    long int far *act_pos_2 = 0xC0000014;
    long int far *act_pos_z = 0xC0000018;
    long int far *act_pos_r = 0xC000001C;
    int far *t_1 = 0xC0000020;           /* pointer to DPR torque Regs. */
    int far *t_2 = 0xC0000022;
    int far *t_z = 0xC0000024;
    int far *t_r = 0xC0000026;
    int far *timing_a = 0xC0000002;    /* pointer to DPR timing_A reg. */
    int far *timing_b = 0xC0000004;    /* pointer to DPR timing_B reg. */
    int far *command  = 0xC0000006;    /* pointer to DPR command reg. */
    long int *c_a_1;    /* pointers to arrays of sampled joint positions */
    long int *s_c_a_1; /* a => actual, d => desired */
    long int *c_d_1;
    long int *s_c_d_1;
    long int *c_a_2;
    long int *s_c_a_2;
    long int *c_d_2;
    long int *s_c_d_2;
    long int *c_a_z;
    long int *s_c_a_z;
    long int *c_d_z;
    long int *s_c_d_z;
    long int *c_a_r;
    long int *s_c_a_r;
    long int *c_d_r;
    long int *s_c_d_r;
    int *c_t_1;                 /* pointers to arrays of sampled torques */
    int *s_c_t_1;
    int *c_t_2;
    int *s_c_t_2;
    int *c_t_z;
    int *s_c_t_z;
    int *c_t_r;
    int *s_c_t_r;
    int i, counter = 0;
    FILE *fp;
    typedef struct{     /* header for MATLAB file */
        long type;
        long mrows;
```

```
        long ncols;
        long imagf;
        long namlen;
    } Fmatrix;
    char *data_file_name="xxxx";
    Fmatrix x;

/* DISPLAY THE GAINS */
    clrscr();
    printf("\t\t\tPROPORTIONAL + DERIVATIVE CONTROL\n\n");
    printf("The error gains are:  Kp_1 = %lf\n", kp_1);
    printf("                      Kp_2 = %lf\n", kp_2);
    printf("                      Kp_z = %lf\n", kp_z);
    printf("                      Kp_r = %lf\n\n", kp_r);
    printf("                      Kv_1 = %lf\n", kv_1);
    printf("                      Kv_2 = %lf\n", kv_2);
    printf("                      Kv_z = %lf\n", kv_z);
    printf("                      Kv_r = %lf\n\n", kv_r);

/* CHANGING THE GAINS PROMPT */
    printf("Change any of the gains? (y/n): ");
    if(getch() == 'y') {
        printf("\n\n");
        kp_1 = choose_new_gain("Kp_1", kp_1);
        kp_2 = choose_new_gain("Kp_2", kp_2);
        kp_z = choose_new_gain("Kp_z", kp_z);
        kp_r = choose_new_gain("Kp_r", kp_r);
        printf("\n");
        kv_1 = choose_new_gain("Kv_1", kv_1);
        kv_2 = choose_new_gain("Kv_2", kv_2);
        kv_z = choose_new_gain("Kv_z", kv_z);
        kv_r = choose_new_gain("Kv_r", kv_r);
    }

/* DISPLAY CURRENT SAMPLING PERIOD AND PROMPT FOR A CHANGE */
    sample_per = (double)(*sample_timer)/1000000;
    printf("\n\nThe sampling period is: %lf sec.\tChange? [y/n]: ",sample_per);
    if(getch() == 'y') {
        printf("\n\nEnter new sample period (in sec.): ");
        scanf("%lf", &sample_per);
        *sample_timer = (unsigned int)(sample_per * 1000000);
        printf("sample_timer = %x \n", *sample_timer);
        printf("press any key to continue ");
        while(!getch());
    }

/* CHECK ROBOT FOR OVER RUN ERRORS */
    robot_error_check();

/* GET TRAJECTORY */
    pp = choose_path_planner(sample_per);
    s_pp = pp;

/* ALLOCATE MEMORY FOR DATA CAPTURE
 * Data from at least 1000 equally spaced samples are captured */
    num_samples = Duration / sample_per;
    if(num_samples < 1000) capture_intrvl = 1;
    else capture_intrvl = (unsigned int)(num_samples/1000);
    num_captures = (unsigned int)(num_samples/capture_intrvl);
    if(!(c_a_1=s_c_a_1=(long int *)malloc(num_captures * 4 + 20))){
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }
    if(!(c_d_1=s_c_d_1=(long int *)malloc(num_captures * 4 + 20))){
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }
    if(!(c_a_2=s_c_a_2=(long int *)malloc(num_captures * 4 + 20))){
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }
    if(!(c_d_2=s_c_d_2=(long int *)malloc(num_captures * 4 + 20))){
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }
    if(!(c_a_z=s_c_a_z=(long int *)malloc(num_captures * 4 + 20))){
```

```
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }
    if(!(c_d_z=s_c_d_z=(long int *)malloc(num_cap ures * 4 + 20))){
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }
    if(!(c_a_r=s_c_a_r=(long int *)malloc(num_captures * 4 + 20))){
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }
    if(!(c_d_r=s_c_d_r=(long int *)malloc(num_captures * 4 + 20))){
        printf("Not enough memory availab'e.  Aborting\n\n");
        goto terminate; }
    if(!(c_t_1 = s_c_t_1 = (int *)malloc(num_captures * 2 + 20))){
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }
    if('(c_t_2 = s_c_t_2 = (int *)malloc(num_captures * 2 + 20))){
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }
    if(!(c_t_z = s_c_t_z = (int *)malloc(num_captures * 2 + 20))){
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }
    if(!(c_t_r = s_c_t_r = (int *)malloc(num_captures * 2 + 20))){
        printf("Not enough memory available.  Aborting\n\n");
        goto terminate; }

/* PROMPT TO BEGIN EXECUTION */
    delline();
    printf("\r(A)bort or (B)egin the move? \n");
    do {
        switch(ch = tolower(getch())) {
            case 'a':  goto terminate;
            case 'b':  break;   /* continue */
            default :  delline();
                       printf("\rINVALID CHOICE. TRY AGAIN: ");
        }
    } while(ch != 'a' && ch != 'b');

/* BEGINNING OF THE MOVE */
    clrscr();
    kv1 = kv_1 / sample_per;
    kv2 = kv_2 / sample_per;
    kvz = kv_z / sample_per;
    kvr = kv_r / sample_per;
    printf("\t\t\tEXECUTING");
    *command = 0x0001;          /* have 196 begin its i_o loop */
    while( (*pp != 0xffff0000) && (*error == 0) ) {

    /* wait to for start of sampling period */
        while('(*timing_a & 0x0001));

    /* update joint position errors */
        el_1 = e_1, el_2 = e_2, el_z = e_z, el_r = e_r;

    /* wait for joint positions */
        while(!(*timing_a & 0x0002));

    /* compute joint torques */
        *c_t_1 = *t_1 = (int)(0x0800 + kp_1*(e_1 = ((*c_d_1 = *pp++) - (*c_a_1 =
*act_pos_1))) + kv1*(e_1-el_1));
        *c_t_2 = *t_2 = (int)(0x0800 + kp_2*(e_2 = ((*c_d_2 = *pp++) - (*c_a_2 =
*act_pos_2))) + kv2*(e_2-el_2));
        *c_t_z = *t_z = (int)(0x0800 + kp_z*(e_z = ((*c_d_z = *pp++) - (*c_a_z =
*act_pos_z))) + kvz*(e_z-el_z));
        *c_t_r = *t_r = (int)(0x0800 + kp_r*(e_r = ((*c_d_r = *pp++) - (*c_a_r =
*act_pos_r))) + kvr*(e_r-el_r));
        *timing_b = 0x0001;  /* tell 196 that torques are ready */

    /* capture data every capture interval */
        if(!capture_counter) {
            c_d_1++;
            c_d_2++;
            c_d_z++;
            c_d_r++;
```

```
            c_a_1++;
            c_a_2++;
            c_a_z++;
            c_a_r++;
            c_t_1++;
            c_t_2++;
            c_t_z++;
            c_t_r++;
            counter++;
            capture_counter = capture_intrvl - 1;
        }
        else {
            capture_counter--;
        }

    /* indicate end of sample period */
        *timing_a = 0x0000;
    }

/* INDICATE END OF COMMAND */
    *command = 0x0000;
    *timing_b = 0x0000;

/* DISPLAY SUCCESS OF EXECUTION */
    delline();
    if(*error) {
        printf("\n\n\tCommand terminated prematurely because:\n\n");
        display_errors();
    }
    else {
        printf("\n\n\t\t\tCommand executed normally.\n\n");
        printf("\t\tSaving %i data samples to disk.  Please wait.\n\n", counter);
        c_a_1 = s_c_a_1; /* reset the pointers */
        c_d_1 = s_c_d_1;
        c_a_2 = s_c_a_2;
        c_d_2 = s_c_d_2;
        c_a_z = s_c_a_z;
        c_d_z = s_c_d_z;
        c_a_r = s_c_a_r;
        c_d_r = s_c_d_r;
        c_t_1 = s_c_t_1;
        c_t_2 = s_c_t_2;
        c_t_z = s_c_t_z;
        c_t_r = s_c_t_r;
/* open file for saving to disk */
        if((fp=fopen("robot.mat","wb"))==NULL) {
            printf("cannot open file. Aborting\n");
            goto terminate;
        }
        x.imagf = 0;
        x.namlen = 5;
        x.mrows = counter;
/* save a vector called 'tor1' (joint 1 torque) to disk */
        x.type = 30;    /* data is signed int type */
        x.ncols = 1;
        data_file_name = "tor1";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_t_1, 2, counter, fp);
/* save a vector called 'tor2' (joint 2 torque) to disk */
        data_file_name = "tor2";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_t_2, 2, counter, fp);
/* save a vector called 'torz' (joint z torque) to disk */
        data_file_name = "torz";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_t_z, 2, counter, fp);
/* save a vector called 'torr' (joint r torque) to disk */
        data_file_name = "torr";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
```

```
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_t_r, 2, counter, fp);
/* save sample period and capture interval to disk */
        x.type = 0;     /* data is double precision */
        x.mrows = 1;
        data_file_name = "sper";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(&sample_per, 8, 1, fp);
        x.type = 40;    /* data is unsigned 16-bit */
        data_file_name = "intv";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(&capture_intrvl, 2, 1, fp);
/* save a matrix called 'pos1' (joint 1 position) to disk
 * column 1 = desired position of joint 1
 * column 2 = actual position of joint 1 */
        x.type = 20;    /* data is long int type */
        x.ncols = 2;
        x.mrows = counter;
        data_file_name = "pos1";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen, fp);
        fwrite(c_d_1, 4, counter, fp);
        fwrite(c_a_1, 4, counter, fp);
/* save a matrix called 'pos2' (joint 2 position) to disk
 * column 1 = desired position of joint 2
 * column 2 = actual position of joint 2 */
        data_file_name = "pos2";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen, fp);
        fwrite(c_d_2, 4, counter, fp);
        fwrite(c_a_2, 4, counter, fp);
/* save a matrix called 'posz' (joint z position) to disk
 * column 1 = desired position of joint z
 * column 2 = actual position of joint z */
        data_file_name = "posz";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen, fp);
        fwrite(c_d_z, 4, counter, fp);
        fwrite(c_a_z, 4, counter, fp);
/* save a matrix called 'posr' (joint 2 position) to disk
 * column 1 = desired position of joint r
 * column 2 = actual position of joint r */
        data_file_name = "posr";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen, fp);
        fwrite(c_d_r, 4, counter, fp);
        fwrite(c_a_r, 4, counter, fp);

        fclose(fp);
        }

/* FREE ALLOCATED MEMORY */
terminate:
    if(s_pp) farfree(s_pp);
    if(s_c_a_1) free(s_c_a_1);
    if(s_c_d_1) free(s_c_d_1);
    if(s_c_a_2) free(s_c_a_2);
    if(s_c_d_2) free(s_c_d_2);
    if(s_c_a_z) free(s_c_a_z);
    if(s_c_d_z) free(s_c_d_z);
    if(s_c_a_r) free(s_c_a_r);
    if(s_c_d_r) free(s_c_d_r);
    if(s_c_t_1) free(s_c_t_1);
    if(s_c_t_2) free(s_c_t_2);
    if(s_c_t_z) free(s_c_t_z);
    if(s_c_t_r) free(s_c_t_r);
    printf("\t\t\tPress any key to continue ");
    while(!getch());
    return;
}
```

```
/*****************************************************************************
 *
 * SERAJI.ROB - Decentralized Adaptive Control Algorithm */
 * From Trans. Robotics & Automation, April 1989, Vol. 5, No. 2, pp. 183-201
 *
 *****************************************************************************/

#include "alloc.h"
#include "stdio.h"

long int huge *choose_path_planner(double);  /* function prototypes */
double choose_new_gain(char *name, double gain);

extern double Duration;  /* Global variable declared in utility.rob */

seraji_decen_adapt_ctrl()
{
    double kp_1=0, kp_2=0, kp_z=0, kp_r=0;            /* position gains */
    double kp_11,  kp_21,  kp_z1,  kp_r1;             /* last "     "   */
    double kv_1=0, kv_2=0, kv_z=0, kv_r=0;            /* velocity gains */
    double kv_11,  kv_21,  kv_z1,  kv_r1;             /* last "     "   */
    double e_1 =0, e_2 =0, e_z =0, e_r =0;            /* position errors */
    double e_11,   e_21,   e_z1,   e_r1=0;            /* last "     "    */
    double edot_1=0, edot_2=0, edot_z=0, edot_r=0;    /* velocity errors */
    double edot_11,  edot_21,  edot_z1,  edot_r1;     /* last "     "    */
    double r_1=0,  r_2=0,  r_z=0,  r_r=0;             /* weighted error */
    double r_11,   r_21,   r_z1,   r_r1;              /* last "     "   */
    double f_1=20,f_2=20,f_z=20,  f_r = 20;           /* auxiliary signals */
    double f_11,   f_21,   f_z1,   f_r1;              /* last "       "    */
    static double wp_1=80, wp_2=8, wp_z=1, wp_r=1; /* weighting factors */
    static double wv_1=40, wv_2=2,  wv_z=1, wv_r=1;
    static double d_1= 175,d_2= 175,d_z= 175,d_r= 175; /* integ. adapt gains */
    static double a0_1=350,a0_2=350,a0_z=350,a0_r=350;
    static double a1_1=8, a1_2=8, a1_z=8,  a1_r=8;
    static double rho_1=0, rho_2=0, rho_z=0, rho_r=0; /* prop. adapt. gains */
    static double b0_1= 0, b0_2= 0, b0_z= 0, b0_r= 0;
    static double b1_1= 0, b1_2= 0, b1_z= 0, b1_r= 0;
    double d_1spo2,  d_2spo2,  d_zspo2,  d_rspo2;
    double a0_1spo2, a0_2spo2, a0_zspo2, a0_rspo2;
    double a1_1spo2, a1_2spo2, a1_zspo2, a1_rspo2;
    static double sample_per;  /* sample period */
    double spo2;
    char ch;
    int not_enough_memory;
    long int huge *pp;  /* pointer to array of desired joint positions */
    long int huge *s_pp; /* a place to save the value of this pointer */
    unsigned int far *sample_timer = 0xC0000000; /* ptr to DPR sample_period Reg. */
    double num_samples;
    unsigned int num_captures;
    unsigned int capture_counter = 0;
    unsigned int capture_intrvl;
    int far *error = 0xC0000008;
    long int far *act_pos_1 = 0xC0000010;  /* pointers to DPR actual position Regs. */
    long int far *act_pos_2 = 0xC0000014;
    long int far *act_pos_z = 0xC0000018;
    long int far *act_pos_r = 0xC000001C;
    int far *t_1 = 0xC0000020; /* pointers to DPR torque registers */
    int far *t_2 = 0xC0000022;
    int far *t_z = 0xC0000024;
    int far *t_r = 0xC0000026;
    int far *timing_a = 0xC0000002; /* pointer to DPR timing_A register */
    int far *timing_b = 0xC0000004; /* pointer to DPR timing_B register */
    int far *command  = 0xC0000006; /* pointer to DPR command register */
    long int *c_a_1;          /* pointers to arrays of sampled joint positions */
    long int *s_c_a_1;        /* a => actual, d => desired */
    long int *c_d_1;
    long int *s_c_d_1;
    long int *c_a_2;
    long int *s_c_a_2;
    long int *c_d_2;
    long int *s_c_d_2;
    long int *c_a_z;
```

```
    long int *s_c_a_z;
    long int *c_d_z;
    long int *s_c_d_z;
    long int *c_a_r;
    long int *s_c_a_r;
    long int *c_d_r;
    long int *s_c_d_r;
    int *c_t_1;                         /* pointers to arrays of sampled torques */
    int *s_c_t_1;
    int *c_t_2;
    int *s_c_t_2;
    int *c_t_z;
    int *s_c_t_z;
    int *c_t_r;
    int *s_c_t_r;
    int i, counter = 0;
    FILE *fp;
    typedef struct{     /* MATLAB header information */
        long type;
        long mrows;
        long ncols;
        long imagf;
        long namlen;
    } Fmatrix;
    char *data_file_name = "????";
    Fmatrix x;

/* DISPLAY PARAMETERS */
    clrscr();
    printf("\t\tSERAJI DECENTRALIZED ADAPTIVE CONTROL\n\n");
    printf("Joint\t Weighting Factors    Integral Adaptation Gains    ");
    printf("Prop Adaptation Gains");
    printf("  #\t     Wp\t    Wv\t\tdelta  alpha0  alpha1\t      rho");
    printf("  beta0  beta1\n\n");
     printf("   1\t     %05.2f   %05.2f\t%05.2f    %05.2f    %05.2f\t      %05.2f %05.2f
%05.2f\t",
        wp_1, wv_1, d_1, a0_1, a1_1, rho_1, b0_1, b1_1);
     printf("   2\t     %05.2f   %05.2f\t%05.2f    %05.2f    %05.2f\t      %05.2f %05.2f
%05.2f\n",
        wp_2, wv_2, d_2, a0_2, a1_2, rho_2, b0_2, b1_2);
     printf("   z\t     %05.2f   %05.2f\t%05.2f    %05.2f    %05.2f\t      %05.2f %05.2f
%05.2f\n",
        wp_z, wv_z, d_z, a0_z, a1_z, rho_z, b0_z, b1_z);
     printf("   r\t     %05.2f   %05.2f\t%05.2f    %05.2f    %05.2f\t      %05.2f %05.2f
%05.2f\n",
        wp_r, wv_r, d_r, a0_r, a1_r, rho_r, b0_r, b1_r);

/* CHANGE PARAMETER PROMPT */
    printf("\nChange any parameters? (y/n): ");
    if(getch() == 'y') {
        printf("\n\n");
        printf("Change Weighting Factors? (y/n): ");
        if(getch() == 'y') {
            printf("\n\n");
            wp_1 = choose_new_gain("wp_1", wp_1);
            wp_2 = choose_new_gain("wp_2", wp_2);
            wp_z = choose_new_gain("wp_z", wp_z);
            wp_r = choose_new_gain("wp_r", wp_r);
            printf("\n");
            wv_1 = choose_new_gain("wv_1", wv_1);
            wv_2 = choose_new_gain("wv_2", wv_2);
            wv_z = choose_new_gain("wv_z", wv_z);
            wv_r = choose_new_gain("wv_r", wv_r);
        }
        printf("\n");
        printf("Change Integral Adaptation Gains? (y/n): ");
        if(getch() == 'y') {
            printf("\n\n");
            d_1 = choose_new_gain("delta_1", d_1);
            d_2 = choose_new_gain("delta_2", d_2);
            d_z = choose_new_gain("delta_z", d_z);
            d_r = choose_new_gain("delta_r", d_r);
```

```
            printf("\n");
            a0_1 = choose_new_gain("alpha0_1", a0_1);
            a0_2 = choose_new_gain("alpha0_2", a0_2);
            a0_z = choose_new_gain("alpha0_z", a0_z);
            a0_r = choose_new_gain("alpha0_r", a0_r);
            printf("\n");
            a1_1 = choose_new_gain("alpha1_1", a1_1);
            a1_2 = choose_new_gain("alpha1_2", a1_2);
            a1_z = choose_new_gain("alpha1_z", a1_z);
            a1_r = choose_new_gain("alpha1_r", a1_r);
        }
        printf("\n");
        printf("Change Proportional Adaptation Gains? (y/n): ");
        if(getch() == 'y') {
            printf("\n\n");
            rho_1 = choose_new_gain("rho_1", rho_1);
            rho_2 = choose_new_gain("rho_2", rho_2);
            rho_z = choose_new_gain("rho_z", rho_z);
            rho_r = choose_new_gain("rho_r", rho_r);
            printf("\n");
            b0_1 = choose_new_gain("beta0_1", b0_1);
            b0_2 = choose_new_gain("beta0_2", b0_2);
            b0_z = choose_new_gain("beta0_z", b0_z);
            b0_r = choose_new_gain("beta0_r", b0_r);
            printf("\n");
            b1_1 = choose_new_gain("beta1_1", b1_1);
            b1_2 = choose_new_gain("beta1_2", b1_2);
            b1_z = choose_new_gain("beta1_z", b1_z);
            b1_r = choose_new_gain("beta1_r", b1_r);
        }
    }

/* DISPLAY INITIAL VALUES OF THE AUXILIARY SIGNALS */
    printf("\n\nAuxiliary signal 1 initially %lf\n", f_1);
    printf("    \"       \"    2    \"     %lf\n", f_2);
    printf("    \"       \"    z    \"     %lf\n", f_z);
    printf("    \"       \"    r    \"     %lf\n", f_r);
    printf("\nChange? (y/n): ");
    if(getch() == 'y') {
        printf("\n\n");
        f_1 = choose_new_gain("aux. sig. 1", f_1);
        f_2 = choose_new_gain("aux. sig. 2", f_2);
        f_z = choose_new_gain("aux. sig. z", f_z);
        f_r = choose_new_gain("aux. sig. r", f_r);
    }

/* DISPLAY CURRENT SAMPLING PERIOD AND PROMPT FOR A CHANGE */
    sample_per = (double)(*sample_timer)/1000000;
    printf("\n\nThe sampling period is: %lf sec \tChange? [y/n]: ",sample_per);
    if(getch() == 'y') {
        printf("\n\nEnter new sample period (in sec.): ");
        scanf("%lf", &sample_per);
        *sample_timer = (unsigned int)(sample_per * 1000000);
        printf("sample_timer = %x \n", *sample_timer);
        printf("press any key to continue ");
        while(!getch());
    }

/* CHECK ROBOT FOR OVER RUN ERRORS */
    robot_error_check();

/* GET TRAJECTORY */
    pp = choose_path_planner(sample_per);
    s_pp = pp;

/* ALLOCATE MEMORY FOR DATA CAPTURE
 * Data from at least 1000 equally spaced samples are captured */
    num_samples = Duration / sample_per;
    if(num_samples < 1000) {
        capture_intrvl = 1; }
    else {
        capture_intrvl = (unsigned int)(num_samples/1000);}
```

```
    num_captures = (unsigned int)(num_samples/capture_intrvl);
    if(!(c_a_1=s_c_a_1=(long int *)malloc(num_captures * 4 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_d_1=s_c_d_1=(long int *)malloc(num_captures * 4 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_a_2=s_c_a_2=(long int *)malloc(num_captures * 4 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_d_2=s_c_d_2=(long int *)malloc(num_captures * 4 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_a_z=s_c_a_z=(long int *)malloc(num_captures * 4 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_d_z=s_c_d_z=(long int *)malloc(num_captures * 4 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_a_r=s_c_a_r=(long int *)malloc(num_captures * 4 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_d_r=s_c_d_r=(long int *)malloc(num_captures * 4 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_t_1 = s_c_t_1 = (int *)malloc(num_captures * 2 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_t_2 = s_c_t_2 = (int *)malloc(num_captures * 2 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_t_z = s_c_t_z = (int *)malloc(num_captures * 2 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }
    if(!(c_t_r = s_c_t_r = (int *)malloc(num_captures * 2 + 20))){
       printf("Not enough memory available.  Aborting\n\n");
       goto terminate; }

/* PROMPT TO BEGIN EXECUTION */
    delline();
    printf("\r(A)bort or (B)egin the move? \n");
    do {
        switch(ch = tolower(getch())) {
            case 'a':   goto terminate;
            case 'b':   break;   /* continue */
            default :   delline();
                        printf("\rINVALID CHOICE. TRY AGAIN: ");
        }
    } while(ch != 'a' && ch != 'b');

/* BEGINNING OF THE MOVE */
    clrscr();
    printf("\t\t\tEXECUTING");
    spo2 = sample_per / 2;
    if(*(pp+28) - *(pp+0) < 0) f_1 = -f_1;
    if(*(pp+29) - *(pp+1) < 0) f_2 = -f_2;
    if(*(pp+30) - *(pp+2) < 0) f_z = -f_z;
    if(*(pp+31) - *(pp+3) < 0) f_r = -f_r;
    d_1spo2 = d_1 * sample_per / 2;
    d_2spo2 = d_2 * sample_per / 2;
    d_zspo2 = d_z * sample_per / 2;
    d_rspo2 = d_r * sample_per / 2;
    a0_1spo2= a0_1* sample_per / 2;
    a0_2spo2= a0_2* sample_per / 2;
    a0_zspo2= a0_z* sample_per / 2;
    a0_rspo2= a0_r* sample_per / 2;
    a1_1spo2= a1_1* sample_per / 2;
    a1_2spo2= a1_2* sample_per / 2;
    a1_zspo2= a1_z* sample_per / 2;
    a1_rspo2= a1_r* sample_per / 2;
    *command = 0x0001;          /* have 196 begin its i_o loop */
```

```
while( (*pp != 0xffff0000) && (*error == 0) ) {

/* wait for signal from slave to begin a new sampling period */
    while(!(*timing_a & 0x0001));

/* store current parameters as previous (last) parameters */
    f_1l = f_1, f_2l = f_2, f_zl = f_z, f_rl = f_r;
    r_1l = r_1, r_2l = r_2, r_zl = r_z, r_rl = r_r;
    kp_1l = kp_1, kp_2l = kp_2, kp_zl = kp_z, kp_rl = kp_r;
    kv_1l = kv_1, kv_2l = kv_2, kv_zl = kv_z, kv_rl = kv_r;
    e_1l = e_1, e_2l = e_2, e_zl = e_z, e_rl = e_r;
    edot_1l = edot_1, edot_2l = edot_2, edot_zl = edot_z, edot_rl = edot_r;

/* wait for position */
    while(!(*timing_a & 0x0002));

/* compute joint position error in degrees and capture robot state */
    e_1 = ((*c_d_1 = *pp++) - (*c_a_1 = *act_pos_1))/872.2222;
    e_2 = ((*c_d_2 = *pp++) - (*c_a_2 = *act_pos_2))/444.4444;
    e_z = ((*c_d_z = *pp++) - (*c_a_z = *act_pos_z))/380.9600;
    e_r = ((*c_d_r = *pp++) - (*c_a_r = *act_pos_r))/227.5555;

/* compute joint velocity error */
    edot_1 = (e_1 - e_1l)/sample_per;
    edot_2 = (e_2 - e_2l)/sample_per;
    edot_z = (e_z - e_zl)/sample_per;
    edot_r = (e_r - e_rl)/sample_per;

/* compute weighting errors */
    r_1 = wp_1*e_1 + wv_1*edot_1;
    r_2 = wp_2*e_2 + wv_2*edot_2;
    r_z = wp_z*e_z + wv_z*edot_z;
    r_r = wp_r*e_r + wv_r*edot_r;

/* compute auxiliary signals */
    f_1 = f_1l + d_1spo2*(r_1+r_1l) + rho_1*(r_1-r_1l);
    f_2 = f_2l + d_2spo2*(r_2+r_2l) + rho_2*(r_2-r_2l);
    f_z = f_zl + d_zspo2*(r_z+r_zl) + rho_z*(r_z-r_zl);
    f_r = f_rl + d_rspo2*(r_r+r_rl) + rho_r*(r_r-r_rl);

/* compute gains */
    kp_1 = kp_1l + a0_1spo2*(r_1*e_1 + r_1l*e_1l) +
           b0_1*(r_1*e_1 - r_1l*e_1l);
    kp_2 = kp_2l + a0_2spo2*(r_2*e_2 + r_2l*e_2l) +
           b0_2*(r_2*e_2 - r_2l*e_2l);
    kp_z = kp_zl + a0_zspo2*(r_z*e_z + r_zl*e_zl) +
           b0_z*(r_z*e_z - r_zl*e_zl);
    kp_r = kp_rl + a0_rspo2*(r_r*e_r + r_rl*e_rl) +
           b0_r*(r_r*e_r - r_rl*e_rl);
    kv_1 = kv_1l + a1_1spo2*(r_1*edot_1 + r_1l*edot_1l) +
           b1_1*(r_1*edot_1 - r_1l*edot_1l);
    kv_2 = kv_2l + a1_2spo2*(r_2*edot_2 + r_2l*edot_2l) +
           b1_2*(r_2*edot_2 - r_2l*edot_2l);
    kv_z = kv_zl + a1_zspo2*(r_z*edot_z + r_zl*edot_zl) +
           b1_z*(r_z*edot_z - r_zl*edot_zl);
    kv_r = kv_rl + a1_rspo2*(r_r*edot_r + r_rl*edot_rl) +
           b1_r*(r_r*edot_r - r_rl*edot_rl);

/* compute torques */
    *c_t_1 = *t_1 = (int)(0x0800 + e_1*kp_1 + edot_1*kv_1 + f_1);
    *c_t_2 = *t_2 = (int)(0x0800 + e_2*kp_2 + edot_2*kv_2 + f_2);
    *c_t_z = *t_z = (int)(0x0800 + e_z*kp_z + edot_z*kv_z + f_z);
    *c_t_r = *t_r = (int)(0x0800 + e_r*kp_r + edot_r*kv_r + f_r);
    *timing_b = 0x0001;  /* tell 196 that torques are ready */

/* capture data every capture interval */
    if(!capture_counter) {
        *c_d_1++;
        *c_d_2++;
        *c_d_z++;
        *c_d_r++;
        *c_a_1++;
```

```
            *c_a_2++;
            *c_a_z++;
            *c_a_r++;
            *c_t_1++;
            *c_t_2++;
            *c_t_z++;
            *c_t_r++;
            counter++;
            capture_counter = capture_intrvl - 1;
        }
        else {
            capture_counter--;
        }

    /* indicate end of sample period */
        *timing_a = 0x0000;
    }

/* INDICATE END OF COMMAND */
    *command = 0x0000;
    *timing_b = 0x0000;

/* DISPLAY SUCCESS OF EXECUTION */
    delline();
    if(*error) {
        printf("\n\n\tCommand terminated prematurely because:\n\n");
        display_errors();
    }
    else {
        printf("\n\n\t\t\tCommand executed normally.\n\n");
        printf("\t\tSaving %i data samples to disk.  Please wait.\n\n", counter);
        c_a_1 = s_c_a_1; /* reset the pointers */
        c_d_1 = s_c_d_1;
        c_a_2 = s_c_a_2;
        c_d_2 = s_c_d_2;
        c_a_z = s_c_a_z;
        c_d_z = s_c_d_z;
        c_a_r = s_c_a_r;
        c_d_r = s_c_d_r;
        c_t_1 = s_c_t_1;
        c_t_2 = s_c_t_2;
        c_t_z = s_c_t_z;
        c_t_r = s_c_t_r;
/* open file for saving to disk */
        if((fp=fopen("robot.mat","wb"))==NULL) {
            printf("cannot open file. Aborting\n");
            goto terminate;
        }
        x.imagf = 0;
        x.namlen = 5;
        x.mrows = counter;
/* save a vector called 'torl' (joint 1 torque) to disk */
        x.type = 30;    /* data is signed int type */
        x.ncols = 1;
        data_file_name = "torl";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_t_1, 2, counter, fp);
/* save a vector called 'tor2' (joint 2 torque) to disk */
        data_file_name = "tor2";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_t_2, 2, counter, fp);
/* save a vector called 'torz' (joint z torque) to disk */
        data_file_name = "torz";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_t_z, 2, counter, fp);
/* save a vector called 'torr' (joint r torque) to disk */
        data_file_name = "torr";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
```

```
        fwrite(c_t_r, 2, counter, fp);
/* save sample period and capture interval to disk */
        x.type = 0;     /* data is double precision */
        x.mrows = 1;
        data_file_name = "smpl";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(&sample_per, 8, 1, fp);
        x.type = 0040;  /* data is unsigned 16-bit */
        data_file_name = "invl";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(&capture_intrvl, 2, 1, fp);
/* save a matrix called 'pos1' (joint 1 position) to disk
 * column 1 = desired position of joint 1
 * column 2 = actual position of joint 1 */
        x.type = 20;    /* data is long int type */
        x.ncols = 2;
        x.mrows = counter;
        data_file_name = "pos1";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_d_1, 4, counter, fp);
        fwrite(c_a_1, 4, counter, fp);
/* save a matrix called 'pos2' (joint 2 position) to disk
 * column 1 = desired position of joint 2
 * column 2 = actual position of joint 2 */
        data_file_name = "pos2";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_d_2, 4, counter, fp);
        fwrite(c_a_2, 4, counter, fp);
/* save a matrix called 'posz' (joint z position) to disk
 * column 1 = desired position of joint z
 * column 2 = actual position of joint z */
        data_file_name = "posz";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_d_z, 4, counter, fp);
        fwrite(c_a_z, 4, counter, fp);
/* save a matrix called 'posr' (joint r position) to disk
 * column 1 = desired position of joint r
 * column 2 = actual position of joint r */
        data_file_name = "posr";
        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(data_file_name, sizeof(char), x.namlen,fp);
        fwrite(c_d_r, 4, counter, fp);
        fwrite(c_a_r, 4, counter, fp);
        fclose(fp);
        }

/* FREE ALLOCATED MEMORY */
terminate:
    if(s_pp) farfree(s_pp);
    if(s_c_a_1) free(s_c_a_1);
    if(s_c_d_1) free(s_c_d_1);
    if(s_c_a_2) free(s_c_a_2);
    if(s_c_d_2) free(s_c_d_2);
    if(s_c_a_z) free(s_c_a_z);
    if(s_c_d_z) free(s_c_d_z);
    if(s_c_a_r) free(s_c_a_r);
    if(s_c_d_r) free(s_c_d_r);
    if(s_c_t_1) free(s_c_t_1);
    if(s_c_t_2) free(s_c_t_2);
    if(s_c_t_z) free(s_c_t_z);
    if(s_c_t_r) free(s_c_t_r);
    printf("\t\t\tPress any key to continue ");
    while(!getch());
    return;
}
```

```
/*****************************************************************************
 *
 * CUBIC.ROB - A cubic spline path generator that incorporates via points.
 * Velocity at the via points is automatically chosen.
 *
 *****************************************************************************/

#include "alloc.h"
#include "stdio.h"

void get_cartesian_position(); /* function prototypes */
void get_joint_position();
void get_time();
cmp_sign(double, double);

extern double Duration;          /* global variable declared in utility.rob */

long int huge *cubic_spline(double sample_per)
{
    char ch;
    int i, j, error, num_via_points;
    double s1, s2; /* slope 1, slope 2 */
    long int pos[10][4];         /* A two dimensional position array.
                                  * A row for each specified path point.
                                  * column 0 = joint 1 pulse count
                                  *   "    1 =   "   2   "      "
                                  *   "    2 =   "   z   "      "
                                  *   "    3 =   "   r   "      "   */
    long int position[4];
    double vel[10][4] = {        /* A two dimensional velocity array */
        0,0,0,0,                 /* A row for each specified path point */
        0,0,0,0,                 /* column 0 = joint 1 velocity */
        0,0,0,0,                 /*    "   1 =   "   2    "      */
        0,0,0,0,                 /*    "   2 =   "   z    "      */
        0,0,0,0,                 /*    "   3 =   "   r    "      */
        0,0,0,0,
        0,0,0,0,
        0,0,0,0,
        0,0,0,0,
        0,0,0,0 };
    double a[4][4]; /* two dimensional array of coefficients */
    double time[10];
    double t, t2, t3, tf, tf2, tf3;
    long int huge *path_point; /* pointer to starting address of path */
    long int huge *s_path_point;       /* a place to save this value */
    unsigned long int num_bytes_mem_req;
    long int far *current_pos = 0xC0010000;

/* DISPLAY CURRENT POSITION */
    clrscr();
    printf("\tCUBIC SPLINE PATH GENERATOR WITH VIA POINT CAPABILITY\n\n");
    printf("Features:  *current robot position is automatically taken ");
    printf("as the initial\n");
    printf("            position of the move\n");
    printf("           *velocity at the via points is automatically computed\n\n");
    current_position();

/* GET CURRENT POSITION (from DPR) */
    pos[0][0] = *(current_pos++);
    pos[0][1] = *(current_pos++);
    pos[0][2] = *(current_pos++);
    pos[0][3] = *(current_pos);
    time[0] = 0;

/* GET # OF VIA POINTS AND CHOICE OF SPACE */
    printf("\nWill you be entering data in (C)artesian or (J)oint space? ");
    ch = getch();

    printf("\nEnter total number of via points (Max. 8): ");
    scanf("%i", &num_via_points);

/* GET DURATION OF THE MOVE AND CREATE STORAGE AREA FOR THE PATH */
```

```
    printf("Enter duration of the move (in sec.): ");
    do {
        scanf("%lf", &time[num_via_points+1]);
        Duration = time[num_via_points+1];
        num_bytes_mem_req=(unsigned long int)(4*4*Duration/sample_per+20+100);
            /* 4 = four joints
             * 4 = four bytes per joint per sample
             * Duration/sample_per = total # of samples
             * 20 = four bytes extra per joint + 4 byte flag
             * 100 = safety margin */
        path_point=(long int huge *)farmalloc(num_bytes_mem_req);
        if(!path_point) {
            printf("Insufficient memory for a %lf",Duration);
            printf("second move.\nEnter a shorter duration: ");
        }
    } while (!path_point);

/* PROMPT FOR VIA POINTS AND FINAL POINT */
    for(i=1; i<=num_via_points+1; i++) { /* for each via point */
        if(i != num_via_points+1) printf("\nVia point no. %i:\n", i);
        else                      printf("\nFinal point:\n");
        if(ch == 'c') get_cartesian_position(&pos[i][0]);
        else          get_joint_position(&pos[i][0]);
        if(i != num_via_points+1) get_time(&time[i]);
    }

/* COMPUTE THE VELOCITIES AT THE VIA POINTS */
    for(i=1; i<=num_via_points; i++) {  /* for each via point */
        for(j=0; j<=3; j++){            /* for each joint */
            s1 = (double)(pos[i+1][j]-pos[i][j]) / (time[i+1] - time[i]);
            s2 = (double)(pos[i][j]-pos[i-1][j]) / (time[i] - tim (i-1]);
            if(cmp_sign(s1,s2)) vel[i][j] = 0;
            else vel[i][j] = (s1 + s2)/2;  /* the average of the two */
        }
    }

/* COMPUTE AND STORE THE PATH */
    printf("\n\n\t\tCOMPUTING THE PATH.  PLEASE WAIT");
    s_path_point = path_point;
    t = 0.0;
    tf = 0;
    for(i=0; i<=num_via_points; i++) { /* for all points (beginning at 0) */
        t = t - tf;
        tf = t  [i+1] - time[i];
        tf2 =  * tf;
        tf3 = tf2 * tf;
        for(j=0; j<=3; j++) { /* for each joint */
           a[j][0] = pos[i][j];
           a[j][1] = vel[i][j];
           a[j][2] =  3*(pos[i+1][j]-a[j][0])/tf2 - (2*a[j][1]+vel[i+1][j])/tf;
           a[j][3] = -2*(pos[i+1][j]-a[j][0])/tf3 + (vel[i+1][j]+a[j][1])/tf2;
        }
        for(; t <= tf+sample_per/2; t += sample_per) {
            t2 = t * t;
            t3 = t2 * t;
            *path_point++ =(long int)(a[0][0]+a[0][1]*t+a[0][2]*t2+a[0][3]*t3+0.5);
            *path_point++ =(long int)(a[1][0]+a[1][1]*t+a[1][2]*t2+a[1][3]*t3+0.5);
            *path_point++ =(long int)(a[2][0]+a[2][1]*t+a[2][2]*t2+a[2][3]*t3+0.5);
            *path_point++ =(long int)(a[3][0]+a[3][1]*t+a[3][2]*t2+a[3][3]*t3);
        }
    }
    *path_point = 0xffff0000;  /* end-of-path flag */

    return(s_path_point);
}
```

```
/*****************************************************************************
 *
 * CYCLOID.ROB - Cycloidal path generator
 *
 *****************************************************************************/

#include "alloc.h"
#include "stdio.h"
#include "math.h"

void get_cartesian_position();          /* fucntion prototypes */
void get_joint_position();

extern double Duration;          /* global variabl declared in utility.rob */

long int huge *cycloid(double sample_per)
{
    char ch;
    long int pos[4]; /* a vector to hold desired final position */
    long int huge *path_point; /* starting address of path */
    long int huge *s_path_point;       /* place to save this value */
    unsigned long int num_bytes_mem_req;
    long int far *initial_pos = 0xC0010000;
    long int i_1p = *(initial_pos + 0);
    long int i_2p = *(initial_pos + 1);
    long int i_zp = *(initial_pos + 2);
    long int i_rp = *(initial_pos + 3);
    double d1, d2, dz, dr;
    double t, w, factor;

/* DISPLAY CURRENT POSITION */
    clrscr();
    printf("\t\t\tCYCLOIDAL PATH GENERATOR\n\n");
    current_position();

/* GET FINAL POSITION */
    printf("\nWill you be entering data in (C)artesian or (J)oint space? ");
    ch = getch();
    printf("\n\nFinal point:\n");
    if(ch == 'c') get_cartesian_position(&pos[0]);
    else          get_joint_position(&pos[0]);

/* GET DURATION OF THE MOVE AND CREATE STORAGE AREA FOR THE PATH */
    do {
        printf("\n\tEnter Duration of the move (in sec.): ");
        scanf("%lf", &Duration);
        num_bytes_mem_req=(unsigned long int)(4*4*Duration/sample_per+20+100);
            /* 4 = four joints
             * 4 = four bytes per joint per sample
             * Duration/sample_per = total # of samples
             * 20 = four bytes extra per joint + 4 byte flag
             * 100 = safety margin */
        path_point=(long int huge *)farmalloc(num_bytes_mem_req);
        if(!path_point) printf("Not enough memory in far heap\n");
    } while (!path_point);

/* COMPUTE AND STORE THE PATH */
    printf("\n\n\t\tCOMPUTING THE PATH.  PLEASE WAIT");
    s_path_point = path_point;
    w = 2 * M_PI / Duration;
    d1 = (pos[0]-i_1p)/(2*M_PI), d2 = (pos[1]-i_2p)/(2*M_PI);
    dz = (pos[2]-i_zp)/(2*M_PI), dr = (pos[3]-i_rp)/(2*M_PI);
    for(t = 0.0; t <= Duration + sample_per/2; t += sample_per) {
        factor = (w*t - sin(w*t));
        *path_point++ = (long int)(i_1p + d1*factor + 0.5);
        *path_point++ = (long int)(i_2p + d2*factor + 0.5);
        *path_point++ = (long int)(i_zp + dz*factor + 0.5);
        *path_point++ = (long int)(i_rp + dr*factor + 0.5);
    }
    *path_point = 0xffff0000;  /* flag at end of path */
    return(s_path_point);
}
```