Appendix A

Configuring the PC for UW DigiScope

Danial J. Neebel

This appendix provides the steps necessary to set up the various hardware systems that are supported by UW DigiScope. DigiScope can interface to three different types of devices: (1) the Real Time Devices ADA2100 analog and digital I/O interface card that is installed internally in the IBM PC or compatible, (2) an external single board computer with serial RS232 communications (Motorola S68HC11EVBU Student Design Kit and MCM68HC11EVB Evaluation Kit), and (3) a virtual I/O device (data files). When you install UW DigiScope with the `INSTALL.EXE` program, you identify which physical devices are available in your system. This selection can be changed later if you add a conversion device.

Figure A.1 shows the three different devices and the capabilities of each. A sampling rate of approximately 500 samples per second (sps) can be achieved with a 12-MHz 80286-based IBM PC/AT compatible. The maximum sampling rate of a slower 8088-based IBM PC or compatible running at 4.77 MHz is approximately 250 sps. Thus, depending on your driving habits, mileage may vary.

Item	Internal device (ADA 2100)	External device (Motorola EVBU)	Virtual I/O device
Sampling rate (sps)	1 to 500	31 to 500	1 to 500
Analog input range (V)	–5 to +5	0 to +5	N/A
Number of analog inputs	8	4	20
Number of analog outputs	2	none	none
Analog output range (V)	–10 to +10	N/A	N/A
Number of digital inputs	4	8	none
Number of digital outputs	4	8	none

Figure A.1 Comparison of the three types of devices.

An important step in installing any of the three devices in your system is running the `INSTALL` program. `INSTALL` sets up all the software for DigiScope. If the Real Time Devices ADA2100 is installed in the system, you should place a call to `DACINIT.COM` in the `AUTOEXEC.BAT`. `DACINIT.COM` resets the ADA2100 timer so that the timer does not cause interrupts. The documentation with the ADA2100 explains why it is necessary to do this.

In the following three sections, we provide the steps necessary to get a PC system ready for Data Acquisition using DigiScope. First we cover configuring and installing the ADA2100. Next we present the steps to connect the Motorola EVBU and EVB to a PC using RS232 communications. Finally, a few pointers are presented for setting up a good system to perform Virtual Data Acquisition. Before discussing any of these hardware setups, it is important to know how to organize the work area correctly to do the job right and safely. Read through the information in the following box.

> ***Warning: Preparing a Work Area***
>
> Safety first. Unplug all devices that you are working on. Never remove the case of any device without making sure that the device is unplugged. This will help protect you as well as your equipment.
>
> Installing and configuring your system for either the ADA2100 or the Motorola EVBU will require you to handle circuit boards containing delicate integrated circuits (ICs). Proper care should be taken to limit the likelihood of any device becoming damaged. Start with a clear work area. Items such as an antistatic mat and wristband can help limit static electricity. If you do not have these items then always make sure you have grounded yourself before touching any ICs or circuit boards. This will help reduce the amount of static electricity in your body.
>
> Many of the ICs on the ADA2100 and the EVBU are CMOS technology. Even a small amount of static electricity can damage a CMOS part permanently. The part may not show any signs of damage but it will fail to operate properly.

A.1 INSTALLING THE REAL TIME DEVICES ADA2100 IN AN IBM PC

Installing the Real Time Devices ADA2100 interface card in an IBM PC is a three-step process. Each step must be taken with great care. The first step is to configure the ADA2100. Next is the physical installation of the ADA2100 into the bus of the IBM PC. The final step is to set up the CONFIG.WDS file. This is done automatically by the INSTALL.EXE program and can be changed by running ADINSTAL.EXE.

A.1.1 Configuration of the ADA2100

DigiScope makes some assumptions about how the ADA2100 is configured. Figure A.2 gives the jumper and switch settings for the device. It is also important to check the rest of the internal cards in the system to determine if the ADA2100 settings will conflict with devices already installed in your system. The settings for the ADA2100 are flexible enough that this should not be a problem. ADA2100 interface card

Jumper/Switch	Function	Required settings
P2	Base I/O address	As defined by CONFIG.WDS
S1	Analog input signal type	Pos 1, 2, 3 UP and Pos 4 down
P3	PIT I/O header connector	Chain the timers together; see text below
P5	PIT interrupt header	As defined by CONFIG.WDS
P6	EOC monitor header	PA7
P7	EOC interrupt header	EOC not connected to any IRQ
P9	A/D converter voltage	10 V
P10	D/A converter voltage	Both set to ±

Figure A.2 RTD ADA2100 jumper configurations. ADA2100 interface card

The Base I/O Address (P2) must be set to the value defined during installation of the software. The analog input signal type (S1) should be set so that there are eight single-ended channels with ± polarity. The PIT I/O Header (P3) connector should be set so that OUT0 is connected to CLK1, OUT1 is connected to CLK2, and OUT2 can be connected to CO2 or –CO2. The EG0, EG1, and EG2 lines should be connected to +5 V. The PIT interrupt header, P5, must be configured so that OUT2 is connected to the IRQ line defined during installation. The EOC monitor header, P6, should be set to PA7. The EOC interrupt header, P7, must be configured so that EOC is not connected to any interrupt. The A/D converter voltage range, P9, should be set to 10 V. Finally, the D/A converter output voltage range, P10, should have AOUT1 and AOUT2 both connected to ±.

A.1.2 Installing the ADA2100

In this section we do not give step-by-step instructions for removing the cover and installing a card. We assume that the reader is familiar enough with an IBM PC to be able to perform these tasks. If not, your system manual should give a good explanation of how to install an I/O card. In this section we give some indications of things to look out for. It is important to make sure that no other cards conflict with the settings on the ADA2100. The first of these is the Base I/O address. Make sure that no other card in the system is configured to use the addresses in the range Base I/O Address to Base I/O Address +0x17. If, for example, the Base I/O Address of the ADA2100 is 0x240, then there should be no other cards using addresses between 0x240 and 0x257. These addresses must be dedicated to the ADA2100. Also make sure no other card is using the IRQ line set on P5, the PIT Interrupt Header.

A.1.3 Installing ADA2100 power-up initialization

The AUTOEXEC.BAT file must be modified to execute DACINIT.COM. This can be done with almost any text editor. DACINIT.COM will set the 82C54 timer on the ADA2100 to a known state. The 82C54 timer does not automatically reset to a known state on power up. This makes it possible for the 82C54 to power up in a mode that will interrupt the system if the system does not disable interrupts. The system boot should disable this interrupt and set the interrupt vector to a "dummy" interrupt. If for some reason the system does not perform these operations, problems could occur. By running DACINIT.COM the 82C54 will not cause an interrupt. Also, DigiScope will disable the 8254 on exit. If DigiScope is not allowed to exit properly (Control Break is used), then the timer may still be running and causing interrupts to occur.

A.1.4 Connecting to the ADA2100

Figure A.3 shows the connection points for the inputs and outputs. The "–" signal pins are the ground lines. All the ground lines are the same on the ADA2100.

Signal	+ Signal pins	– Signal pins
Analog input channels 1 to 8	1 to 8	21 to 28
Analog output channel 1, 2	10, 11	30, 31
Digital inputs 0 to 3	36, 16, 35, 15	37
Digital outputs 0 to 3	34, 14, 33, 13	37

Figure A.3 Signal connections for the ADA2100.

Thus, if several channels share the same ground, only one ground lead need be connected to the ADA2100. The analog inputs are configured as eight single ended ± polarity channels. Do not exceed the input voltage range of ±5 V. Digital signals are to/from an 82C55 directly. Do not exceed the 0 to +5 V level of the 82C55. Also, a CMOS device like the 82C55 cannot drive a large amount of current. We recommend using a noninverting buffer such as the 74LS244 to protect the 82C55. See Appendix C of the ADA2100 User's Manual for more information on the inputs and outputs of the 82C55.

A.2 CONFIGURING THE MOTOROLA 68HC11EVBU

The Motorola EVBU Student Design Kit was chosen as an inexpensive data acquisition and control unit. This device was chosen because of its availability and low cost. At the time of this printing, Motorola is selling this kit for $68.11. This price is not likely to change. The EVBU is designed to emulate a Motorola 68HC11. The processor used in the EVBU has 8 channels of 8-bit A/D, an RS232 Serial Communications Interface, and 24 bits of digital I/O. As implemented in DigiScope, the EVBU has 4 channels of A/D (0 to +5 V). The capability exists for 8 digital inputs and 8 digital outputs (0 to +5 V).

Connecting the EVBU to a host computer is straightforward. All that is required is a cable and a +5-V dc power supply. The +5-V supply can be replaced by a battery. Instructions for using a battery to power the EVBU are included with the kit.

The EVBU does not come with a cable, but a good explanation of how to make a cable to connect the EVBU to the IBM PC is included with the kit. We offer a different cable design. Since our software does not require hardware handshaking, a cable with only three wires and some jumpers can be constructed. The jumpers are to ensure proper operation of PC serial communication cards which expect hardware handshaking. Figure A.4 shows cable designs for both 25-pin and 9-pin PC connectors.

A.2.1 Configuration of the EVBU

The initial configuration is shown in Figure A.5. Some minor changes will be made later after some setup of the program memory in the EVBU. All of these settings are defaults (or can be) except for J2 and J4. J2 need not be removed but a jumper is needed across J4 to put the 68HC11 into bootstrap mode. J2 is used by the BUFFALO monitor to determine if the monitor should be executed or if the processor should jump to the EEPROM at location 0xB600. Simply remove J2 and place it across J4.

Where the words installed or removed are used, there is a physical jumper that must be handled. Where the words shorted or opened are used, there is no physical jumper supplied but there may be a cut-trace short located on the printed circuit

board solder side (bottom). It is not necessary to cut any traces on the circuit board. In fact, the only jumpers that need be changed are J2 and J4.

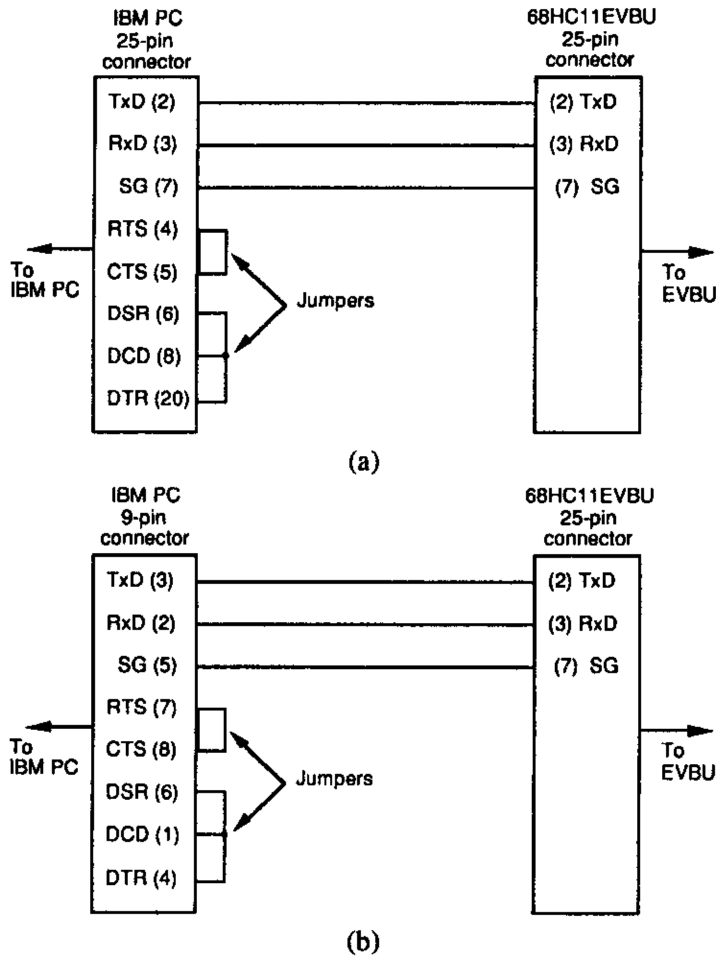

Figure A.4 Cable designs for connecting a PC to the 68HC11EVBU board. (a) 25-pin PC connector. (b) 9-pin PC connector.

Jumper	Function	Required settings
J1	Input power select header	Set as required for supply type
J2	Program execution select	Must be moved to J3
J3,J4	MCU mode select header	Must both be installed
J5,J6	MCU clock reconfiguration	Must both be open
J7	Trace enable header	Can be installed or removed
J8,J9	SCI reconfiguration	Both shorted
J10,11,12,13	SPI reconfiguration headers	Can be installed or removed
J14	Real time clock INT* header	Must be open
J15	TxD reconfiguration header	Must be shorted
J5	Terminal baud rate select	Across pins 11 and 12
J6	Host port Rx signal disable	Can be installed or removed

Figure A.5 Motorola EVBU jumper configurations

A.2.2 Connecting the IBM PC to the EVBU

You will need to provide a cable as described in the Motorola EVBU User's Manual. The Digiscope program checks the file `CONFIG.WDS` to determine the serial port used to communicate with the EVBU. The default is serial port configured as COM1. You should run the `ADINSTAL.EXE` program to create `CONFIG.WDS` if you have not done so already.

A.2.3 Installing EDAC 68HC11 program into the EVBU

The files `EDAC.S19` and `EDAC.ASM` are on the disk that you received with this textbook. `EDAC.ASM` is the source code for the program that will reside on the EVBU and communicate with DigiScope. `EDAC.S19` contains the hex code for `EDAC.ASM`. `EDAC.S19` is in Motorola S-record format. A good description of the S-record file format can be found in Appendix A of the EVBU manual.

Included with the student project kit is a software development utility called `pcBug11`. `pcBug11` can be used to develop software for the 68HC11. `pcBug11` can be used to download programs from a PC to the 68HC11 RAM, EPROM or EEPROM using the 68HC11 bootstrap mode. A macro called `LOAD.MCR` for `pcBug11` has been provided with DigiScope to allow simple programming of the 68HC11 EEPROM with `EDAC.S19`. After loading `EDAC.S19` into the EEPROM of the 68HC11, it is necessary to use `pcBug11` to start the `EDAC` program running on the 68HC11.

If the above is not workable, then the monitor may be programmed into the EPROM using a 12 V power supply and a 100 Ω resistor. With the monitor programmed into the EPROM, J2 may be configured such that the processor will start executing the code in EEPROM after reset. This means that the reset switch need only be pushed instead of rerunning `pcBug11`.

> *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.

A.2.4 Connecting signals to the EVBU

Figure A.6 shows the pin numbers for connecting analog and digital signals to the Motorola EVBU. Note that analog input channels 1 to 4 are connected to port E bits 4 through 7.

Signal	+ Signal pins	– Signal pins
Analog input channels 1 to 4	44, 46, 48, 50	1
Digital inputs 0 to 7	9-16	1
Digital outputs 0 to 7	42, 41, 40, 39, 38, 37, 36, 35	1

Figure A.6 Signal connections for the Motorola EVBU.

Although it is not absolutely necessary, we recommend to those using an EVBU for data acquisition and control to add some protection to the inputs and outputs. CMOS devices such as the 68HC11 are very sensitive to voltages outside the range of 0 to +5 V dc and are not able to sink or source a large amount of current.

Protecting analog inputs

Figure A.7 gives an example of a simple protection circuit for analog inputs. To keep the input voltage from rising above +5 V or dropping below 0 V, we have

suggested clamping diodes. Point A will not go below –0.7 V or above +5.7 V. This is enough to protect the 68HC11. Between the clamping diodes and the input on the IC, a current limiting resistor is used. A value of 10 kΩ will limit the current without changing the input impedance too much. Finally, a 0.1 μF ceramic capacitor is placed from the input to ground. This capacitor will help dampen the effects of any high-frequency noise that may be present.

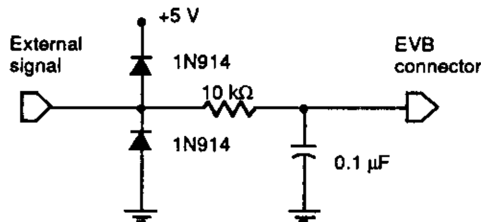

Figure A.7 Example of a simple protection circuit for a Motorola EVBU analog input.

Protecting digital inputs and outputs

Digital inputs and outputs are much easier to protect. A noninverting buffer can be placed between the outside world and the microcontroller. If an input or output is subjected to a damaging voltage, the buffer will be damaged and not the microcontroller. Also, a buffer made of Low Power Schottky TTL (LSTTL) technology is more resistant to out-of-range voltages. A 74LS244 works very well as a buffer. There are 8 buffers in a single package. Only one 74LS244 is required for each of the input and output banks.

A.3 CONFIGURING THE MOTOROLA 68HC11EVB

Setting up the Motorola EVB Evaluation Kit is much the same as setting up the system with the EVBU Student Project Kit. The EVB is also designed to emulate a Motorola 68HC11.

There are different jumpers to configure on the EVB since this is a different circuit board. The software setup for the EVB is similar to the software setup for the EVBU. The major difference is that a monitor program is already programmed into an external EPROM on the EVB. There is no need to program the internal EPROM of the 68HC11. The monitor is then used to program the EEPROM.

A.3.1 Setting up the EVB

There are two steps to configuring the EVB. The first is setting jumpers and building and connecting the RS232 cable. The second step is to connect the power supplies to the EVB. The EVB requires +5, +12, and -12 V supplies. The jumpers on the EVB should be configured as shown in Figure A.8. All of these settings are defaults (or can be). J4 will need to be changed to EEPROM after EEPROM has been programmed with `EDAC.S19`. This jumper is used by the `BUFFALO` monitor to determine if the monitor should be executed or if the processor should jump to the EEPROM at location 0xB600. Initially the jumper should be in position TBD to indicate that the program in the EPROM should be executed.

Jumper	Function	Required settings
J1	Reset select header	Can be installed or removed
J2	Clock select header	Across pins 2 and 3
J3	RAM select header	Can be installed or removed
J4	Program execution select	Must set to EPROM initially
J5	Terminal baud rate select	Across pins 11 and 12
J6	Host port Rx signal disable	Can be installed or removed

Figure A.8 Motorola EVB jumper configurations.

A.3.2 Installing EDAC 68HC11 program into the EVB

The files `EDAC.S19` and `EDAC.ASM` are on the disk you received with this textbook. `EDAC.ASM` is the source code for the program that will reside on the EVBU and communicate with Digiscope. `EDAC.S19` contains the hex code for `EDAC.ASM`. `EDAC.S19` is in Motorola S-record format, A good description of the S-record file format can be found in Appendix A of the EVBU manual.

Included with the student project kit is a software development utility called **pcBug11**. **pcBug11** can be used to develop software for the 68HC11. **pcBug11** can be used to download programs from a PC to the 68HC11 RAM or EPROM using the 68HC11 bootstrap mode. A macro called **LOAD.MCR** for **pcBug11** has been provided with Digiscope to allow simple programming of the 68HC11 EEPROM with **EDAC.S19**. After loading **EDAC.S19** into the EEPROM of the 68HC11 it is necessary to use `pcbug11` to start the `EDAC` program running on the 68HC11.

A.3.3 Connecting the IBM PC to the EVB

You will need to provide a cable as described in the Motorola EVB User's Manual. The DigiScope program checks the file `CONFIG.WDS` to determine the serial port used to communicate with the EVB. The default is serial port configured as COM1. You should run the `ADINSTAL.EXE` program to create `CONFIG.WDS` if you have not done so already.

A.3.4 Connecting signals to the EVB

See section A.2.4 for a description of how to connect signals to the EVB. The same basic principles apply for protecting analog inputs and digital output and digital inputs. Figure A.9 shows the pin numbers for connecting analog and digital signals to the Motorola EVB.

Signal	+ Signal pins	– Signal pins
Analog input channels 1 to 8	43, 45, 47, 49, 44, 46, 48, 50	1
Digital inputs 0 to 7	9-16	1
Digital outputs 0 to 7	42, 41, 40, 39, 38, 37, 36, 35	1

Figure A.9 Signal connections for the Motorola EVB.

A.4 VIRTUAL INPUT/OUTPUT DEVICE (DATA FILES)

Although it may seem a bit strange to talk about hardware configuration for a virtual device, this section gives you some indication of how a PC should be set up to run a virtual I/O device. The minimum requirements are a PC with 640 kbytes of memory, VGA or Hercules monochrome graphics, and a single floppy drive. For best performance, we strongly recommend using the program with a hard disk drive. If DigiScope must read from a floppy disk every time it reads a sample point then "real-time data acquisition" will not look very real.

A.5 PUTTING A HEADER ON A BINARY FILE: ADDHEAD

Files created by DigiScope have a unique structure. You can view information in a file header with the UW DigiScope `stat (U) s` command. A special program is also provided for creating file headers.

If you have a data file that contains 16-bit integers in strictly binary format (e.g., C-language 16-bit integers), you can use ADDHEAD to put a header on the file. ADDHEAD will prompt the user for the information to be placed in the header of the file. If the user does not enter any information for a field, ADDHEAD will use the default. The defaults for each field are printed with the prompt in parenthesis. After all information has been entered, ADDHEAD reads the binary file and writes the header and data to a new file called `filename.dat`.

A.6 REFERENCES

M68HC11EVBU Universal Evaluation Board User's Manual. 1986. Motorola Literature Distribution, P.O. Box 20912, Phoenix, AZ 85036.

ADA2100 User's Manual. 1990. Real Time Devices, Inc, 820 North University Dr., P.O. Box 906, State College, PA 16804, 1990.

M68HC11 User's Manual. 1990. Motorola Literature Distribution, P.O. Box 20912, Phoenix, AZ 85036.

Appendix B

Data Acquisition and Control Routines

Danial J. Neebel

This appendix describes the routines that can be used to perform data acquisition and control provided in DACPAC.LIB. The routines in DACPAC were developed for two purposes. The first was for use in UW DigiScope. The second was to provide simple software interfaces to the signal conversion devices for use by other programs. DACPAC.LIB and the required headers are included on the floppy disk you received with this text. The DACPAC routines perform analog signal acquisition, digital signal acquisition, analog signal output, digital signal output, and timing.

The DACPAC routines provide an interface to three different devices. These three devices are an Analog and Digital I/O interface card from Real Time Devices that is installed inside the IBM PC or compatible, an external single board computer with RS232 communications (Motorola S68HC11EVBU Student Project Kit), and a virtual I/O device (data files).

There are some similarities between the three different devices. All device handling is done with four basic operations. These are OPEN, GET, PUT, and CLOSE. OPEN initializes the device. GET retrieves a piece of data from the device. PUT gives the device a piece of data to output. CLOSE terminates all operations being performed on or by the device, including closing a file, disabling interrupts, and removing communication links.

There are five sections in this appendix. In the first section we present the data structures used in DACPAC.LIB. The next section describes the top-level routines used to call the routines specific to each device. The following three sections discuss the routines provided in DACPAC. The descriptions of the routines for handling devices are divided up into subsections for each type of routine. Along with the four types above, we have added digital input and digital output as two special types of GET and PUT routines. Each subsection describes in detail what data are required by the routine and what data are set by the routine.

Before charging off into details, we give one simple warning. Always make sure to close all devices that have been opened before exiting the program. This means that the use of the C-library function `exit()` should be used with great care. A device left open after program exit could cause your PC to hang up.

B.1 DATA STRUCTURES

Here are the important data structures used by the routines described in this section. The most important is the **Header** data structure. This structure contains all the information about the data that has been or will be gathered from a device.

The title, creator, source, and type are all strictly character strings to be used as the programmer sees fit. The package routines do not use these fields. When opening any of the devices supported by DACPAC, the user must be careful to initialize some parts of the **Header** data structure passed to the OPEN routine and note also what parts of the data structure are initialized by the OPEN routine. Figure B.1 shows the data structures used by the data acquisition routines in DACPAC.

```
typedef short DATATYPE;    /* data type will be 16 bit integer */

typedef enum {ECG,EMG,EEG,CV,RESP,EKG,ABC,ERROR}  chantype_t;

typedef struct ChannelRecord {
   chantype_t   type;
   float        offset;
   float        gain;
} CHANTYPE;

typedef struct HeaderRecord {
   char    title[80];         /* title to be used for display */
   char    filename[40];      /* filename containing data */
   FILE    *fd;               /* file pointer returned by fADopen */
   char    creator[80];       /* name of person who gathered data */
   char    source[80];        /* name of A/D card or other device */
   char    type[80];          /* i.e. 12-lead ECG  */
   float   volthigh;          /* High limit of input voltage */
   float   voltlow;           /* Low limit of input voltage */
   int     stepsize;          /* step size used by data compress. */
   char    compression;       /* Data compression type */
   int     rate;              /* positive integer */
   int     resolution;        /* number between 8 and 16 */
   int     num_channels;      /* number of channels (in array) */
   int     num_samples;       /* number of samples */

   CHANTYPE channel[20];      /* pointer to array of Channel info*/
   DATATYPE    *data;         /* pointer to data buffer */
} DataHeader_t;
```

Figure B.1 Data header and channel type data structures (from **defns.h**).

An important part of the **Header** structure is the array of CHANTYPE. This array of structures contains all the information that is unique to each channel. Part of the

HANTYPE structure is an enumerated typed variable called **type**. This field is not sed by any of the DACPAC routines. The **type** field is provided for the user.

The defined type DATATYPE is more than just type short. The convention for all ariables of type DATATYPE is that they be 2's complement 16-bit integers, a value f 0 corresponds to 0 V. To calculate the voltage given from a value of DATATYPE, ultiply the value by the resolution (i.e., bits/V) of the data from device being ad. To calculate the number of bits/V, divide the voltage range by 2 raised to the ower of the resolution. The voltage range is found by subtracting **Header-voltlow** from **Header->volthigh**. The resolution is taken from **Header-resolution**.

.2 TOP-LEVEL DEVICE ROUTINES

ACPAC contains routines that call specified device handlers. These routines are cluded in DAC.C. The header file that contains the prototypes for the routines is AC.H. The prototypes are shown in Figure B.2. Examples of the calling onventions for each type of device are given in Figure B.3. For determining what alues to send these routines, look at the section corresponding to the device you e using. The top-level routine PUT() calls **fADput_buffer()** with a size of one.

```
char OPEN(DataHeader_t *Header, char *dir, int device);
char CLOSE(DataHeader_t *Header, int device);
char GET(DataHeader_t *Header, DATATYPE *data, int device);
char PUT(DataHeader_t *Header, DATATYPE *data,int channel, int
device);
```

Figure B.2 Prototypes for the top level routines (from DAC.H).

.3 INTERNAL I/O DEVICE (RTD ADA2100)

he RTD (Real Time Devices) ADA2100 can perform more I/O functions than the ther two devices described in this appendix. Using the routines below, the DA2100 is capable of reading eight single-ended analog inputs, driving two nalog outputs ranging from –10 to +10 V, reading four digital inputs, and driving ur digital outputs. The digital I/O uses standard TTL levels ranging from 0 to 5 V.

```
#define VIRTUAL    1
#define EXTERNAL   2
#define INTERNAL   3

/* calling conventions for Internal device */

OPEN(&Header, "", INTERNAL);
CLOSE(&Header, INTERNAL);
GET(&Header, &data, INTERNAL);

/* calling conventions for External device */

OPEN(&Header, "r", EXTERNAL);
CLOSE(&Header, EXTERNAL);
GET(&Header, &data, EXTERNAL);

/* calling conventions for Virtual device */

OPEN(&Header, "rt", VIRTUAL);
CLOSE(&Header, VIRTUAL);
GET(&Header, &data, VIRTUAL);
```

Figure B.3 Calling conventions for the three devices.

See Appendix A for a description of how to configure the ADA2100. So items such as configuring for digital input and output are done via the software DACPAC. Figure B.4 shows the routines provided to perform the above functic and the calling convention for each. The prototypes for all the routines shown the figure are in IDAC.H. All routines return a value of 1 if the operation successful and 0 if the operation is unsuccessful. Some routines will always successful and will always return a value of 1.

```
char Iopen(DataHeader_t *Header);
char Iclose(DataHeader_t *Header);
char Iget(DATATYPE *data);
```

Figure B.4 Internal card calling conventions (from IDAC.H).

B.3.1 Opening the device: Iopen()

The `Iopen()` routine initializes the ADA2100 to perform digital input and out and analog input at the given sample rate. The timer is set to provide interrupts the given sample rate. The interrupt vector is set to point to a small routine that s a flag. The flag is checked by `Iget()`.

Figure B.5 shows the how the `Header` data structure is used and what eleme of the structure must be initialized and what elements are initialized by `Iopen`

Any elements not listed are not initialized or used by `Iopen()` and so can be used at the discretion of the calling routine.

The following items must be initialized before the device is opened.

```
Header->rate
Header->num_channels
```

The following items are initialized by the Iopen(); routine.

```
Header->volthigh = 5.0;
Header->voltlow = -5.0;

Header->resolution = 12;
if (channels > 7) channels = 7; /* should fix this sometime */
Header->num_channels = channels;
Header->num_samples = 0;

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

Figure B.5 `Iopen()` routine description.

B.3.2 Closing the device: Iclose()

The `Iclose()` routine stops the timer on the ADA2100 and resets the interrupt vector to the value set before `Iopen()` was called. `Iclose()` does not modify nor does it require any elements of the `Header` data structure. `Header` is passed into `Iclose()` only for commonality.

B.3.3 Taking an analog input reading: Iget()

The `Iget()` routine checks to see if the flag has been set by the timer interrupt routine. If the flag has been set, `Iget()` reads the channels requested by `Iopen()`, puts the data from those channels in an array of `DATATYPE` pointed to by data and returns a 1. If the flag has not been set , `Iget()` returns a 0. If a rate of 0 is selected, then `Iget()` will always read a value into data and return a 1.

B.4 EXTERNAL I/O DEVICE (MOTOROLA 68HC11EVBU)

The routines that interface to either of the Motorola devices do so by performing serial communications on an RS232 link. Both the EVBU and EVB use the same

interface. The routines provided in DACPAC are given in Figure B.6. These are similar to the routines provided for the ADA2100. There are two major differences.

Although it is not absolutely necessary, we recommend to those using an EVBU for data acquisition and control that some protection be placed on the inputs and outputs. See section A.2.4 for some examples of simple protection circuits.

```
char Eopen(DataHeader_t *Header, char *dir);
char Eclose(DataHeader_t *Header);
char Eget(DATATYPE *data);
```

Figure B.6 Edac calling conventions (from EDAC.H).

B.4.1 Opening the device: Eopen()

The `Eopen()` routine initializes the serial port and sends a soft reset command followed by setup commands to the EVBU. These setup commands tell the software running on the EVBU at what rate to sample the analog inputs and which inputs are to be sampled. Since the EVBU can only communicate at a maximum of 9600 bits/s, it is necessary in some cases to buffer the data and send the block to the PC when the PC has time to read the data. So a last item included in these commands is whether real-time transfer or block transfer is to be used. To set the EVBU for real-time transfer, call `Eopen()` with `dir` set to `r`. To set the EVBU for block transfers, call `Eopen()` with `dir` set to `b`.

Figure B.7 shows how the `Header` data structure is used, what elements of the structure must be initialized, and what elements are initialized by `Eopen()`.

The following items must be initialized before the device is opened.

```
Header->rate
Header->num_channels
```

The following items are initialized by the Eopen(); routine.

```
Header->volthigh = 5.0;
Header->voltlow = -5.0;

Header->resolution = 8;
if (channels > 8) channels = 8;
Header->num_channels = channels;
Header->num_samples = 0;

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

Figure B.7 Eopen routine description.

Any elements not listed are not initialized or used by `Eopen()` and so can be used at the discretion of the calling routine.

If you do not wish to perform analog input, simply give `Eopen()` a sample rate of 0. If this is done, no analog inputs will be read. In this case `Eget()` should not be used. `Eopen()` assumes the caller does not want to perform analog input. Digital input and output may still be performed.

B.4.2 Closing the device: Eclose()

The `Eclose()` routine will stop all analog signal acquisition on the EVBU. The EVBU will be given a soft reset command. This means that the EVBU will stop reading analog inputs and performing digital I/O until `Eopen()` is called again and the EVBU receives setup information again.

B.4.3 Taking an analog input reading: Eget()

The `Eget()` routine checks to see if data has arrived from the EVBU. If data has arrived without error, then the `Eget()` puts the data from the channels in `DATATYPE` variable pointed to by data and returns a 1. If no data has arrived from the EVBU, `Eget()` returns a 0.

B.5 VIRTUAL I/O DEVICE (DATA FILES)

Figure B.8 shows the calling conventions for each of the routines available to the user. These routines were developed to serve two purposes. The first was to allow anyone to do real-time data acquisition even if they do not have any data acquisition hardware. The second purpose was to allow the storage, labeling, and retrieval of signals gathered using the above data acquisition routines. The routines described here do not require any special hardware. If you have an IBM PC/XT/AT or compatible with a floppy drive, 640 kbytes of RAM, and either a Hercules monochrome or VGA color monitor, you should be able to run these routines.

```
char fADopen(DataHeader_t *Header, char *dir);
char fADclose(DataHeader_t *Header);
int fADget_buffer(DataHeader_t *Header,int size,DATATYPE *buffer);
char fADget(DataHeader_t *Header, DATATYPE *signal);
int fADput_buffer(DataHeader_t *Header,int size,DATATYPE *buffer);
void fADputfile(DataHeader_t *Header);
void fADgetfile(DataHeader_t *Header);
```

Figure B.8 `fAD` calling conventions (from `fAD.H`).

B.5.1 File structure

The file structure uses an ASCII text header to describe the data in the file. The header is terminated by a single blank line. The data immediately follow this blank line. The data are stored in the file in binary. Figure B.9 gives an example `Header`. Note that for `fADopen()` to operate properly, the field names must be exactly as shown in Figure B.9. Even though the headers are stored in text format, the files cannot be edited using a normal text editor since the data are stored in the files in binary. Section A.5 shows how to use the program called ADDHEAD to add a header to an existing file.

```
Title:  ECG data from file ecg105
Creator:  unknown
Source:  unknown
Type:  ECG single channel
Volthigh:  12
Voltlow:  -12
Step:  0
Compress: N
Resolution:  12
Rate:  200
Channels: 1
Samples: 12000
Chan:  0   Gain   1.0000  Ofst   0.0000   Type ECG
```

Figure B.9 Example file header.

B.5.2 Opening the device: fADopen()

The `fADopen()` routine opens a file and initializes the timer to provide software delays to simulate real-time data acquisition. If you do not wish to perform timed analog input, simply call `fADopen()` with `dir` set to `r` for read only. If timed input is desired, then call `fADopen()` with `dir` set to `rt` for read with timed input. If file output is desired, then call `fADopen()` with `dir` set to `w` for write only. When opening a file for output, *all* elements of the `Header` structure must be set. When opening a file for read only `Header->filename` need be set. As shown in Figure B.10, all other elements will be read in from the file. Function `fADopen()` will only open a file for reading (options `r` and `rt`) or writing (option `w`). This figure shows how the `Header` data structure is used and what elements of the structure must be initialized and what elements are initialized by `fADopen()`. Any elements not listed are not initialized or used by `fADopen()` and so can be used at the discretion of the calling routine.

The following items must be initialized before the device is opened.

```
Header->filename;
```

The following items are initialized by the Iopen(); routine.

```
Header->title;
Header->creator;
Header->source;
Header->type;
Header->fd;
Header->stepsize;
Header->compression;
Header->rate;
Header->resolution;
Header->num_samples;
Header->volthigh;
Header->voltlow;
Header->compression;      /* no compression */
Header->resolution;
Header->channels = channels;
Header->num_samples = 0;

for (i=0;i<Header->num_channels;i++) {
   ChanOffset(Header,i);
   ChanGain(Header,i);
   ChanType(Header,i);
```

Figure B.10 `fADopen` routine description.

B.5.3 Closing the device: fADclose()

The `fADclose()` routine performs two very important operations. First, the obvious, it closes the file pointed to by `Header->fd`. Second, if `fADopen()` was called with `dir` pointing to `t`, `fADclose()` will remove the timer routine from the time-of-day interrupt line.

B.5.4 Taking an analog input reading: fADget(), fADget_buffer(), fADgetfile()

The `fADget()` routine checks the flag set by the interrupt service routine. If the flag has been set, then `fADget()` reads one sample from each of the channels in the file pointed to by `Header->fd` and returns a 1. The samples are then placed in an array pointed to by data. It is the responsibility of the calling routine to allocate enough space for the data. If the flag has not been set by the interrupt service routine, then `fADget()` returns a 0.

If no timing is required and all of the data needs to read in for processing, then `fADgetfile()` can be used. `fADgetfile()` will open the file specified by `Header-`

`>filename`, read the file header into the `Header` data structure, and read all the data into an array pointed to by `Header->data`. Note also that `fADgetfile()` will allocate memory for the data pointed to by `Header->data`.

If the file is too large to read all at once or only some of the data is needed from the file, then `fADget_buffer()` is very useful. `fADget_buffer()` will read as many samples as are requested by size. If successful, `fADget_buffer()` returns the number of samples read from the file. A sample is one reading on each channel.

B.5.5 Writing data to a file: fADput_buffer() and fADputfile()

For writing data out to a file, there are two routines—`fADput_buffer()`, and `fADputfile()`. The routines used for writing to a file do not use timing. There is no `fADput()` to correspond with `fADget()` since calling `fADput_buffer()` with a buffer size of 1 will write one sample of each channel into the file pointed to by `Header->fd` just as one would expect a routine called `fADput()` to do. Remember that a sample is one reading for each channel. The top-level routine `PUT()` calls `fADput_buffer()` with a size of one.

`fADputfile()` makes storing the data to a file simple. `fADputfile()` will open the file name `Header->filename` for writing, write all header information to the file, and write all data to the file and then close the file. `fADputfile()` does not free the memory pointed to by `Header->data`.

B.6 REFERENCES

ADA2100 User's Manual. 1990. Real Time Devices, Inc, 820 North University Dr., P.O. Box 906, State College, PA 16804. Ph: (814) 234-8087; Fax: (814) 234-5218.

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.

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.

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.

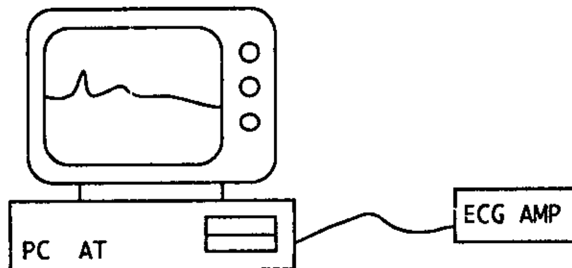

Figure C.1 Internal card data acquisition and control system.

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.

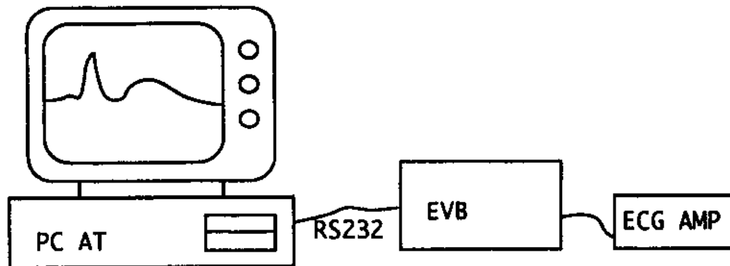

Figure C.3 Motorola EVBU data acquisition and control system.

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.

nput data

\n input data routine must give the EVBU a request to read data and wait for a re-ponse. Remember that the host should always limit the amount of time spent wait-ng for a response so as not to hang the computer waiting for an event that will ever occur.

)utput data

\n output data routine must send the EVBU a command telling the EVBU to out-ut a specified piece of data.

:.2.2 Serial communications using Turbo C

ierial communication is accomplished via the **bioscom()** routine. Function **ioscom()** is a multipurpose routine that can be used to initialize the serial port, heck status, read the serial port, send a byte out on the serial port. The Turbo C eference manual gives some very helpful examples of how to perform each of hese operations. Figure C.6 shows some examples from the code developed for his 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.

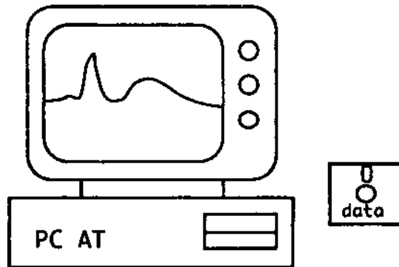

Figure C.8 A microcomputer and hardware for virtual analog input.

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 these 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.

Appendix D

UW DigiScope User's Manual

Willis J. Tompkins and Annie Foong

UW DigiScope is a program that gives the user a range of basic functions typical of a digital oscilloscope. Included are such features as data acquisition and storage, sensitivity adjustment controls, and measurement of waveforms. More important, this program is also a digital signal processing package with a comprehensive set of built-in functions that include the FFT and filter design tools. For filter design, pole-zero plots assist the user in the design process. A set of special advanced functions is also included for QRS detection, signal compression, and waveform generation. Here we concentrate on acquainting you with the general functions of UW DigiScope and its basic commands. Before you can use UW DigiScope, you need to install it on your hard disk drive using the `INSTALL` program. See the directions on the DigiScope floppy disk. Also be sure to read the `README.DOC` file on the disk for additional information about DigiScope that is not included in this book.

D.1 GETTING AROUND IN UW DIGISCOPE

To run the program, go to the `DIGSCOPE` directory and type `SCOPE`. Throughout this appendix, the words `SCOPE`, DigiScope, and UW DigiScope are used interchangeably.

D.1.1 Main Display Screen

Figure D.1 shows the main display screen of `SCOPE`. There is a menu on the left of the screen and a command line window at the bottom. There are two display channels. The one with the dashed box around it is called the active channel, which can be selected with the `(A)ctive Ch` menu command. Operations generally manipulate the data in the active channel. The screen shows an ECG read from a disk file displayed in the top channel and the results of processing the ECG with a derivative algorithm in the bottom (i.e., active) channel.

Maneuvering in **SCOPE** is accomplished through the use of menus. The **UP** and **DOWN ARROWS** move the selection box up and down a menu list. Hitting the **RETURN** key selects the menu function chosen by the selection box. Alternately, a menu item can be selected by striking the key indicated in parenthesis for each command (e.g., key **F** immediately executes the `(F)ilters` command).

An `e(X)it` from the current menu to its parent (i.e., the previous menu) can be achieved either by placing the box on the `e(X)it` item and striking the **RETURN** key, by hitting the **X** key, or by simply hitting the **ESC** key. In fact the **ESC** key is used throughout **SCOPE** to exit from the current action, and the **RETURN** key is used to execute a selected function.

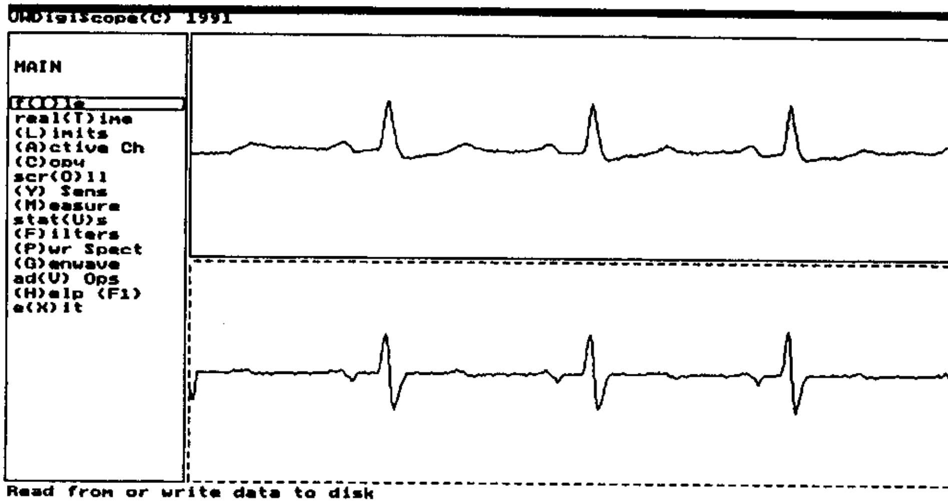

Figure D.1 UW DigiScope's main display screen. The dashed box indicates the active channel. In this case, the ECG in the top channel was read from a disk file, then a derivative algorithm was applied to processed the ECG and produce the waveform in the bottom (active) channel.

D.1.2 Communicating with SCOPE

Sometimes it is necessary for the user to enter data via the keyboard. Such data entry is done in the command line window at the bottom of the screen. Entry of data either ends with a **RETURN**, upon which the data are accepted, or an **ESC** which allows the user to quit data entry and make an escape back to the previous screen. Correction can be done with the **BACKSPACE** key prior to hitting **RETURN**. **SCOPE** also provides information to the user via short text displays in this window.

D.2 OVERVIEW OF FUNCTIONS

Figure D.2 shows how main menu functions branch to other menus. Command `f(I)le` permits reading or writing disk data files. Function `real (T)ime` lets you select the source of sampled data to be either from a disk file or, if the computer has the proper hardware installed, from an external Motorola 68HC11 microcontroller card or an internal Real Time Devices signal conversion card.

Function `(L)imits` lets you choose whether SCOPE functions operate on the whole file or just the portion of the file that is displayed. The default limits at start-up are the 512 data points from the file seen on the display. The maximal file size is 5,120 sampled data points. When you write a file to disk using the `f(I)le (W)rite` command, you are asked if you want to write only the data on the display (512 points) or the whole file. With the `scr(O)ll` function, you can scroll through a file using the arrow keys and select which of the file's 512 data points appear on the display. `(C)opy` performs a copy of one display channel to the other.

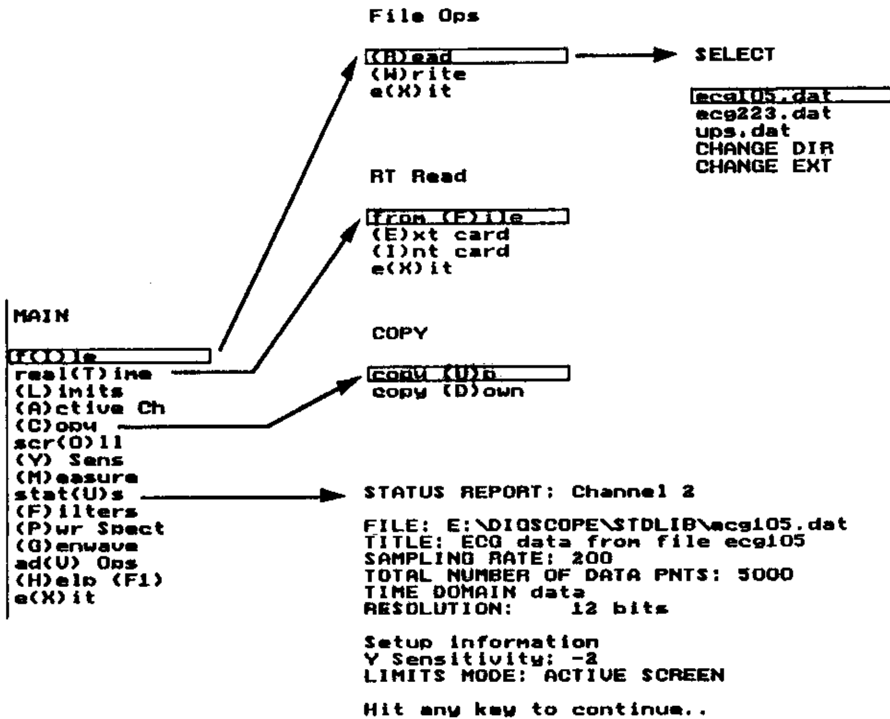

Figure D.2 Branches to submenus from the main menu.

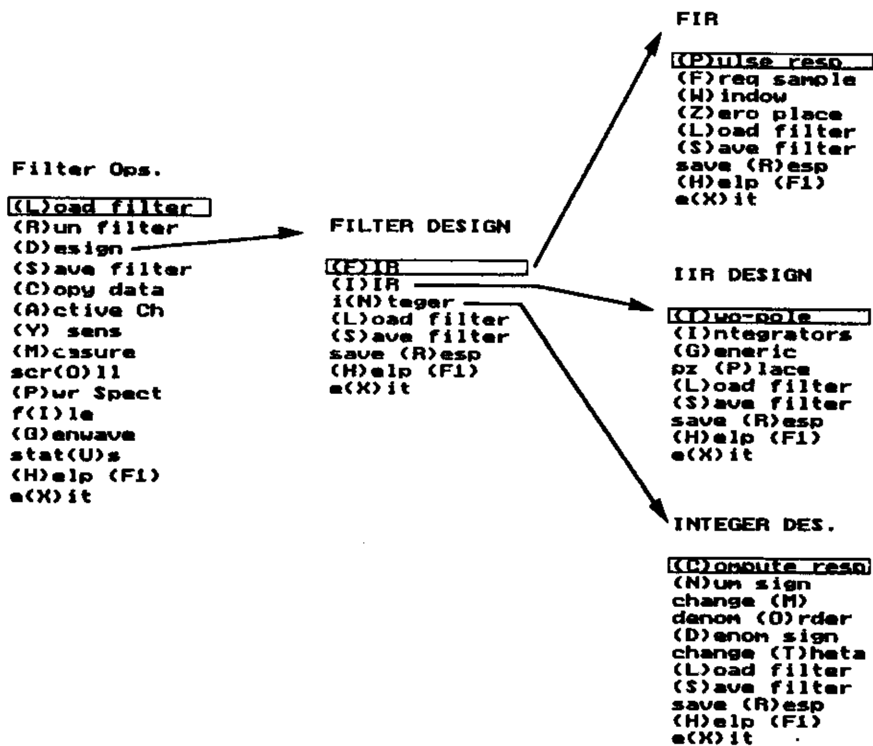

Figure D.3 Filter design menus.

To adjust the amplitude of the active channel, select `(Y) Sens` and increase or decrease the sensitivity of the channel by a factor of two each time you strike the up or down arrow on the keyboard. This function operates like the sensitivity control on an oscilloscope. `(M)easure` superimposes two cursors on the waveform in the active channel that you can move with the arrow keys. At the bottom of the display, a window shows the time and amplitude values of the cursors.

Command `stat(U)s` provides a summary screen of information about the characteristics of the current data display. This information is recorded in the header of a data file when it is written to disk.

Function `(P)wr Spect` computes and displays the power spectrum of a signal. A `(H)elp` function briefly explains each of the commands. Selecting to `e(X)it` from

this main menu returns you to DOS. Choosing e(X)it on any submenu returns you to the previously displayed menu.

Figure D.3 shows menus for performing filter design that branch from the (F)ilters command in the main menu. Any filter designed with these tools can be saved in a disk file and used to process signals. Tools are provided for designing the three filter classes, FIR, IIR, and integer-coefficient filters. (F)IR and (I)IR each provide four design techniques, and I(N)teger fully supports development of this special class of filter. The most recently designed filter is saved in memory so that selecting (R)un filter executes the filter process on the waveform in the active window. (L)oad filter loads and runs a previously designed filter that was saved on disk.

Figure D.4 shows the filter design window for a two-pole IIR filter. In this case, we first selected bandpass filter from a submenu and specified the radius and angle for placement of the poles. The SCOPE program then displayed the pole-zero plot, response to a unit impulse, magnitude and phase responses, and the difference equation for implementing the filter (not shown). By choosing e(X)it or by hitting ESC, we can go back to the previous screen (see Figure D.3) and immediately execute (R)un filter to see the effect of this filter on a signal.

Note that the magnitude response is adjusted to 0 dB, and the gain of the filter is reported. High-gain filters or cascades of several filters (e.g., running the same filter more than once or a sequence of filters on the same signal data) may cause integer overflows of the signal data. These usually appear as discontinuities in the output waveforms and are due to the fact that the internal representation of signal data is 16-bit integers (i.e., values of approximately ±32,000). Thus, for example, if you have a 12-bit data file (i.e., values of approximately ±2,000) and you pass these data through a filter or cascade of filters with an overall gain of 40 dB (i.e., an amplitude scaling by a factor of 100), you produce numbers in the range of ±200,000. This will cause an arithmetic overflow of the 16-bit representation and will give an erroneous output waveform. To prevent this problem, run the special all-pass filter called atten40.fil that does 40-dB attenuation *before* you pass the signal through a high-gain filter. The disadvantage of this operation is that signal bit-resolution is sacrificed since the original signal data points will be divided by a factor of 100 by this operation bringing a range of ±2,000 to ±20 and discarding the least-significant bits.

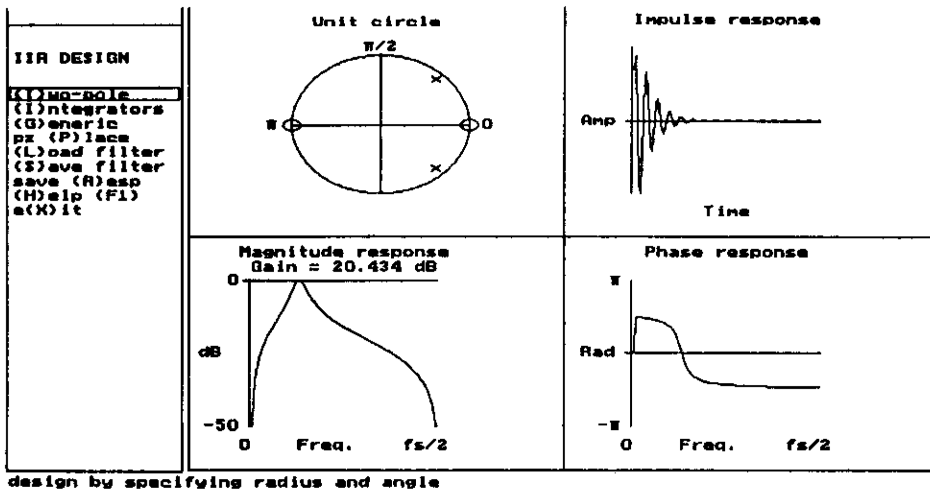

Figure D.4 UW DigiScope screen image.

Figure D.5 shows the set of special functions called from the main menu with the advanced options selection `ad(V) Ops`. These advanced features are frequency analysis, crosscorrelation, QRS detection, data compression, and sampling theory.

`(F)req anal` does power spectral analysis of a waveform segment (called a template) selected by the use of two movable cursors. This utility illustrates the effects of zero-padding and/or windowing of data. A template is selected with the cursors, and zero-padded outside of the cursors. Any dc-bias is removed from the zero-padded result. A window can be applied to the existing template. The window has a value of unity at the center of the template and tapers to zero at the template edges according to the chosen window. Function `re(S)tore` recopies the original buffer into the template.

`(C)orrelation` crosscorrelates a template selected from the top channel with a signal. A template selected from the upper channel is crosscorrelated with the signal on the upper channel, leaving the result in the lower channel. If you do not read in a new file after selecting the template, then the template is crosscorrelated with the file from which it came (an autocorrelation of sorts). After a template has been selected, you can read in a new file to perform true crosscorrelation. The output is centered around the selected template.

`(Q)RS detect` permits inspection of the time-varying internal filter outputs in the QRS detection algorithm described in the book. This algorithm is designed for signals sampled at 200 sps. QRS detection operates on the entire file. When it encounters the end of the data file, it resets the threshold and internal data registers of the filters and starts over, so you may observe a "standing" wave if the data file is a short one.

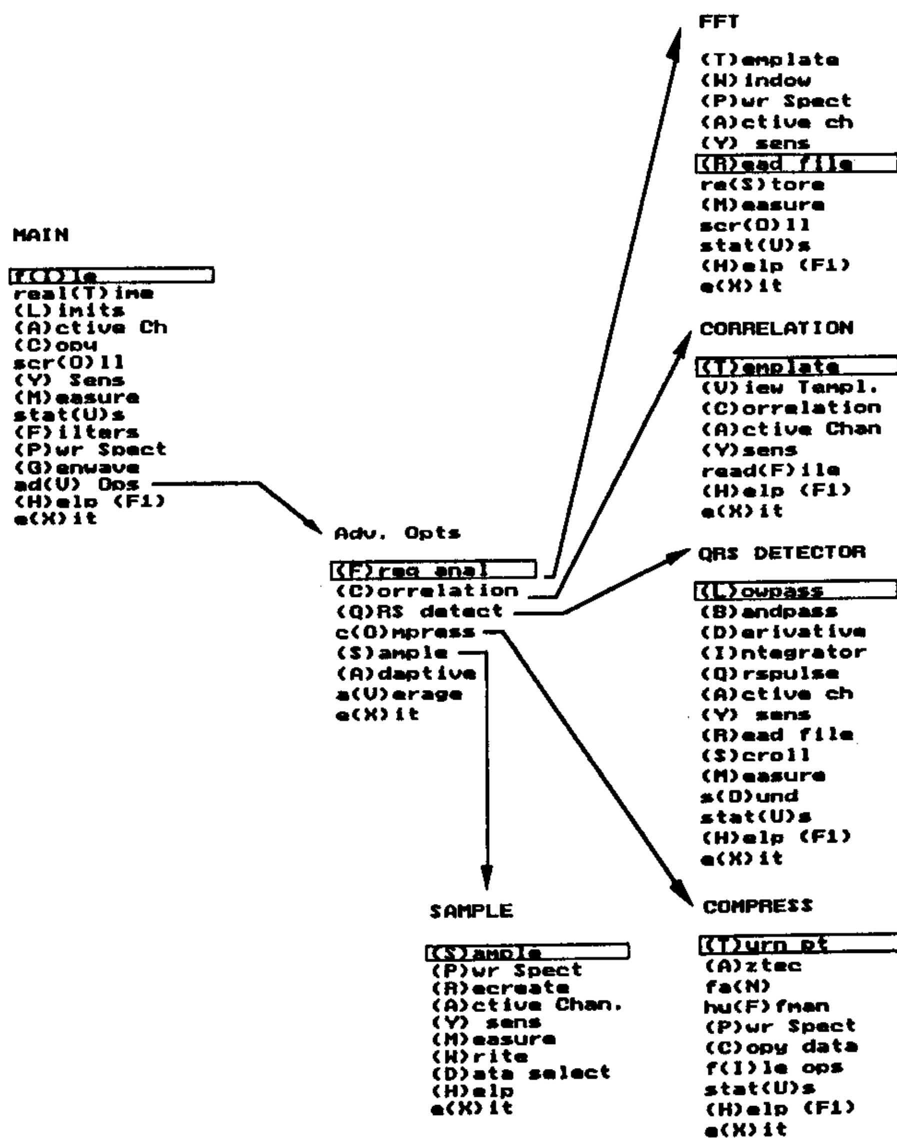

Figure D.5 Advanced options menus.

Function c(O)mpress provides the option to do data compression of a waveform using the turning point, AZTEC, Fan, and Huffman coding algorithms. The data reduction techniques compute approximations to the data in the upper channel. The algorithms operate only on the displayed data. The Turning Point algorithm reduces the number of data points by a factor of two, keeping critical points. The FAN and AZTEC algorithms require a threshold, which determines a trade-off between data reduction and distortion. The user is prompted to enter a value for the threshold, preferably a fraction of the data range (which is displayed). Huffman coding is a lossless algorithm that creates a lookup coding table based on frequency of occurrence of data values. A lookup table must be computed by (M)ake before the data can be compressed by (R)un. (R)un actually compresses then decompresses the data and displays the data reduction ratio. When making the table, first differencing can be used, which generally reduces the range of the data and improves data reduction. If the data range is too great when executing (M)ake , the range of the lookup table will be truncated to 8 bits, and values outside this range will be placed in the infrequent set and prefixed, so data reduction will be poor. The best thing to do in this case is to attenuate the data to a lower range (with a filter like atten40.fil) and then repeat the process.

(S)ample facilitates study of the sampling process by providing the ability to sample waveforms at different rates and reconstruct the waveforms using three different techniques. This module uses one of three waveforms which it generates internally, so you cannot use this module to subsample existing data. The data is generated at 5,000 sps, and you may choose a sampling rate from 1 to 2,500 sps at which to subsample. When you take the power spectrum of data that has been sampled (but not reconstructed), the program creates a temporary buffer filled with 512 points of the original waveform sampled at the specified rate. In other words, even though the displayed data is shown with the intersample spaces, the power spectrum is not computed based on the displayed data (and the 5,000 sps rate) but rather with the data that would have been created by sampling the analog waveform at the specified sample rate. The reason for this is to illustrate a frequency domain representation based on the specified sampling rate, and not the more complicated power spectrum that would result from computing the FFT of the actual displayed data.

Function (A)daptive demonstrates the basic principles of adaptive filtering, and a(V)erage illustrates the technique of time epoch signal averaging.

Figure D.6 shows the GENWAVE function that provides a waveform generator. Signals with controlled levels of ran(D)om and 60-H(Z) noise can be synthesized for testing filter designs. In addition to (S)ine, (T)riangle, and s(Q)uare waves, ECGs and other repetitive template-based waveforms can be generated with the t(E)mplate command. Figure D.7 shows two signals synthesized using this function. Figure D.8 shows the nine different templates that are provided for synthesizing normal and abnormal ECG signals. For more details about the GENWAVE function, see Appendix E.

```
GENWAVE              GENWAVE PARAMETERS

(S)ine               ECG1
(T)riangle           Heart rate is 120 BPM
s(Q)uare             Sampling rate is 200 Hz
t(E)mplate           There are  5.1 beats
(F)req/BPM           there is 0% random and 0% 60-Hz noise
sample (R)ate        resolution is 8 bits
(P)ts/(B)eats        amplitude of waveform is 100%
ran(D)om
60-H(Z)
(A)mplitude
resol(U)tion
(G)enerate
(W)rite file
(H)elp (F1)
e(X)it

ECG or other TEMPLATEs from default.tpl
```

Figure D.6 Waveform generation.

(a)

(b)

Figure D.7 Waveforms generated using (G)enwave. (a) Two-Hz sine wave with 20% random noise. (b) ECG based on ECG WAVE 1 with 5% 60-Hz noise.

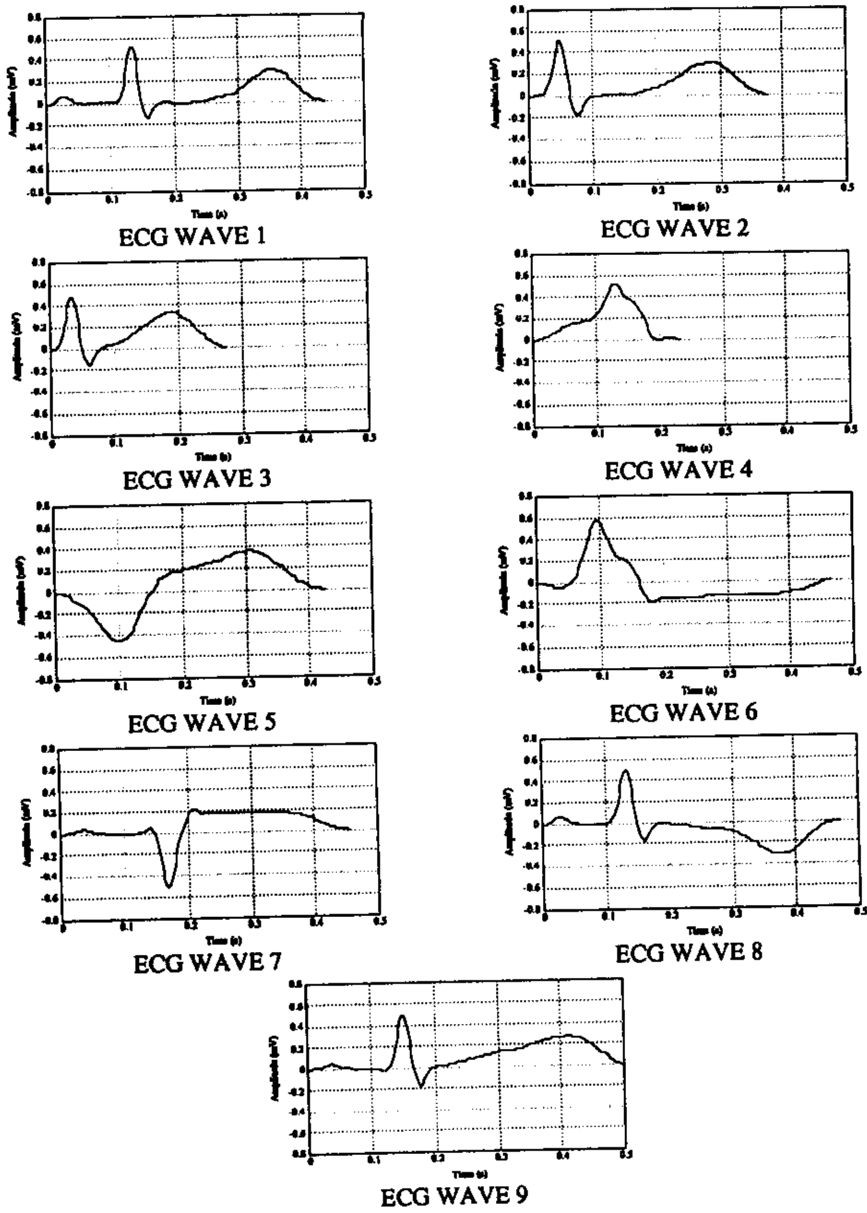

Figure D.8 Templates used by the GENWAVE function to generate ECG waveforms. Template ECG WAVE 1 is normal, the rest are abnormal.

Appendix E

Signal Generation

Thomas Y. Yen

In the process of designing digital filters and signal detection algorithms, it is important to have techniques for testing their performance. The testing process often involves applying signals to the input of the filter or algorithm and observing the resulting signal at the output. For biomedical applications, the use of physiological signals like the ECG and EEG are often preferable to basic signals like sine, triangle, and square waves. Physiological signals are not usually readily available making testing of the filters and algorithms difficult. It is, however, possible to use a computer and software to generate realistic physiological signals. This appendix describes how standard signals and physiological signals are generated by the **GENWAVE** function in UW DigiScope.

E.1 SIGNAL GENERATION METHODS

Two basic techniques for synthesizing signals are the model equation method and the waveform template method. In the model equation method, a mathematical model of the signal is used to calculate a sequence of values that make up the signal. This method of signal generation is very flexible, allowing signals of any frequency and amplitude to be produced. However, accurately modeling a particular signal such as the ECG can be difficult. If the signal is very complex, the model may require a large set of equations and hence a large computation time. We use the equation method in UW DigiScope for generating sine, square, and triangle waves.

On the other hand, the use of waveform templates is a very simple, flexible, and fast method of generating complex, repetitive signals. A waveform template is easily obtained by sampling one cycle of a signal with an analog-to-digital converter and saving the sampled data points as a template. This method allows any type of repetitive signal, regardless of complexity, to be reproduced. We used this template approach for ECG signal simulation in UW DigiScope because of its straightfor-

ward implementation and because of the ability to synthesize new signals by simply adding templates.

E.2 SIGNAL GENERATOR PROGRAM (GENWAVE)

GENWAVE is the program module embedded within UW DigiScope that is called with the **(G)enwave** command. It allows the creation of various types of signal waveforms. The user can specify the number of cycles, sampling rate, waveform repetition frequency, noise level, amplitude, and bit resolution. The different types of signals available are determined by the waveform templates available. New waveform templates can either be added to the default template file or saved in separate template files. The following sections describe how to create template files and how to use the **GENWAVE** program.

E.2.1 Creating template files

Two programs provided for the creation of templates are **ASC2TPL.EXE** and **TPL2ASC.EXE**. **ASC2TPL.EXE** converts an ASCII file of the proper format into a binary template file. **TPL2ASC.EXE** converts a binary template file, such as the default template file **DEFAULT.TPL** into an ASCII file so that it can be edited with an editor or word processing program. The default template file that comes with DigiScope includes nine ECG templates. New templates can be added to this file by first converting the file into an ASCII file, then adding new template data to the end of the file, and finally converting the ASCII file back to the binary default template format. In the conversion to binary, the **ASC2TPL.EXE** program automatically scales the waveforms to a 12-bit range to obtain maximal resolution of the waveform. Figure E.1 shows the structure of the ASCII template file format.

```
Waveform Name              (80 characters max.)
Waveform ID number         (1-12 are reserved; max. number 50)
Number of Datapoints       (32767 max.)
Sample Rate of Signal      (in Hz.-32767 max.)
Data(1)
Data(2)                    (the first and the last datapoints
.                          must be the same baseline values)
.
.
Data(Number of Datapoints)
(Repeat above for each waveform template)
```

Figure E.1 ASCII template file format.

To convert the default template file to an ASCII template file, enter the follo ing syntax:

```
TPL2ASC DEFAULT.TPL <ASCII File Name>
```

where **ASCII File Name** is the name by which the default template data is stor The file name **DEFAULT.TPL** is the binary template file from which the wavefo templates are read. An example of the format of this command is

```
TPL2ASC DEFAULT.TPL ASCTEMP.ASC
```

This command converts the binary template file **DEFAULT.TPL** into an AS(template file called **ASCTEMP.ASC**. Both file names must be specified, and different binary template file name other than **DEFAULT.TPL** may be used.

To convert an ASCII template file to the binary default template file, fi rename the current default template file, then type the following at the comma prompt:

```
ASC2TPL <ASCII File Name> DEFAULT.TPL
```

where **ASCII File Name** is the file that is to be converted to the default templ file. The new **DEFAULT.TPL** file must then be placed in the **DIGSCOPE\STDLIB** (rectory in order for DigiScope to use it. An example of the format of this comma is

```
ASC2TPL ASCTEMP.ASC DEFAULT.TPL
```

converts the ASCII template file into the template file called **DEFAULT.TPL**. Bc file names must be specified, and a different binary template file name other th **DEFAULT.TPL** may be used.

A template file can have a maximum of 50 waveform templates. Each wavefo must have a waveform ID. The ID numbers in a given file should start with o and increase by one for each template in the file. The waveform name can be up 80 characters in length. To ensure amplitude-matching of the start and the end o template, the first and last waveform values must be the same. Figure E.2 shows example of an ASCII template file.

E.2.2 Using the GENWAVE function

When the signal generator **GENWAVE** creates a waveform, several parameters a attached. If the signal is saved as a file, these parameters are placed in the fi header (defaults are in brackets).

1. Output filename (8 characters with 3 letter extension) [**WAVEFORM.OUT**]
2. Input filename (8 characters with 3 letter extension) [**DEFAULT.TPL**]
3. Output sampling rate in sps (10,000 max.) [500]
4. Output waveform frequency in cycles per minute (10,000 max.) [60]

5. Number of data points (5,000 max.) [512]
6. Noise: % of full scale (0 to ±100) [0]
   (a) Random noise
   (b) 60 Hz noise
7. Waveform ID (1 to 50) [1]
8. Full-scale resolution in bits (3–12 max.) [8]

GENWAVE is always able to produce sine, triangle, and square wave signals because they are built into the program even if the **DEFAULT.TPL** file is not found. Since **GENWAVE** always reads this file if it is present, should you wish to create a new default file, save the old **DEFAULT.TPL** file by another name and rename your new file **DEFAULT.TPL**.

```
Normal ECG        (Waveform Name)
4                 (Waveform ID)
157               (Number of Datapoints)
360               (Samplerate)
0                 (Data1)
0                 (Data2)
0                 (Data3)
624               (Data4)
1560              (Data5)
.                 .
.                 .
.                 .
920               (Data155)
25                (Data156)
0                 (Data157 Last)
PVC               (Waveform Name)
5                 (Waveform ID)
83                (Number of Datapoints)
360               (Samplerate)
0                 (Data1)
0                 (Data2)
43                (Data3)
324               (Data4)
260               (Data5)
.                 .
.                 .
.                 .
320               (Data1)
35                (Data82)
0                 (Data83 Last)
```

Figure E.2 Example of an ASCII template file.

Appendix F

Finite-Length Register Effects

Steven Tang

Digital filter designs are either implemented using specific-purpose hardware or general-purpose computers such as a PC. Both approaches involve the use of finite-length data registers (FLR) which can represent only a limited number of values. In Chapter 3, we discussed the process of analog-to-digital conversion and some of the quantization effects due to the finite number of representable magnitudes. This is one of several problems dealing with FLR effects that we must consider when designing our own digital filter on a signal processor or microprocessor. This appendix discusses overflow characteristics, roundoff noise, limit cycles, scaling and I/O variations due to fixed-point registers. We also compare the functional advantages of floating-point and fixed-point registers.

F.1 QUANTIZATION NOISE

Fixed-point registers, when used in digital filters, store a finite number of representable integer numbers. There are two consequences of this type of representation: (1) the state variables that make up the filter can only represent an integral multiple of the smallest quantum, and (2) there is a maximal value that the register can represent in a one-to-one correspondence. The first effect is known as quantization, the second, as overflow (Oppenheim and Schafer, 1975; Roberts and Mullis, 1987).

Quantization errors can occur in a fixed-point register whenever there is a multiply and accumulate function. For example, if we are trying to implement an FIR filter, the output would be the sum of weighted tapped inputs. The final summation is representable only to the precision of the smallest value. The smallest quantum that we can represent with a register of bit-length $B+1$ is

$$q = \Delta 2^{-B} \tag{F.1}$$

This value q is frequently called the quantization step size. We can arbitrarily define q by choosing the value of Δ. If we are trying to design an integer filter, the value of Δ is 2^B, and $q = 1$. The advantage of choosing a large Δ is that we can expand the domain of representable values. However, we consequently lose precision since we also increase the quantization step size. Figure F.1 shows the quantizer characteristics for the rounding of a three-bit register.

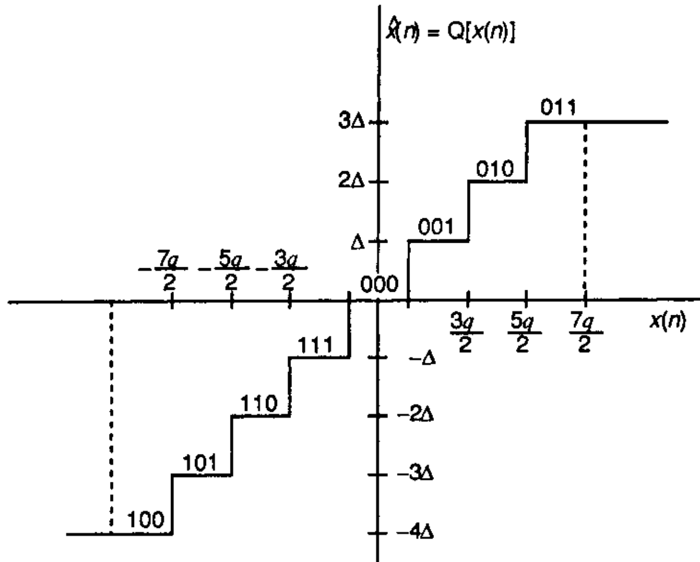

Figure F.1 2's complement rounding quantization effect for 3-bit register.

The error between the real number $x(n)$ and its finite binary representation is

$$e = x(n) - Q[x(n)] \tag{F.2}$$

This is known as the quantization error due to FLR. This presumably also occurs during data acquisition with any A/D converter. However, A/D quantization is a hardware problem; FLR quantization is due to software restrictions.

F.1.1 Rounding

One method of implementing a quantizer is to round off the true value to the nearest representable quantum level. The quantization error is then bounded by $q/2$ and $-q/2$. Thus, quantization is functionally equivalent to adding some random noise in the range $-q/2 < \alpha < q/2$ to the original real number. We can think of this noise as

being randomly generated and having the probably density function (PDF) shown in Figure F.2.

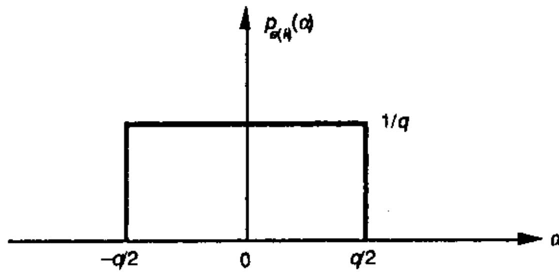

Figure F.2 The probability density function of quantization error for rounding.

Theoretical studies and numerical simulations have shown that this model is accurate for inputs with a relatively broad frequency spectrum and wide PDF, so that the input does not remain between the same quantization levels for a long duration. We can thus calculate a theoretical value for noise power associated with each quantized multiply and accumulate function.

$$\sigma_e^2 = E[e(k)^2] = \int_{-q/2}^{q/2} \frac{\alpha^2}{q} d\alpha = \frac{q^2}{12} \tag{F.3}$$

F.1.2 Truncation

The other method of quantizing is to simply truncate the value, or reduce it until we find a representable level. This has a similar effect to rounding except that the probability density function of such quantization error has a shifted mean. The noise range has limits of $-q$ and 0. While there is no inherent advantage to using either rounding or truncation, rounding is preferred because of its zero mean. For theoretical calculations, this is a much easier method to use since the noise power is simply the variance of the noise.

F.2 LIMIT CYCLES

If a digital filter amplifies the input, there will be internal gain among the state variables. If the accumulated value at a register is beyond the highest binary repre-

sentation available, an overflow characteristic must be implemented. Figure F.3 shows several different ways of describing a function for the values outside the representable range. Saturation overflow acts as a strict output limiter. Two's complement overflow has the characteristic of a wraparound effect. Zero after overflow simply suppresses the output to zero.

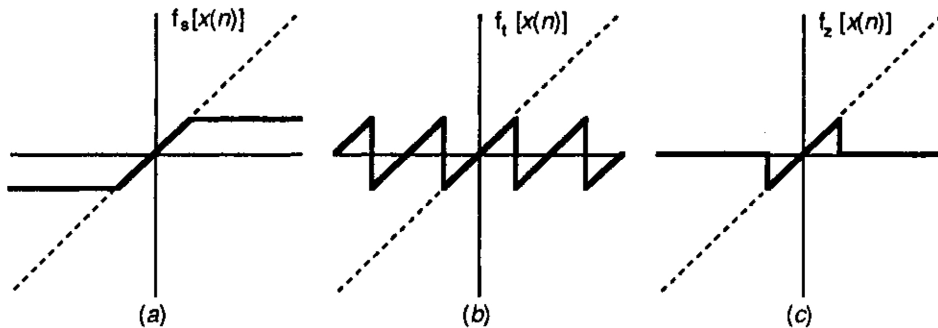

Figure F.3 Overflow characteristics. (a) Saturation, (b) Two's complement, (c) Zero after overflow.

In each case, there are competing advantages that offer a compromise between total roundoff error and the likelihood of continual overflow. Saturation overflow provides the closest output to the actual value (shown by the dotted line). However, by maintaining the output at the largest quantized level, there is greater potential for another overflow after the next filter iteration. The two's-complement overflow scheme is a natural characteristic of two's complement representation; the largest positive representable value is one bit smaller than the most negative value available. While two's-complement creates greater roundoff error, there is equal probability that the output will fall anywhere within the representable quantization range. The zero after overflow is a compromise between the saturation and two's-complement characteristics.

F.2.1 Overflow oscillations

From the different types of overflow characteristics, we see that the possibility of repeated overflows can happen in all three cases. When overflows occur continually, and the output does not converge to zero after an initial nonzero input, there exists an overflow oscillation. The implication of an overflow oscillation is that the output is no longer dependent on the input.

Figure F.4 shows a second-order filter example of jumping from an allowable state space to an overflowed state, applying two's-complement overflow, and fi-

space equations by setting the input to be zero. We can represent any filter by the equations

$$\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}u(k) \quad \text{(F.4)}$$

$$y(k) = \mathbf{C}\mathbf{x}(k) + Du(k) \quad \text{(F.5)}$$

Thus, the **A** is the matrix that maps the current state variable vector, $\mathbf{x}(k)$, to the next, $\mathbf{x}(k+1)$. If the eigenvalues of **A** [i.e., the poles of H(z)] are greater than unity, the filter is unstable and overflow will occur. There are particular regions in the state space for which $\mathbf{x}(k+1)$ will continually be mapped out of range for zero input. These are the areas for which overflow oscillations occur. Having a nonzero input can sometimes change the characteristics of the mapping function so that it jumps out of this cycle. However, the input is usually not very large in comparison to the total range of the state space.

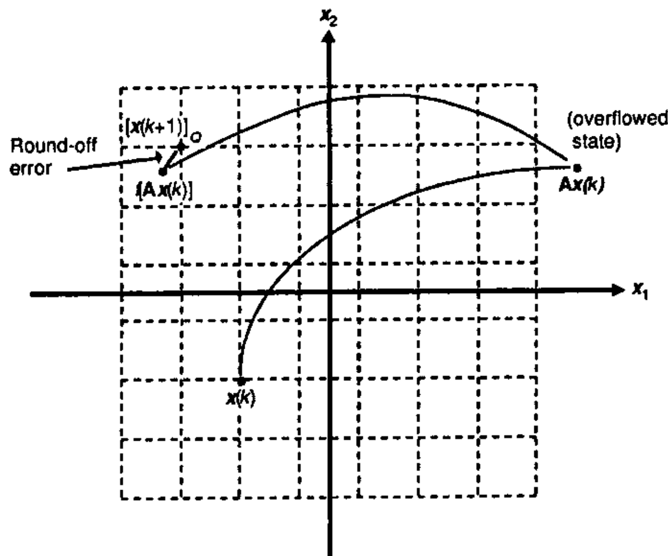

Figure F.4 State space description of second-order digital filter showing overflow and roundoff.

F.2.2 Deadband effect

Overflow is not the only type of limit cycle that can occur from FLR effects. Recursive IIR filters with constant input can produce steady-state output after quantization when they should actually continually decay. This problem is a result

of roundoff errors that cause the state variable vector to get stuck in the same state space. Thus, there is essentially a *deadband* where constant input prevents a linear I/O mapping of the filter characteristics. This is not as hazardous as overflow oscillation in the amount of roundoff error that occurs, but it still is problematic in giving an erroneous output response.

Here is a first order example of such a filter. Let

$$y(nT) = -0.96y(nT - T) + x(nT) \qquad \text{for } x(0) = 13 \text{ and } x(nT) = 0 \text{ for } n > 0$$

Clearly, without quantization, the output should eventually decay to zero. However, if we assume rounding to the nearest integer at the output, we will have the following I/O characteristics.

n	$y(nT) = -0.96[y(nT - T)]_Q$	$[y(nT)]_Q$
0	13	13
1	−12.48	−12
2	11.52	12
3	−11.52	−12
4	11.52	12

Such a filter is said to have a limit cycle period of two. A filter with limit cycle of periodicity one would have a state vector that remains inside the same grid location continuously.

One solution to deadband effects is to add small amounts of white noise to the state vector $x(nT)$. However, this also means that the true steady state of a filter response will never be achieved. Another method is to use magnitude truncation instead of rounding. The problem here, as well, is that truncation may introduce new deadbands while eliminating the old ones.

F.3 SCALING

There are several ways to avoid the disastrous effect of limit cycles. One is to increase Δ so that the state space grid covers a greater domain. However, this also increases the quantization noise power. The most usual cause of limit cycles are filters that have too much gain. Scaling is a method by which we can reduce the chances of overflow, still maintain the same filter transfer function, and not compromise in limiting quantization noise.

Figure F.5 shows how to scale a node v' so that it does not overflow. By dividing the transfer function $F(z)$ from the input u to the node by some constant β, we scale

the node and reduce the probability of overflow. To maintain the I/O characteristics of the overall filter, we must then add a gain of β to the subsequent transfer function $G(z)$.

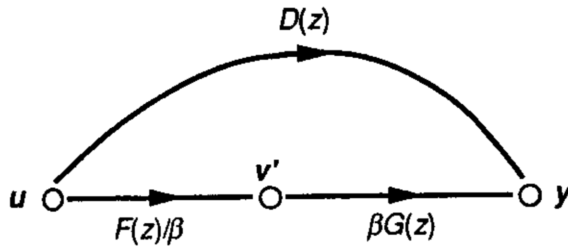

Figure F.5 State variable description of scale node variable v'.

There are two different scaling rules called l_1 and l_2 scaling. The first rule corresponds to bounding the absolute value of the input. The second rule maintains a bounded energy input. The two rules are

$$\beta = \|f\|_1 = \sum_{l=0}^{\infty} |f(l)| \tag{F.6}$$

$$\beta = \delta \|f\|_2 = \delta \left[\sum_{l=0}^{\infty} f^2(l) \right]^{1/2} \tag{F.7}$$

The parameter δ can be chosen arbitrarily to meet the desired requirements of the filter. It can be regarded as the number of standard deviations representable in the node v' if the input is zero mean and unit variance. A δ of five would be considered very conservative.

F.4 ROUNDOFF NOISE IN IIR FILTERS

Both roundoff errors and quantization errors get carried along in the state variables of IIR filters. The accumulated effect at the output is called the roundoff noise. We can theoretically estimate this total effect by modeling each roundoff error as an additive white noise source of variance $q^2/12$.

If the unit-pulse response sequence from node i to the output is g_i, and if quantization is performed after the accumulation of products in a double length accumulator, the total output roundoff noise is estimated as

$$\sigma^2_{tot} = \frac{q^2}{12}\left\{\sum_{i=1}^{n} \|g_i\|^2 + 1\right\} \qquad \text{(F.8)}$$

The summation in this equation is sometimes called the noise gain of the filter. Choosing different forms of filter construction can improve noise gain by as much as two orders of magnitude. Direct form filters tend to give higher noise gains than minimum noise filters that use appropriate scaling and changes to g_i to reduce the amount of roundoff noise.

F.5 FLOATING-POINT REGISTER FILTERS

Floating-point registers are limited in their number of representable states, although they offer a wider domain because of the exponential capabilities. The state space grid no longer looks uniform but has a dependency on the distance between the state vector and the origin. Figure F.6 demonstrates the wider margins between allowable states for numbers utilizing the exponent range.

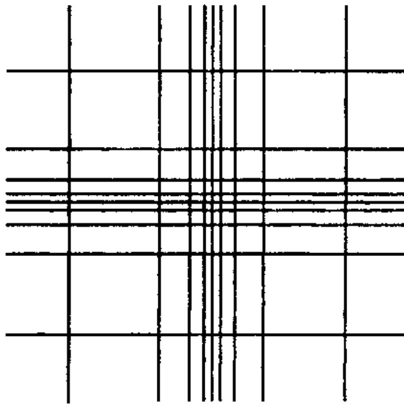

Figure F.6 State space grid for floating-point register.

Floating point differs from fixed point in two ways: (1) there is quantization error present during addition, and (2) the output roundoff noise variance is

proportional to the size of the number in the state variable. Although floating point greatly expands the domain of a filter input, the accumulated roundoff errors due to quantization are considerably greater than for fixed-point registers.

F.6 SUMMARY

In many real-time applications, digital signal processing requires the use of FLRs. We have summarized the types of effects and errors that arise as a result of using FLRs. These effects depend on the type of rounding and overflow characteristics of a register, whether or not it is fixed or floating point, and if there is scaling of the internal nodes.

Figure F.7 compares the total error for a filter with variable scaling levels. For no scaling, we expect to have greater probability of overflow, unless the input is well bounded. As we increase the scaling factor δ, overflow is less prevalent, but the roundoff error from quantization begins to increase because the dynamic range of the node register is decreased. To minimize total error output, we must find a compromise that decreases both errors.

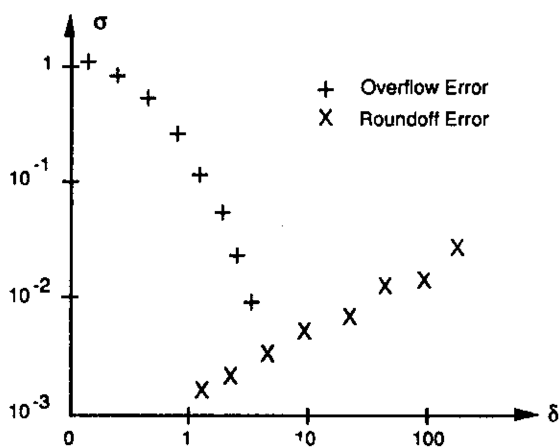

Figure F.7 Comparison of overflow and roundoff error in total error output.

F.7 LAB: FINITE-LENGTH REGISTER EFFECTS IN DIGITAL FILTERS

Write a subroutine that quantizes using the rounding feature and a subroutine that simulates 2's-complement overflow characteristics for an 8-bit integer register. Try implementing a high-pass IIR filter with the transfer function

$$H(z) = \frac{1}{1 + 0.5z^{-1} + z^{-2}}$$

Using a sinusoidal input, find the amplitude at which the filter begins to overflow. Examine the output of the filter for such an input. Does the overflow characterize a 2's-complement response?

F.8 REFERENCES

Oppenheim, A. V. and Schafer, R. W. 1975. *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall.

Roberts, R. A. and Mullis, C. T. 1987. *Digital Signal Processing*, Reading, MA: Addison-Wesley.

Appendix G

Commercial DSP Systems

Annie Foong

A wide variety of commercial data acquisition hardware and software is currently available in the market. Most comes in the form of full-fledged data acquisition systems that support various hardware cards in addition to data analysis and display capabilities. Basically a complete data acquisition system consists of three modules: acquisition, analysis, and presentation.

G.1 DATA ACQUISITION SYSTEMS

G.1.1 Acquisition

Four common ways of acquiring data use (1) an RS-232 serial interface, (2) the IEEE 488 (GPIB) parallel instrumentation interface, (3) the VXI bus, or (4) a PC-bus plug-in data acquisition card.

RS-232 interface

This approach consists of a serial communication protocol for simple instruments such as digital thermometers, panel meters, and data loggers. They are useful for controlling remote data acquisition systems from long distances at data rates lower than 1 kbyte/s. Since the RS-232 interface comes standard on most computers, no extra hardware is necessary.

IEEE 488 (GPIB) interface

Many sophisticated laboratory and industrial instruments, such as data loggers and digital oscilloscopes, are equipped with GPIB interfaces. Devices communicate through cables up to a maximum length of 20 meters using an 8-bit parallel protocol with a maximum data transfer rate of two Mbyte/s. This interface supports both control and data acquisition. IEEE 488 uses an ASCII command set (Baran, 1991).

VXI bus

This bus is a high-performance *instrument-on-a-card* architecture for sophisticated instruments. Introduced in 1987, this architecture has been driven by the need for physical size reduction of rack-and-stack instrumentation systems, tighter timing and synchronization between multiple instruments, and faster transfer rates. This standard is capable of high transfer speeds exceeding 10 Mbyte/s.

Plug-in data acquisition boards

Data acquisition boards plug directly into a specific computer type, such as the PC or the Macintosh. This method combines low cost with moderate performance. These boards usually support a wide variety of functions including A/D conversion, D/A conversion, digital I/O, and timer operations. They come in 8–16 bit resolution with sampling rates of up to about 1 MHz. They offer flexibility and are ideal for general-purpose data acquisition.

G.1.2 Analysis and presentation

Data analysis transforms raw data into useful information. This book is principally about data analysis. Most software packages provide such routines as digital signal processing, statistical analysis, and curve fitting operations.

Data presentation provides data to the user in an intuitive and meaningful format. In addition to presenting data using graphics, presentation also includes recording data on strip charts and generation of meaningful reports on a wide range of printers and plotters.

G.2 DSP SOFTWARE

The trend is toward using commercial DSP software that provides the entire process of data acquisition, analysis, and presentation. Here we discuss commercially available software for general plug-in PC data acquisition boards. Because of the flexibility of such a scheme of data acquisition, there is a huge market and many suppliers for such software. In addition, many vendors offer complete training programs for their software.

Software capabilities vary with vendors' emphasis and pricing. Some companies, for example, sell their software in modules, and the user can opt to buy whatever is needed.

Some common capabilities of commercial DSP software include the following:

1. Support of a wide variety of signal conversion boards.
2. Comprehensive library of DSP algorithms including FFT, convolution, low-pass, high-pass, and bandpass filters.
3. Data archiving abilities. The more sophisticated software allows exporting data to Lotus 123, dBase, and other common analysis programs.
4. Wide range of sampling rates.
5. Impressive graphics displays and menu and/or icon driven user interface.
6. User-programmable routines.
7. Support of high-level programming in C, BASIC, or ASCII commands.
8. Customizable report generation and graphing (e.g., color control, automatic or manual scaling).

A few interesting software packages are highlighted here to give the reader a flavor of what commercial DSP software offers.

SPD by Tektronix is a software package designed for Tektronix digitizers and digital oscilloscopes and the PEP series of system controllers or PC controllers. It offers in its toolset over 200 functions including integration and differentiation, pulse measurements, statistics, windowing, convolution and correlation, forward and inverse FFTs for arbitrary length arrays, sine wave components of an arbitrary waveform, interpolation and decimation, standard waveform generation (sine, square, sinc, random), and FIR filter generation.

DADiSP by DSP Development Corporation offers a version that operates in the protected mode of Intel 80286 or 80386 microprocessors, giving access to a full 16 Mbytes of addressability. Of interest is the metaphor that DADiSP uses. It is viewed as an interactive graphics spreadsheet. The spreadsheet is for waveforms, signals, or graphs instead of single numbers. Each cell is represented by a window containing entire waveforms. For example, if window 1 (W1) contains a signal, and W2 contains the formula DIFF(W1) (differentiate with respect to time), the differentiated signal will then be displayed in W2. If the signal in W1 changes, DADiSP automatically recalculates the derivative and displays it in W2. It also takes care of assigning and managing units of measurement. In the given example, if W1 is a voltage measurement, W1 will be rendered in volts, and W2 in volts per second. As many as 100 windows are allowed with zoom, scroll, and cursoring abilities. The number of data points in any series is limited only by disk space, as DADiSP automatically pages data between disk and memory.

DspHq by Bitware Research Systems is a simple, down-to-earth package that includes interfaces to popular libraries such as MathPak87 and Numerical Recipes.

MathCAD by MATHSoft, Inc. is a general software tool for numerical analysis. Although not exactly a DSP package, its application packs in electrical engineering and advanced math offer the ability to design IIR filters, perform convolution and correlation of sequences, the DFT in two dimensions, and other digital filtering.

A more powerful software package, MatLAB by Math Works, Inc., is also a numerical package, with an add-on Signal Processing Toolbox package having a rich collection of functions immediately useful for signal processing. The Toolbox's features include the ability to analyze and implement filters using both direct and FFT-based frequency domain techniques. Its IIR filter design module allows the user to convert classic analog Butterworth, Chebyshev, and elliptic filters to their digital equivalents. It also gives the ability to design directly in the digital domain. In particular, a function called *yulewalk*() allows a filter to be designed to match any arbitrarily shaped, multiband, frequency response. Other Toolbox functions include FIR filter design, FFT processing, power spectrum analysis, correlation function estimates and 2D convolution, FFT, and crosscorrelation. A version of this product limited to 32 × 32 matrix sizes can be obtained inexpensively for either the PC or Macintosh as part of a book-disk package (*Student Edition of MatLAB*, Prentice Hall, 1992, about $50.00).

ASYST by Asyst Software Technologies supports A/D and D/A conversion, digital I/O, and RS-232 and GPIB instrument interfacing with a single package. Commands are hardware independent. It is multitasking and allows real-time storage to disk, making it useful for acquiring large amounts of data at high speeds.

The OMEGA SWD-RTM is a real-time multitasking system that allows up to 16 independent timers and disk files. This is probably more useful in a control environment that requires stringent timing and real-time capabilities than for DSP applications.

LabWindows and LabVIEW are offered by National Instruments for the PC and Macintosh, respectively. LabWindows provides many features similar to those mentioned earlier. However, of particular interest is LabVIEW, a visual programming language, which uses the concept of a virtual instrument. A virtual instrument is a software function packaged graphically to have the look and feel of a physical instrument. The screen looks like the front panel of an instrument with knobs, slides, and switches. LabVIEW provides a library of controls and indicators for users to create and customize the look of the front panel. LabVIEW programs are composed of sets of graphical functional blocks with interconnecting wiring. Both the virtual instrument interface and block diagram programming attempt to shield engineers and scientists from the syntactical details of conventional computer software.

G.3 VENDORS

Real Time Devices, Inc.
State College, PA
(814) 234-8087

BitWare Research Systems
Inner Harbor Center, 8th Floor, 400 East Pratt Street,
Baltimore, MD 21202-3128
(800) 848-0436

Asyst Software Technologies
100 Corporate Woods, Rochester, NY 14623
(800) 348-0033

National Instruments
6504 Bridge Point Parkway, Austin, TX 78730-5039
(800) IEEE-488

Omega Technologies
One Omega Drive, Box 4047, Stamford, CT 06907
(800) 826-6342

DSP Development Corporation
One Kendall Square, Cambridge, MA 02139
(617) 577-1133

Tektronix
P.O. Box 500, Beaverton, OR 97077
(800) 835-9433

MathSoft, Inc.
201 Broadway Cambridge, MA 02139
(800) MathCAD

The MathWorks, Inc.
21 Eliot St, South Natick, MA 01760
(508) 653-1415

G.4 REFERENCES

Baran, N. 1991. Data acquisition: PCs on the bench. *Byte*, 145–49, (May).
Coffee, P. C. 1990. Tools provide complex numerical analysis. *PC Week*, (Oct. 8).
National Instruments. 1991. IEEE-488 and VXI bus control. *Data Acquisition and Analysis*.
Omega Technologies 1991. *The Data Acquisition Systems Handbook.*
The MathWorks, Inc. 1988. *Signal Processing Toolbox User's Guide.*

Index

A

accelerometer 14
adaptive filter 101, 174
AHA database 280
alarm 3
algorithm 12
alias, definition 57
alphanumeric data 1
Altair 8800 9
ambulatory patient 272
amplifier
    ECG 47
        schematic diagram 48
    instrumentation 28, 73
    micropower 74
    operational 50
amplitude accuracy (see converter, resolution)
analog filter 44 (see filter, analog)
analog multiplexer (see converter, analog-to-digital)
analog-to-digital converter 12 (see converter)
ANN 15
antialias filter (see filter, analog)
apnea 180
Apple II 9
application software 16
application specific integrated circuit 291, 293
architecture 21
array processor 287
arrhythmia 24, 266
    monitor 13, 273
artificial neural network 11, 15, 22
ASIC (see application specific integrated circuit)
assembly language 19, 20
ASYST 359
Atanasoff 7
autocorrelation 224
automaton 244
averaging
    signal 184
AZTEC algorithm (see data reduction)

B

Babbidge 7
BASIC 18, 20, 358
Basic Input/Output System 11, 19
battery power 13
benchmark 12
binomial theorem 81, 126
biomedical computing 5, 17
BIOS (see Basic Input/Output System)
bit-reversal 222
bit-serial processor 288
Blackman-Tukey estimation method 232
blind person 15
block diagram language 21
blood pressure 2
blood pressure tracings 1
body core temperature 2
bradycardia 277
brain 11

C

C language 19, 21
C++ language 21
C-language program
    bandpass filter 168
    bit-reversal computation 223
    convolution 230
    crosscorrelation 226
    derivative 254
    Fan data reduction algorithm 204
    Hanning filter 107
    high-pass filter 251
    low-pass filter 140, 169, 248
    moving window integrator 258
    power spectrum estimation 234

real-time Hanning filter 108
TP data reduction algorithm 196
trapezoidal integrator 135
two-pole bandpass filter for QRS detection 239
CAD 293
calories 14
CALTRAC 14
cardiac
atria 30
atrioventricular (AV) node 30
depolarization 30, 271
equivalent generator 27
output 216
pacemaker (see pacemaker), 47
Purkinje system 30
repolarization 271
sinoatrial (SA) node 30
spread of excitation 30
ventricles 31
ventricular muscle 30
cardiogenic artifact 180, 227
cardiology 25
cardiotachometer 43, 239
cascade (see filter, cascade)
catheter 4
circular buffer 169
clinical engineering 14
clinician 4
closed loop control 3
CMOS 275
CMRR (see common mode rejection ratio)
coefficient sensitivity 102
common mode rejection ratio 36, 73
compressed spectral array 232
compression (see data reduction)
concurrent execution 287
converter
analog-to-digital 275 (see RTD ADA2100 interface card), (see 68HC11EVBU interface card)
analog multiplexer 73
counter 67
dual-slope 67
flash 71
parallel 71
successive approximation 69
tracking 67
charge scaling 65
differential linearity 63
digital-to-analog
design 65, 66
interpolation methods 66
dynamic properties 63
gain error 61
integral linearity 61
integrated circuit 64
monotonicity 61
offset error 61
resolution 63
settling time 63
static properties 61
voltage reference 64
voltage scaling 64
convolution 117, 226, 358
C-language program 230
direct 102
fast 102
correlation
autocorrelation 224, 232
coefficient 223
crosscorrelation 223, 243
C-language program 226
UW DigiScope command 337
CORTES algorithm (see data reduction)
CP/M 9, 18
CRYSTAL 293
CSA (see compressed spectral array)
cutoff frequency 138, 151

D
DADiSP 358
damping factor 138
data compression 15 (see data reduction)
data reduction 193
adaptive coding 209
AZTEC algorithm 197, 202
compressed signal 193
CORTES algorithm 202
facsimile 211
Fan algorithm 202
C-language program 204
Huffman coding 206
frequent set 208
infrequent set 208
translation table 208
lossless algorithm 193, 208
lossy algorithm 194
LZW algorithm 209
reduction ratio 193
residual difference 193, 210
run-length encoding 211
SAPA algorithm 206
signal reconstruction 200
significant-point-extraction 194
turning point algorithm 194, 202
C-language program 196
UW DigiScope commands 339
dBase 358
deadband effect 351

decimation in time 219
derivative (see filter, digital)
desktop computer 6
DFT 218
diabetic 14
diagnosis 4
diathermy 216
differentiator (see filter, digital, derivative)
DigiScope (see UW DigiScope)
digital filter (see filter, digital)
digital signal processing 12
digital signal processor 283
 chip 19
 Motorola DSP56001 285, 286
 Texas Instruments TMS320 285
digital-to-analog converter (see converter)
dipole current source 27
Dirac delta function 80, 216
disk operating system 16
DOS (see disk operating system)
DRAM (see RAM)
DSP (see digital signal processing)
DSP chip (see digital signal processor)
DspHq 358

E
ECG (see electrocardiogram, electrocardiography)
EEG (see electroencephalogram), 184
EEPROM 304
EGG (see electrogastrogram)
Einthoven 25
 equilateral triangle 27
EIT (see electrical impedance tomography)
electrical impedance tomography 15
electrical stimulator 4
electrocardiogram 1, 12, 25
 analysis of 170
 body surface 24
 clinical 24
 electrode 25
 fetal 174, 180
 interpretation of 265
  average beat 267
  decision logic 268
  feature extraction 266
  knowledge base 269
  measurement matrix 267
  median beat 267
  statistical pattern recognition 268
  waveform recognition 267
 isoelectric region 30, 271
 J point 272
 monitoring 24
 power spectrum 236
 ST level 271
 standard 12-lead clinical 24
electrocardiography
 acquisition cart 265
 amplifier 47
  schematic diagram 48
 arrhythmia analysis 277
 clinical 24
 database 261
 electrogram 4
 forward problem 29
 Holter recording 236
 inverse problem 30
 late potential 43, 187
 lead coefficients (see lead)
 lead system (see lead system)
 objective of 24
 QRS detector 236, 246, 277
  adaptive threshold 260
  amplitude threshold 247
  automata theory 244
  bandpass filter 238
  crosscorrelation 242
  first and second derivatives 241
  schematic diagram 49
  searchback technique 260
  template matching 243
  UW DigiScope function 337
 recording bandwidth 43, 187
 rhythm analysis 42
 ST-segment analysis 271
 transfer coefficients 29
 vector 28
electrode 13, 15, 25
 ECG 25
 nonpolarizable 27
 offset potential 25
 polarizable 25
 silver-silver chloride (Ag-AgCl) 27
electroencephalogram 1, 162, 184 (see EEG)
electrogastrogram 232
electrolyte 25
electromyogram 43
electrosurgery 216
emergency room 14
EMG (see electromyogram)
ENIAC 7
EPROM 304
equipment database 14
ESIM 293
Euler's relation 152
evoked response 185
 auditory 189

F
facsimile technology 211
Fan algorithm (see data reduction)
fast Fourier transform (see FFT) 102
FFT 102, 216, 236, 286, 358
  butterfly graph 222
fibrillation 180
fiducial point 187, 242
FIFO 169
filter 57
  adaptive 101
  analog 44
    antialias 87
    bandpass 46
    bandstop 46
    differentiator 46
    high-pass 46
    integrator 44
    low-pass 44
    notch 46
    QRS detector 47
    schematic diagrams 45
  cascade 141, 161, 166
    definition 105
  comb 102, 189
  digital (see also C-language program)
    adaptive 174
    advantages 78
    all-pass 91, 159
    amplitude response 94
    band-reject 111, 138, 161, 175
    bandpass 92, 138, 160, 167, 238, 246, 247
      C-language program 239
      integer 239
    bilinear transform 141
    derivative 111, 241, 246, 253, 271
      C-language program 254
      least-squares polynomial 114
      three-point central difference 114
      two-point difference 114
    difference equation 85
    feedback 125
    finite impulse response (see FIR filter) 100
    frequency sampling design 119
    Hanning 103
    high-pass 138, 158, 159, 160, 246, 250, 355
      C-language program 251
    infinite impulse response (see IIR filter) 100, 125
    integer coefficient 105, 151, 246
    integrator
      rectangular 131
      Simpson's rule 135
      trapezoidal 133
    learning 174
    least-squares polynomial smoothing 108
    low-pass 103, 128, 138, 140, 157, 159, 227, 246, 248, 271
      C-language program 248
    minimax design 120
    moving average 103
    moving window integrator 256
      C-language program 258
    nonrecursive 84
    notch (see band-reject)
    parabolic 109
    phase response 97
    recursive 84, 125
    rubber membrane concept 89
    second derivative 116
    smoothing 103
    stability 86
    time delay 155
    transfer function, definition 84
    transform table design 142
    transversal 176
    two-pole 137, 238
    window design 117
  gain 167
finite element resistivity model 15
finite-length register effect 102, 346
FIR filter 155
  definition 100
firmware 19
floppy disk 9
fluid loss 4
FM modem 265
foldover frequency 87
Forth 21
FORTRAN 20, 265
Fourier transform (see also FFT), 189
  inverse discrete-time 117

G
galvanometer 25
Gibb's phenomenon 117
GPIB (see IEEE 488 bus)
graphical user interface 10
gray-scale plot 232
GUI (see graphical user interface)

H
hardware interrupt structure 19
health care 4
hearing aid, digital 291

heart (see also cardiac)
  disease 24
  monitor 13
  rate 24
high-level language 20
Holter
  real-time 280
  recording 236, 273
  scanning 274
  tape recorder 273
home monitoring 272
Huffman coding (see data reduction)
human-to-machine interface 19

I
IBM PC 9, 295, 317, 319, 325
IBM PC/AT 9, 295
icon 21
ICU (see intensive care unit)
IDTFT (see Fourier transform, inverse discrete-time)
IEEE 488 bus 317, 356
IIR filter 125, 352
  definition 100
  design of 136
implanted pacemaker 14
infusion pump 4
insensate feet 14
integrator (see filter, digital)
Intel
  4004 5
  80286 9, 295, 358
  80386 358
  8080 9
  8086/8088 18
  8088 9, 283, 295
  80C86 275
  i486 5, 9
  microprocessor 5
intensive care unit 3, 14, 17, 24, 273
interactive graphics 7

K
Kirchhoff's voltage law 32

L
L'Hôpital's Rule 167
LabVIEW 21, 359
LabWindows 359
LAN (see local area network)
Laplace transform 80, 130
laptop PC 5
LCD display 14
lead
  augmented limb 34
  coefficients 30
  definition 24
  frontal limb 28
  limb 32
  orthogonal 24
  redundant 33, 266
lead system
  12-lead 39
  Frank 40
  monitoring 42
  standard clinical 39
  VCG 40
learning filter 174
Least Mean Squared algorithm 175
Lempel-Ziv-Welch algorithm 209
limit cycle 102, 351
LINC 7
LINC-8 8
linear phase response 119
Lisa 17
LMS algorithm (see Least Mean Squared algorithm)
local area network 1, 19
Lotus 123 358
LZW algorithm 209

M
MAC time 285
Macintosh 7, 10, 15, 17, 357
MAGIC 293
magnetic resonance imaging 1, 290
math coprocessor 141
MathCAD 358
MatLAB 359
medical care system 4
medical history 4
medical instrument 2, 16
MFlops 285
microcomputer 5
microcomputer-based instrument 5
microcomputer-based medical instrument 5
microprocessor 2 (see Intel, Motorola, Zilog)
minicomputer 5
MIPS 9, 285
MIT/BIH database 261, 280
model of cardiac excitation 24
modeling of physiological systems 14
Moore's Law 5
MOSIS 293

Motorola
  68000 285
  68040 5
  Student Design Kit (see 68HC11EVBU interface board)
MS DOS 9, 18
multitasking operating system 17
multiuser 17

N
native code 21
NEXT computer 19
noise cancellation 174
noninvasive procedure 24
nonrecursive, definition 100
Nyquist frequency 56

O
object-oriented software 21
offset potential 25
op amp (see operational amplifier)
open heart surgery 14
open loop control 3
operating room 14
operating system 16
operational amplifier 50
optical disks 1
OS/2 19
overdamping 138
overflow 349

P
pacemaker 3, 4, 47
PACS 1
palmtop PC 10
parallel processing 6
parallel processor 11, 287
Parseval's theorem 117, 230
Pascal 9, 20
pattern recognition 22
PC DOS 18
PDP-12 8
PDP-8 8
peak detector algorithm 260
percent root-mean-square difference, 193
personal supercomputing 22
phase distortion 101
phase response
  linear 153, 164, 167
    definition 101
  nonlinear 136
physiological signal 2
pipeline 287
pneumography 180, 224
portable arrhythmia monitor 13, 274
portable device 6
portable instrument 5, 13
power spectrum estimation 231
  C-language program 234
PRD (see percent root-mean-square difference)
premature ventricular contraction 277
pressure distribution under the foot 14
pressure sensor 14
probably density function 348
punched card 7

Q
Q of a bandpass filter 98, 161, 238, 240
  definition 46
QRS detector (see electrocardiography)
quantization error 102, 161, 346

R
RAM 11, 286, 304
Real Time Devices (see RTD ADA2100 interface card)
real-time
  application 17
  computing 17
  digital filtering 19, 151, 170
  environment 17
  patient monitoring 236
  processing 17, 193
  signal processing algorithm 17
recursive, definition 125
reduction (see data reduction)
rehabilitation engineering 15
Remez algorithm 120
reprogramming 7
residual difference 193, 210
resolution (see converter)
resonator 162
  digital 103
respiratory signal 224
RF interference 216
RISC processor 287
rolloff 102, 136
ROM 10, 19
roundoff noise 102, 346, 352
RS232 interface 295, 317, 356
RTD ADA2100 interface card 295
  C-language calls 318
  calling conventions 310
  configuration 297
  installation 297
  interrupts 319
  jumper and switch settings 297
rubber membrane concept 89, 138, 157
run-length encoding 211