

Appendix C

Data Acquisition and Control—Some Hints

Danial J. Neebel

In the previous two appendices we described how to set up the hardware to use with UW DigiScope and how some of the routines in **DACPAC** can be used. Here we provide some helpful hints on how to develop some of these routines and set up the hardware to perform DAC. We are not describing everything that must be done. We give the reader some hints and advice as to what to do and what not to do. We will also give some good references for performing some of the operations needed to do data acquisition and control with the IBM PC.

This appendix should give the reader some idea of how to go about setting up a simple data acquisition and control system. It includes information on the basic elements required to read analog and digital signals into a computer using the three different types of devices presented in the previous appendix. The first type is an internal device. An internal device is connected to and communicates with the PC via the internal expansion bus. An I/O device can also be external. An external device communicates with the PC via either a serial or parallel communication port. The most common communication ports are RS232 (serial) and IEEE 488 (parallel). In this text, we discuss only RS232 communications since almost all IBM PC architecture machines have an RS232 serial port available. The last type of device is a virtual I/O device. DigiScope uses data files and a timer interrupt to simulate analog data acquisition. These same file utilities are used to store and retrieve data gathered using internal and external analog input devices.

There are four basic operations involved in using an input/output device: Open Device, Input Data, Output Data, and Close Device. Open Device will initialize the device for the type of I/O requested and set up any interrupts that may be needed to perform exact timing. Input Data will determine if the data requested is available and retrieve the data from the device. Output Data will give the device a piece of data to output. Close Device will terminate all operations being performed by the device and disable any interrupts set up by Open Device. In the discussions that follow, we describe exactly what must be done to perform each of these operations for each of the three devices presented. The last section gives helpful hints for writing your own interface to be used with DigiScope.

All programming for the PC is done in Turbo C. We refer the reader to the Turbo C manuals for information on such things as serial communication routines and file input/output.

C.1 INTERNAL I/O DEVICE (RTD ADA2100)

This type of device requires installation inside the chassis of the IBM PC. Figure C.1 shows the connection of an ECG amplifier to an I/O card. Note that no extra hardware is required.

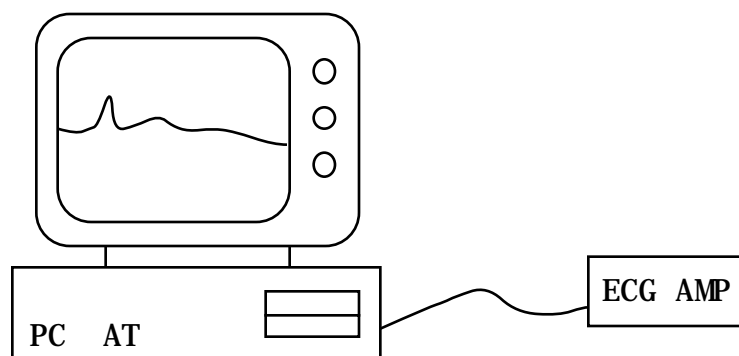


Figure C.1 Internal card data acquisition and control system.

C.1.1 Interfacing to an I/O card with Turbo C

Interfacing to a card installed in the PC is done using the library routines provided in Turbo C that read and write from and to the I/O space of the processor. For 8-bit I/O operations, `inportb()` and `outportb()` are used for input and output respectively. Functions `inport()` and `outport()` are used for 16-bit input and output. The ADA2100 is an 8-bit I/O card so we have used `inportb()` and `outportb()`. We refer the reader to the Turbo C Reference Guide for more information on these routines. Figure C.2 shows examples of using `inportb()` and `outportb()` to set up the 8259 interrupt controller. The 8259 is part of the PC system, but the operations of reading and writing using `inportb()` and `outportb()` are the same as reading and writing to a card.

C.1.2 Handling interrupts on the IBM PC with Turbo C

We discuss some of the basic operations required to properly set up interrupts and—possibly more important—how to make sure that interrupts are disabled when we do not want them to occur. Programming with interrupts is difficult because an interrupt can occur at any point in the execution of a program. We have limited control over when an interrupt can occur. An interrupt is used when an external event needs to stop whatever process is running and cause another process to

execute. There are many sources of interrupts. An interrupt can come from the keyboard, a disk controller, a serial port, the time of day interrupt, or many other sources. In this case we would like something to happen at very regular time intervals.

Interfacing to the interrupt controller

The IBM PC has one Intel 8259 interrupt controller. The interrupt controller is located at 0x20 and 0x21 in the I/O space of the processor. Note that we use the C-language convention for specifying hexadecimal (base 16) constants. The interrupt controller is the device that tells the processor that an external process has requested an interrupt. If interrupts are enabled, the processor will acknowledge the interrupt. The interrupt controller will then tell the processor where to look for the interrupt vector. The vector is the address of the interrupt service routine.

The 8259 interrupt controller has many capabilities, but we recommend that you only change the interrupt mask register, IMR. For a more detailed description of the 8259, see the Intel *Microprocessor and Peripheral Handbook, Volume I*, or Eggbrecht (1983). The IMR is located at 0x21 in I/O space. The IBM PC/AT architecture uses two 8259 interrupt controllers. The master interrupt controller is located at 0x21 and the slave is located at 0x70. The two interrupt controllers are cascaded to provide 15 different interrupt levels. IRQ2 of the master interrupt controller is connected to the slave.

When masking or unmasking an interrupt, it is very important to only change the mask of the interrupt of interest. It is equally important to mask the interrupt after use. Figure C.2 shows one method of unmasking an interrupt at the beginning of a program and returning the mask to the original setting at the end of the program.

Interfacing to the operating system—interrupt handling

Along with unmasking the interrupt in the interrupt controller, we must also initialize the interrupt vector to point to the proper interrupt handler. Turbo C provides two routines that make this very easy. Before setting the interrupt vector to the new interrupt handler, the current interrupt vector must be saved so that before the program exists to the operating system, the interrupt vector can be returned to the original value.

An important item to remember when using interrupts on the PC is to always return the system interrupt handler and interrupt vectors to the state they were in when the program started. To do this we save the current IMR and interrupt vector into two global variables and reset the IMR and interrupt vector to these values upon termination of the program. Exiting without resetting the IMR and interrupt vector could cause serious problems. If this event occurs for whatever reason, the user should reboot the system.

```

main(char *argv[], int argc)
{
    disable();                /* disables all interrupts */

    OldVector = getvect(10);  /* set interrupt vector */
    setvect(10, Itimer);

    OldMask = inportb(0x21);  /* unmask IRQ 2 on 8259 */
    mask = OldMask & 0xFB;
    outportb(0x21,mask);
    enable();                /* enables all interrupts */

    /* code that can be interrupted by INTERRUPT 10 */

    disable();

    outportb(0x21,OldMask);  /* return mask to original value
*/
    setvect(10, OldVector);  /* return interrupt vector to
*/
                                /* original value
*/
    enable();
}

```

Figure C.2 Setting the interrupt vector and interrupt mask register.

C.2 EXTERNAL I/O DEVICE (MOTOROLA 68HC11EVBU)

For an external input/output device, we have chosen the Motorola S68HC11EVBU student project kit. This device was chosen because of its availability and low cost. The drawback of using this device is the difficulty in setting it up. To turn the EVBU into an I/O device, a 68HC11 assembly language program is needed to communicate with the PC and perform the necessary input and output. Fortunately we have done this part of the task for you. An example system is shown in Figure C.3. Also, the EVBU is not a difficult device to program. The student evaluation kit includes enough tools and documentation so that anyone who has experience writing assembly language code should be able to program the EVBU.

An advantage of using this device is that all critical timing can be done by the 68HC11. This means we can avoid using interrupts on the PC altogether. However, the 68HC11 has a time-based interrupt. Interrupts are a little easier to handle on a microcontroller than on a PC; an operating system and disk drives and other items make a PC more complicated.

In this section we give a short introduction to using serial communications with Turbo C. Finally, we give an example of two routines, one written in Turbo C for the PC and one in assembly language for the EVBU; they each perform a simple communication sequence.

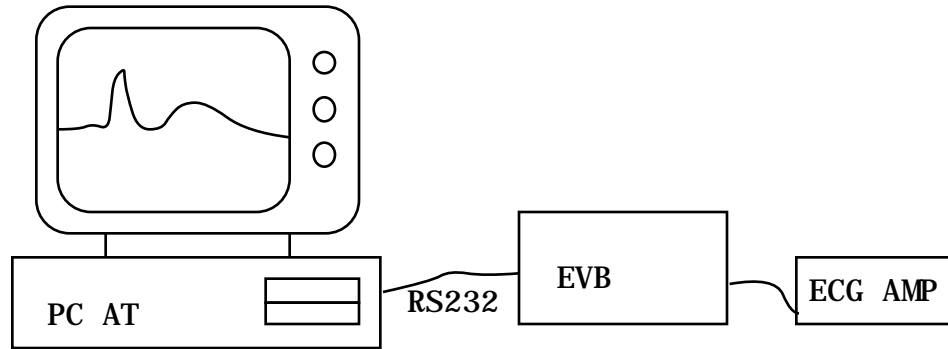


Figure C.3 Motorola EVBU data acquisition and control system.

C.2.1 Common operations

Here we briefly describe the operations needed to perform the basic operations of **OPEN**, **CLOSE**, **GET**, and **PUT**.

```
#define COM1      0
#define COM2      1
#define COM3      2
#define COM4      3
#define SETTINGS (0xE0|0x03|0x00|0x00)
                /* 9600 Bd 8 bits 1 stop no parity */

                /* CPORT contains COM port is being used */
int  CPORT=COM1;

                /* initialize serial port */
bioscom(0, SETTINGS, CPORT);

                /* read the status of serial port */
status = bioscom(3,0,CPORT);

                /* read the input buffer of serial port */
port = bioscom(2,0,CPORT);

                /* Send an ASCII Q out on serial port */
bioscom(1,'Q',CPORT);
```

Figure C.4 Use of the `bioscom()` Turbo C library routine.

Opening the device

An open device routine must initialize the RS232 serial port, establish a communication link with the EVBU, and initialize the EVBU for any timing and data taking that need to be performed. Initializing the serial port is easily done using the `bioscom()` routine provided by Turbo C (see Figure C.4). In section C.3.3 we give

an example of how this is done. Figure C.5 shows the communication sequence that takes place when the external device is opened.

- PC sends reset command "R" to EVBU for soft reset.
- EVBU gets "R" and jumps to reset vector. As part of reset, EVBU sends "R" to PC to echo Reset command.
- PC sends channel mask "Cx" to EVBU. x is an 8-bit mask with 1's in positions corresponding to analog input channels to be read.
- EVBU echoes channel mask "Cx" and saves it.
- PC sends delay setting, "Dxxxx" to EVBU. xxxx is in Hex and is timer ticks of 68HC11 timer.
- EVBU echoes "Dxxxx" command and saves delay setting.
- PC sends go command "G" to EVBU.
- EVBU echoes "G" command and starts timer interrupts.

Figure C.5 Communication sequence for initializing EVBU for real-time data transfer.

Closing the device

A close device routine need only send a command to tell the EVBU to terminate all data acquisition and stop sending data to the PC host.

Important: Starting EDAC

Each time the EVBU is reset it is necessary to use **pcBug11** to restart the **EDAC** program. The command to do this is:

PCBUG11 -E port=N macro=go.

Where N is the number of the COM port that the EVBU is connected to. N must be either 1, 2, 3, or 4. Also the following files must be in the current directory:

**PCBUG11.EXE, TALKE.XOO, TALKE.BOO, and
GO.MCR.**

All these files except for **GO.MCR** are included on the **pcBug11** disk you received with the EVBU. **GO.MCR** is included on your DigiScope disk.

Input data

An input data routine must give the EVBU a request to read data and wait for a response. Remember that the host should always limit the amount of time spent waiting for a response so as not to hang the computer waiting for an event that will never occur.

Output data

An output data routine must send the EVBU a command telling the EVBU to output a specified piece of data.

C.2.2 Serial communications using Turbo C

Serial communication is accomplished via the `bioscom()` routine. Function `bioscom()` is a multipurpose routine that can be used to initialize the serial port, check status, read the serial port, send a byte out on the serial port. The Turbo C reference manual gives some very helpful examples of how to perform each of these operations. Figure C.6 shows some examples from the code developed for this text.

```

send_command("R");          /* reset the EVBU */

/* Tell EVBU which channels the host wants to read. */
/* The EVBU will send this many bytes to the TERMINAL */
/* at the sample rate */

for (i=0, mask=0;i<Header->num_channels;i++) {
    mask |= 1 << i;
}
sprintf(buf, "C%X\r", mask);
send_command(buf);

/* calc delay betw. samples */
if (Header->rate != 0) {
    delay = 2000000 / Header->rate;
} else {
    delay = 0;          /* if rate is 0 send 0 delay */
}
sprintf(buf, "D%X\r", delay);    /* Tell EVBU delay */
send_command(buf);

send_command("G");          /* Tell EVBU to start sampling */

```

Figure C.6 Turbo C code for the PC to execute the sequence in Figure C.5.

C.2.3 A sample communication sequence and the code to execute it

Here we present the communication sequence used to open the EVBU device used by `Eopen()`. We have removed some of the error checking to make the code easier to understand. The sequence being executed is exactly the same as that performed by `Eopen()` on the PC and `EDAC.ASM` on the EVBU. The sequence is outlined in Figure C.5. The PC must provide the EVBU with sample rate and number of channels, and indicate if block transfer mode or real time mode is to be used. The code used to execute the sequence on the PC and EVBU is given in Figures C.6 and C.7 respectively. The `send_command()` routine shown in Figure C.8 sends a NULL-terminated string to the EVBU.

Note that a one-byte mask is sent to tell the EVBU which channels to read. Each bit position corresponds to an analog input channel. The bits and channels are numbered 0–7. If bit 2 is the only bit set in the mask, then channel 2 will be the only channel read.

	LDAA	#'R'	send signon character to host
	JSR	DATOUT	
SETUP	JSR	DATIN	
	BRCLR	FLAGS	
	RCVDAT	SETUP	wait for a character
	JSR	DATOUT	echo to host for host's error checking
	CMPA	#'D'	Check for delay
	BNE	SETUP1	
	JSR	GETDELAY	Read in the hex value and save
	BRA	SETUP	
SETUP1	CMPA	#'C'	Check for number of channels
	BNE	SETUP2	
	JSR	GETCHAN	Read in the hex value and save
	BRA	SETUP	
SETUP3	CMPA	#'G'	Check for Start
	BNE	SETUP	
* set up A/D converter and start interrupts			

Figure C.7 Assembly language code for 68HC11 to execute the sequence in Figure C.5.

C.3 VIRTUAL I/O DEVICE (DATA FILES)

File I/O is a normal operation to most programmers. Here we show one method of using files to simulate a physical I/O device. The first task is to make file I/O operations look like operations involving a physical I/O device. Next we need to provide some type of timing operation so that input and output take place at a specific rate. Providing timing is the difficult part of simulating a physical device

with data files. To perform the timing operation of virtual analog input, your system must have the time-of-day interrupt compatible with an IBM PC or PC/AT.

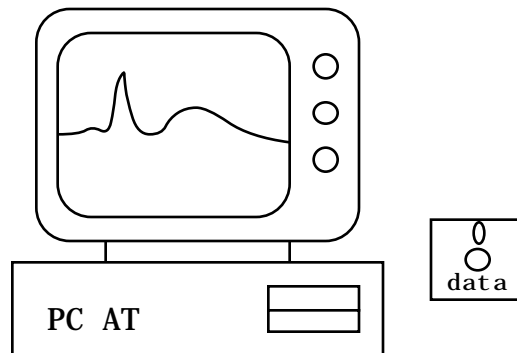


Figure C.8 A microcomputer and hardware for virtual analog input.

C.3.1 Stealing the time of day interrupt

Figure C.9 shows how to steal the time-of-day interrupt. The old interrupt vector is saved so that it can be restored and also so that the new timer interrupt routine can call the interrupt service routine approximately 18.2 times/s. This keeps the time-of-day clock running at a rate close to the correct rate. If this was not done the time-of-day would be set to an unknown value after the program was finished using the interrupt. For more information about what types of things can be done by stealing the time of day interrupt, see Bovens and Brysbaert (1990).

C.3.2 Initializing the 8253 for a specified sample rate

Figure C.10 shows a routine to set up the timer to a specified rate. The operations required are to calculate the number of clock ticks required between samples and then set up the timer. Timer setup requires setting **TIMER0** in **mode 2** and writing the least-significant byte then the most-significant byte to **TIMER0**. The timer control is located at 0x43 in I/O space and **TIMER0** is located at 0x40 in I/O space. The 8253 has three timers. **TIMER1** is used for the speaker output and **TIMER2** is used for dynamic memory refresh. It is very important that both of these timers remain undisturbed. We refer the reader to the *Intel Microprocessor and Peripheral Handbook, Volume II* for more information on the Intel 8253.

```

#define TIMER0      0x40      /* I/O mem locations for 8253
*/
#define TIMER_CTRL  0x43

void interrupt(*OldTimer)(void); /* global to save old ISR */

    disable();                /* disable interrupts */
    OldTimer = getvect(0x08);  /* save the old ISR*/
    setvect(0x08, SuperTimer); /* set new ISR to our routine
*/

    SetUpTimer(rate);        /* initialize timer to rate */

    enable();                /* enable interrupts */

/***** code that can be interrupted goes here *****/

    disable();                /* disable interrupts */
    setvect(0x08, OldTimer);  /* restore ISR */

    outportb(TIMER_CTRL, 0x36); /* timer 0, mode 3, LSB and MSB
*/
    outportb(TIMER0, 0x00);
    outportb(TIMER0, 0x00);

    enable();                /* enable interrupts */

```

Figure C.9 Code to steal the time-of-day interrupt (INT 8).

```

#define TIMER0      0x40      /* I/O mem locations for 8253
*/
#define TIMER_CTRL  0x43

#define TIMER_CLOCK (long)1192755 /* Hz crystal for 8253 */
#define BIOS_TIC    (double)18.2 /* in ticks per second */

int old_timer_call;

void SetupTimer(frequency)
int frequency;
{
    long          divisor;
    int data;

    old_timer_call = (int) (((double) frequency) / BIOS_TIC);

    divisor = TIMER_CLOCK / ((long) frequency);

    outportb(TIMER_CTRL, 0x34); /* timer 0, mode 3, LSB and MSB */
    data = (int) (divisor & 0xFF);
    outportb(TIMER0, data);
    data = (int) ((divisor >> 8) & 0xFF);
    outportb(TIMER0, data);
}

```

Figure C.10 Code to set the 8253 **TIMER0** to for a specified sample rate.

C.3.3 Writing an interrupt service routine in Turbo C

It is important that an ISR return the system to the original state the system was in before the interrupt service routine was executed. This means that an ISR must save the current state of the processor when the ISR was started. Fortunately, Turbo C makes sure this is done if the routine is declared with “void interrupt.”

```
void interrupt SuperTimer()
{
    static int    super_counter = 0;

    TIME_OUT = TRUE;

    if (++super_counter == old_timer_call)
    {
        OldTimer();          /* execute old timer approx. 18.2 */
        super_counter = 0;    /* times per second */
    } else {
        outportb(0x20, 0x20); /* interrupt acknowledge signal */
    }
}
```

Figure C.11 Interrupt service routine to set flag and call real-time clock approximately 18.2 times/s.

Figure C.11 shows an ISR that could be used for virtual A/D in **DACPAC.LIB**. Note that the **SuperTimer()** routine only sets a flag and updates the real-time clock. By limiting the amount of work done by an ISR, we can eliminate some of the problems caused by using interrupts.

C.4 WRITING YOUR OWN INTERFACE SOFTWARE

One of the most common needs of users of the data acquisition software available with DigiScope will be adding a new interface device. In this section, we give some directions for adding an interface to a different internal card to DigiScope.

C.4.1 Writing the interface routines

Section C.2 gives some hints for one method of interfacing Turbo C code to an internal card. Most I/O cards will be shipped with some interface routines and/or a manual and examples for writing these routines. The trick to making the internal card work with DigiScope will be to match the card interfaces to the interfaces to DigiScope. Appendix B shows all of the function prototypes used in DigiScope for interfacing to the various types of I/O devices. For example, if you wish to write an

interface to an internal card, you will need to write the routines included in **IDAC.OBJ**. A list of those functions is given in Figure B.4.

Examples of the code included in the module **IDAC.OBJ** are given in Figures C.12(a), C.12(b), and C.13. Figure C.12 shows the code used to open the device and set up necessary timing and I/O functions. You should take note of the items in the internal data structure **Header** that are initialized.

```

/* Global Variables */
unsigned int    BaseAddress = 200; /* base address for RTD card */
unsigned int    IRQline = 3;
static char    AD_TIME_OUT;
static char    DA_TIME_OUT;
static int     OldMask;
static int     NUM_CHANNELS;      /* used to remember how many */
                                  /* channels to read */
static char    OPEN=NO;
static char    TIMED=NO;

/* RTD board addresses */

#define         PORTA          BaseAddress + 0
#define         PORTB          BaseAddress + 1
#define         PORTC          BaseAddress + 2
#define         PORT_CNTRL    BaseAddress + 3

#define         SOC12          BaseAddress + 4
#define         SOC8           BaseAddress + 5
#define         AD_MSB         BaseAddress + 4
#define         AD_LSB         BaseAddress + 5

#define         DA1_LSB        BaseAddress + 8
#define         DA1_MSB        BaseAddress + 9
#define         DA2_LSB        BaseAddress + 0xA
#define         DA2_MSB        BaseAddress + 0xB
#define         UPDATE         BaseAddress + 0xC
#define         CLEAR_DA       BaseAddress + 0x10

#define         TIMER0         BaseAddress + 0x14
#define         TIMER1         BaseAddress + 0x15
#define         TIMER2         BaseAddress + 0x16
#define         TIMER_CNTRL    BaseAddress + 0x17

char Iopen(DataHeader_t *Header)
{
    int register i,j,in, out;
    int delay, num_channels;
    char buf[20];
    int mask;

```

Figure C.12(a) Beginning of routine to open internal card for analog I/O.

```

AD_TIME_OUT = FALSE;
DA_TIME_OUT = FALSE;

Header->volthigh = 5.0; /* initialize header data structure
*/
Header->voltlow = -5.0;

Header->resolution = 12;
if (Header->num_channels > 8)
    Header->num_channels = 8;
NUM_CHANNELS = Header->num_channels;
Header->num_samples = 0;
Header->data = NULL;

/* initialize internal card */
/* PORT C low is input */
/* PORT C high is output */

outportb(PORT_CNTRL,0x91);

/* set gain and offset for each channel during Iget routine */
/* type should be set by user */
/* for now the gain is always set to one */

for (i=0;i<Header->num_channels;i++) {
    ChanOffset(Header,i) = 0.0;
    ChanGain(Header,i) = 1.0;
}

if (!CheckTimer()) return(NO); /* This routine is used to
*/
/* check if the board is installed
*/

/* setup timing function
*/
if (Header->rate != 0) {
    disable(); /* only perform timing if rate is */
/* nonzero */
    ISetupTimer(Header->rate); /* setup 8253 timer */

    OldVector = getvect(IRQnumber); /* set interrupt vector */
    setvect(IRQnumber, Itimer);
/* unmask interrupt mask */
    OldMask = inportb(0x21);
    mask = OldMask & ~(1 << IRQline) & 0xFF;
    outportb(0x21,mask);
    enable();
    TIMED=YES;
} else { /* rate is zero => don't use timing
*/
    TIMED=NO;
}
OPEN=YES;
return(YES);

```

```
} /* Iopen() */
```

Figure C.12(b) End of routine to open internal card for analog I/O.

```
char Iget(DATATYPE *data)
{
    int msb, lsb, i;

    if (!OPEN) {
        DEVICE_NOT_OPEN();
        return(NO);
    }
    if (AD_TIME_OUT || !TIMED) {

        for (i=0;i<NUM_CHANNELS;i++) {
            /* read all channels requested */
            /* for now the gain is tied to 1 */
            outportb(PORTB,0x00 | i );          /* select channel */
            outportb(SOC12,0);                  /* start a conversion */
            while (!(inportb(PORTA) & 0x80));
                /* wait for conversion to end */
                /* after EOC then read MSB and LSB */
            msb = inportb(AD_MSB)*16; /* read in data from RTD card */
            lsb = inportb(AD_LSB)/16; /* the card should be in +/- mode */
            data[i] = msb + lsb - 2048;
        }

        AD_TIME_OUT = FALSE;      /* Clear AD_TIME_OUT flag */
        return(YES);
    } else {
        return(NO);
    }
} /* Iget */
```

Figure C.13 Routine to get analog input data from internal card.

C.4.2 Including the new interface in UW DigiScope

The file **DACPAC.LIB** contains all of the UW DigiScope data acquisition routines. To replace the current interface for internal I/O card, you need only replace the **IDAC.OBJ** module in **DACPAC.LIB**. To replace the external device and virtual device, replace the **EDAC.OBJ** and **FAD.OBJ** modules respectively. You should include all of the functions that are listed as included with those modules in Appendix B. Failure to include all the functions will at the least cause a compiler error and at worst a run-time error. If you do not wish to use a function, you may simply insert a dummy function in its place. For additional details, see the **README** file on the UW DigiScope disk.

C.5 REFERENCES

- ADA2100 User's Manual*. 1990. Real Time Devices, Inc., 820 North University Dr., P.O. Box 906, State College, PA 16804.
- Bovens, N. and Brysbaert, M. 1990. IBM PC/XT/AT and PS/2 Turbo Pascal timing with extended resolution. *Behavior Research Methods, Instruments, & Computers*, **22**(3): 332–34.
- Eggbrecht, L. C. 1983. *Interfacing to the IBM Personal Computer*. Indianapolis, IN: Howard W. Sams.
- M68HC11 User's Manual*. 1990. Motorola Literature Distribution, P.O. Box 20912, Phoenix, Arizona 85036.
- M68HC11EVB Evaluation Board User's Manual*. 1986. Motorola Literature Distribution, P.O. Box 20912, Phoenix, AZ 85036.
- Microprocessor and Peripheral Handbook, Volumes I and II*. 1988. Intel Literature Sales, P.O. Box 8130, Santa Clara CA, 95052-8130.
- Turbo C Reference Guide*. 1988. Borland International, Inc., 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95055-0001.
- Turbo C User's Guide*. 1988. Borland International, Inc., 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95055-0001.