The Communication Links in ProCell

- Intel386EX Peripheral Device Interfaces

by

Ning Ye

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering

and Master of Engineering in Electrical Engineering

at the Massachusetts Institute of Technology

January 10, 2000

[June 2000]

Author _____
Department of Electrical Engineering and Computer Science

Certified by _____
Kamal Youcef-Toumi
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

The Communication Links in ProCell
- Intel386EX Peripheral Device Interfaces
by
Ning Ye

Submitted to the
Department of Electrical Engineering and Computer Science

January 10, 2000

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering
and Master of Engineering in Electrical Engineering

ABSTRACT

The communication links in a photoresist processing system are crucial issues of the automation in semiconductor production facilities. On the module control level in ProCell, the Intel386EX microprocessor interfaces with the peripheral devices via three different data transfer protocols. Both the hardware and the software for these three protocols are discussed in detail. The device drivers for two of the protocols, RS-232 serial communication and Controller Area Network are written in C in a VxWorks, real-time operating system. A set of diagnostic procedures called *echo* is developed to test the driver for a serial device SC26C92 in RS-232 serial communication. Overall testing was done in both hardware and software.

Thesis Supervisor: Kamal Youcef-Toumi

Title: Associate Professor

Acknowledgements

The author wishes to acknowledge some of the people who contributed to the completion of this thesis. The Track Division of Silicon Valley Group, Inc. in San Jose, California provided a wonderful opportunity for me to carry out the work for this thesis. I must thank the person who helped me the most, Charles Lee, my supervisor at SVG who was instrumental in helping me define the research topic and providing a stimulating research environment. Many engineers at SVG who worked on the ProCell system contributed to the thesis work both directly and indirectly.

I would like to thank Professor Kamal Youcef-Toumi for traveling all the way to San Jose to help me with writing of this thesis and directing the course of the thesis work. His time and patience is greatly appreciated.

Additionally, I would like to thank my parents for their unending encouragement and support. Without them I surely would never have reached this point.

Ning Ye
Massachusetts Institute of Technology
January, 2000

Table of Contents

List of Figures

Chapter 1: Introduction

Semiconductor manufacturing has been such a rapidly growing industry that it has enormous impacts on even the most mundane tasks in our daily life. Semiconductor industry's products nowadays are found not only in the high tech electronic goods which semiconductors have been traditionally associated with, but in toys and household appliances as well. The semiconductor equipment industry produces the process tools that are used in the complex manufacturing of semiconductors. With the continued expansion of Internet-related technologies and increasing demand for semiconductors in new electronics, semiconductor equipment industry has been growing tremendously in the last decade as a result of chip manufacturer's increasing production.

This thesis addresses the photoresist processing tools in the semiconductor equipment industry. A photoresist processing system is usually composed of several process modules, i.e. coaters, developers, etc. The communications among the individual modules, as well as the communications between modules and the module controllers, have always been crucial issues of the automation in semiconductor production facilities, and must be done in a timely and precise way. Failure in communications in the control system can lead to severe consequences to possibly cause paralysis of the entire machine.

This thesis presents the architecture of the software control system in an automated photoresist processing machine. It focuses on the communications and interfaces between the host microprocessors and the modules. Thesis work includes development of the drivers for serial communication devices, diagnostic procedures to

ensure the proper functioning of the driver, and parallel communication between the microprocessor and a motion control chipset.

Chapter 2 gives a rudimentary introduction on photoresist process and Silicon Valley Group's ProCell, a photoresist processing system. Chapter 3 focuses on the module controller board in ProCell. It illustrates how the microprocessor used in module control interfaces with the modules. Three communication protocols are introduced at the end of Chapter 3. Chapter 4, Chapter 5 and Chapter 6 address these three types of communication protocols, RS-232 serial communication, Controller Area Network (CAN) and parallel communication respectively.

Chapter 2: Photoresist Process

2.1 Introduction

This chapter provides a basic introduction to the integrated circuit (IC) fabrication and Silicon Valley Group (SVG)'s ProCell system. SVG's Track Systems Division[1] is a world's leading manufacturer of the complete family of photoresist processing products. A brief description of the essential process in IC fabrication will help understanding SVG's ProCell system, the latest photoresist processing system, designed and manufactured by SVG's Track Systems Division.

2.2 Photoresist Process

The fundamental unit of IC manufacturing is a silicon wafer. The essential process in IC fabrication is the imaging, alignment, and transfer of complex patterns onto the silicon wafer. A photolithography machine called stepper is used to transfer patterns from an optical plate called a mask, to photoresist on the wafer. Photoresist is a photosensitive material that coats the top surface of the wafer [Howe '97]. It dissolves in developer if it has been exposed to light. Patterns are transferred to a wafer by covering the wafer with photoresist, exposing a pattern in the photoresist and then using the patterned photoresist as a mask through which to implant dopants or etch the material. Photoresist processing is a pivotal function, which includes numerous steps performed repeatedly, such as coating, developing and baking of silicon wafers.

[1] SVG's Track Division is where the author's thesis work was done.

2.3 SVG's ProCell System

ProCell from SVG's Tracks Systems Division is an evolutionary photolithography processing system that provides a complete line of photoresist processes, including vapor prime, vacuum-dehydration bake, photoresist coat, hot plate bake, and wafer temperature control using chill plates. Designed around the principle of symmetry, ProCell's cluster configuration jump-starts productivity by departing from the previous linear track models. This design eliminates the bottlenecks that plague conventional track systems. Wafers flow smoothly through ProCell with the minimum number of moves. In fact, this is currently the only platform that can complete the lithography process in just 12 perfectly balanced moves [Silicon '99]. Figure 1 is a picture of SVG's ProCell. Figure 2 shows the top view of ProCell's machine configuration.

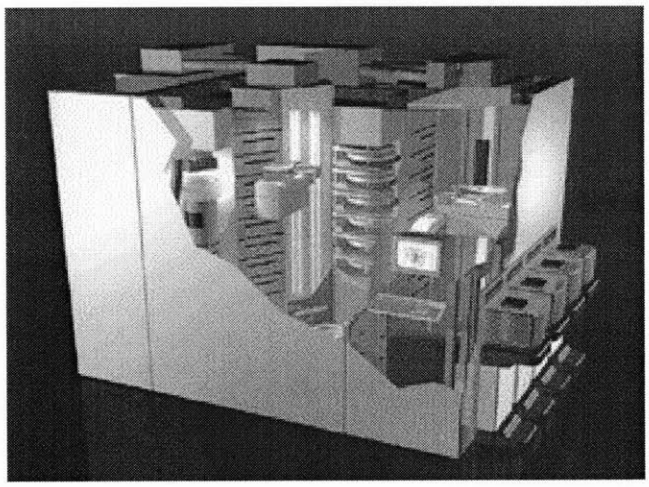

Figure 1: Silicon Valley Group's ProCell system.[2]

[2] Courtesy of Silicon Valley Group.

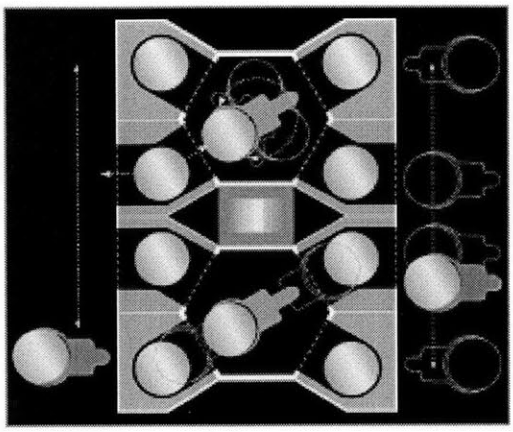

Figure 2: View of ProCell from top down.[3]

2.4 ProCell's Control System Architecture

This section gives a brief description of SVG's ProCell's architecture of its control system. This description gives a background of the thesis work, and shows where the work done in this thesis fits in the big picture of ProCell's overall control system.

Figure 3 illustrates ProCell's software control system. At the top layer, System Controller Board is a Pentium based single board computer running Windows NT 4.0. This is used to host the system control software, where the user displays and operator controls as well as factory interfaces are provided. The mid-layer is the Machine Controller Board, which is also the same basic Pentium based SBC board, hosting the machine control software. It provides high-level module controls, load ports and robot controls. A Module Controller Board is based on an Intel386EX microprocessor with I/O

[3] Courtesy of Silicon Valley Group.

and motion controllers to control each process module. The Module Controller Boards share the same VxWorks real time operating system environment as the Machine Controller Board.

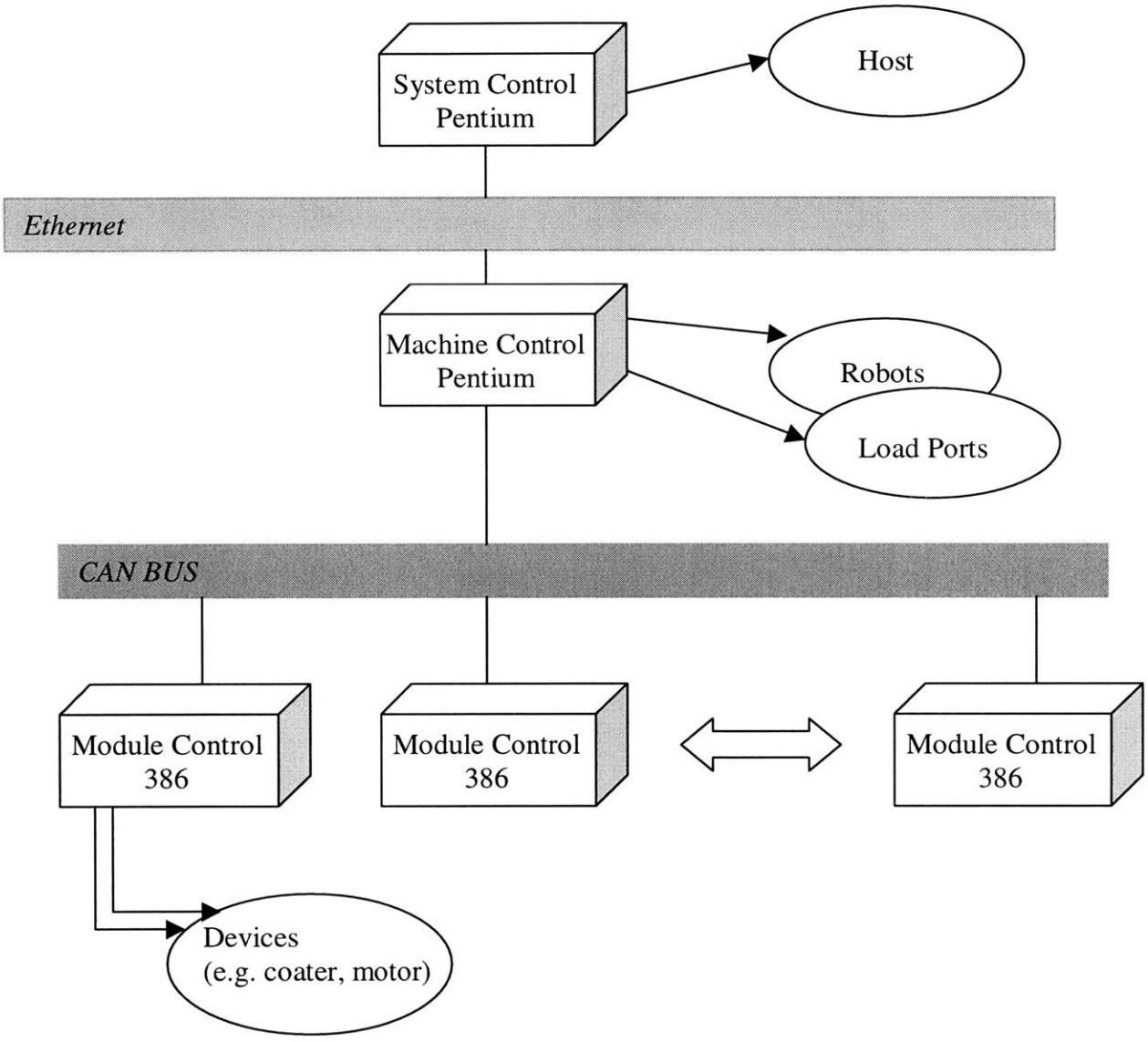

Figure 3: ProCell Systems Architecture / Communication Diagram.[4]

[4] Courtesy of Silicon Valley Group.

There are 7 main communication links in ProCell Control Systems.

1) Pentium Window NT to Pentium VxWorks communication links over Ethernet.
2) Pentium Window NT to Pentium peripherals and VxWorks communication links via VME BUS.
3) Pentium VxWorks to 386 VxWorks communication links via CAN-BUS and VME-BUS.
4) Pentium VxWorks to OEM (Original Equipment Manufacturer) communication links via serial or parallel signals.
5) 386 VxWorks to 386 VxWorks peer to peer communication links via CAN-BUS.
6) 386 to I/O control chips and OEM communication links via serial, parallel, or electrical signals.
7) Pentium Windows NT to GEM/SECS (General Equipment Manufacturer / Semi-conductor Equipment Communication Set) hose via HSMS protocol over Ethernet.

2.5 Summary

This chapter gave a brief introduction to semiconductor industry's photoresist process and described ProCell, SVG Track Systems Division's new photoresist processing system on the whole, as well as its overall control system architecture. The following chapters will go on to discuss the module controller system in ProCell, with a focus placed on the communications in the module controller system.

Chapter 3: Module Controller System

3.1 Overview

The Module Controller Boards in ProCell system are based on an Intel386EX microprocessor with I/O and motion controllers to control each process module in ProCell (link 6). Refer to Figure 4 for the block diagram of ProCell's overall control system to see where the Module Controller Boards are located at in the overall control system.

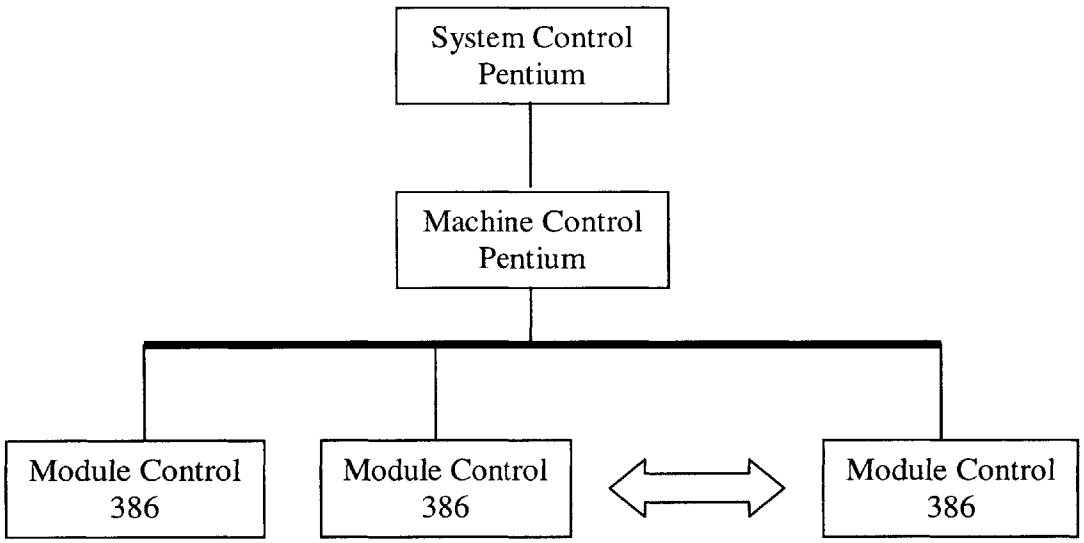

Figure 4: Block diagram of ProCell's control system.

Controls of modules at module level reduce the operation delays by the controller system, which used to be a key reliability/process concern in the past, because the delays have been seen to arise from the fact that all controls were at high level in the previous

systems. Distributed module control provides stand-alone module tests. Decisions are made at module level so that there is less communication to system control level.

3.2 Module Controller Board

Link 6, which is the interface between the Intel386EX-based Module Controller Board and the modules, together with link 5's CAN-BUS are where the work of this thesis has been focused on at SVG's Track Division. The emphasis of the thesis will be placed on 386 peripheral device interfaces, mostly on the device drivers' side. The Intel386EX-based Module Controller Boards share the same standard VxWorks real time operating system.

Figure 5 shows the block diagram for typical interfaces between Intel386EX microprocessor and the peripheral devices, and communication between two microprocessors.

The heart of the Module Controller Board is an Intel386EX microprocessor. In addition to memory and A/D converters, the microprocessor is directly interfaced with a Universal Asynchronous Receiver/Transmitter (UART), an 82527 serial communication controller which is a device that performs serial communication according to CAN protocol, and an advanced multi-axis motion control chipset. They will be discussed in further detail in later chapters.

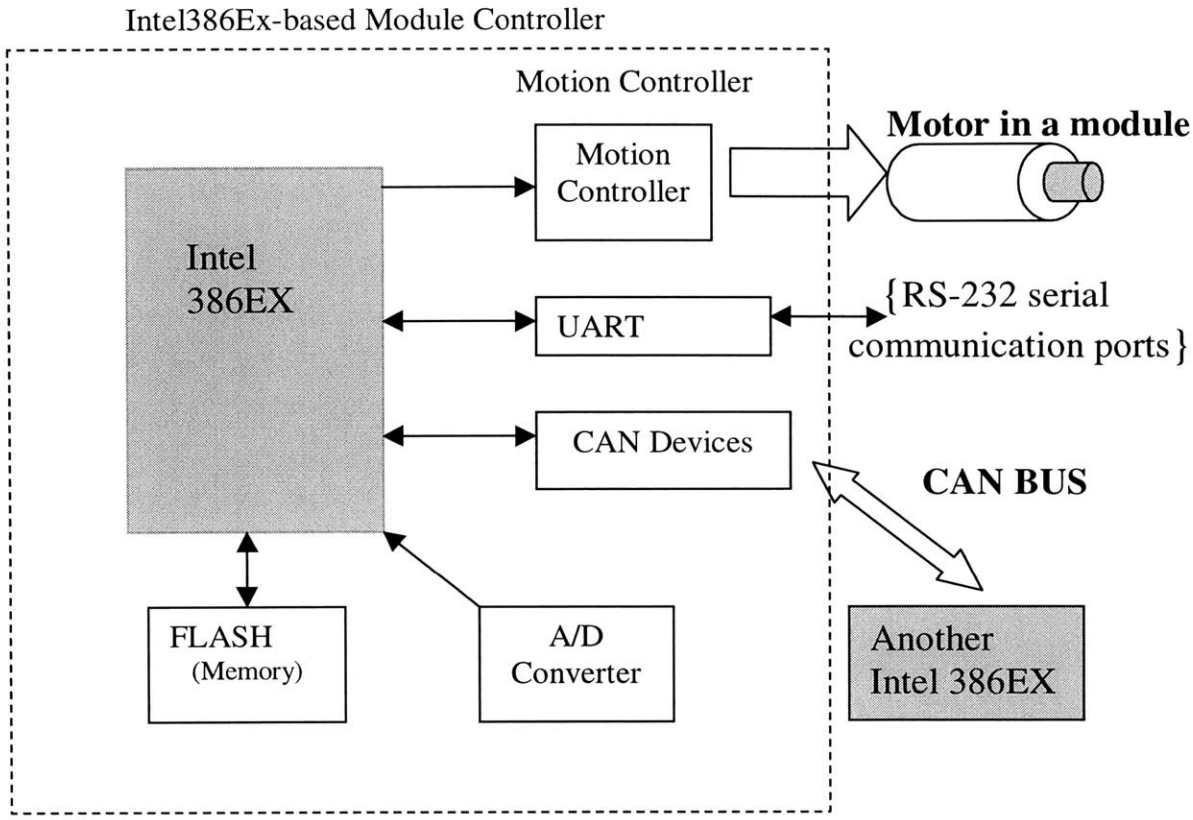

Figure 5: The Module Controller Board in ProCell and the Interfaces.

3.2 Communication Protocols

RS-232 Serial Communication

Most of the modules communicate with the Module Controller Boards via RS-232 serial ports. In order for the RS-232 serial ports to be connected to the microprocessor, serial transmission needs to be converted to parallel transmission via UART. A Dual UART device SC26C92 is used here. Chapter 4 will discuss the serial device driver in VxWorks and diagnostic procedures for the device driver.

Controller Area Network (CAN)

In ProCell, peer module controllers are connected via a serial bus using Controller Area Network (CAN) protocol. On the Module Controller Board, the microprocessor is connected with a CAN controller 82527 which, in turn, connects to a CAN Transceiver to perform the serial communication. The protocol for CAN, the driver for CAN devices in VxWorks, as well as the physical CAN connection are described in Chapter 5.

Parallel Communication

The MC1401A used here is a 2-IC general-purpose motion control chipset that provides closed-loop digital servo control for motors. The chipset is controlled by the host microprocessor that interfaces with the chipset via an 8-bit, bi-directional port. All communications to/from the chipset consist of packet-oriented messages. A packet is a sequence of transfers to/from the microprocessor resulting in a chipset action or data transfer. Chapter 6 will address the protocol and the hardware interface for the parallel communication.

3.2 Summary

This Chapter gave an overall illustration of the Intel386EX-based Module Controller Board, as well as the major components on the board. Three communication protocols were introduced in this Chapter, RS-232 serial communication, Controller Area Network (CAN) and parallel communication. The next three chapters will be discussing each communication protocol in substantial detail.

Chapter 4: Device and Driver for RS-232 Serial Communication

4.1 Introduction

This Chapter presents the thesis work done on the RS-232 serial communication device driver in VxWorks on the module control level in ProCell. It begins by discussing the advantages of serial over parallel data transmission in the ProCell. Then it moves on to describe a DUART Philips SC26C92 and its device driver in VxWorks, a real-time operating system run on ProCell. An introduction on I/O system and device driver in VxWorks in general is given in section 4.4 for a better understanding of *SCC* driver. Section 4.5 focuses on the *SCC* test, a diagnostic procedure written for testing of *SCC* driver and the serial communication between two RS-232 ports.

4.2 Advantages of Serial Data Transmission in ProCell

Serial communication is vastly used in ProCell, between the module controllers and the modules. Using serial data transfer rather than parallel transfer in ProCell has its advantages. Serial cable can be much longer than parallel cables. The serial port transmits a '1' as -3 to -25 volts and a '0' as +3 to +25 volts whereas a parallel port transmits a '0' as 0v and a '1' as 5v. Therefore the serial port can have a maximum swing of 50V compared to the parallel port which has a maximum swing of 5 Volts. Thus cable loss is not going to be as much of a problem for serial cables than they are for parallel ones. This is especially desirable in large wafer processing machines like ProCell, where the host computers could be placed far away from the process modules in the system. Moreover, serial transmission doesn't require as many wires as parallel transmission. When a device has to be mounted a great distance away from the computer, a 3-core

cable (Null Modem Configuration) is a lot cheaper that running 19-core or 25-core cable. The above two features are both desirable in setting up test environment during the engineering process as well.

4.3 Philips SC26C92 DUART

The Philips Semiconductors SC26C92 Dual Universal Asynchronous Receiver/Transmitter (DUART) is a single-chip serial communications device that provides two full-duplex asynchronous receiver/transmitter channels in a single package. It interfaces directly with the Intel386EX microprocessor. When the SC26C92 is conditioned to transmit data, the transmitter converts the parallel data from the microprocessor to a serial bit stream on the TxD output pin. It automatically sends a start bit followed by the programmed number of data bits, an optional parity bit, and the programmed number of stop bits. The least significant bit is sent first. When the SC26C92 is conditioned to receive data, the receiver looks for a mark-to-space transition of the start bit on the RxD input pin. If a transition is detected, the state of the RxD pin starts getting sampled every certain time interval depending on the clock until the proper number of data bits and parity bit (if any) have been assembled, and one stop bit has been detected. The least significant bit is received first. The data is then transferred to the Receive buffer [Philips '97].

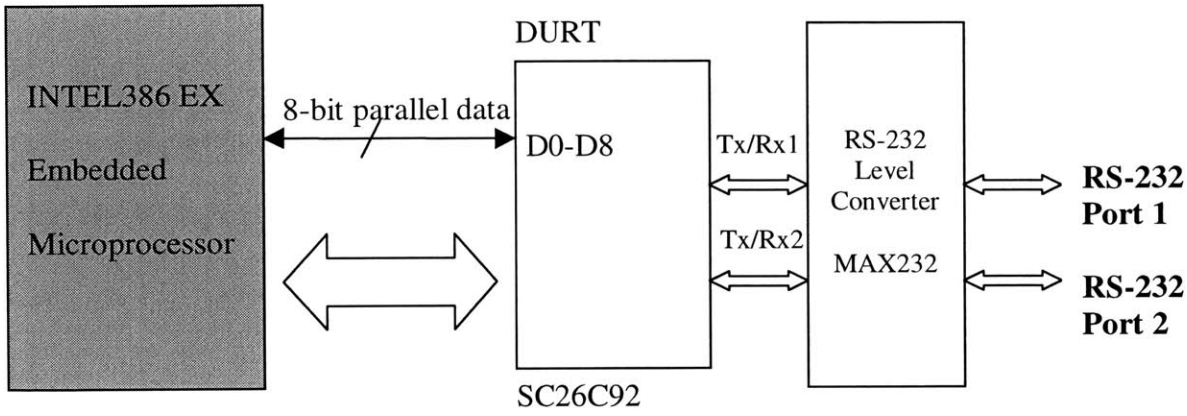

Figure 6: Interfacing RS-232 ports to devices.

Figure 6 shows how to interface RS-232 ports to devices. Note that almost all digital devices require either TTL or CMOS logic levels. Therefore, the first step to connecting a device to the RS-232 port is usually transforming the RS-232 levels back into 0 and 5 Volts. This is done by an RS-232 Level Converter, MAX232 here.

4.4 Device Driver for Philips SC26C92 DUART

Device drivers are program routines that let peripheral devices communicate with operating systems. The driver for the Philips SC26C92 is implemented as a standard VxWorks terminal driver making use of the **`tyLib()`** system library. How VxWorks provides terminal devices and device drivers and VxWorks' I/O system will be described in further detail in the next section.

Refer to **`Scc.c`** in Appendices for the **`tty`** driver for the Philips SC26C92 DUART. In **`Scc.c`**, there are two important routines: **`tySCCDrv`** and **`tySCCDevCreat.`** They are described in the text below.

tySCCDrv

Before a diver is used to drive a device, it must be initialized by calling the program routine: **tySCCDrv()**. The initialization routine installs the driver in the I/O system, connects to any interrupts used by the devices serviced by the driver, and performs any necessary hardware initialization. A single driver for a serial communications device can often handle many separate channels. This routine should be called exactly once before any **read**'s, **write**'s, or **tySCCDevCreate**'s.

tySCCDevCreate

This routine adds terminal devices that are to be serviced by the driver (initialized by routine **tySCCDrv**) to the I/O system. Before a terminal device can be used, it has to be created first. This is done through calling the routine **tySCCDrvCreate**. Each serial communication port to be used will have exactly one terminal device associated with it after this routine has been called.

TySCCDrvCreate takes five arguments, and returns OK (1) if a terminal device has been successfully created, or Error (0) otherwise.

```
STATUS tySCCDevCreate (name, channel, rdBufSize,
     wBufSize, baudRate)
```

Name	name to use for this terminal device, i.e. "**/tySCC/0**"
Channel	physical channel for this device
RdBufSize	read buffer (input buffer) size, in bytes
WbufSize	write buffer (output buffer) size, in bytes
Baudrate	baud rate to create the terminal device with, i.e. 9600

The arguments for **tySCCDevCreate** routine.

For instance, to create a terminal device "**/tySCC/0**", with input and output buffers of size 512 bytes, at 9600 baud rate, the proper call is:

```
TySCCDevCreate ("/tySCC/0", 0, 512, 512, 9600);
```

4.5 Introduction to Devices and Drivers in VxWorks

4.5.1 I/O system in VxWorks

VxWorks is a high-performance networked real-time operating system. It runs on a variety of hardware, including the Intel 386 based system. This section gives a background on VxWorks I/O system, with an emphasis on devices and drivers in VxWorks.

In VxWorks, applications access I/O devices by opening named files. A file can refer to one of the two things:

A *"raw" device* such as a serial communication channel.

A *logical file* on a random-access device containing a file system.

Consider the following named files:

/usr/file1 **/pipe/mypipe/** **/tyCo/0**

The first one refers to a file called *file1* on a disk device called **/usr**. The second one is a pipe named **mypipe**. The third one refers to a physical serial channel. However, I/O can be done to or from any of these in the same way. Within VxWorks, they are all called files, even though they refer to very different physical objects [Wind '97].

Basic I/O is the lowest level of I/O in Vxworks. The following are a few basic I/O calls.

open(): Open a file/device.

close(): Close a file/device.

read(): Read a previously created or opened file.

write(): Write a previously created or opened file.

At the basic I/O level, files are referred to by a file descriptor, or *fd*. An *fd* is a small integer returned by a call to **open()**. The other basic I/O calls take an *fd* as a parameter to specify the intended file. Before I/O can be performed to a device, an *fd* must be opened to that device by invoking the **open()** routine.

After an *fd* is obtained by invoking **open()**, tasks can read bytes from a file with **read()** and write bytes to a file with **write()**. The arguments to **read()** are the *fd*, the address of the buffer to receive input, and the maximum number of bytes to read:

```
int  nBytes = read (fd, &buffer, maxBytes);
```

The **read()** routine waits for input to be available from the specified file/device, and returns the number of bytes actually read.

The arguments to **write()** are the *fd*, the address of the buffer that contains the data to be output, and the number of bytes to be written:

```
int actualBytes = write (fd, &buffer, nBytes);
```

The **write()** routine ensures that all specified data is at least queued for output before returning to the caller, though the data may not yet have been written to the device (that is driver dependent).

Another very important basic I/O routine is I/O control: **ioctl()** routine. It is an open-ended mechanism for performing any I/O functions that do not fit the other basic I/O calls. Examples include determining how many bytes are currently available for

output and setting device-specific options. The arguments to the **ioctl()** routine are the *fd*, a code that identifies the control function requested, and an optional function-dependent argument.

```
Result = ioctl (fd, function, arg);
```

For example, the following call uses the **FIOFLUSH** function to discard all the bytes in the input or output buffer:

```
Status = ioctl (fd, FIOFLUSH, 0);
```

See **scctest.cpp** in the Appendices for how all the basic functions mentioned above are used.

4.5.2 Devices and Drivers

Devices are handled by program modules called drivers. The VxWorks I/O system is flexible, allowing specific device drivers to handle the basic I/O functions mentioned in the last section.

For serial I/O devices, VxWorks provides terminal device drivers (**tty** drivers). VxWorks serial I/O devices are buffered serial byte streams. Each device has a ring buffer (circular buffer) for both input and output. Reading from a **tty** device extracts bytes from the input ring. Writing to a **tty** device adds bytes to the output ring. The size of each ring buffer is specified when the device is created during system initialization. The **tty** devices respond to the **ioctl()** functions mentioned in the previous text.

A device driver implements the basic I/O functions for a particular kind of a device. In general, drivers have routines that implement each of the basic functions, such

as **read()**, **write()**, etc. although some of the routines can be omitted if the functions are not operative with that device.

In VxWorks, some drivers are capable of servicing many instances of a particular kind of device. For example, a single driver for a serial communications device can handle many separate channels that differ only in a few parameters, such as device address.

Devices are added to the I/O system dynamically by calling the internal I/O routine **isoDevAdd()**. The arguments to **iosDevAdd()** are the address of the device descriptor for the new device, the device's name, and the driver number of the driver that services the device. Refer to file **scc.c** in the Appendices for the use of **iosDevAdd().**

4.5.3 Opening a File/Device

Files/devices are opened with **open()**. The I/O system searches the device list for a device name that matches the file name (or an initial sub-string) specified by the caller. If a match is found, the I/O system uses the driver number contained in the corresponding device header and call the driver's open routine in the driver table.

The I/O system must establish an association between the file descriptor used by the caller in subsequent I/O calls, and the driver that services it. The I/O system maintains these associations in a table called *fd* table. This table contains the driver number and an additional driver-determined 4-byte value. The driver value is the internal descriptor returned by the driver's open routine, and can be any nonnegative value the driver requires to identify the file. In subsequent calls to the driver's other I/O functions,

such as **read()**, **write()**, **ioctl()** and **close()**, this value is supplied to the driver in place of the *fd* in the application-level I/O calls.

	drvnum	value
0		
1		
2		
3		
4		
.	.	.
.	.	.
.	.	.

fd Table

Example of Opening a Device

In Figure7 and Figure 8 [Wind '97], a user calls open() to open the device /xx0. The following series of actions take place in the I/O system:

1) It searches for the device list for a device name that matches the specified device name: **/xx0**.
2) It reserves a slot in the *fd* table, which is used if the open is successful.
3) It then looks up the address of the driver's open routine, **xxOpen()**, and calls that routine. Note that the arguments to **xxOpen()** are transformed by the I/O system from the user's original arguments to **open()**. The first argument to **xxOpen()** is a pointer to the device descriptor the I/O system located in the full file name search. The next parameter is the remainder of the file name specified by the user, after removing the initial substring that matched the device name. In this case, because the device name matched the entire file name, the remainder passed to the driver is a null

string. The last parameter is the file access flag, in the case **O_RDONLY**; this is, the file is opened for reading only.

4) It executes **xxOpen(),** which returns a value that subsequently identifies the newly opened file. In the case, the value is the pointer to the device descriptor. This value is supplied to the driver in subsequent I/O calls that refer to the driver being opened.
5) The I/O system then enters the driver number and the value returned by **xxOpen()** in the reserved slot in the *fd* table. Again, the value entered in the *fd* table has meaning only for the driver, and is arbitrary as far as the I/O system is concerned.

4.5.4 Reading Data from the File/Device

The routine **read()** is used to obtain input data from the device. The specified *fd* is the index to the *fd* table for this device. The I/O system uses the driver number contained in the table to locate the driver's read routine, **xxRead()**. The I/O system calls **xxRead()**, passing it the identifying value in the *fd* table that was returned by the driver's open routine, **xxOpen()**. Again, in this case the value is the pointer to the device descriptor. The driver's read routine then does whatever is necessary to read data from the device. The process for user calls to **write()** and **ioctl()** follow the same procedure. Refer to Figure 9 for an example of reading Data from a device [Wind '97].

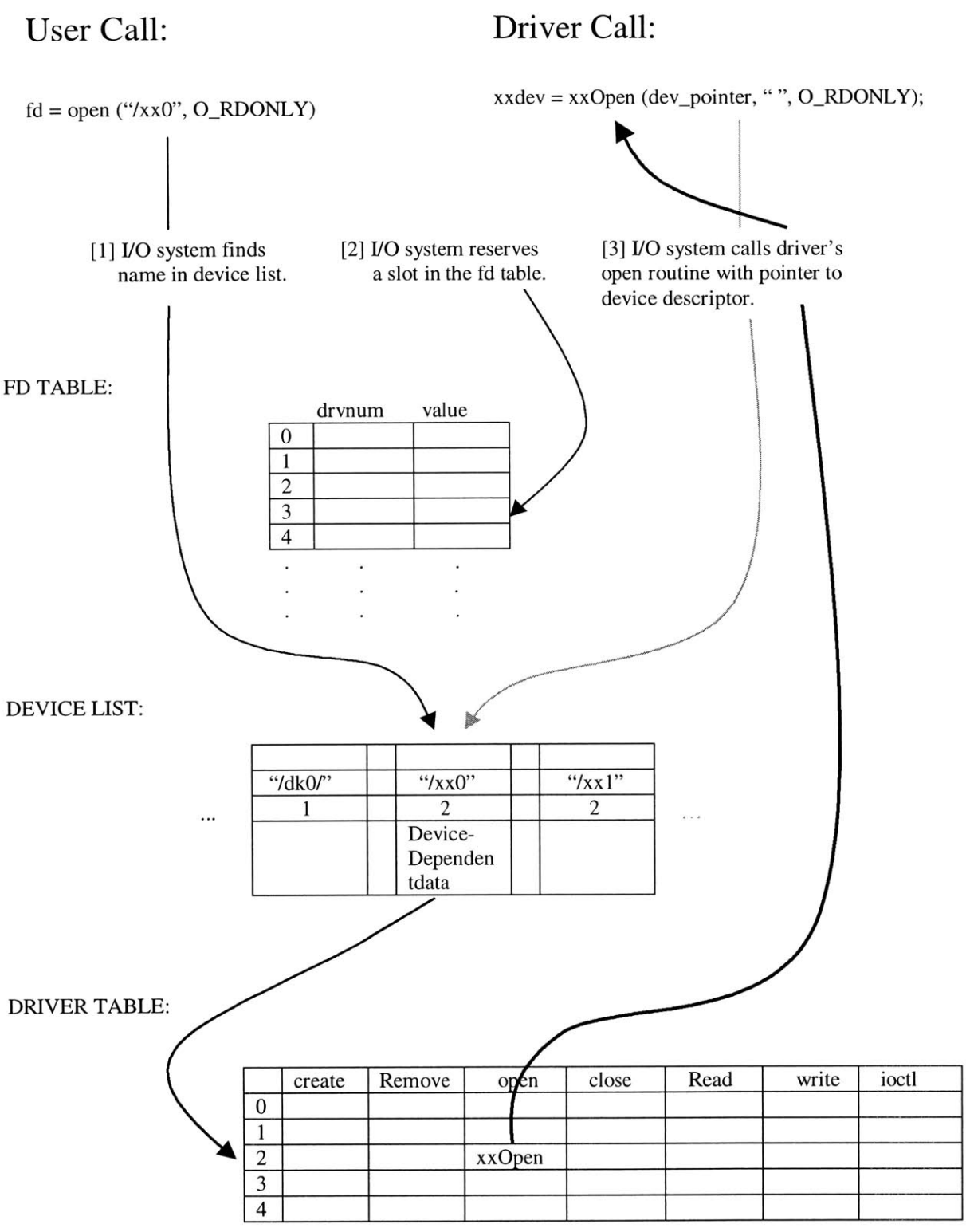

Figure 7: Call to I/O Routine **`Open()`**. [part 1]

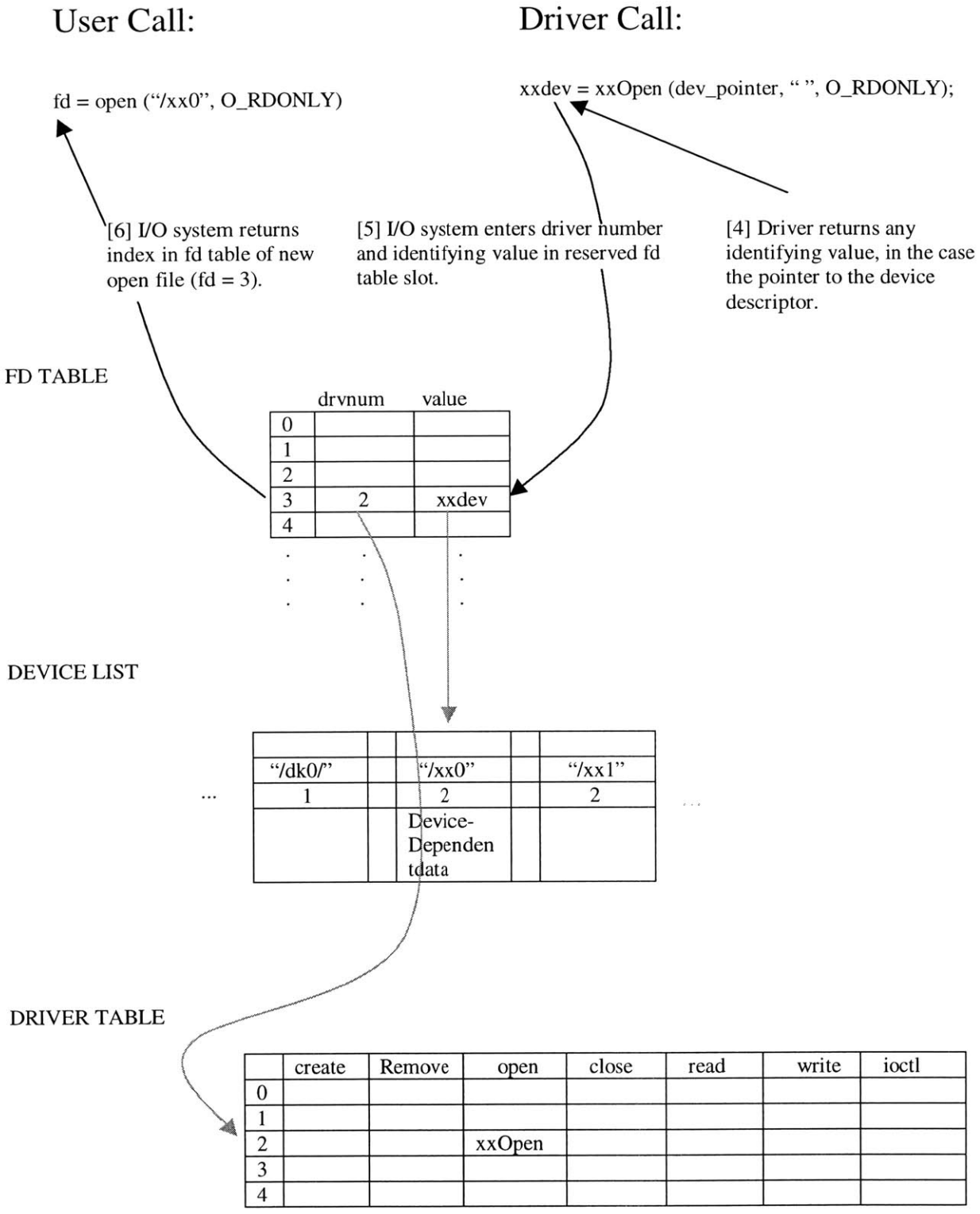

Figure 8: Call to I/O Routine **`Open()`**. [part 2]

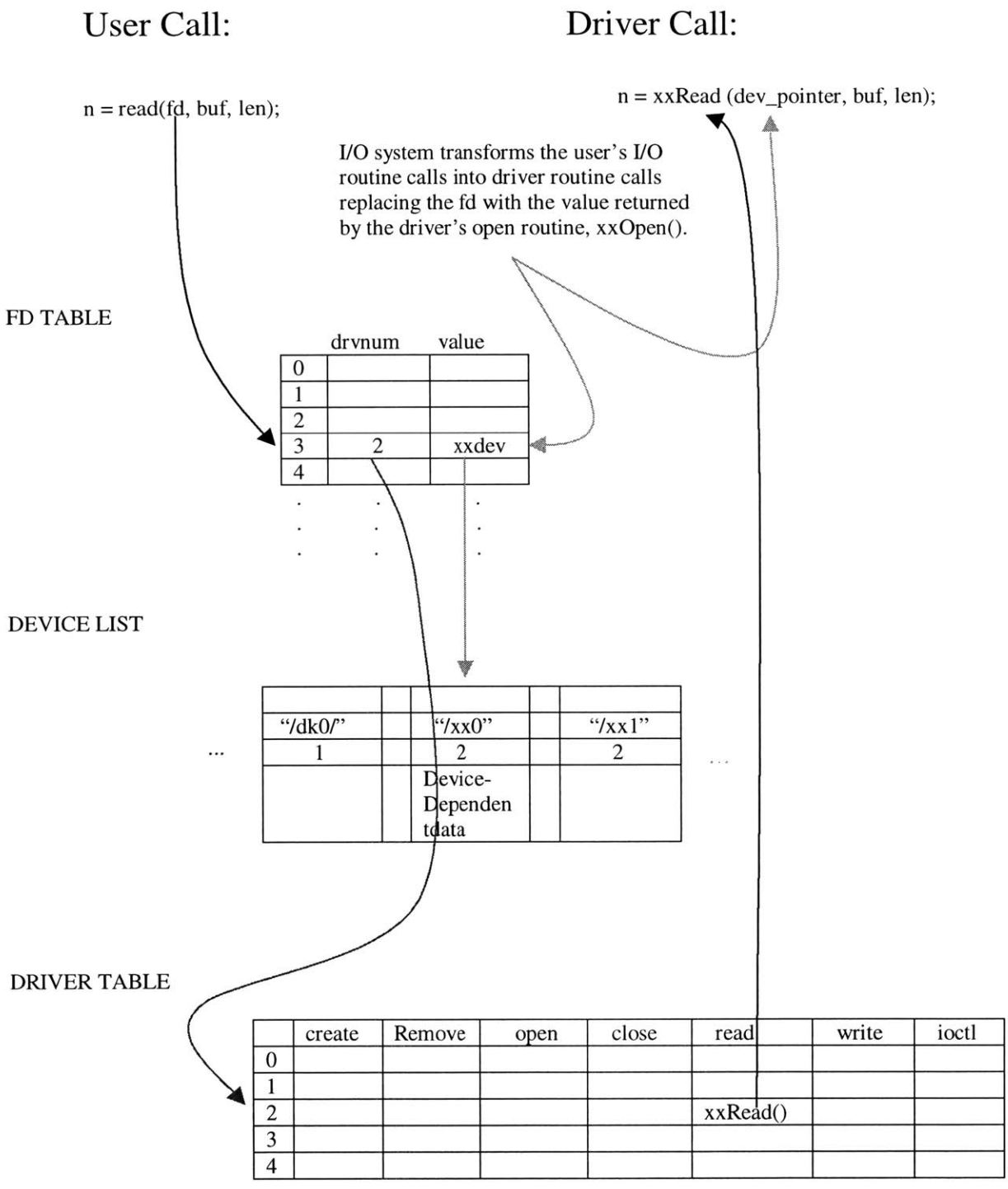

Figure 9: Call to I/O **`read`** Routine.

4.5.5 Closing a File

The user terminates the use of a file by calling **`close()`**. As in the case of **`read()`**, the I/O system uses the driver number contained in the *fd* table to locate the driver's close routine. In the example driver, no close routine is specified; thus no driver routines are called. Instead, the I/O system marks the slot in the *fd* table as being available. Any subsequent references to that *fd* cause an error. However, subsequent calls to **`open()`** can reuse that slot.

4.6 Test for SC26C92 Driver

Most of the modules on Procell communicate with the Intel386EX-based Module Controller Board via serial communication. Therefore, it is pivotal to ensure the reliability of the serial device driver.[5]

4.6.1 Test Set-up

One way to test the *SCC* driver is the *echo* method. This test method involves a very simple set-up: one Module Controller Board, an extension I/O board, and a cable connecting the two communication devices/channels to be tested. See Figure 10 for the illustration of the set-up. These two communication devices/channels are wired up with the method of Null Modem (Figure 11). The aim is to make the computer think it is talking to a modem rather than another computer. Any data transmitted from the first communication device/channel must be received by the second thus TD is connected to RD. The second device/channel must have the same set-up thus RD is connected to TD.

Signal Ground (SG) must also be connected so both grounds are common to each device/channel.

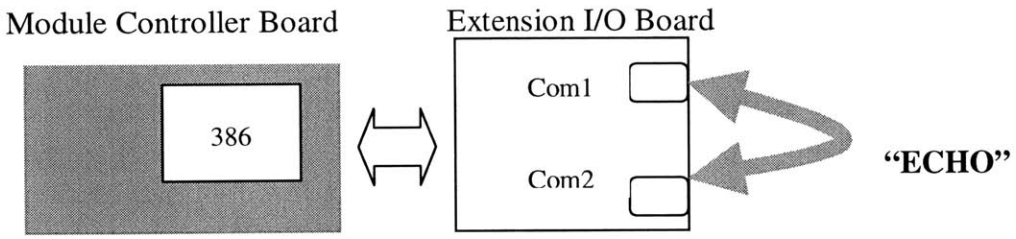

Figure 10: Set-up for "Echo" Test.

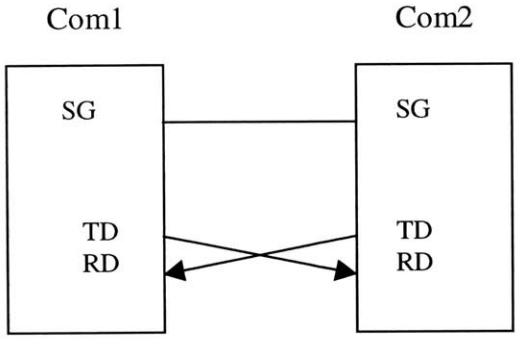

Figure 11: Wiring diagram between communication channel 1&2 (Null Modem Wiring).

4.6.2 "Echo"

The following text describes *echo* in five steps, which are also illustrated in a block diagram in Figure 12.

I. Write a message in the output buffer of communication channel 1 using the **`write()`** I/O routine.

[5] SC26C92 driver is referred to as *SCC* driver in this thesis.

II. Since channel 1's TD line is wired together with channel 2's RD line, reading from channel 2 via a **`read()`** routine gets the message sent from channel 1 and puts it in channel 2's input buffer.

III. Then use the **`write()`** routine to place that message read from channel 2 into channel 2's output buffer.

IV. Likewise, since channel 2's TD line is wired together with channel 1's RD line, reading from channel 1 now will get that message.

V. Compare this received message with the originally sent message at the beginning of the process to see if they have the same content. If they do, the echo test returns a positive result.

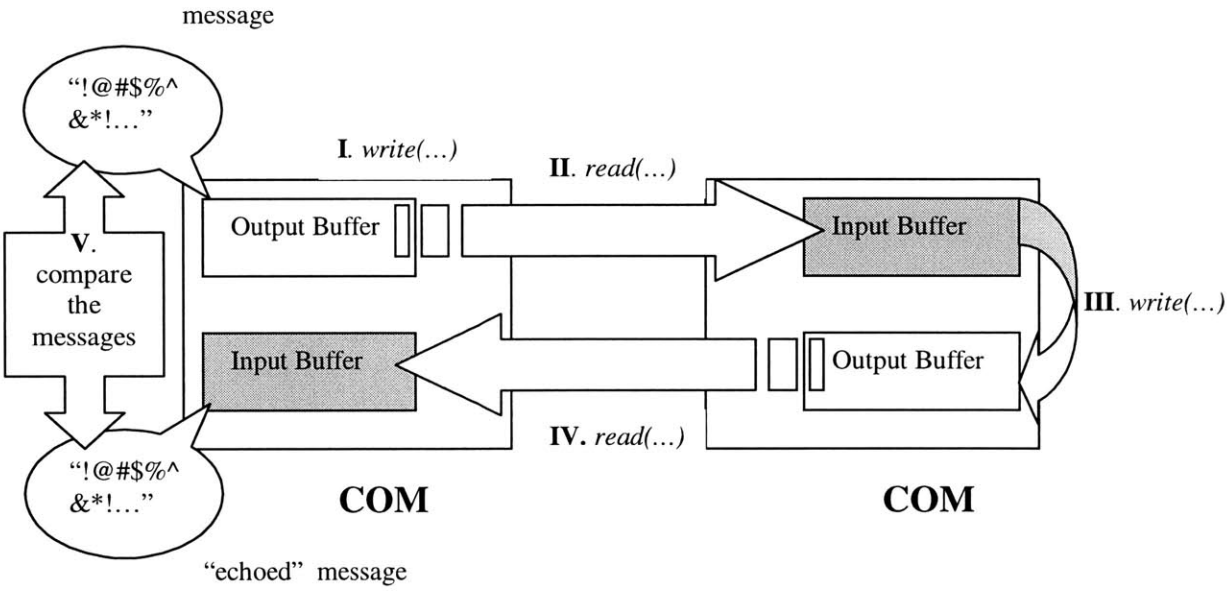

Figure 12: Block Diagram of *Echo* between Com1 and Com2.

4.6.3 *SCC* Test

This section describes in detail, in terms of VxWorks, how to implement the idea of echo method in doing the *SCC* test.

- Initialize the *SCC* driver via a call to a **tySCCDrv()** routine. As mentioned in an earlier section, **tySCCDrv()** has to be called exactly once before any terminal devices can be created.
- Create two terminal devices: channel 1 and channel 2. Specify the input/output buffer size for each terminal, i.e. 1024 bytes for buffer size. Specify the baud rate, i.e. 19200bps. The devices have to be successfully created in order for any data transmission to happen. Every device has a unique name. No multiple devices can share the same device name. If a device with a specified name has already been created, an error message will be returned at a second attempt to create a device with that name. In VxWorks window, a **devs** command at the prompt will show the user a list of currently existing devices.
- Before I/O can be performed to a device, an *fd* must be opened to that device by invoking the **open()** routine (discussed in detail in the previous section). The arguments to **open()** are the file/device name, the type of access (open for reading only, writing only or both), and, when necessary the mode.
- Discard all bytes in the input and output buffers in each device. A basic I/O command **ioctl()** can be used here:

  i.e. **ioctl = (fd, FIOFLUSH, 0);**

  FIOFLUSH is an I/O control function that does the above job. This is an important step, because the input and output buffers in the devices may have data garbage in

them. To begin the echo test, all the buffers need to be cleared of garbage, or the result of the echo test will not reflect accurately the functionality of the serial device driver.

- Start the echo process. First, write a message to Com 1 using the **`write()`** routine. The **`write()`** routine takes three arguments: the file descriptor (*fd*) of the device the message is to be written to, the address of the buffer that contains the data message to be output, and the number of bytes to be written. **`write()`** puts data in device's output buffer.

  i.e. **`write(fd, InBuffer, strlen(InBuffer));`**

- Read from Com 2. Note that Com 1's data transmission (TD) line is connected with Com 2's data receiving (RD) line. The **`read()`** routine takes three arugements: the file descriptor (*fd*) of the device, the address of the buffer to receive input, and the maximum number of bytes to read.

  i.e. **`read(fd, &EchoBuffer, maxBytes);`**

  The **`read()`** routine waits for input to be available from the specified device, and returns the number of bytes actually read. However, in the echo test development stage, it's been noticed that the data written to Com 1's output buffer is not available for input at Com 2 right away. The **`read()`** routine sometimes only returns the first part of the data written to Com 1. In order to prevent the above from occuring, enough time ought to be set aside for the data to be available for input at Com 2 before the **`read()`** routine is called. VxWorks, being a real-time operating system, provides direct control over a task's execution. The routine **`taskDelay()`** postpones a task in VxWorks. Calling the routine **`taskDelay()`** between writing

to Com 1 and reading from Com 2 guarantees the integrity of the data to be transmitted. The read data are saved in **`EchoBuffer`**.

- Here starts the second half of the echo test. Transfer what is in the EchoBuffer to the output buffer of Com 2 via calling another **`write()`** routine follwed by calling another **`taskDelay()`**. As mentioned before, a **`taskDelay()`** routine will make sure the complete data are available in Com 2's output buffer for input to another channel later.
- Read from Com 1. Note that Com 2's data transmission (TD) line is connected with Com 1's data receiving (RD) line. The data that have just been read get to be saved in the input buffer of Com 1.
- Compare the content in the input buffer of Com 1 with the original message written to Com 1 at the beginning of the echo process. If they are the same, the echo test returns a positive result for Com 1 and Com 2's device driver. Otherwise, the test returns error result.

4.7 Summary

This chapter focused on the devices and drivers for RS-232 serial communications in ProCell. Philips SC26C92 DUART was desribed in section 4.3. A general introduction to devices and drivers in VxWorks was given in section 4.5. In section 4.6, a set of diagnostic procedures called *echo* was introduced to test the functionality of the *SCC* driver. The source code for the *SCC* driver and the *SCC* test can be found in the Appendices.

Chapter 5: Controller Area Network (CAN)

5.1 Introduction

In the ProCell system, peer Module Controller Boards communicate with one another via CAN. CAN is a protocol for serial data transfer. Developed in automobile industry, CAN was first adopted for serial communication in vehicles. The CAN network protocol detects and corrects transmission errors caused by electromagnetic interference. Additional advantages such as low cost, high real-time capabilities and ease of use make CAN chosen by manufacturers in semiconductor and semiconductor equipment industry as well many other industries. Section 5.2 describes how the CAN network functions. 5.3 and 5.4 focus on the CAN controller 82527 and its driver in VxWorks.

5.2 How CAN Works

5.2.1 Principles of Data Exchange

When data are transmitted by CAN, the content of the data messages does not contain addresses of either the transmitting mode, or of any receiving node. Instead, the message is labeled by an identifier that is unique throughout the network. The identifier defines not only the content but also the priority of the message. The lower the numerical value of the identifier is, the higher the priority the message has. A message with the highest priority is guaranteed to gain bus access as if it were the only message being transmitted. This is important for bus allocation when several stations are competing for bus access.

If the CPU of a given station wishes to send a message to one or more stations, it passes the data to be transmitted and their identifiers to the assigned CAN controller chip. This is all the CPU has to do to initiate data exchange. The message is constructed and transmitted by CAN controller. As soon as the CAN chip receives the bus allocation, all other stations on the CAN network become receivers of this message. Each station in the CAN network, having received the message correctly, conducts an acceptance test to decided whether the data received are relevant for that station. If the data are of no significance for the station concerned, they are ignored. Otherwise they get processed.

The above described content-oriented addressing scheme of the CAN network has the advantage of a high degree of system and configuration flexibility. When more stations are added to the existing CAN network, there is usually no need to make any hardware or software modifications to the already existing stations, as long as the new stations are pure receivers. Because the data transmission protocol does not require physical destination addresses for the individual components, it permits multiple reception and the synchronization of distributed processes. For instance, measurements needed as information by several modules in the system can be transmitted via the network, in such a way that it is not necessary for each module to have its own sensor.

5.2.2 CAN Protocol

In a CAN system, data are transmitted and received using Message Frames. Message Frames carry data from a transmitting node to one or more receiving nodes in the CAN network.

The CAN protocol supports two message frame formats: the Standard CAN protocol (version 2.0A) and the Extended CAN protocol (version 2.0B). The only essential difference between these two protocols is in the length of the identifier. In the standard format the length of the identifier is 11 bits and it is 29 bits in the extended format.

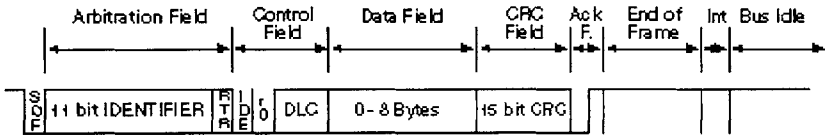

Figure 13: Message frame for standard format (CAN specification 2.0A).

The message frame for transmitting message on the bus consists of seven main fields. A message in the standard format (see Figure 13) [CiA '99] begins with the start bit "start of frame" (SOF). It is followed by the "arbitration field", which contains the identifier and the "RTR" (remote transmission request) bit, which indicates whether is a data frame or a request frame without any data bytes. The "control field" contains the IDE ("identifier extension) bit, which indicates either standard format or extended format, a bit reserved for future extensions and a four bit Data Length Code (DLC), a count of the data bytes in the data field. The "data field" ranges from 0 to 8 bytes in length and is followed by the "CRC field", which is used as a frame security check for detecting bit errors. The "ACK field" comprises the ACK slot (1 bit) and the ACK delimiter (1 recessive bit). The bit in the ACK slot is sent as a recessive bit and is overwritten as a dominant bit by those receivers who have at this time received the data correctly (positive acknowledgement). Correct messages are acknowledged by the

receivers regardless of the result of the acceptance test. The end of the message is indicated by "end of frame", seven recessive bits. "Intermission" is the minimum number of bit periods separating consecutive messages. If there is no following bus access by any consecutive message, the bus remains idle ("bus idle").

5.2.3 Error Detection

One of the outstanding features of the CAN protocol is its high transmission reliability. The CAN controller registers a station's error and evaluates it statistically in order to take appropriate measures.

CAN implements five error detection mechanisms; three at the message level and two at the bit level. The ones at message level are Cyclic Redundancy Checks (CRC), Frame Checks, and Acknowledgement Error Checks. At the bit level, there are bit monitoring and bit stuffing.

Refer to Figure 13, every transmitted message contains a 15 bit "CRC field". The CRC is computed by the transmitter and is based on the message content. All receivers that accept the message performs a similar calculation and flag any error.

There are certain predefined bit values that must be transmitted at certain points within any CAN Message Frame. When the frame check of a receiver detects an invalid bit in one of these positions, a Format Error will be flagged.

In a Acknowledgement (ACK) Error Check, if a transmitter determines that a message has not been acknowledged, then an ACK Error is flagged.

What bit monitoring does is that any transmitter automatically monitors and compares the actual bit level on the bus with the level that is transmitted. If the two levels are not the same, a bit error is flagged.

Bit stuffing is a technique CAN uses to check on communication integrity. After five consecutive identical bit levels have been transmitted, the transmitter will automatically inject (stuff) a bit of the opposite polarity into the bit stream. Receiver of the message will automatically delete (de-stuff) such bits before processing the message in any way. Because of the bit stuffing rule, if any receiving node detects six consecutive bits of the same level, a stuff error is flagged.

5.3 CAN Controller 82527

The 82527 serial communication controller is a highly integrated device that performs serial communication according to the CAN protocol. It supports both the standard and extended message frames in CAN specification.

The 82527 consists of six functional blocks including the CAN controller, RAM, CPU interface logic, clockout and two 8-bit ports. The CPU interface logic manages the interface between the CPU (Intel 386EX microprocessor) and the 82527 using an address/data bus. The CAN controller interfaces to the CAN bus and implements the protocol rules of the CAN protocol for the transmission and reception of messages. The RAM is the interface layer between the CPU and the CAN bus. The two port blocks provide 8-bit low speed I/O capability. The clockout block allows the 82527 to drive other chips [Intel '95]. See Figure 14 for the physical CAN connection [CiA '99].

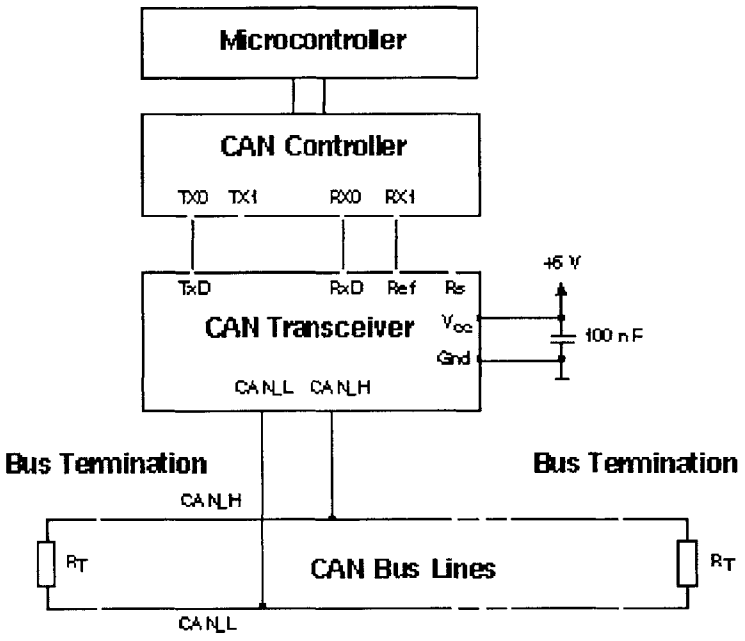

Figure 14: Physical CAN connection.

5.4 Driver for CAN Controller 82527

In section 4.5 there was a brief description on devices, drivers and I/O system in VxWorks. The driver for the 82527 in VxWorks is very similar to the driver for Philips SC26C92 DUART discussed in Chapter 4. This section describes the CAN driver-level interface to the I/O system.

`CanDrv()`

`CanDrv()` installs the CAN driver in the I/O system. It is the first thing that must be done before any devices are added to the system or any I/O request is performed. It returns either OK or ERROR.

CanDevCreate()

This routine is called to add a device to the system that will be serviced by the CAN driver. This function must be called before performing any I/O request to this device. There are several device-dependent arguments required for the device initialization and allocation of system resources.

The CAN driver supports the basic I/O interface functions, including **open()**, **read()**, **write()** and **ioctl()**. A few other important I/O control functions are described below.

FIO_SETFILTER

This I/O control function modifies the acceptance filter masks of the CAN controller. The acceptance masks allow message objects to receive messages with a range of message identifiers instead of just a single message identifier. A "0" means "don't care" and accepts a "0" or "1" for that position, while "1" value means that the incoming bit value "must-match" identically to the corresponding bit in the message identifier. The syntax looks like:

```
int result = ioctl (fd, FIO_SETFILTER, &AcceptMasks);
```

fd is the file descriptor (see Chapter 4 for definition), **AcceptMasks** is a pointer to a data structure which contains the specification for the masks.

FIO_GETFILTER

This I/O control function returns the contents of the acceptance filter masks of the CAN controller in the data structure **AcceptMasks**. The syntax, for example, can be:

```
int result = ioctl (fd, FIO_GETFILTER, &AcceptMasks);
```

FIO_BUSON

After an abnormal rate of occurrences of errors on the CAN bus, the CAN controller enters the *busoff* state. This I/O control function resets the Initial bit the Control register. The CAN controller begins the *busoff* recovery sequence. This bus recovery sequence resets the transmit and receive error counters. If the CAN controller counts 128 packets of 11 consecutive recessive bits on the CAN bus, the *busoff* state is exited. Here is an example:

```
int result = ioctl (fd, FIO_BUSON, 0);
```

FIO_BITTIMING

This I/O control function modifies the bit timing register of the CAN controller. Here is how the function is used:

```
int result = ioctl (fd, FIO_BITTIMG, &dc);
```

dc is a data structure that holds the new values for the bit timing register 0 (bit 8 to 15) and the bit timing register 1 (bit 0 to 7). Possible transfer rates are between 5kbit per second and 1.6 Mbit per second.

The above listed are a few of the important I/O control functions the CAN driver supports. Refer to **Canif.cpp** in the appendix for the implementation of those functions in the CAN interface.

5.5 Summary

Controller Area Network (CAN) is a protocol originally developed in automobile industry for serial data transfer. Its low cost and high reliability makes CAN a widely used network protocol in various industries. This chapter explained how CAN works and

described its unique error detection features. In section 5.3 and 5.4, CAN controller 82527 and its device driver in VxWorks were described.

Chapter 6: Parallel Communication

6.1 Introduction

The Intel386EX microprocessor communicates with a motion control chipset on the module controller board via parallel data transfer. This chapter will discuss the interface between the microprocessor and two of coater's arms and describe the parallel communication between the microprocessor and the chipset that controls the movement of the arms.

6.2 Interface between 386 and Coater's Arms

Coater is a module that applies photoresist and various chemicals on wafers uniformly, and gives the surface of a wafer the characteristics similar to a piece of photographic paper. It consists of two robot arms: a Dispense Arm, which dispenses the chemicals onto the wafer, and a Top Edge Bead Remover (TEBR) Arm, which does the job of cleaning and smoothing the edge of the wafer after chemicals get dispensed on the surface of the wafer.

Both the Dispense Arm and the TEBR Arm are controlled by an advanced multi-axis motion control chipset, PMD's MC1401A. The 386, which serves as a host processor, interfaces with the motion control chip via an 8-bit bi-directional bus and various control signals. Host communication is coordinated by a ready/busy signal, which indicates when communication is allowed.

Refer to **`pmdio.c`** in the Appendix for the low-level routines that enable the communication between 386 and the motion control chipset. **`pmdio.c`** contains low-level routines that write/read a single byte command/data to the chipset, routines that

write/read a 2-byte data word to/from the chipset and maintains the checksum word (Checksum will be explained in a later section), and the main routine called **`send_chipset_cmd`**, which is called to send complete commands to the chipset to control motion. On the top of **`pmdio.c`**, a **`motion.c`** program (Refer to the Appendix.) allows users to move the Dispense Arm and TEBR Arm to the positions desired and completes the motion control interface.

6.3 Advanced Multi-Axis Motion Control Chipset, MC1401A by PMD

The MC1401A is a 2-IC general purpose motion control chipset, packaged in 2 68-pin PLCC packages. The Peripheral Input/Output IC (I/O chip) is responsible for interfacing to the host processor (386 in this case), and to the position input encoders. The Command Processor IC (CP chip) is responsible for all host command, profile and servo computations, as well as for outputting the PWM and DAC signals.

Figure 15 shows a typical system block diagram, along with the pin connectors between the I/O chip and the CP chip.

The CP and I/O chips function together as one integrated motion processor. The major components connected to the chip set are the Encoder, the motor amplifier, and the host processor (386). The 386 is interfaced with the chipset via an 8-bit bi-directional bus and various control signals. The communication between the chipset and the 386 is coordinated by a ready/busy signal, which indicates when communication is allowed.

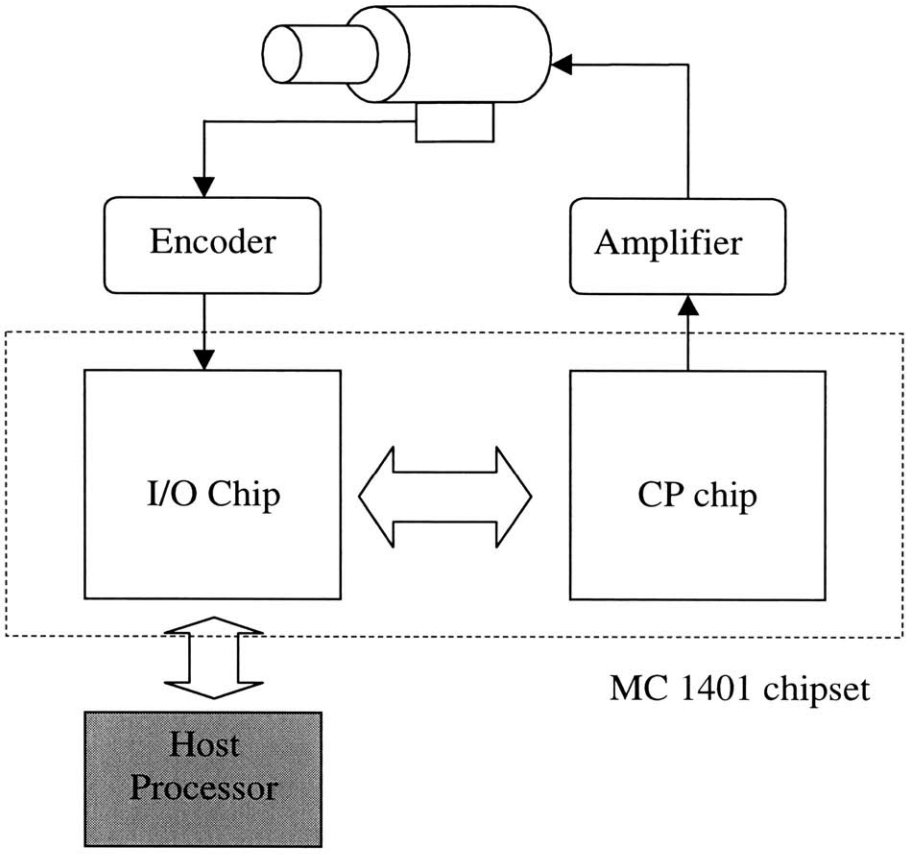

Figure 15: Block diagram of the MC 1401 chipset.

6.4 Data Transfer Protocol

All communications to/from the chip set take the form of packets. A packet is a sequence of transfers to/from the host resulting in a chip set action or data transfer. Packets can consist of a command with no data (Dataless Command), a command with associated data that is written to the chip set (Write Command) or a command with associated data that is read from the chip set (Read Command).

The following charts [Performance '97] show the generic command packet sequence for a Dataless Command, a Write Command, and a Read Command.

Dataless Command				
Time	->	->	->	->
Command Write:		Command byte		
Data Write:				
Data Read:		[packet checksum]		

Write Command				
Time	->	->	->	->
Command Write:		Command byte		
Data Write:		word 1	[word 2]	
Data Read:				[packet checksum]

Read Command				
Time	->	->	->	->
Command Write:		Command byte		
Data Write:				
Data Read:		Word 1	[Word 2]	[packet checksum]

The above charts show that at the end of each packet, a checksum word is available for reading, which provides a reliability enhancement. This checksum consists of a 16-bit sum of all previous communications that have occurred for the associated command. The command byte is included in the low byte of the 1st checksum word (with high byte set to 0). Data words are added as is to the checksum value. For example, if a

SET_VEL (set velocity) command (which takes two 16-bit words of data) was set with a data value of fedcba98, the checksum would be [Performance '97]:

```
  0011 (code for SET_VEL command)
+ fedc (high data word)
+ ba98 (low data word)
-----------------------------------------------
  1b985
checksum = b985 (keep the bottom 16 bits only)
```

6.5 Summary

Chapter 6 contained the description of the parallel communication between the host microprocessor and a motion control chipset. The MC1401A chipset provides closed-loop digital servo control for motors used in moving coater's arms in ProCell system. The interface both in software and hardware between the Intel386EX and the motors were described in section 6.2 and 6.3. The data transfer protocol was illustrated in section 6.4. The **`pmdio.c`** and **`motion.c`** in the Appendices will help better understanding the low-level interface.

Chapter 7: Conclusions

This chapter briefly reviews the communication links in SVG's photoresist process system, ProCell. ProCell's software control system contains a multi-layered architecture. From its system control that provides user-interface on the top operating level, to the middle machine control, to the distributed module control, communication links stand to be very important issues in the successful automation of wafer handling. The focus of this thesis was placed on the module control level in the ProCell system, mainly the three communication protocols associated with the Intel386EX host microprocessor.

Distributed module control, as opposed to control of modules at high system level, not only enables stand-alone module tests to be carried out, but also reduces the information and signals that communicate to the system control level. On ProCell's module control level, the Intel386EX microprocessor serves as the core of the module controller. Those microprocessors not only directly interface with peripheral devices to pass commands and receive feedbacks from the modules, but also communicate with one another to coordinate the tasks of wafer handling from system level.

The Intel386EX microprocessor communicates with the modules through two types of communication protocols, RS-232 serial communication and 8-bit bi-directional parallel communication. The communication between any two module controller boards are done via Controller Area Network (CAN). The peripheral devices for RS-232 serial communication and CAN are SC26C92 and 82527 respectively. The thesis work

includes program routines that let these two devices communicate with the operating system, in this case VxWorks. These program routines can be found in the Appendix.

ProCell is to replace SVG's 90 Series Photoresist Processing System (9X) and 200 Advanced Processing System (200APS). Distributed control (control of modules at module level) has operated successfully on 9X platform for over ten years. Due to the limited time of the author's internship at SVG, ProCell system was still in its development phase when this thesis was done. There was still a large amount of work ahead to be done before ProCell would finally be up and running. For instance, in the area related to the work of this thesis, diagnostic procedures need to be developed for communication link 5's Controller Area Network, to ensure successful data trasfer between modules. These procedures would be similar to yet more complex than the *SCC* test illustrated in Chapter 4 due to CAN's unique features.

Go, ProCell!

Appendices

Appendix A: Source Code for *SCC* Driver (`scc.c`)

```
/* scc.c - tty driver for the Philips SC26C92 serial controller board */

/*
DESCRIPTION
This is the driver for the Philips SC26C92 DUART controller.

USER CALLABLE ROUTINES
Most of the routines in this driver are accessible only through the I/O
system.  Two routines, however, must be called directly: tySCCDrv () to
initialize the driver, and tySCCDevCreate to create devices.

TYSCCDRV
Before using the driver, it must be initialized by calling the routine:
.CS
    tySCCDrv ()
.CE
This routine should be called exactly once, before any reads, writes, or
tySCCDevCreate's.  Normally, it is called from usrRoot (2) in =
usrConfig.c.

CREATING TERMINAL DEVICES
Before a terminal can be used, it must be created.
This is done with the tySCCDevCreate call.
Each port to be used should have exactly one device associated with it,
by calling this routine.

.CS
 STATUS tySCCDevCreate (name, channel, rdBufSize, wBufSize, baudRate)
     char *name;      // Name to use for this device *
     int channel;     // Physical channel for this device (0 - 1) *
     int rdBufSize;   // Read buffer size, in bytes *
     int wBufSize;    // Write buffer size, in bytes *
     int baudRate;    // Baud rate to create device with *

.CE

For instance, to create the device "/tySCC/0", with buffer sizes of 512
=
bytes,
at 9600 baud, the proper call would be:
.CS
   tySCCDevCreate ("/tySCC/0", 0, 512, 512, 9600);
.CE
IOCTL
This driver responds to all the same ioctl codes as a normal ty driver.
All baud rates between 110 and 19200 are available.
*/



#include "vxWorks.h"
#include "iv.h"
#include "ioLib.h"
#include "iosLib.h"
#include "tylib.h"
#include "sccLib.h"
#include "intLib.h"
```

```
#include "errnoLib.h"
#include "sysLib.h"
#include "stdio.h"

#include "sc26c92.h"

#define INT_NUM_IRQ0          0x20

#define RX0_INT_LVL           0x01
#define RX1_INT_LVL           0x05
#define RX2_INT_LVL           0x06
#define RX3_INT_LVL           0x07
#define XMT_INT_LVL           0x09
#define MUX_INT_LVL           0x0D

#define RX0_INT_VEC           ( INT_NUM_IRQ0 + RX0_INT_LVL )
#define RX1_INT_VEC           ( INT_NUM_IRQ0 + RX1_INT_LVL )
#define RX2_INT_VEC           ( INT_NUM_IRQ0 + RX2_INT_LVL )
#define RX3_INT_VEC           ( INT_NUM_IRQ0 + RX3_INT_LVL )
#define MUX_INT_VEC           ( INT_NUM_IRQ0 + MUX_INT_LVL )

#define SCC_BASE_0            0x1000
/* <@ WCS033099 - used to 0x14A0 - changed to avoid problem with Motion
=
Control $> */
/*#define SCC_BASE_1          0x18A0 change again on 10/25 according to
Sun Phi*/
#define SCC_BASE_1            0x14A0
#define SCC_BASE_2            0x1CA0

typedef struct
{
     int        rate;
     int        preset;
}    BAUD;


typedef struct
{
     USHORT     ipcr;             /* input port change register */
     USHORT     isr;              /* interrupt status register */
     USHORT     ctu;              /* counter/timer upper */
     USHORT     ctl;              /* counter/timer lower */
     USHORT     ipr;              /* input port */
     USHORT     startcntr;        /* start counter command */
     USHORT     stopcntr;         /* stop counter command */
     USHORT     acr;              /* aux. control register */
     USHORT     imr;              /* interrupt mask register */
     USHORT     ctpu;             /* c/t upper preset register */
     USHORT     ctpl;             /* C/T lower preset register */
     USHORT     opcr;             /* output port conf. register */
     USHORT     sop12;            /* set output port bits register */
     USHORT     rop12;            /* reset output port bits register */
     USHORT     imr_d;            /* data of imr */
}    TY_SCC_DEV;

typedef struct                  /* TY_SCC_DEV */
{
     TY_DEV     tyDev;
     TY_SCC_DEV *   ptySCC;
     BOOL       created;
     BOOL       isA;
     USHORT     rcv_int_lvl;
     USHORT     rcv_int_vec;
```

```
    USHORT    mr012;
    USHORT    sr;
    USHORT    rxfifo;
    USHORT    csr;
    USHORT    cr;
    USHORT    txfifo;
}   TY_SCC_CHL;

LOCAL TY_SCC_DEV    tySCCdev[3] =
    {{
     SCC_BASE_0 + 0x08,
     SCC_BASE_0 + 0x0A,
     SCC_BASE_0 + 0x0C,
     SCC_BASE_0 + 0x0E,
     SCC_BASE_0 + 0x1A,
     SCC_BASE_0 + 0x1C,
     SCC_BASE_0 + 0x1E,
     SCC_BASE_0 + 0x08,
     SCC_BASE_0 + 0x0A,
     SCC_BASE_0 + 0x0C,
     SCC_BASE_0 + 0x0E,
     SCC_BASE_0 + 0x1A,
     SCC_BASE_0 + 0x1C,
     SCC_BASE_0 + 0x1E
     },
     {
     SCC_BASE_1 + 0x08,
     SCC_BASE_1 + 0x0A,
     SCC_BASE_1 + 0x0C,
     SCC_BASE_1 + 0x0E,
     SCC_BASE_1 + 0x1A,
     SCC_BASE_1 + 0x1C,
     SCC_BASE_1 + 0x1E,
     SCC_BASE_1 + 0x08,
     SCC_BASE_1 + 0x0A,
     SCC_BASE_1 + 0x0C,
     SCC_BASE_1 + 0x0E,
     SCC_BASE_1 + 0x1A,
     SCC_BASE_1 + 0x1C,
     SCC_BASE_1 + 0x1E
     },
     {
     SCC_BASE_2 + 0x08,
     SCC_BASE_2 + 0x0A,
     SCC_BASE_2 + 0x0C,
     SCC_BASE_2 + 0x0E,
     SCC_BASE_2 + 0x1A,
     SCC_BASE_2 + 0x1C,
     SCC_BASE_2 + 0x1E,
     SCC_BASE_2 + 0x08,
     SCC_BASE_2 + 0x0A,
     SCC_BASE_2 + 0x0C,
     SCC_BASE_2 + 0x0E,
     SCC_BASE_2 + 0x1A,
     SCC_BASE_2 + 0x1C,
     SCC_BASE_2 + 0x1E
     },
    };

LOCAL TY_SCC_CHL    tySCCchl[6] =
    {
       {
       {{{NULL}}},
       &tySCCdev[0],
```

```
FALSE,
TRUE,
RX0_INT_LVL,
RX0_INT_VEC,
SCC_BASE_0 + 0x00,
SCC_BASE_0 + 0x02,
SCC_BASE_0 + 0x06,
SCC_BASE_0 + 0x02,
SCC_BASE_0 + 0x04,
SCC_BASE_0 + 0x06
},
{
{{{NULL}}},
&tySCCdev[0],
FALSE,
FALSE,
RX1_INT_LVL,
RX1_INT_VEC,
SCC_BASE_0 + 0x10,
SCC_BASE_0 + 0x12,
SCC_BASE_0 + 0x16,
SCC_BASE_0 + 0x12,
SCC_BASE_0 + 0x14,
SCC_BASE_0 + 0x16
},
{
{{{NULL}}},
&tySCCdev[1],
FALSE,
TRUE,
RX2_INT_LVL,
RX2_INT_VEC,
SCC_BASE_1 + 0x00,
SCC_BASE_1 + 0x02,
SCC_BASE_1 + 0x06,
SCC_BASE_1 + 0x02,
SCC_BASE_1 + 0x04,
SCC_BASE_1 + 0x06
},
{
{{{NULL}}},
&tySCCdev[1],
FALSE,
FALSE,
RX3_INT_LVL,
RX3_INT_VEC,
SCC_BASE_1 + 0x10,
SCC_BASE_1 + 0x12,
SCC_BASE_1 + 0x16,
SCC_BASE_1 + 0x12,
SCC_BASE_1 + 0x14,
SCC_BASE_1 + 0x16
},
{
{{{NULL}}},
&tySCCdev[2],
FALSE,
TRUE,
RX2_INT_LVL,
RX2_INT_VEC,
SCC_BASE_2 + 0x00,
SCC_BASE_2 + 0x02,
SCC_BASE_2 + 0x06,
SCC_BASE_2 + 0x02,
```

```
        SCC_BASE_2 + 0x04,
        SCC_BASE_2 + 0x06
        },
        {
        {{{NULL}}},
        &tySCCdev[2],
        FALSE,
        TRUE,
        RX3_INT_LVL,
        RX3_INT_VEC,
        SCC_BASE_2 + 0x10,
        SCC_BASE_2 + 0x12,
        SCC_BASE_2 + 0x16,
        SCC_BASE_2 + 0x12,
        SCC_BASE_2 + 0x14,
        SCC_BASE_2 + 0x16
        }
    };

LOCAL BAUD baudTable[] =
    {
#if  0
    /* <@ WCS033099 - ACR[7] = 1 $> */
    { 50, 0 }, { 110, 0x11 }, { 200, 0x33 }, { 300, 0x44 }, { 600, 0x55
=
}, { 1200, 0x66 },
    {1050, 0x77 }, { 2400, 0x88 }, { 4800, 0x99 }, { 7200, 0xAA }, { =
9600, 0xBB }, { 38400, 0xCC }
#else
    /* <@ WCS033099 - ACR[7] = 0 $> */
    { 50, 0 }, { 110, 0x11 }, { 150, 0x33 }, { 300, 0x44 }, { 600, 0x55
}, { 1200, 0x66 },
    {2000, 0x77 }, { 2400, 0x88 }, { 4800, 0x99 }, { 1800, 0xAA }, {
9600, 0xBB }, { 19200, 0xCC }
#endif
    };

LOCAL int tySCCDrvNum;        /* driver number assigned to this driver
*/


/* forward declarations */

LOCAL int tySCCOpen ();
LOCAL int tySCCRead ();
LOCAL int tySCCWrite ();
LOCAL int tySCCIoctl ();
LOCAL VOID tySCCStartup ();
LOCAL VOID tySCCrxInt ();
LOCAL VOID tySCCtxInt ();
LOCAL VOID tySCCMuxInt ();
LOCAL VOID tySCCHrdInit();

/*******************************************************************************
*=
*******
*
* tySCCDrv - install Philips SC26C92 driver
*
* RETURNS:
*    OK or
*    ERROR if board not present or unable to install driver
*/
```

```
STATUS tySCCDrv ()

{
     static BOOL done;    /* FALSE = not done; TRUE = done */


     if (!done)
     {
        /* connect to interrupt level and initialize hardware */

        intConnect( INUM_TO_IVEC( tySCCchl[0].rcv_int_vec ), tySCCrxInt,
(int) &tySCCchl[0] );
        intConnect( INUM_TO_IVEC( tySCCchl[1].rcv_int_vec ), tySCCrxInt,
(int) &tySCCchl[1] );
        intConnect( INUM_TO_IVEC( tySCCchl[2].rcv_int_vec ), tySCCrxInt,
(int) &tySCCchl[2] );
        intConnect( INUM_TO_IVEC( tySCCchl[3].rcv_int_vec ), tySCCrxInt,
(int) &tySCCchl[3] );

        /* connect to mux interrupt for channel 5 & 6 */
        intConnect( INUM_TO_IVEC( MUX_INT_VEC ), tySCCMuxInt, (int)
&tySCCchl[0] );

        tySCCHrdInit();

        /* install driver */

        tySCCDrvNum = iosDrvInstall (tySCCOpen, (FUNCPTR) NULL,
                                     tySCCOpen, (FUNCPTR) NULL,
                                     tySCCRead, tySCCWrite, tySCCIoctl);
     }

     return (tySCCDrvNum == ERROR ? ERROR : OK);
}

int  MyOutByte( Port, onebyte )
     int  Port;
     char onebyte;
{
     printf( "<%x:%x>,", Port, onebyte );
     sysOutByte( Port, onebyte );
}

/*******************************************************************************
*=
*******
*
* tySCCDevCreate - create a device for a channel
*/

STATUS tySCCDevCreate (name, channel, rdBufSize, wrtBufSize, baudRate)
     char *name;    /* Name to use for this device */
     int  channel;        /* Physical channel for this device (0 - 3) */
     int  rdBufSize;      /* Read buffer size, in bytes */
     int  wrtBufSize;     /* Write buffer size, in bytes */
     int  baudRate;       /* Baud rate to create device with */

{
     STATUS         status;
     FAST TY_SCC_CHL     *pTyChl = &tySCCchl[channel];

     if ( channel >= N_SCC_CHANNELS )
     {
        return( ERROR );
```

```
    }

    if ( tySCCDrvNum < 1 )
    {
       errnoSet (S_ioLib_NO_DRIVER);
       return (ERROR);
    }

    /* if device exist, don't create it */

    if ( pTyChl->created )
       return (ERROR);

    /* initialize ty device descriptors and baud rate */

    status = tyDevInit ( &pTyChl->tyDev,
                  rdBufSize, wrtBufSize,(FUNCPTR) tySCCStartup );

    if ( status != OK )
       return (ERROR);

    /* enable the receiver and receiver error */

    tySCCIoctl (pTyChl, FIOBAUDRATE, baudRate);

    if ( pTyChl->isA )
    {
       pTyChl->ptySCC->imr_d &= 0x00F0;
       pTyChl->ptySCC->imr_d |= IM_TXA_RDY;
       MyOutByte( pTyChl->ptySCC->imr, (char) pTyChl->ptySCC->imr_d );
    }
    else
    {
       pTyChl->ptySCC->imr_d &= 0x000F;
       pTyChl->ptySCC->imr_d |= IM_TXB_RDY;
       MyOutByte( pTyChl->ptySCC->imr, (char) pTyChl->ptySCC->imr_d );
    }

    if ( pTyChl->rcv_int_vec )
    {
       sysIntEnablePIC( pTyChl->rcv_int_lvl );
    }

    pTyChl->created = TRUE;

    return (iosDevAdd ((DEV_HDR *) pTyChl, name, tySCCDrvNum));
}

/*******************************************************************************
*=
*******
*
* TySCCHrdInit - initialize the SC26C92 controller
*
* This routine initializes the Philips SC26C92.
* same as for the Microbar UNIX system.
*/

#define P1LTC          0xf862

LOCAL VOID tySCCHrdInit ()

{
    FAST TY_SCC_CHL     *pTyChl = &tySCCchl[0];
```

```
    int         i;
    int         oldLevel = intLock();

    /* Reset the SC26C92 */
    MyOutByte( P1LTC, 0 );

    for ( i = 0; i < N_SCC_CHANNELS; i++ )
    {
       MyOutByte( pTyChl->cr, CMD_TX_DISABLE | CMD_RX_DISABLE );
       MyOutByte( pTyChl->cr, CMD_RST_RX );
       MyOutByte( pTyChl->cr, 0 );      /* delay */
       MyOutByte( pTyChl->cr, 0 );
       MyOutByte( pTyChl->cr, 0 );
       MyOutByte( pTyChl->cr, CMD_RST_TX );
       MyOutByte( pTyChl->cr, 0 );      /* delay */
       MyOutByte( pTyChl->cr, 0 );
       MyOutByte( pTyChl->cr, 0 );
       MyOutByte( pTyChl->cr, CMD_SET_MR0 );
       MyOutByte( pTyChl->cr, 0 );      /* delay */
       MyOutByte( pTyChl->cr, 0 );
       MyOutByte( pTyChl->cr, 0 );

       /* Watch Dog Timer Enabled, TX FIFO 6 or more bytes empty,
          RX FIFO 3 or more bytes in FIFO */

       MyOutByte( pTyChl->mr012, MR0_WTD_ENB | MR0_TX_FIFO_6 );

       /* RX FIFO Interrupt Level = 3 or more bytes,
          no parity, 8 bit per character */

       MyOutByte( pTyChl->mr012, MR1_RX_INT0 | MR1_PAR_MODE_NO |
MR1_BITS_CHAR_8 );

       /* normal mode, stop bit */
       MyOutByte( pTyChl->mr012, MR2_CHAN_MODE_NORM | MR2_STOP_BITS_1
);

       /* 9600 baud */
       MyOutByte( pTyChl->csr, RX_CLK_9600 | TX_CLK_9600 );

       if ( pTyChl->isA )
       {
            /* Counter mode, source = External( IP2 ) */
            MyOutByte( pTyChl->ptySCC->acr, 0x80 );      /* <@
WCS033099 - =
19.2K support $> */

            MyOutByte( pTyChl->ptySCC->opcr, OPCR_RXA_INT |
OPCR_RXB_INT | OPCR_TXA_INT | OPCR_TXB_INT );
       }

       MyOutByte( pTyChl->cr, CMD_RX_ENABLE );
       pTyChl++;
    }

    sysIntEnablePIC( MUX_INT_LVL );
    intUnlock( oldLevel );
}

/*******************************************************************
*
* TySCCOpen - open device
*
* ARGSUSED
```

```
*/

LOCAL int tySCCOpen (pTyChl, name, mode)
     TY_SCC_CHL *pTyChl;
     char   *name;
     int    mode;
{
     return ((int) pTyChl);
}

/*******************************************************************************
*=
*******
*
* TySCCRead - task level read routine for Philips SC26C92
*
* This routine fields all read calls to the Philips SC26C92
* It calls tyRead with a pointer to the appropriate element of TySCCchl.
*/

LOCAL int tySCCRead (pTyChl, buffer, maxbytes)
     TY_SCC_CHL *pTyChl;
     char   *buffer;
     int    maxbytes;

{
     return (tyRead( &pTyChl->tyDev, buffer, maxbytes));
}

/*******************************************************************************
*=
*******
*
* TySCCWrite - task level write routine for Intel 534 board
*
* This routine fields all write calls to the Intel 534.
* It calls tyWrite with a pointer to the appropriate element of TySCCDv.
*/

LOCAL int tySCCWrite (pTyChl, buffer, nbytes)
     TY_SCC_CHL     *pTyChl;
     char       *buffer;
     int        nbytes;

{
     return (tyWrite( &pTyChl->tyDev, buffer, nbytes));
}

/*******************************************************************************
*=
*******
*
* TySCCIoctl - special device control
*/

LOCAL int tySCCIoctl (pTyChl, request, arg)
     TY_SCC_CHL *pTyChl;
     int request;  /* request code */
     int arg;            /* some argument */

{
     int i, count;

     switch (request)
```

```
    {
    case FIOBAUDRATE:

       for ( i = 0; i < NELEMENTS( baudTable ); i++ )
       {
            if ( baudTable[i].rate == arg )
            {
               MyOutByte( pTyChl->csr, baudTable[i].preset );
               return( OK );
            }

       }

       break;

    default:
       return (tyIoctl ( &pTyChl->tyDev, request, arg));
    }

    return ERROR;
}

/*******************************************************************************
*=
*******
*
* TySCCrxInt - receive interrupt level processing
*
* This routine handles an interrupt from the 534 board.  The interrupt
* is decoded and the appropriate routine invoked.  The interrupt is
* terminated with a 'specific EOI'.
*/

LOCAL VOID tySCCrxInt (
    TY_SCC_CHL     *pTyChl
    )
{
    char interruptSR;
    char mask;

    if ( pTyChl->isA )
       mask = 0x2;
    else
       mask = 0x20;

    interruptSR = sysInByte( pTyChl->ptySCC->isr ) & mask;

    do
    {
       if ( interruptSR & mask )
       {
            if ( pTyChl->created )
               tyIRd( &pTyChl->tyDev, sysInByte( pTyChl->rxfifo ) );
            else
               sysInByte( pTyChl->rxfifo );
       }

       interruptSR = sysInByte( pTyChl->ptySCC->isr );

    } while( interruptSR & mask );
}

/*******************************************************************************
*=
```

```
*******
*
* TySCCtxInt - receive interrupt level processing
*
*/

LOCAL VOID tySCCtxInt (
     TY_SCC_CHL      *pTyChl
     )
{
     int         i;
     char        outchar;
     char        interruptSR;
     char        mask;


     for ( i = 0; i < N_SCC_CHANNELS; i++ )
     {
        if ( pTyChl->isA )
             mask = 0x1;
        else
             mask = 0x10;
        interruptSR = sysInByte( pTyChl->ptySCC->isr );
        while ( interruptSR & mask )
        {
             if ( pTyChl->created && ( tyITx( &pTyChl->tyDev, &outchar )
== OK ) )
             {
                MyOutByte( pTyChl->txfifo, outchar );
             }
             else
             {
                MyOutByte( pTyChl->cr, CMD_TX_DISABLE );
             }
             interruptSR = sysInByte( pTyChl->ptySCC->isr );
        }

        pTyChl++;
     }
}

LOCAL VOID tySCCMuxInt(
     TY_SCC_CHL      *pTyChl
     )
{
     int         i;
     char        outchar;
     char        interruptSR;
     char        mask;
     TY_SCC_CHL      *pTyChl5;

#if   0
     pTyChl5 = pTyChl + 4;
     /* take care of receive interrupt */
     for ( i = 0; i < 2; i++ )
     {
        if ( pTyChl5->isA )
             mask = 0x2;
        else
             mask = 0x20;

        interruptSR = sysInByte( pTyChl5->ptySCC->isr );
        while ( interruptSR & mask )
        {
```

```
            if ( pTyCh15->created )
               tyIRd( &pTyCh15->tyDev, sysInByte( pTyCh15->rxfifo ) );
            else
               sysInByte( pTyCh15->rxfifo );

            interruptSR = sysInByte( pTyCh15->ptySCC->isr );
         }

         pTyCh15++;
      }
#endif
      /* take care of transmit interrupt */
      for ( i = 0; i < N_SCC_CHANNELS; i++ )
      {
         if ( pTyCh1->isA )
            mask = 0x1;
         else
            mask = 0x10;
         interruptSR = sysInByte( pTyCh1->ptySCC->isr );
         while ( interruptSR & mask )
         {
            if ( pTyCh1->created && ( tyITx( &pTyCh1->tyDev, &outchar )
== OK ) )
            {
               sysOutByte( pTyCh1->txfifo, outchar );
            }
            else
            {
               sysOutByte( pTyCh1->cr, CMD_TX_DISABLE );
               break;
            }
            interruptSR = sysInByte( pTyCh1->ptySCC->isr );
         }

         pTyCh1++;
      }

}

/*******************************************************************************
*=
*******
*
* TySCCStartup - transmitter startup routine
*
* Call interrupt level character output routine for
*/

LOCAL VOID tySCCStartup( pTyCh1 )
    TY_SCC_CHL *pTyCh1;             /* ty device to start up */

{

      MyOutByte( pTyCh1->cr, CMD_TX_ENABLE );

#if   0

      /* any character to send ? */

      while ( ( tyITx ( &pTyCh1->tyDev, &outchar) == OK ) )
      {
         if ( sysInByte( pTyCh1->sr ) & SR_TXRDY )
            MyOutByte( pTyCh1->txfifo, outchar );  /* output the
character =
```

```
*/
        else
             break;
     }

#endif

}

/* The code below is added by Tony Wang */
static char caDevName[2][20];

char * getPortDeviceName(int iPortID, int iBaudRate)
{
      char caDev[2];
      static BOOL bIsDeviceCreated[2] = { FALSE, FALSE };

      /* Make sure tySCCDrv being called once.*/
      if (!bIsDeviceCreated[0] && !bIsDeviceCreated[1])
      {
/* nina */
            if (tySCCDrv() == ERROR)
                  printf("Device installation failed.\n");
      }

      if (iPortID > 1 || iPortID < 0)
            return NULL;

      if (bIsDeviceCreated[iPortID])
            return caDevName[iPortID];

      caDev[0] = '0' + iPortID;
      caDev[1] = 0;

      sprintf( caDevName[iPortID], "/tyCoSCC/%s", caDev );
/* nina */
      if ( tySCCDevCreate( caDevName[iPortID], iPortID, 1024, 1024,
iBaudRate ) != OK )
      {
            printf( "scctest:tySCCDevCreate:Device[%d]:FAIL\n\r",
iPortID );
            return NULL;
      }
      else
            bIsDeviceCreated[iPortID] = TRUE;

      return caDevName[iPortID];
}
```

Appendix B: Source Code for *echo* (`scctest.ccp`)

```
/* scctest.cpp - Test program for SCC driver */
/* channel 0 and channel 1 */

#include "vxWorks.h"
#include "sys/types.h"
#include "iolib.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "sysLib.h"
#include "scclib.h"

void pmdHrdInit(int waitState)
{
      short int adl = 0x0600 | (waitState % 16);
      sysOutWord( 0xf400, adl );
      sysOutWord( 0xf402, 0x0040 );
      sysOutWord( 0xf406, 0x3f );
      sysOutWord( 0xf404, 0xfc01 );
}

void sccDriver()
{
     tySCCDrv();        /* initialize the scc lib */
}

void test(int channel)  /*written for Binh N. for testing */
/* this test() routine continuously sends data to output buffer*/

{    int       i=0, j=0;
     int       fhdl[6] = { -1, -1, -1, -1, -1, -1 };
     char      devname[20];
     char      dev[2];
     char      buf[]="0123456789";

for ( i = 0; i < 4; i++ )
      {

            dev[0] = '0' + i;
            dev[1] = 0;
            sprintf( devname, "/tyCoSCC/%s", dev );
            if ( tySCCDevCreate( devname, i, 1024, 1024, 19200 ) != OK )
            {
                  printf( "scctest:tySCCDevCreate:Device[%d]:FAIL\n\r",
i );
            }
      }

for ( i = 0; i < 4; i++ )
      {
            dev[0] = '0' + i;
            dev[1] = 0;
            sprintf( devname, "/tyCoSCC/%s", dev );
            if ( ( fhdl[i] = open( devname, O_RDWR, 0 ) ) == ERROR )
                  printf( "scctest:open:[%s]:FAIL\n\r", devname );

      }

      while (1)
      {     ioctl(fhdl[channel], FIOFLUSH, 0);  // discard all bytes in
the input and output buffers
            j = write( fhdl[channel], buf, strlen(buf));
```

```
            printf("j=[%d], wrote to scc(%d):[0123456789]\n",j,
channel);

            taskDelay(10); // make sure all characters have been sent to
the buffer
            i=i++;
            if (i==100000)
                  break;
      }
      close( fhdl[0] );
      close( fhdl[1] );
      close( fhdl[2] );
      close( fhdl[3] );

      printf("It's written 100000 times.  END for now. BYE!");
}



void echo01()
{
     int        j;
     int        fhdl[6] = { -1, -1, -1, -1, -1, -1 };
     char       devname[20];
     char       dev[2];
     char       buffer[128];
     char       buf[]="0123456789";

      ULONG timeOutTicks=100;
      ULONG tickCount=0;
      ULONG countRight=0;

while (1)
      {

      ioctl(fhdl[0], FIOFLUSH, 0);  // discard all bytes in the input
and output buffers
      ioctl(fhdl[1], FIOFLUSH, 0);

      j = write( fhdl[0], buf, strlen(buf));
            printf("j=[%d], wrote to scc(0):[0123456789]\n",j);

      taskDelay(10); // make sure all characters have been sent to the
buffer

      j = read( fhdl[1], buffer, 10);
            buffer[j]=0; printf("read from scc(0):[%s]\n", buffer);

      j = write(fhdl[1], buffer, strlen(buffer));
            buffer[j]=0; printf("wrote to scc(1):[%s]\n", buffer);

      taskDelay(10);
      j = read( fhdl[0], buffer, 10);
            buffer[j]=0; printf("j=%d, read:[%s]\n", j, buffer);

      if ((buffer[0]==
'0')&&(buffer[1]=='1')&&(buffer[2]=='2')&&(buffer[3]=='3')&&(buffer[4]==
'4') && (buffer[5]=='5') && (buffer[6]=='6') && (buffer[7]=='7')&&
(buffer[8]== '8') && (buffer[9]=='9') && (buffer[10]=='\0'))
      {
        countRight++;
      }
      tickCount++;
      if (tickCount >= timeOutTicks)
```

```
            break;
      }
        close( fhdl[0] );
        close( fhdl[1] );
printf("*********************************\n");
printf("%d messages out of %d messages have been sent and read correctly
between channel 0 and 1\n", countRight, timeOutTicks);
printf("*********************************\n");

}




void echo()
// two echo tests, between channel 0 and 1, and between channel 2 and 3
go
// on at the same time.

{
      char  OutBfr01[100], InBfr01[100], EchoBfr01[100];
      char  OutBfr23[100], InBfr23[100], EchoBfr23[100];
      int   i, j, m;
      int   readError01=0, readError23=0;
      int   fhdl[6] = { -1, -1, -1, -1, -1, -1 };
      char  devname[20];
      char  dev[2];

/*************************************************************************
****/



for ( i = 0; i < 3; i++ )
      {

            dev[0] = '0' + i;
            dev[1] = 0;
            sprintf( devname, "/tyCoSCC/%s", dev );
            if ( tySCCDevCreate( devname, i, 1024, 1024, 19200 ) != OK )
            {
                  printf( "scctest:tySCCDevCreate:Device[%d]:FAIL\n\r",
i );
            }
      }

for ( i = 0; i < 3; i++ )
      {
            dev[0] = '0' + i;
            dev[1] = 0;
            sprintf( devname, "/tyCoSCC/%s", dev );
            if ( ( fhdl[i] = open( devname, O_RDWR, 0 ) ) == ERROR )
                  printf( "scctest:open:[%s]:FAIL\n\r", devname );

      }

/*************************************************************/

for ( i=0; i <999; i++)

      {
            for (j=0; j<99; j++)
            {
```

```
                InBfr01[j]    = InBfr23[j]   =(char) ((i+25+j) % 50
+35);
                EchoBfr01[j]  = EchoBfr23[j] =(char) ((i+24+j) % 50
+35);
                OutBfr01[j]   = OutBfr23[j]  =(char) ((i+23+j) % 50
+35);
            }

            InBfr01[99] = InBfr23[99]=0;

            ioctl(fhdl[0], FIOFLUSH, 0);  // discard all bytes in the
input and output buffers
            ioctl(fhdl[1], FIOFLUSH, 0);
            ioctl(fhdl[2], FIOFLUSH, 0);
            ioctl(fhdl[3], FIOFLUSH, 0);

            j = write( fhdl[0], InBfr01, strlen(InBfr01));
                  InBfr01[j]=0;
            j = write( fhdl[2], InBfr23, strlen(InBfr23));
                  InBfr23[j]=0;

            taskDelay(10); // make sure all characters have been sent to
the buffer
            // it seems like 4 is the minimum for 100 characters =)

            j = read( fhdl[1], EchoBfr01, 100);
                  EchoBfr01[j]=0;
            j = read( fhdl[3], EchoBfr23, 100);
                  EchoBfr23[j]=0;

            if ((m = memcmp(EchoBfr01, InBfr01, 100)) != 0)
                  readError01++;
            if ((m = memcmp(EchoBfr23, InBfr23, 100)) != 0)
                  readError23++;

            j = write(fhdl[1], EchoBfr01, strlen(EchoBfr01));
                  EchoBfr01[j]=0;
            j = write(fhdl[3], EchoBfr23, strlen(EchoBfr23));
                  EchoBfr23[j]=0;

            taskDelay(4);

            j = read( fhdl[0], OutBfr01, 100);
                  OutBfr01[j]=0;
            j = read( fhdl[2], OutBfr23, 100);
                  OutBfr23[j]=0;

            if ((m = memcmp(OutBfr01, EchoBfr01, 100)) != 0)
                  readError01++;
            if ((m = memcmp(OutBfr23, EchoBfr23, 100)) != 0)
                  readError23++;
      }

            printf("There are %d reading errors between Channel 0 and
1\n", readError01);
            printf("There are %d reading errors between Channel 2 and
3\n", readError23);

            close( fhdl[0] );
            close( fhdl[1] );
            close( fhdl[2] );
            close( fhdl[3] );
}
```

```
/* The following is the routine echo() divided to different tasks:
i.e.  writeCom(0, "1234");
      readCom(1,4)
      writeCom(1,"1234");
      readCom(0,4);

*/
int writeCom(int channel, char *buffer)
{
      int   i, j;
      int   fhdl[6] = { -1, -1, -1, -1, -1, -1 };
      char  devname[20];
      char  dev[2];

/*********************************************************************
****/

            dev[0] = '0' + channel;
            dev[1] = 0;
            sprintf( devname, "/tyCoSCC/%s", dev );
            if ( tySCCDevCreate( devname, channel, 1024, 1024, 19200 )
!= OK )
            {
                  printf( "scctest:tySCCDevCreate:Device[%d]:FAIL\n\r",
channel );
            }

            dev[0] = '0' + channel;
            dev[1] = 0;
            sprintf( devname, "/tyCoSCC/%s", dev );
            if ( ( fhdl[channel] = open( devname, O_RDWR, 0 ) ) == ERROR
)
                  printf( "scctest:open:[%s]:FAIL\n\r", devname );


      ioctl(fhdl[channel], FIOFLUSH, 0);  // discard all bytes in the
input and output buffers


      j = write(fhdl[channel], buffer, strlen(buffer));
                  buffer[j]=0; printf("j=[%d], wrote:[%s]\n",j, buffer);


      return(j);
}



int readCom(int channel, int len)
{
      int   i, j;
      int   fhdl[6] = { -1, -1, -1, -1, -1, -1 };
      char  devname[20];
      char  dev[2];
      char    buf[128];
/*********************************************************************
****/

            dev[0] = '0' + channel;
            dev[1] = 0;
            sprintf( devname, "/tyCoSCC/%s", dev );
```

```
            if ( tySCCDevCreate( devname, channel, 1024, 1024, 19200 )
!= OK )
            {
                  printf( "scctest:tySCCDevCreate:Device[%d]:FAIL\n\r",
channel );
            }

            dev[0] = '0' + channel;
            dev[1] = 0;
            sprintf( devname, "/tyCoSCC/%s", dev );
            if ( ( fhdl[channel] = open( devname, O_RDWR, 0 ) ) == ERROR
)
                  printf( "scctest:open:[%s]:FAIL\n\r", devname );


/*****************************************************************/

      while(1)
      {
            int nBytesUnread;
            ioctl (fhdl[channel], FIONREAD, (int) &nBytesUnread);
            printf("There are %d bytes unread\n", nBytesUnread);

            if (nBytesUnread > 0)
                  j+=read(fhdl[channel], buf+j, len);
            else
                  break;
      }
        buf[j]=0;
      printf("read:[%s]\n", buf);
      close(fhdl[channel]);
      return j;
}
```

Appendix C: Source Code for PMD Chipset's Interface (pmdio.c)

```
/* FileName::PMDIO.C */

#include <taskLib.h>
#include "vxWorks.h"
#include "iv.h"
#include "ioLib.h"
#include "iosLib.h"
#include "intLib.h"
#include "errnoLib.h"
#include "sysLib.h"
#include "semLib.h"
#include "stdio.h"
#include "motion.h"
#include "taskLib.h"

/* defines */
#define inp_x(ioAddr)            sysInByte(ioAddr)
#define outp_x(ioAddr, data)     sysOutByte(ioAddr,data)
#define DIAG                     1

/* global variables */
int check_sum_err = 0;

/*
-------------------------------------
  LOW-LEVEL HOST I/O ROUTINES
-------------------------------------
*/


WORD  PMDReady(void)
{
   int   in_val;
      in_val = inp_x(STATUS_PORT_ADDRESS);
      return(in_val & READY_BUSY_BIT_MASK);
}

/*
-------------------------------------------------------------------------
---
Routine:       wait_until_not_busy

Function:
               This routine polls the ready/busy host bit and returns
               when it is ready. If the chipset is not receiving
               power, or if there is some other hardware I/O problem,
               this routine will report error after time out.

Arguments:
               Time out.

Return Arguments:
```

```
                OK if the the ready bit has been set before time out or
                                ERROR if not.
-------------------------------------------------------------------------
---
*/

STATUS  wait_until_not_busy(int iTimeOut)
{
      int in_val, iTick;

      /* poll ready port, if not ready, loop; chip should ready within
12us */
      for (iTick = 0; iTick < iTimeOut; iTick++)
      {
            in_val = (WORD)(inp_x(STATUS_PORT_ADDRESS) &
inp_x(STATUS_PORT_ADDRESS) & inp_x(STATUS_PORT_ADDRESS));
            if ((in_val & READY_BUSY_BIT_MASK) != 0)
            {
                  return OK;
            }
            taskDelay(1);
      }

#ifdef DIAG
      printf("<Wait until not busy failed.>\n");
#endif

      return ERROR;
}


/*
-------------------------------------------------------------------------
---
Routine:       write_cmd

Function:
               This is the low-level routine that writes a single byte
command
               to the chipset.

Arguments:
   the_cmd     is the command byte to write to the chipset.

Return Arguments:
               none
-------------------------------------------------------------------------
---
*/

void  write_cmd(BYTE  the_cmd)
{
#ifdef DIAG
      printf("The writing command: %x.\n", the_cmd);
#endif

   outp_x(COMMAND_PORT_ADDRESS, the_cmd);
```

```
}

/*
------------------------------------------------------------------------
---
Routine:        write_byte

Function:
                This is the low-level routine that writes a single byte
of data
                to the chipset.

Arguments:
   the_byte     is the data byte to write to the chipset.

Return Arguments:
                none
------------------------------------------------------------------------
---
*/

void  write_byte(BYTE  the_byte)
{

   outp_x(DATA_PORT_ADDRESS, the_byte);

}

/*
------------------------------------------------------------------------
---
Routine:        read_byte

Function:
                This is the low-level routine that reads a single byte
of
                data from the chipset.

Arguments:
                none

Return Arguments:
                the byte read from the chipset
------------------------------------------------------------------------
---
*/

BYTE  read_byte(void)
{
   BYTE  the_byte;

   the_byte = inp_x(DATA_PORT_ADDRESS);

   return(the_byte);
}
```

```
/*
-------------------------------------------------------------------------
---
Routine:        write_a_word

Function:
                This routine writes a 2-byte data word to the chipset,
and
                maintains the checksum word.

Arguments:
   the_word        is the data word to write to the chipset.
   the_checksum    is a pointer to the running checksum.

Return Arguments:
                OK if writing is sucessful or ERROR if not.
-------------------------------------------------------------------------
---
*/

STATUS  write_a_word(WORD     the_word,
                     WORD     *the_checksum)
{
#ifdef DIAG
      int in_val;
      int i, time = 0;
#endif

      /* poll ready port, if not ready, loop; chip should ready within
12us */
   if (wait_until_not_busy(12) == ERROR)
      {
#ifdef DIAG
            printf("<Write a word failed.>\n");
#endif
            return ERROR;
      }

   write_byte((BYTE) ((the_word & 0xff00) >> 8));
   write_byte((BYTE) (the_word & 0xff));

#ifdef DIAG
/*    while (TRUE) */
      {
            in_val = (WORD)inp_x(STATUS_PORT_ADDRESS);
            if ((in_val & READY_BUSY_BIT_MASK) != 0)
            {
                  time++;
                  for (i = 0; i < time; i++);
                  printf("After writnig a word, time issue
occured.\n");
            }
            else
                  printf("After write a word, chip is not ready after
%d.\n", time);
      }
#endif
```

```
   *the_checksum += the_word;

      return OK;
}

/*
------------------------------------------------------------------------
---
Routine:       read_a_word

Function:
               This routine reads a 2-byte data word from the chipset,
and
               maintains the checksum word.

Arguments:
   the_word       is a pointer to the data word to read from the
chipset.
   the_checksum   is a pointer to the running checksum.

Return Arguments:
               OK if reading is sucessful or ERROR if not.
------------------------------------------------------------------------
---
*/

STATUS     read_a_word(WORD     *the_word,
                       WORD     *the_checksum)
{
#ifdef DIAG
      int in_val;
      int i, time = 0;
#endif

   BYTE  lo_byte, hi_byte;

      /* poll ready port, if not ready, loop; chip should ready within
12us */
   if (wait_until_not_busy(12) == ERROR)
      {
#ifdef DIAG
            printf("<Read a word failed.>\n");
#endif
            return ERROR;
      }

   hi_byte = read_byte();
/*   sysInByte(0x1410); */
   lo_byte = read_byte();
/*
#ifdef DIAG
      while (TRUE)
      {
            in_val = (WORD)inp_x(STATUS_PORT_ADDRESS);
            if ((in_val & READY_BUSY_BIT_MASK) != 0)
            {
```

```
                time++;
                for (i = 0; i < time; i++);
                printf("After reading a word, time issue
occured.\n");
            }
            else
                printf("After read a word, chip is not ready after
%d.\n", time);
        }
#endif
*/
#ifdef DIAG
            printf("<Hi byte: %x, Lo byte: %x.>\n", hi_byte, lo_byte);
#endif
   *the_word = ((WORD)(hi_byte & 0xff) << 8) | (lo_byte & 0xff);
   *the_checksum += *the_word;

      return OK;
}


/*
-------------------------------------------------------------------------
---
Routine:        read_n_check_checksum

Function:
                This routine is called at the end of a command sequence.
It
                reads a data word from the chipset (the checksum), and
compares
                it with the expected checksum provided as a calling
argument.
                It returns an error condition if the checksums do not
compare
                correctly.

Arguments:
   checksum          is the expected checksum, based on the previous
I/O operations
   actual_checksum is a pointer to the actual checksum, read from the
chipset

Return Arguments:
                CHECKSUM_BAD if the command checksum is bad,
CHECKSUM_GOOD if
                the command checksum is OK
-------------------------------------------------------------------------
---
*/

WORD read_n_check_checksum(WORD  checksum,
                           WORD  *actual_checksum)
{
   WORD  asic_checksum, scratch;

   read_a_word(&asic_checksum, &scratch);
```

```
      if (asic_checksum == checksum)
    {
#ifdef DIAG
            printf("<Actual checksum: %x, checksum: %x.>\n",
asic_checksum, checksum);
#endif
            return(CHECKSUM_GOOD);
      }
   else
   {
         sysOutByte(0x1402, 0xff);
#ifdef DIAG
            printf("<Actual checksum: %x, checksum: %x.>\n",
asic_checksum, checksum);
#endif
      *actual_checksum = asic_checksum;
#ifdef DIAG
      printf("<Bad checksum.>\n");
/*       exit(0); */
#endif
            return(CHECKSUM_BAD);
   }
}

/*
------------------------------------------------------------------------
---
Routine:       write_a_cmd

Function:
               This routine writes a single command byte to the
chipset, and
               maintains the checksum word.

Arguments:
   the_cmd     is the command to write to the chipset.
   the_checksum   is a pointer to the running checksum. After this
routine
                  executes the running checksum is set equal to the
command
                  byte.

Return Arguments:
               OK if writing a command is sucessful or ERROR if not.
------------------------------------------------------------------------
---
*/

STATUS  write_a_cmd(BYTE     the_cmd,
                    WORD     *the_checksum)
{
      /* poll ready port, if not ready, loop; chip should ready within
12us */
   if (wait_until_not_busy(12) == ERROR)
      {
#ifdef DIAG
```

```
            printf("<Write a command failed.>\n");
#endif
            return ERROR;
      }

   write_cmd(the_cmd);

   *the_checksum = the_cmd;

      return OK;
}

/*
-------------------------------------------------------------------
---
Routine:        send_chipset_cmd

Function:
                This is the main routine used to send a command to the
chipset.
                It accepts read or write commands, with any # of data
words.
                It uses the command checksum for both read and write
operations.
                In case of a checksum error, it will not re-try the
command
                automatically, although it does report the checksum
status of
                the operation, so that the calling routine can (if
desired),
                perform a special recovery sequence upon I/O checksum
error.

Arguments:
   command  is the hex command code
   cmd_type     is the type of the command, the values are CHIPSET_SET
                and CHIPSET_REQUEST
   length       is the length of the read or write data stream in words
   send_data    is a pointer to an array of words where the data to send
                is stored
   rcv_data     is a pointer to an array of words where the data to
receive
                is stored

Return Arguments:
                                OK if sending chipset command sucessful
or ERROR if not.
-------------------------------------------------------------------
---
*/

STATUS send_chipset_cmd(int             command,
                        int             cmd_type,
                        int             length,
                        WORD            *send_data,
                        WORD            *rcv_data)
{
```

```
   WORD            checksum, got;
   int             i;


   /* write the command to the chipset */
   if (write_a_cmd(command, &checksum) == ERROR)
            return ERROR;

   for (i = 0; i < length; i++)
   {
      if (cmd_type == CHIPSET_SET)
            {
         if (write_a_word(send_data[i], &checksum) == ERROR)
                        return ERROR;
      }
            else
            {
                  if (read_a_word(&rcv_data[i], &checksum) == ERROR)
                        return ERROR;
            }
   }


      /* Temperary delay some time. */
      for (i = 0; i < 1200; i++);

   /* get & check the checksum */

   if ((read_n_check_checksum(checksum, &got) == CHECKSUM_BAD) &&
(command != RESET))
   {

#ifdef DIAG
         printf("<I/O CHECKSUM ERROR OCCURRED>\n\r");
#endif
      check_sum_err++;
      return ERROR;
   }

   return OK;
}



/*
-----------------------------------------------
AXIS CONTROL COMMANDS
-----------------------------------------------
*/

LONG  set_1(void)
{
   WORD  axis_status, scratch;

      if (send_chipset_cmd(SET_1, CHIPSET_REQUEST, 1, &scratch,
&axis_status) == ERROR)
```

```
      {
#ifdef DIAG
            printf("<Setting axis to 1 failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("setting axis to 1, status : %x\n", axis_status);
#endif

   return(((LONG) axis_status) & 0xffff);
}

LONG  set_2(void)
{
   WORD  axis_status, scratch;

      if (send_chipset_cmd(SET_2, CHIPSET_REQUEST, 1, &scratch,
&axis_status) == ERROR)
      {
#ifdef DIAG
            printf("<Setting axis to 2 failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("setting axis to 2, status : %x\n", axis_status);
#endif

   return((LONG) axis_status);
}

LONG  set_3(void)
{
   WORD  axis_status, scratch;

      if (send_chipset_cmd(SET_3, CHIPSET_REQUEST, 1, &scratch,
&axis_status) == ERROR)
      {
#ifdef DIAG
            printf("<Setting to axis 3 failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("setting axis to 3, status : %x\n", axis_status);
#endif

   return((LONG) axis_status);
}

LONG  set_4(void)
{
   WORD  axis_status, scratch;
```

```
      if (send_chipset_cmd(SET_4, CHIPSET_REQUEST, 1, &scratch,
&axis_status) == ERROR)
      {
#ifdef DIAG
            printf("<Setting to axis 4 failed.>\n");
#endif
      return ERROR;
      }

#ifdef DIAG
      printf("setting axis to 4, status : %x\n", axis_status);
#endif

   return((LONG) axis_status);
}

LONG  set_i(void)
{
   WORD  axis_status, scratch;

      if (send_chipset_cmd(SET_I, CHIPSET_REQUEST, 1, &scratch,
&axis_status) == ERROR)
      {
#ifdef DIAG
            printf("<Setting axis to i failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("setting to interrupting axis, status : %x\n",
axis_status);
#endif

   return((LONG) axis_status);
}

/*
-----------------------------------------------
PROFILE GENERATION
-----------------------------------------------
*/

STATUS set_prfl_s_crv(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting profile to s-curve\n");
#endif

   if (send_chipset_cmd(SET_PRFL_S_CRV, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting profile to s-curves failed.>\n");
```

```
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_prfl_trap(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting profile to trapezoidal\n");
#endif

      if (send_chipset_cmd(SET_PRFL_TRAP, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting profile to trapezoidal failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_prfl_vel(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting profile to velocity contouring\n");
#endif

   if (send_chipset_cmd(SET_PRFL_VEL, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting profile to velocity contouring.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_prfl_gear(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting profile to electronic gear\n");
#endif

      if (send_chipset_cmd(SET_PRFL_GEAR, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
```

```
      {
#ifdef DIAG
            printf("<Setting profile to electronic gear failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_pos(LONG   the_value)
{
   WORD  data[2], scratch;

#ifdef DIAG
      printf("setting destination position to: %lx\n", the_value);
#endif

   data[HIGH] = (WORD) ( (the_value >> 16)   & 0xffffL);
   data[LOW] = (WORD) (the_value & 0xffffL);

   if (send_chipset_cmd(SET_POS, CHIPSET_SET, 2, data, &scratch) ==
ERROR)
      {
#ifdef DIAG
            printf("<Setting destination position to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_vel(LONG   the_value)
{
   WORD  data[2], scratch;

#ifdef DIAG
      printf("setting max. velocity to: %lx\n", the_value);
#endif

   data[HIGH] = (WORD) ((the_value >> 16) & 0xffffL);
   data[LOW] = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_VEL, CHIPSET_SET, 2, data, &scratch) ==
ERROR)
      {
#ifdef DIAG
            printf("<Setting max velocity to %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}
```

```
STATUS set_acc(LONG   the_value)
{
   WORD  data[2], scratch;

#ifdef DIAG
      printf("setting acceleration to: %lx\n", the_value);
#endif

   data[HIGH] = (WORD) ((the_value >> 16) & 0xffffL);
   data[LOW] = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_ACC, CHIPSET_SET, 2, data, &scratch) ==
ERROR)
      {
#ifdef DIAG
            printf("<Setting acceleration to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_max_acc(LONG  the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("setting max. acceleration to: %lx\n", the_value);
#endif

   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_MAX_ACC, CHIPSET_SET, 1, &data,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Setting max. acceleration to: %lx failed.>",
the_value);
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_jerk(LONG  the_value)
{
   WORD  data[2], scratch;

#ifdef DIAG
      printf("setting jerk to: %lx\n", the_value);
#endif

   data[HIGH] = (WORD) ((the_value >> 16) & 0xffffL);
```

```
   data[LOW] =  (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_JERK, CHIPSET_SET, 2, data, &scratch) ==
ERROR)
      {
#ifdef DIAG
            printf("<Setting jerk to: %lx failed.>", the_value);
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_ratio(LONG the_value)
{
   WORD  data[2], scratch;

#ifdef DIAG
      printf("setting target ratio to: %lx\n", the_value);
#endif

   data[HIGH] = (WORD ) ((the_value >> 16)   & 0xffffL);
   data[LOW] = (WORD) (the_value & 0xffffL);

   if (send_chipset_cmd(SET_RATIO, CHIPSET_SET, 2, data, &scratch) ==
ERROR)
      {
#ifdef DIAG
            printf("<Setting target ration to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}

STATUS clr_prfl(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("clear profile...\n");
#endif

      if (send_chipset_cmd(CLR_PRFL, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Clearing profile failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}
```

```
STATUS synch_prfl(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("synch profile...\n");
#endif

      if (send_chipset_cmd(SYNCH_PRFL, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Synch profile failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS zero_pos(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("zeroing position...\n");
#endif

      if (send_chipset_cmd(ZERO_POS, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Zeroing position failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

LONG  get_pos(void)
{
   WORD  data[2], scratch;
   LONG  pos;

   if (send_chipset_cmd(GET_POS, CHIPSET_REQUEST, 2, &scratch, data) ==
ERROR)
      {
#ifdef DIAG
            printf("<Getting position failed.>\n");
#endif
            return ERROR;
      }

   pos = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] & 0xffffL);
```

```
#ifdef DIAG
      printf("position: %lx\n", pos);
#endif

   return(pos);
}

LONG  get_vel(void)
{
   WORD  data[2], scratch;
   LONG  vel;

   if (send_chipset_cmd(GET_VEL, CHIPSET_REQUEST, 2, &scratch, data) ==
ERROR)
      {
#ifdef DIAG
            printf("<Getting valocity failed.>\n");
#endif
            return ERROR;
      }

   vel = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] & 0xffffL);

#ifdef DIAG

      printf("velocity: %lx\n", vel);
#endif

   return(vel);
}

LONG  get_acc(void)
{
   WORD  data[2], scratch;
   LONG  acc;

   if (send_chipset_cmd(GET_ACC, CHIPSET_REQUEST, 2, &scratch, data) ==
ERROR)
      {
#ifdef DIAG
            printf("<Getting acdeleration failed.>\n");
#endif
            return ERROR;
      }

   acc = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] & 0xffffL);

#ifdef DIAG
      printf("acceleration: %lx\n", acc);
#endif

   return(acc);
}

LONG  get_max_acc(void)
{
   WORD  max_acc, scratch;
```

```
   if (send_chipset_cmd(GET_MAX_ACC, CHIPSET_REQUEST, 1, &scratch,
&max_acc) == ERROR)
      {
#ifdef DIAG
            printf("<Getting max. acceleration failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("max. acceleration: %x\n", max_acc);
#endif

   return((LONG) max_acc);
}

LONG  get_jerk(void)
{
   WORD  data[2], scratch;
   LONG  jerk;

   if (send_chipset_cmd(GET_JERK, CHIPSET_REQUEST, 2, &scratch, data)
== ERROR)
      {
#ifdef DIAG
            printf("<Getting jerk failed.>\n");
#endif
            return ERROR;
      }

   jerk = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] & 0xffffL);

#ifdef DIAG
      printf("jerk: %lx\n", jerk);
#endif

   return(jerk);
}

LONG  get_ratio(void)
{
   WORD  data[2], scratch;
   LONG  ratio;

   if (send_chipset_cmd(GET_RATIO, CHIPSET_REQUEST, 2, &scratch, data)
== ERROR)
      {
#ifdef DIAG
            printf("<Getting ratio failed.>\n");
#endif
            return ERROR;
      }

   ratio = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] & 0xffffL);

#ifdef DIAG
```

```
      printf("ratio: %lx\n", ratio);
#endif

   return(ratio);
}

LONG  get_trgt_pos(void)
{
   WORD  data[2], scratch;
   LONG  target_pos;

   if (send_chipset_cmd(GET_TRGT_POS, CHIPSET_REQUEST, 2, &scratch,
data) == ERROR)
      {
#ifdef DIAG
            printf("<Getting target position failed.>\n");
#endif
            return ERROR;
      }

   target_pos = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] & 0xffffL);

#ifdef DIAG
      printf("target position: %ld\n", target_pos);
#endif

   return(target_pos);
}

LONG  get_trgt_vel(void)
{
   WORD  data[2], scratch;
   LONG  target_vel;

   if (send_chipset_cmd(GET_TRGT_VEL, CHIPSET_REQUEST, 2, &scratch,
data) == ERROR)
      {
#ifdef DIAG
            printf("<Getting target velocity failed.>\n");
#endif
            return ERROR;
      }

   target_vel = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] & 0xffffL);

#ifdef DIAG
      printf("target velocity: %ld\n", target_vel);
#endif

   return(target_vel);
}

/*
-----------------------------------------------
DIGITAL FILTER
-----------------------------------------------
*/
```

```
STATUS set_fltr_pid(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting filter to: PID\n");
#endif

      if (send_chipset_cmd(SET_FLTR_PID, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting filter to: PID failed.>");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_fltr_pivff(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting filter to: PIVFF\n");
#endif

      if (send_chipset_cmd(SET_FLTR_PIVFF, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting filter to: PIVFF failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_kp(LONG the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("setting kp to: %lx\n", the_value);
#endif

   data = (WORD)(the_value & 0xffffL);

      if (send_chipset_cmd(SET_KP, CHIPSET_SET, 1, &data, &scratch) ==
ERROR)
      {
#ifdef DIAG
            printf("<Setting kp to: %lx failed.>\n", the_value);
#endif
```

```
            return ERROR;
      }

      return OK;
}

STATUS set_kd(LONG the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("setting kd to: %lx\n", the_value);
#endif

   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_KD, CHIPSET_SET, 1, &data, &scratch) ==
ERROR)
      {
#ifdef DIAG
            printf("<Setting kd to: %lx failed.>\n", the_value);
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_ki(LONG the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("setting ki to: %lx\n", the_value);
#endif

   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_KI, CHIPSET_SET, 1, &data, &scratch) ==
ERROR)
      {
#ifdef DIAG
            printf("<Setting ki to: %lx failed.>\n", the_value);
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_kv(LONG the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("setting kv to: %lx\n", the_value);
#endif
```

```
   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_KV, CHIPSET_SET, 1, &data, &scratch) ==
ERROR)
      {
#ifdef DIAG
            printf("<Setting kv to: %lx failed.>\n", the_value);
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_kvff(LONG  the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("setting kvff to: %lx\n", the_value);
#endif
   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_KVFF, CHIPSET_SET, 1, &data, &scratch)
== ERROR)
      {
#ifdef DIAG
            printf("<Setting kvff to: %lx failed.>\n", the_value);
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_i_lm(LONG  the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("setting i_limit to: %lx\n", the_value);
#endif

   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_I_LM, CHIPSET_SET, 1, &data, &scratch)
== ERROR)
      {
#ifdef DIAG
            printf("<Setting i_limit to: %lx failed.>\n", the_value);
#endif
            return ERROR;
      }

      return OK;
}
```

```
STATUS set_pos_err(LONG  the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("setting position error to: %lx\n", the_value);
#endif

   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_POS_ERR, CHIPSET_SET, 1, &data,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Setting position error to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}

LONG  get_kp(void)
{
   WORD  kp, scratch;

   if (send_chipset_cmd(GET_KP, CHIPSET_REQUEST, 1, &scratch, &kp) ==
ERROR)
      {
#ifdef DIAG
            printf("<Getting kp failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("Kp: %04x\n", kp);
#endif

   return((LONG) kp);
}

LONG  get_kd(void)
{
   WORD  kd, scratch;

   if (send_chipset_cmd(GET_KD, CHIPSET_REQUEST, 1, &scratch, &kd) ==
ERROR)
      {
#ifdef DIAG
            printf("<Getting kd failed.>\n");
#endif
            return ERROR;
      }
```

```
#ifdef DIAG
      printf("Kd: %04x\n", kd);
#endif

   return((LONG) kd);
}

LONG  get_ki(void)
{
   WORD  ki, scratch;

   if (send_chipset_cmd(GET_KI, CHIPSET_REQUEST, 1, &scratch, &ki) ==
ERROR)
      {
#ifdef DIAG
            printf("<Getting ki failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("Ki: %04x\n", ki);
#endif

   return((LONG) ki);
}

LONG  get_kv(void)
{
   WORD  kv, scratch;

   if (send_chipset_cmd(GET_KV, CHIPSET_REQUEST, 1, &scratch, &kv) ==
ERROR)
      {
#ifdef DIAG
            printf("<Getting kv failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("Kv: %x\n", kv);
#endif

   return((LONG) kv);
}

LONG  get_kvff(void)
{
   WORD  kvff, scratch;

   if (send_chipset_cmd(GET_KVFF, CHIPSET_REQUEST, 1, &scratch, &kvff)
== ERROR)
      {
#ifdef DIAG
            printf("<Getting kvff failed.>\n");
#endif
```

```
            return ERROR;
      }

#ifdef DIAG
      printf("Kvff: %04x\n", kvff);
#endif

   return((LONG) kvff);
}

LONG  get_i_lm(void)
{
   WORD  i_lim, scratch;

   if (send_chipset_cmd(GET_I_LM, CHIPSET_REQUEST, 1, &scratch, &i_lim)
== ERROR)
      {
#ifdef DIAG
            printf("<Getting i_limit failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("i_lim: %04x\n", i_lim);
#endif

   return((LONG) i_lim);
}

LONG  get_pos_err(void)
{
   WORD  pos_err, scratch;

   if (send_chipset_cmd(GET_POS_ERR, CHIPSET_REQUEST, 1, &scratch,
&pos_err) == ERROR)
      {
#ifdef DIAG
            printf("<Getting position error failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("max. position err: %04x\n", pos_err);
#endif

   return((LONG) pos_err);
}

LONG  get_intgr(void)
{
   WORD  integration_value, scratch;

   if (send_chipset_cmd(GET_INTGR, CHIPSET_REQUEST, 1, &scratch,
&integration_value) == ERROR)
      {
```

```
#ifdef DIAG
            printf("<Getting integration failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("integration value: %04x\n", integration_value);
#endif

   return((LONG) integration_value);
}

LONG  get_actl_pos_err(void)
{
   WORD  actl_pos_err, scratch;

   if (send_chipset_cmd(GET_ACTL_POS_ERR, CHIPSET_REQUEST, 1, &scratch,
&actl_pos_err) == ERROR)
      {
#ifdef DIAG
            printf("<Getting actural position error failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("Actual filter position error: %04x\n", actl_pos_err);
#endif

   return((LONG) actl_pos_err);
}

STATUS set_auto_stop_off(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("turn auto_stop off...\n");
#endif

      if (send_chipset_cmd(SET_AUTO_STOP_OFF, CHIPSET_SET, 0,
&scratch1, &scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Turning auto_stop off failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_auto_stop_on(void)
{
   WORD  scratch1, scratch2;
```

```
#ifdef DIAG
      printf("turn auto_stop on...\n");
#endif

      if (send_chipset_cmd(SET_AUTO_STOP_ON, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Turning auto_stop on failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

/*
-----------------------------------------------
PARAMETER UPDATE
-----------------------------------------------
*/

STATUS set_time_brk(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting time breakpoint\n");
#endif

      if (send_chipset_cmd(SET_TIME_BRKPNT, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting timr breakpoint failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_pos_brk(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting positive target position breakpoint\n");
#endif

      if (send_chipset_cmd(SET_POS_BRKPNT, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting positive target position breakpoint
failed.>\n");
#endif
```

```
            return ERROR;
      }

      return OK;
}

STATUS set_neg_brk(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting negative target position breakpoint\n");
#endif

      if (send_chipset_cmd(SET_NEG_BRKPNT, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting negative target positon breakpoint
failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_actl_pos_brk(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting positive actual position breakpoint\n");
#endif

      if (send_chipset_cmd(SET_ACTL_POS_BRK, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting positive actural position breakpoint
failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_actl_neg_brk(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting negative actual position breakpoint\n");
#endif
```

```
      if (send_chipset_cmd(SET_ACTL_NEG_BRK, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting negative actual position breakpoint
failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_brk_off(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting breakpoint off\n");
#endif

      if (send_chipset_cmd(SET_BRK_OFF, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting breakpoint off failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_brk_pnt(LONG  the_value)
{
   WORD  data[2], scratch;

#ifdef DIAG
      printf("setting break point value to: %lx\n", the_value);
#endif

   data[HIGH] = (WORD) ((the_value >> 16) & 0xffffL);
   data[LOW] = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_BRK_PNT, CHIPSET_SET, 2, data, &scratch)
== ERROR)
      {
#ifdef DIAG
            printf("<Setting break point value to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}
```

```
STATUS update(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("update target info\n");
#endif

      if (send_chipset_cmd(_UPDATE_, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Updating target info failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS multi_update(LONG the_value)
{
   WORD  mask, scratch;

#ifdef DIAG
      printf("starting multiple axis...; using mask:  %lx\n",
the_value);
#endif

   mask = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(MULTI_UPDATE, CHIPSET_SET, 1, &mask,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Multi-updating failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_auto_update_on(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting auto update on...\n");
#endif

      if (send_chipset_cmd(SET_AUTO_UPDATE_ON, CHIPSET_SET, 0,
&scratch1, &scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting auto update failed.>\n");
#endif
```

```
            return ERROR;
      }

      return OK;

}

STATUS set_auto_update_off(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting auto update off...\n");
#endif

      if (send_chipset_cmd(SET_AUTO_UPDATE_OFF, CHIPSET_SET, 0,
&scratch1, &scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting auto update off failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

LONG  get_brk_pnt(void)
{
   WORD  data[2], scratch;
   LONG  breakpoint_value;

   if (send_chipset_cmd(GET_BRK_PNT, CHIPSET_REQUEST, 2, &scratch,
data) == ERROR)
      {
#ifdef DIAG
            printf("<Getting breakpoint failed.\n>");
#endif
            return ERROR;
      }

   breakpoint_value = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] &
0xffffL);

#ifdef DIAG
      printf("breakpoint: %lx\n", breakpoint_value);
#endif

   return(breakpoint_value);
}

/*
-----------------------------------------------
INTERRUPT PROCESSING
-----------------------------------------------
*/
```

```
STATUS set_intrpt_mask(LONG the_value)
{
   WORD  mask, scratch;

#ifdef DIAG
      printf("setting interrupt mask to: %lx\n", the_value);
#endif

   mask = (WORD) (the_value & 0xffffL);
      if (send_chipset_cmd(SET_INTRPT_MASK, CHIPSET_SET, 1, &mask,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Setting interrupr mask to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}

LONG  get_intrpt(void)
{
   WORD  axis_status, scratch;

   if (send_chipset_cmd(GET_INTRPT, CHIPSET_REQUEST, 1, &scratch,
&axis_status) == ERROR)
      {
#ifdef DIAG
            printf("<Getting interrupr status failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("getting interrupt status : %x\n", axis_status);
#endif

   return((LONG) axis_status);
}

STATUS rst_intrpt(LONG the_value)
{
   WORD  mask, scratch;

#ifdef DIAG
      printf("resetting interrupt using mask: %lx\n", the_value);
#endif

   mask = (WORD) (the_value & 0xffffL);
      if (send_chipset_cmd(RST_INTRPT, CHIPSET_SET, 1, &mask, &scratch)
== ERROR)
      {
#ifdef DIAG
            printf("Resetting interrupt using mask: %lx failed.>\n",
the_value);
```

```
#endif
            return ERROR;
      }

      return OK;
}

LONG  get_intrpt_mask(void)
{
   WORD  interrupt_mask, scratch;

   if (send_chipset_cmd(GET_INTRPT_MASK, CHIPSET_REQUEST, 1, &scratch,
&interrupt_mask) == ERROR)
      {
#ifdef DIAG
            printf("<Getting interrupt mask failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("Interrupt mask: %04x\n", interrupt_mask);
#endif

   return((LONG) interrupt_mask);
}

/*
-----------------------------------------------
STATUS/MODE PROCESSING
-----------------------------------------------
*/

STATUS clr_status(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("clearing status...\n");
#endif

      if (send_chipset_cmd(CLR_STATUS, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Clearing status failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS rst_status(LONG the_value)
{
   WORD  scratch1, scratch2;
```

```
#ifdef DIAG
      printf("clearing status of selected bits using mask: %lx\n\r",
the_value);
#endif

      if (send_chipset_cmd(RST_STATUS, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Clearing status of selected bits using mask: %lx
failed.>\n", the_value);
#endif
            return ERROR;
      }

      return OK;
}

LONG  get_status(void)
{
   WORD  axis_status, scratch;

   if (send_chipset_cmd(GET_STATUS, CHIPSET_REQUEST, 1, &scratch,
&axis_status) == ERROR)
      {
#ifdef DIAG
            printf("<Getting status failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("getting status : %x\n", axis_status);
#endif

   return((LONG) axis_status);
}

LONG  get_mode(void)
{
   WORD  misc_mode_info, scratch;

   if (send_chipset_cmd(GET_MODE, CHIPSET_REQUEST, 1, &scratch,
&misc_mode_info) == ERROR)
      {
#ifdef DIAG
            printf("<Getting mode failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("mode: %x\n", misc_mode_info);
#endif

   return((LONG) misc_mode_info);
}
```

```
/*
-----------------------------------------------
ENCODER
-----------------------------------------------
*/

STATUS set_cnts(LONG  the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("setting # counts / rev to: %lx\n", the_value);
#endif

   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_CNTS, CHIPSET_SET, 1, &data, &scratch)
== ERROR)
      {
#ifdef DIAG
            printf("<Setting # counts / rev to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_capt_index(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting capture mode to index signal\n");
#endif

      if (send_chipset_cmd(SET_CAPT_INDEX, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting capture mode to index signal failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_capt_home(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting capture mode to home signal\n");
#endif
```

```
      if (send_chipset_cmd(SET_CAPT_HOME, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAF
            printf("<Setting capture mode to home signal failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

LONG  get_cnts(void)
{
   WORD  counts, scratch;

   if (send_chipset_cmd(GET_CNTS, CHIPSET_REQUEST, 1, &scratch,
&counts) == ERROR)
      {
#ifdef DIAG
            printf("<Getting counts failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("# counts/rev: %04x\n", counts);
#endif

   return((LONG) counts);
}

LONG  get_capt(void)
{
   WORD  data[2], scratch;
   LONG  index;

   if (send_chipset_cmd(GET_CAPT, CHIPSET_REQUEST, 2, &scratch, data)
== ERROR)
      {
#ifdef DIAG
            printf("<Getting capture failed.>\n");
#endif
            return ERROR;
      }

   index = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] & 0xffffL);

#ifdef DIAG
      printf("index: %lx\n", index);
#endif

   return((LONG) index);
}

/*
```

```
----------------------------------------
MOTOR
----------------------------------------
*/

STATUS set_output_pwm(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting output mode to PWM...\n");
#endif

      if (send_chipset_cmd(SET_OUTPUT_PWM, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting output mode to PWM failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_output_dac12(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting output mode to DAC12...\n");
#endif

   if (send_chipset_cmd(SET_OUTPUT_DAC12, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting output mode to DAC12 failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_output_dac16(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting output mode to DAC16...\n");
#endif

      if (send_chipset_cmd(SET_OUTPUT_DAC16, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
```

```
            printf("<Setting output mode to DAC16 failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS mtr_on(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("turn motor on...\n");
#endif

      if (send_chipset_cmd(MTR_ON, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Turning motor on failed.>n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS mtr_off(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("turn motor off...\n");
#endif

      if (send_chipset_cmd(MTR_OFF, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Turning motor off failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS set_mtr_cmd(LONG the_value)
{
   WORD  mtr_cmd, scratch;

#ifdef DIAG
      printf("setting motor cmd to:  %lx\n", the_value);
#endif

   mtr_cmd = (WORD) (the_value & 0xffffL);
```

```
      if (send_chipset_cmd(SET_MTR_CMD, CHIPSET_SET, 1, &mtr_cmd,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Setting motor command to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}

/* this routine is only available on the -b chipsets */

STATUS set_buf_mtr_cmd(LONG the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("setting buffered motor command to: %lx\n\r", the_value);
#endif

   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_BUF_MTR_CMD, CHIPSET_SET, 1, &data,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Setting buffered motor command to: %lx
falied.>\n", the_value);
#endif
            return ERROR;
      }

      return OK;
}

/* this routine is only available on the -b chipsets */

LONG  get_buf_mtr_cmd(void)
{
   WORD  buf_mtr_cmd_value, scratch;

      if (send_chipset_cmd(GET_BUF_MTR_CMD, CHIPSET_REQUEST, 1,
&scratch, &buf_mtr_cmd_value) == ERROR)
      {
#ifdef DIAG
            printf("<Getting buffer motor command failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("\n\r\rdouble buffered motor cmd value: %04x\n\r",
buf_mtr_cmd_value);
```

```
#endif
   return ((LONG) buf_mtr_cmd_value);
}

LONG  get_output_mode(void)
{
   WORD  output_mode, scratch;

   if (send_chipset_cmd(GET_OUTPUT_MODE, CHIPSET_REQUEST, 1, &scratch,
&output_mode) == ERROR)
      {
#ifdef DIAG
            printf("<Getting output mode failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("Output_mode: %s\n", (output_mode == 0) ? "PWM" :
((output_mode == 1) ? "DAC12" : "DAC16"));
#endif

   return((LONG) output_mode);
}

/*
----------------------------------------
MISCELLANEOUS
----------------------------------------
*/

STATUS axis_on(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("turn axis on...\n");
#endif

   if (send_chipset_cmd(AXIS_ON, CHIPSET_SET, 0, &scratch1, &scratch2)
== ERROR)
      {
#ifdef DIAG
            printf("<Turnning axis on failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS axis_off(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("turn axis off...\n");
```

```
#endif

      if (send_chipset_cmd(AXIS_OFF, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Turnning axis off failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

LONG  get_actl_pos(void)
{
   WORD  data[2], scratch;
   LONG  actual_position;

   if (send_chipset_cmd(GET_ACTL_POS, CHIPSET_REQUEST, 2, &scratch,
data) == ERROR)
      {
#ifdef DIAG
            printf("<Getting actual position failed.>\n");
#endif
            return ERROR;
      }

   actual_position = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] &
0xffffL);

#ifdef DIAG
      printf("actual_position: %ld\n", actual_position);
#endif

   return(actual_position);
}

STATUS set_lmt_sense(LONG the_value)
{
   WORD  limit_sense, scratch;

#ifdef DIAG
      printf("setting limit sense to: %lx\n", the_value);
#endif

   limit_sense = (WORD) (the_value & 0xffffL);

   if (send_chipset_cmd(SET_LMT_SENSE, CHIPSET_SET, 1, &limit_sense,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Setting limit sense to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }
```

```
      return OK;
}

LONG  get_lmt_swtch(void)
{
   WORD  limit_switch, scratch;

   if (send_chipset_cmd(GET_LMT_SWTCH, CHIPSET_REQUEST, 1, &scratch,
&limit_switch) == ERROR)
      {
#ifdef DIAG
            printf("<Getting limint switich failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("limit switch:     %04x\n\n", limit_switch);
#endif

   return((LONG) limit_switch);
}

STATUS lmts_on(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting limit switches to on...\n");
#endif

      if (send_chipset_cmd(LMTS_ON, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting limit switich to on failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

STATUS lmts_off(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("setting limit switches to off...\n");
#endif

      if (send_chipset_cmd(LMTS_OFF, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting limit switched to off failed.>\n");
```

```
#endif
            return ERROR;
      }

      return OK;
}


LONG  get_home(void)
{
   WORD  home_status, scratch;

   if (send_chipset_cmd(GET_HOME, CHIPSET_REQUEST, 1, &scratch,
&home_status) == ERROR)
      {
#ifdef DIAG
            printf("<Getting home status failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("Home Status: %04x\n", home_status);
#endif

   return((LONG) home_status);
}


STATUS set_smpl_time(LONG the_value)
{
   WORD  sample_time, scratch;

#ifdef DIAG
      printf("setting sample time to: %lx\n", the_value);
#endif

   sample_time = (WORD) (the_value & 0xffffL);
      if (send_chipset_cmd(SET_SMPL_TIME, CHIPSET_SET, 1, &sample_time,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Setting sample time to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}

LONG  get_smpl_time(void)
{
   WORD  sample_time, scratch;

   if (send_chipset_cmd(GET_SMPL_TIME, CHIPSET_REQUEST, 1, &scratch,
&sample_time) == ERROR)
```

```
      {
#ifdef DIAG
            printf("<Getting sample time failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("Sample time: %04x\n", sample_time);
#endif

   return((LONG) sample_time);
}

STATUS reset(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("resetting chipset...\n");
#endif

      if (send_chipset_cmd(RESET, CHIPSET_SET, 0, &scratch1, &scratch2)
== ERROR)
      {
#ifdef DIAG
            printf("<Resetting failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}

LONG  get_vrsn(void)
{
   WORD  data, scratch;
   WORD  generation, num_axis, part_num, dash_num, maj_version_num,
min_version_num;

   if (send_chipset_cmd(GET_VRSN, CHIPSET_REQUEST, 1, &scratch, &data)
== ERROR)
      {
#ifdef DIAG
            printf("<Getting version failed.>\n");
#endif
            return ERROR;
      }

   generation = (data & 0xc000) >> 14;
   num_axis = ((data & 0x3800) >> 11)+1;
   part_num = (data & 0x0700) >> 8;
   dash_num =  (data & 0xe0) >> 5;
   maj_version_num = (data & 0x18) >> 3;
   min_version_num = data & 7;

#ifdef DIAG
```

```
   {
      printf("P/N: %04x%s, version #: %0d.%0d\n",
         (generation << 12) + (num_axis << 8) + part_num, (dash_num ==
0) ? "" : "-P",   maj_version_num, min_version_num);
   }
#endif

   return((LONG) data);
}

LONG  get_time(void)
{
   WORD  data[5], scratch;
   LONG  chip_time;

   if (send_chipset_cmd(GET_TIME, CHIPSET_REQUEST, 2, &scratch, data)
== ERROR)
      {
#ifdef DIAG
            printf("<Getting time failed.>\n");
#endif
            return ERROR;
      }

   chip_time = ((data[HIGH] & 0xffffL) << 16) | (data[LOW] & 0xffffL);

#ifdef DIAG
      printf("chip time: %lx\n", chip_time);
#endif

   return(chip_time);
}



/*
-------------------------------------
  ADDITIONAL COMMANDS FOR MC1401A
-------------------------------------
*/


STATUS stop(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("stop...\n");
#endif

      if (send_chipset_cmd(STOP, CHIPSET_SET, 0, &scratch1, &scratch2)
== ERROR)
      {
#ifdef DIAG
            printf("<Stoping failed.>\n");
#endif
            return ERROR;
```

```
      }

      return OK;
}


STATUS smooth_stop(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("smooth_stop...\n");
#endif

      if (send_chipset_cmd(SMOOTH_STOP, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Smooth stop failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}


STATUS set_mtn_cmplt_brk(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("set_mtn_cmplt_brk...\n");
#endif

      if (send_chipset_cmd(SET_MTN_CMPLT_BRK, CHIPSET_SET, 0,
&scratch1, &scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting motion complete break failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}



STATUS set_ext_brk(void)
{
   WORD  scratch1, scratch2;

#ifdef DIAG
      printf("set_ext_brk...\n");
#endif
```

```
   if (send_chipset_cmd(SET_EXT_BRK, CHIPSET_SET, 0, &scratch1,
&scratch2) == ERROR)
      {
#ifdef DIAG
            printf("<Setting ecteranl break failed.>\n");
#endif
            return ERROR;
      }

      return OK;
}


STATUS set_mtr_lmt(LONG  the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("set_mtr_lmt to: %lx\n", the_value);
#endif

   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_MTR_LMT, CHIPSET_SET, 1, &data,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Setting motor limit to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }

      return OK;
}



STATUS set_mtr_bias(LONG  the_value)
{
   WORD  data, scratch;

#ifdef DIAG
      printf("set_mtr_bias to: %lx\n", the_value);
#endif

   data = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_MTR_BIAS, CHIPSET_SET, 1, &data,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Setting motor bias to: %lx failed.>\n",
the_value);
#endif
            return ERROR;
      }
```

```
      return OK;
}


STATUS set_actl_pos(LONG   the_value)
{
   WORD  data[2], scratch;

#ifdef DIAG
      printf("setting actual destination position to: %lx\n",
the_value);
#endif

   data[HIGH] = (WORD) ( (the_value >> 16)   & 0xffffL);
   data[LOW] = (WORD) (the_value & 0xffffL);

      if (send_chipset_cmd(SET_ACTL_POS, CHIPSET_SET, 2, data,
&scratch) == ERROR)
      {
#ifdef DIAG
            printf("<Setting actual destination position to: %lx
failed.>\n", the_value);
#endif
      }

      return OK;
}


LONG  get_mtr_lmt(void)
{
   WORD  mtrLimit, scratch;

   if (send_chipset_cmd(GET_MAX_ACC, CHIPSET_REQUEST, 1, &scratch,
&mtrLimit) == ERROR)
      {
#ifdef DIAG
            printf("<Getting motor limit failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("motor limit: %x\n", mtrLimit);
#endif

   return((LONG) mtrLimit);
}


LONG  get_mtr_bias(void)
{
   WORD  mtrBias, scratch;

   if (send_chipset_cmd(GET_MAX_ACC, CHIPSET_REQUEST, 1, &scratch,
&mtrBias) == ERROR)
```

```
      {
#ifdef DIAG
            printf("<Getting motor bias failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("motor bias: %x\n", mtrBias);
#endif

   return((LONG) mtrBias);
}


LONG  get_mtr_cmd(void)
{
   WORD  mtrCommand, scratch;

   if (send_chipset_cmd(GET_MAX_ACC, CHIPSET_REQUEST, 1, &scratch,
&mtrCommand) == ERROR)
      {
#ifdef DIAG
            printf("<Getting motor command failed.>\n");
#endif
            return ERROR;
      }

#ifdef DIAG
      printf("motor limit: %x\n", mtrCommand);
#endif

   return((LONG) mtrCommand);
}


/*
------------------------------------------------------------------------
---
Routine:       hardware_reset

Function:
               This routine sends a hardware reset command to the
chipset.

Arguments:
               none

Return Arguments:
               none
------------------------------------------------------------------------
---
*/

void  hardware_reset(void)
{
   outp_x(RESET_PORT_ADDRESS, RESET_CODE);
```

```
}

void  hardware_unreset(void)
{
   outp_x(RESET_PORT_ADDRESS, UNRESET_CODE);
}


void enable_amps(void)
{
   outp_x(AMP_ENABLE_PORT, AMP_ENABLE_ALL);
}

void enable_amp(int axis)
{
   switch (axis)
   {
        case 1: outp_x(AMP_ENABLE_PORT, AMP_ENABLE_1); break;
        case 2: outp_x(AMP_ENABLE_PORT, AMP_ENABLE_2); break;
        case 3: outp_x(AMP_ENABLE_PORT, AMP_ENABLE_3); break;
        case 4: outp_x(AMP_ENABLE_PORT, AMP_ENABLE_4); break;
   }
}

void disable_amp(void)
{
   outp_x(AMP_ENABLE_PORT, 0);
}


BYTE read_home(void)
{
   return inp_x(HOME_STATUS_PORT_1);
}

/*
------------------------------------------------------------------------
---
Routine:       wait_for_interrupt

Function:
               This routine polls the host interrupt bit of the status
port
               and returns when an interrupt is active (the motion
chipset
               is asserting the host_interrupt signal). Depending on
the
               IRQ selection setting of the board, a chipset host
signal
               may or may not generate a DOS interrupt. In any case
this
               routine only looks at the 'polled' mode bit in the
status
               word. Note that this routine has no waiting period
timeout. If
               the chipset is not receiving power, or if there is some
other
```

```
                hardware I/O problem, this routine many wait in an
"infinite loop".

Arguments:
                none

Return Arguments:
                none
-------------------------------------------------------------------------
---
*/

/*
void  wait_for_interrupt(void)
{
   BYTE  the_byte;

   while (TRUE)
   {
      the_byte = inp_x(INTERRUPT_PORT_ADDRESS);

      if ((the_byte & HOST_INTERRUPT_BIT_MASK)  == INTERRUPT_BIT_SENSE)
         return;

   }
}
*/
```

Appendix D: Source Code for Motion Control (`motion.o`)

```
#include "vxWorks.h"
#include "iv.h"
#include "ioLib.h"
#include "iosLib.h"
#include "intLib.h"
#include "errnoLib.h"
#include "sysLib.h"
#include "semLib.h"
#include "stdio.h"
#include "motion.h"
#include "pmdio.h"
#include "math.h"


/*Global Variables*/
int CurrentAxis = 0;
float CountsPerMM = 100.0;
float MMperCount = 0.0;
long ZeroPos = 0;
long MotionRange = 0;

static long abs(long value)
{
    value = (value<0)? -value : value;
    return value;
}

void WaitForSettle()
{
      unsigned long t1;
      while ((t1=get_time()) > (0xffffffff - 10000));
      while (get_time() < t1+1000);
}

STATUS SetPID(long P, long I, long D, long I_LM)
{
      if (set_kp(P) == ERROR)
            return ERROR;

      if (set_ki(I) == ERROR)
            return ERROR;

      if (set_kd(D) == ERROR)
            return ERROR;

      if (set_i_lm(0) == ERROR)
            return ERROR;

      if (set_kvff(0) == ERROR)
            return ERROR;

      if (set_i_lm(I_LM) == ERROR)
            return ERROR;
```

```
	return OK;
}


STATUS Reset()
{
	hardware_reset();
	hardware_unreset();

	if (reset() == ERROR)
		return ERROR;

	disable_amp();

	return OK;
}


STATUS Stop()
{
	if (stop() == ERROR)
		return ERROR;

	if (GoUpdate() == ERROR)
		return ERROR;

	return OK;
}


STATUS SelectAxis(int Axis)
{
	disable_amp();

	switch(Axis)
	{
		case 1:
			if (set_1() == ERROR)
				return ERROR;
			break;
		case 2:
			if (set_2() == ERROR)
				return ERROR;
			break;
		case 3:
			if (set_3() == ERROR)
				return ERROR;
			break;
		case 4:
			if (set_4() == ERROR)
				return ERROR;
			break;
	}

	if (Stop() == ERROR)
		return ERROR;
```

```
        if (set_smpl_time(SMPL) == ERROR)
                return ERROR;

        if (set_auto_stop_off() == ERROR)
                return ERROR;

        if (axis_on() == ERROR)
                return ERROR;

        if (mtr_on() == ERROR)
                return ERROR;

        enable_amp(Axis);

        CurrentAxis = Axis;

        return OK;
}



/* Move "counts" relative from the current position in "tms"
milliseconds */
STATUS MoveCounts(long counts, int tms)
{
        long destpos;
        long vel;
        long acc;
        float fsmpl_ms;
        float ftemp1, ftemp2;
        float ftms;
        float fcounts;

        if (stop() == ERROR)
                return ERROR;

        if (GoUpdate() == ERROR)
                return ERROR;

        fsmpl_ms = (float) SMPL * 100 / 1000;
        ftms = (float) tms;
        fcounts = (float) abs(counts);

        destpos = get_actl_pos() + counts;

        ftemp1 = fcounts / (ftms / fsmpl_ms);
        vel = (long)(ftemp1 * 65536);

        ftemp2 = (ftms*1000/10000) / fsmpl_ms;
        ftemp1 = ftemp1/ftemp2;
        acc = (long)(ftemp1 * 65535);

#ifdef DIAG
        printf("\n smpl_ms: %10.4f", fsmpl_ms);
        printf("\n ftms: %10.4f", ftms);
        printf("\n fcounts: %10.4f", fcounts);
        printf("\n destpos: %d", destpos);
```

```
        printf("\n vel: %d", vel);
        printf("\n acc: %d\n\n", acc);
#endif

        if (set_prfl_trap() == ERROR)
                return ERROR;
        if (set_pos(destpos) == ERROR)
                return ERROR;
        if (set_vel(vel) == ERROR)
                return ERROR;
        if (set_acc(acc) == ERROR)
                return ERROR;
        if (clr_status() == ERROR)
                return ERROR;
        if (axis_on() == ERROR)
                return ERROR;
        if (mtr_on() == ERROR)
                return ERROR;
        if (GoUpdate() == ERROR)
                return ERROR;

        return OK;
}


/* Move "MM" relative to the "ZeroPos" in "tms" millisec */
STATUS MoveArm(long MM, int tms)
{
        long actualPos;
        long absPos;
        long movecount;
        long counts;

        counts = MM * CountsPerMM;
        absPos = ZeroPos + counts;
        actualPos = get_actl_pos();
        movecount = absPos - actualPos;

#ifdef DIAG
        printf("\nCountsPerMM=%li",CountsPerMM);
        printf("\ncounts=%li",counts);
        printf("\nabsPos=%li",absPos);
        printf("\nactualPos=%li",actualPos);
        printf("\nmovecount=%li",movecount);
        printf("\n");
#endif

        return MoveCounts( movecount, tms );
}


/* Move arm until home sensor is ON using the specified velocity */
STATUS CaptureHome(long vel, short int nDir)
{
        /* stop any motion */
        if (Stop() == ERROR)
```

```
        return ERROR;

/* Abort if home sensor is already ON */
if ((read_home() & (0x01 <<((CurrentAxis-1) * 4))) != 0)
        return OK;

/* setup motion parms */
if (set_prfl_trap() == ERROR)
        return ERROR;
if (set_actl_pos(0) == ERROR)
        return ERROR;
if (set_pos(0) == ERROR)
        return ERROR;
if (set_vel(NORMAL_VEL) == ERROR)
        return ERROR;
if (set_acc(NORMAL_ACC) == ERROR)
        return ERROR;
if (axis_on() == ERROR)
        return ERROR;
if (mtr_on() == ERROR)
        return ERROR;
if (GoUpdate() == ERROR)
        return ERROR;

/* set up for home capture */
if (get_capt() == ERROR) /* reset the capt hw */
        return ERROR;
if (set_capt_home() == ERROR)
        return ERROR;
if (clr_status() == ERROR)
        return ERROR;
if (set_pos( nDir * BACKWARD * ONE_REV_COUNT/2) == ERROR)
        return ERROR;
if (set_vel(vel) == ERROR)
        return ERROR;
if (GoUpdate() == ERROR)
        return ERROR;

/* set auto smooth stop */
if (smooth_stop() == ERROR)
        return ERROR;
if (set_ext_brk() == ERROR)
        return ERROR;

/* wait till home capture is on */
while ( (get_status() & GS_POSITION_CAPTURED) == 0);

/* check if error occured */
if (get_status() & GS_MOTION_ERROR)
{
        /* resume pmd */
        if (clr_status() == ERROR)
                return ERROR;
        if (synch_prfl() == ERROR)
                return ERROR;
        if (GoUpdate() == ERROR)
                return ERROR;
```

```
	}

	return OK;
}

/* Implements the Seek Home algorithm */
STATUS Home(short int nDir)
{
	/* if not at home */
	if ((read_home() & (0x01 << ((CurrentAxis-1) * 4))) == 0)
	{
		/* fast home */
		if (CaptureHome(NORMAL_VEL/10, nDir) == ERROR)
			return ERROR;
	}
	WaitForSettle();

	/* repeat move out N counts until not at home */
	while ((read_home() & (0x01 <<((CurrentAxis-1) * 4))) != 0)
	{
		if (set_act1_pos(0) == ERROR)
			return ERROR;
		if (MoveCounts(nDir * FORWARD * 200,50) == ERROR)
			return ERROR;
		while ((get_status() & GS_MOTION_COMPLETE) == 0);
	}
	WaitForSettle();

	/* slow home */
	if (CaptureHome(NORMAL_VEL/500, nDir) == ERROR)
		return ERROR;
	WaitForSettle();

	/* should be home now */
	if (Stop() == ERROR)
		return ERROR;
	if (clr_status() == ERROR)
		return ERROR;
	if (set_act1_pos(0) == ERROR)
		return ERROR;

	return OK;
}



STATUS CaptureIndex(long vel)
{
	/* stop any motion */
	Stop();

	/* setup motion parms */
	if (set_prfl_trap() == ERROR)
		return ERROR;
	if (set_vel(vel) == ERROR)
		return ERROR;
```

```
      if (set_acc(NORMAL_ACC) == ERROR)
            return ERROR;
      if (axis_on() == ERROR)
            return ERROR;
      if (mtr_on() == ERROR)
            return ERROR;

      /* setup for index capture */
      if (get_capt() == ERROR) /* reset the capt hw */
            return ERROR;
      if (set_capt_index() == ERROR)
            return ERROR;
      if (clr_status() == ERROR)
            return ERROR;
      if (set_pos(get_act1_pos() + (FORWARD * ONE_REV_COUNT/2)) ==
ERROR)
            return ERROR;
      if (GoUpdate() == ERROR)
            return ERROR;

      /* wait till home capture is on */
      while ((get_status() & GS_POSITION_CAPTURED) == 0);

      /* stop the motion */
      if (smooth_stop() == ERROR)
            return ERROR;
      if (GoUpdate() == ERROR)
            return ERROR;

      return OK;
}

LONG GetStatus()
{
      return get_status();
}

void RatioOfMMtoCounts(float mm, float counts)
{
printf("Counts: %f,  MM: %f.\n", counts, mm);
      CountsPerMM = (int)(counts / mm);
printf("CountsPerMM: %f.\n", CountsPerMM);
    MMperCount = mm / counts;
}


void SetArmZeroPosition(long counts)
{
    ZeroPos = counts;
}

void SetMotionRange(long counts)
{
    MotionRange = counts;
}
```

```
STATUS GetArmPosition(long *MM)
{
      LONG lPosition = get_actl_pos();

      if (lPosition == ERROR)
            return ERROR;

      *MM = (lPosition - ZeroPos) * MMperCount;

      return OK;
}

STATUS GetArmCount(long *counts)
{
      *counts = get_actl_pos();

      if (*counts == ERROR)
            return ERROR;

      return OK;
}


/* make sure that the previous data in the buffer is updated
   before the the new data is put into the buffer */

STATUS GoUpdate()
{
  LONG  prev_time, curr_time;

  prev_time = curr_time = get_time();
  /* wait until the next  x servo loop occurs */
  while ( curr_time < (prev_time + DELAY_LOOPS) )
    curr_time = get_time();

  return update();
}

void pmdHrdInit(int waitState)
{
      short int ad1 = 0x0600 | (waitState % 16);
      sysOutWord( 0xf400, ad1 );
      sysOutWord( 0xf402, 0x0040 );
      sysOutWord( 0xf406, 0x3f );
      sysOutWord( 0xf404, 0xfc01 );
}
```

References

[CiA '99] CAN in Automation, http://www.can-cia.de, 1999.

[Howe '93] Roger T. Howe and Charles G. Soldini, *Microelectronics*, Prentice Hall, 1997.

[Intel '95] Intel Corporation, "82527 Serial Communication Controller Architectural Overview", 1995.

[Performance '97] Performance Motion Devices, "Advanced Multi-Axis Motion Control Chipset", 1997.

[Philips '97] Philips, "SC26C92 Dual Universal Asynchronous Receiver/Transmitter (DUART) product specification", 1997.

[Silicon '99] Silicon Valley Group, *http://www.svg.com*, 1999.

[Wind '97] Wind River System, *VxWorks Programmer's Guide 5.3.1*, 1997.