4000 PX11 & DX11

USER'S GUIDE

IM-0887

TABLE OF CONTENTS

CHAPTER 6 I/OBASIC REAL-TIME CONTROL STATEMENTS

CHAPTER 1

INTRODUCTION

1.1 PURPOSE OF THE MANUAL

The purpose of this manual is to provide all the basic information necessary for a user to set up software and hardware for signal conditioning, data acquisition and process control applications.

This user's guide will:

1. Teach you to start up a BASYS system,

2. Describe the features of the I/OBASIC software,

3. Instruct you on how to use I/OBASIC features, and

4. Give you examples of programs in which I/OBASIC features are used.

If you are a first-time user, read the first four chapters of this manual in sequence before running programs on the system. These chapters will describe the operation of the system so you can take full advantage of its features.

1.2 INTENDED AUDIENCE

I/OBASIC users should know BASIC. Your programs will be written essentially in BASIC, but I/OBASIC has convenience features not found in standard BASIC. These enhancements to BASIC are described in Chapter 5 of this manual. Additional information about I/OBASIC is provided in the I/OBASIC Language Reference Manual.

1.3 RELATED DOCUMENTS

Use the ADAC Corporation hardware manuals provided with your system to familiarize yourself with the hardware of the system and to determine the components of your particular system.

The BASYS System manuals will provide sufficient information to enable BASIC users to operate the system, but you may find the following reference manuals useful in future operation of the system:

DEC RT-11 Manuals

DEC BASIC Manuals

1.4 KEYPAD EDITING SOFTWARE

Included with DISKBASYS and PROMBASYS systems is a keypad editor that permits you to create and modify ASCII files, such as I/OBASIC programs or data files. It is not necessary to use the keypad editor to make full use of the features in the BASYS system. The keypad editor program called KED, is not documented here. You can refer to the PDP-11 Keypad Editor User's Guide, published by Digital Equipment Corporation for information on how to run and use KED.

1.5 BASYS FAMILY DESCRIPTION

The ADAC BASYS family of data acquisition and control systems consists of six members; DISKBASYS, PROMBASYS, PX11, DX11, MICROBASYS and PICOBASYS. These six members span a performance range from the larger DISKBASYS systems to the smaller PICOBASYS systems.

A feature shared by all six family members is the I/OBASIC interpreter language. User-written I/OBASIC programs can be developed on any family member and run on anyother family member. The only exception is the MICROBASYS and PICOBASYS systems do not have support for file I/O and virtual arrays, which are available on DISKBASYS, PX11, DX11, and PROMBASYS systems.

Each BASYS family member can be configured with a wide variety of analog and digital I/O options, as well as other peripheral hardware options. These options permit tailoring of the system to the user's particular requirements. The table below lists the maximum amount of I/O supported by each family member:

	PX11	DX11
CPU Type	LSI-11/23 LSI-11/73	LSI-11/23 LSI-11/73
Disk Support	YES	YES
Powerfail Support	NO	NO
Serial Channels	8	8
High-Level Analog Input Channels	128	128
Digital Channels	128	128
Analog Out	128	128
Wide Range A/D	1024	1024

1.5.1 DX11 SYSTEM DESCRIPTION

DX11 is the high-end member of the BASYS family. It is capable of being configured with a wide variety and amount of mass storage. It is based on Digital Equipment Corporation's RT-11 operating system and LSI-11/23 CPU. The system operates from a rotating mass storage device, (a winchester disk drive). The system is suitable for program developement environments because of the large amounts of mass storage available. Applications that require collecting or storing large quantities of data can also make use of the mass storage devices.

1.5.2 PX11 SYSTEM DESCRIPTION

The PX11system is designed for applications that cannot use rotating disk storage because of environmental conditions such as dust, vibration, temperature, etc.

The PX11system is supplied with the system software located in Programmable Read-Only Memory (PROM XM0: DEVICE). This memory is configured to function as a system disk. When the system is powered on the operating system software is automatically read from this disk, loaded into system working memory, and then executed.

User-written programs and data can be stored in read/write CMOS memory that is configured to function as a disk. This memory is battery backed-up so that it can retain its contents for periods of many days without power. For more permanent storage of programs, software is provided with each PX11 & DX11 system that lets the user burn the image of a read/write disk into a PROM disk. A hardware PROM burner is an option that can be supplied by ADAC Corporation for this purpose.

The operation of a PX11 & DX11 system is described more fully in later sections of this manual.

1.6 BASYS COMMUNICATIONS SOFTWARE

Communications software designed to run on a host computer and support all four BASYS systems is available for a number of different operating systems. This software allows programs to be downline loaded to a BASYS system from a host computer over a standard RS-232/RS-423 serial line. Similarly, programs may be upline stored from a BASYS system to a host computer.

The communications software is available for VAX/VMS, MICROVAX, RSX-11M, RSX-11M+, RT-11, DX11, PX11, and IBM PC/XT host computer systems. Programs are developed and executed directly on a target BASYS system, using only the terminal and mass storage devices of the host computer.

Additional information is supplied in the <u>BASYS Communications Utility User's Guide</u>.

CHAPTER 2

SYSTEM STARTUP

Each of the two different BASYS systems consists of both hardware and software components. The hardware components include items such as a central processing unit, a console terminal, one or more storage devices, and analog and digital I/O devices. Refer to the ADAC Corporation Hardware manuals to familiarize yourself with the components in your particular system. The software components consist of an I/OBASIC interpreter and operating system software.

This chapter will explain how to start up both of these BASYS systems so that the hardware and software can be used.

2.1 STARTING A PX11 & DX11 SYSTEM

No special operations are required to start a PX11 or DX11 system since all of the operating system software is located permanently in the system SY: Device. The system can be bootstrapped directly on power up.

2.2 BOOTSTRAPPING YOUR BASYS SYSTEM

Once the BASYS system has been turned on, the system should bootstrap automatically.

If the system bootstraps correctly, the following display should appear on the terminal screen for DX11 and PX11 systems:

```
                  02-E
ADAC I/OBASIC Vnn-x

>
```

The "greater than sign" (>) is the prompt which tells you that the system is ready to receive I/OBASIC commands. When this prompt is present, you can enter and run programs or perform any other I/OBASIC function. Before attempting to do so, however, read the following chapters describing I/OBASIC software and how it differs from BASIC.

2.3 SAMPLE PROGRAMS

Included with DX11 and PX11 systems are several sample I/OBASIC programs. You can run these programs by using the I/OBASIC RUN command as follows:

```
        >RUN SY:progname <ret>
```

where progname is the name of one of the sample programs. SY: is used to identify the system disk as the location of the program. The following are the sample programs provided with DX11 and PX11 systems:

ADDRSS.BAS	Displays I/O page addresses
CODE.BAS	Plays a game of substitution code
LANDER.BAS	The famous lunar lander game
TIME.BAS	Used for computing instruction times

2.4 INDIRECT COMMAND FILE SUPPORT

I/OBASIC commands and statement lines can be read from indirect command files. This feature is very useful and permits I/OBASIC programs to be run automatically at system powerup by using an indirect command file. This support permits a standalone system to be created using programs developed in I/OBASIC.

The indirect command file feature is especially important in PX11 & DX11 system. It allows the user to generate and store commands that control the manner in which a system is restored, once power is applied, or re-applied. The desired sequence of events to occur is stored as a series of RT-11 commands that get executed in from a startup file. (START.COM)

PX11 & DX11 includes code that, during boot-up of the system, looks for the presence of a special file named "START.COM". If this file is found it is used to process RT-11 and I/OBASIC commands. If this file is not found the system automatically starts the I/OBASIC interpretter and displays the ">" prompt.

During boot, the system is designed to search through the file directories of the various semiconductor "disks", looking for START.COM. Once found, the search is ended, and the file is executed. The sequence of search is first through XM1: (RAM), then XM2: (auxilliary PROM), and then XM3: (located on the 1822PROM board), and finally XM4: (the non-volatile RAM disk).

In writing the START.COM indirect command file, a series of commands is issued in exactly the same manner as if giving these commands manually from the keyboard. To simplify the writing of the file, the RT-11 COPY command can be used to "copy" a file generated from the system console keyboard to the non-volatile RAM provided for XM4:. After the file is generated, it is terminated with a "control-z".

As an example, assume that the following sequence is desired after power-up:

1. Load the I/OBASIC interpreter.

2. Run program 'DATTIM.BAS', which asks the operator for the time and date. The program is stored on device XM4:

3. Run program 'STRTUP.BAS' from device XM3: after DATTIM ends. PX11 ONLY.

4. Upon completion of STRTUP, run program 'LEVCTL.BAS' from device XM3:. It is assumed that XM4 is the default device.

EXAMPLE:

```
COPY TT:START.COM XM4:      instructs system to copy inputs from
                            keyboard to RAM disk, XM4:
R IOBAS                     load I/OBASIC interpreter
RUN DATTIM                  run program DATTIM on XM4:
RUN XM3:STRTUP              run STRTUP on XM3: after DATTIM finishes
RUN XM3:LEVCTL              run LEVCTL after STRTUP finishes
<CTRL-Z>                    terminates ind. command file
```

The program is now stored on the default device, XM4:. It will automatically get called up during power-up. If it is desired to run this program again at some other time, type the following at the RT-11 command level:

```
@START <ret>
```

XM1:	RAM (VOLATILE)
XM2:	PROM
XM3:	1822 PROM
XM4:	STATIC RAM.

CHAPTER 3

MASS STORAGE VOLUMES

Mass storage volumes allow you to keep information outside of the computer memory. Information on storage volumes is accessed using mass storage devices (or drives).

PX11 - Storage device solid state battery backed-up memory
DX11 - Storage device, Winchester and solid state battery backed-up memory

3.1 DATA STORAGE

Information is kept on storage volumes in the form of files. A file is simply a collection of data. Programs, parts of programs, input data, output data, or text, such as reports, may be stored in files.

Every file on a storage volume has a unique name to distinguish it from other files stored on the volume. When you create a file you give it whatever name you wish as long as no two files on one volume have the same name. File names can be up to 6 characters long and can be either letters or numbers.

File names are followed by a period and a 3-letter extension which identifies the file type. File extensions tell you whether a file is a user program, system program, data file, or some other type of file. A list of possible file extensions and their meaning is given in the table below.

File Type	Meaning
.BAC	I/OBASIC compiled file
.BAK	Editor (or other) backup file
.BAS	I/OBASIC source file
.COM	Indirect command file
.DAT	I/OBASIC data file
.SAV	Executable background program file
.SYS	System files and handlers
.TXT	Text file

Normally the operating system will provide the appropriate file extensions to files when they are created. However, you should be aware of these extensions, since they help identify different file types.

The system maintains a directory on each storage volume listing all the files on that volume. The directory will tell you the file name, file type, size, and creation date for each file. Each time you alter the contents of a storage volume, the change will be reflected in the directory. See Section 4.6 for more information on storage volume directories.

3.2 HARD DISK WRITE-PROTECTION

On the control panel there is a button marked (WWP) "Winchester Write Protect". When this button is pressed in, it lights up and the disk is write-protected. To write-enable the drive simply press the "WWP" button until it does not light up.

3.3 ACCESSING STORED FILES

To access information on a storage volume it must be loaded into an appropriate storage device unit. Each unit of a storage device is called a drive. Devices are given code names consisting of two letters, a number, and a colon. For example: DL0: is the name of the winchester drive.

3.4 PX11 & DX11 MEMORY DISKS

PX11 & DX11 Systems treat extended memory as though it were several disks. A special device handler supplied with either system does this automatically. The user needs to know several things in order to make effective use of these disks:

1. The disks have physical device names XM0: to XM7:.

2. If no device name is specified in an I/OBASIC file operation, such as SAVE, OLD, or OPEN, then the default device is used. This is also true for RT-11 operations, such as DIR, DELETE, etc. The default device (called DK:) will be assigned to device XM1: if XM1: is present. If device XM1: does not exist then device XM4: is assigned the default device.

3. Device XM0: contains the operating system software in EPROM. It starts at the physical memory address 3.5M bytes, and is 224K bytes long. It occupies the first seven socket pairs on the ADAC 1822PROM board. (Not available in DX11)

4. Device XM1: is a read/write memory disk. It starts at physical memory address 128K bytes, and its length is dynamically set depending on how much contiguous read/write memory is installed in the system after this address. Up to 3.0M bytes may be installed. If there is no memory in the system device XM1: will not exist. Memory can be volatile dynamic RAM or non-volatile CMOS RAM.

5. Device XM2: is available for a user-developed PROM disk starting at physical address 3.25M bytes and extending to 3.5M bytes. A separate ADAC 1822PROM board could be used to hold the contents of this device.

6. Device XM3: is defined as the last (eighth) socket pair on the ADAC 1822PROM board. This device is used to hold the optional BASYS communications utility software. It can also be used as a user-developed PROM disk. This disk will hold up to 56 blocks of file storage (one block is 512 bytes). (Not available in DX11)

7. Device XM4: is the non-volatile RAM from 64K bytes to 128K bytes. It is available to the user. The configuration parameters generated by the CONFIG program are stored in XM4.

8. The other disks are not assigned to any particular extended memory locations, but they may be configured by the user.

NOTE: Any device on the system can be set to the default device by issuing the RT-11 command: ASSIGN XMn: DK:, where XMn is device XM0:, XM1:, XM2:, XM3:, or XM4:. This command can be issued directly by the user or it can be imbedded in a START.COM file set up by the user. The START.COM file should contain the following commands:

```
ASSIGN XMn: DK:
RUN XM0:IOBAS
```

See the section on indirect command file support in Chapter 5 of this manual for more information.

If the need arises to configure an extended memory disk the following RT-11 command may be used:

```
CONFIG SET XMn BASE=aaa LENGTH=bbb
```

Where n is the disk unit to be configured, aaa is the physical base address of the extended memory that will be used as a disk, and bbb is the length of the disk. Parameters aaa and bbb may be specified as one of the following:

1. An address in increments of 64 bytes (or 32 words), as determined by the high 16 bits of the 22-bit physical address, as in 4000, which represents the physical address 400000.

2. A block address, by appending the letter B to the number. A block is 256 words (512 bytes).

3. A kilobyte address, by appending the letters KB to the number. A kilobyte is 1024 bytes.

4. A megabyte address, by appending the letters MB to the number, as in 2MB or 2.MB.

The numbers aaa and bbb may also be either octal or decimal numbers. If a decimal point follows the number (as in 10.), it is assumed to be a decimal number, otherwise it is an octal number.

To show the current extended memory disk configuration, use the following command at the RT-11 level:

```
        CONFIG SHOW
```

The following chart shows the memory configuration of the various XM disks:

```
                                              Octal

+---------------------------+    0KB           0
+  RT-11 Operating System   +                              56KB
+---------------------------+   56KB      160000
+STANDARD, always supplied  +                              72KB maximum
+     XM4:    RAM disk      +                            disk is autosized
+             Non-volatile  +
+---------------------------+  128KB      400000
+                           +
+                           +
+     XM1:    RAM disk      +                             2944KB maximum
+optional; can be volatile  +                            disk is autosized
+  or non-volatile RAM      +
+---------------------------+ 3072KB    14000000
+       <unassigned>        +                              256KB
+---------------------------+ 3328KB    15000000
+                           +                              256KB maximum
+     XM2:    PROM disk     +                            disk is autosized
+                           +                            disk is read only
+---------------------------+ 3584KB    16000000
+                           +                              224KB
+     XM0:  PROMdisk      +                              disk isread only
+                           +
+---------------------------+ 3808KB    16700000
+     XM3:    PROM disk     +                             32KB
+----------------------------+3840KB    17000000      diskisread only
+       <unassigned>        +                             256KB
+---------------------------+ 4088KB    17760000
+         I/O Page          +
+---------------------------+ 4096KB    17777777
```

CHAPTER 4

RT-11 COMMANDS

The operating system, called RT-11, is part of the software of the BASYS System. RT-11 is a collection of programs that make up the environment in which you enter and run your programs. RT-11 coordinates the resources of the computer and lets you control them.

The BASYS System is designed to enable you to bypass much of the operating system in order to decrease the learning time involved in using the system. This chapter provides the RT-11 commands you will need to know, in addition to the I/OBASIC language, in order to use the BASYS System. The RT-11 commands described in this chapter are listed below:

1. ASSIGN
2. COPY
3. DATE
4. DELETE
5. DIRECTORY, /FULL, /BRIEF, /PRINTER
6. FORMAT, /SINGLE
7. INITIALIZE
8. RENAME
9. SQUEEZE
10. TIME
11. TYPE

This chapter tells how and when to perform each of the above commands. Prior to performing any of these commands you must exit from the I/OBASIC interpreter.

All commands given in this chapter are entered by typing a carriage return after the command. A carriage return is symbolized by <ret> in the command format. If an RT-11 command requires that you specify a device name, the device name will be indicated by ddn: where dd is the two-letter device code and n is drive number.

4.1 HOW TO EXIT FROM OR RUN I/OBASIC

To enter any RT-11 command you must exit the I/OBASIC interpreter so you can direct your commands to the RT-11 operating system. To exit the I/OBASIC interpreter and communicate to RT-11, type the following on the terminal in response to the I/OBASIC prompt (>):

```
        BYE <ret>
```

A period (.) will appear on the terminal. This period is the RT-11 prompt. It is similar to the greater-than sign (>) prompt in I/OBASIC; it tells you that the system is ready to receive RT-11 commands. To enter an RT-11 command simply type the command followed by a carriage return in response to an RT-11 prompt.

If your system is running in the RT-11 operating system and you want to return to I/OBASIC, you can do either of two things:

1. Type the following in response to the RT-11 prompt:

   ```
   RUN SY:IOBAS <ret>
   ```

2. Or, rebootstrap (reload) the system software. This can be done either by turning the power off and then on again, or by depressing the RUN button located on the front of the system. Remember that when the system is first turned on, it runs I/OBASIC and you see the greater-than sign (>) prompt.

4.2 ASSIGN COMMAND

The ASSIGN command is used to assign a logical device name to a physical storage device. Use the following command format:

```
        .ASSIGN <ret>

        Physical Device Name?  ddn:  <ret>

        Logical Device Name?  xx:  <ret>
```

ddn: is the physical device name; xx: is a logical device name. For example:

```
        .ASSIGN <ret>

        Physical Device Name?  DL0:  <ret>

        Logical Device Name?  DK:  <ret>
```

results in disk drive DL0: being assigned the logical name DK:. This means that it can be specified by either of the two designations, DL0: or DK:. Use the ASSIGN command to specify which device unit is to be the default (DK:) for RT-11 commands.

When issuing an RT-11 command that assumes the default device specification (DK:), you can override the default by specifying any other device in the command line.

NOTE

> In general, the RT-11 ASSIGN command is not needed by users of a BASYS System. It is possible to successfully operate a BASYS System without ever having to issue this command. It is documented here for configurations that have more than two disk drive units, and where the user wants to make use of logical device names.

4.3 COPY COMMAND

The COPY command copies files between or within storage volumes. The system duplicates the file you specify as input and gives it the name and file extension you specify as output. The original version of the file is not altered in any way.

You can use this command to create backup copies of important data.

To copy a file from one volume to another, the volume you are copying to must be write-enabled (Section 3.2). Use the following command format:

```
.COPY <ret>

From?  ddn:filename.ext <ret>

To?  ddm:copyname.ext <ret>
```

where ddn: and ddm: are physical or logical device names. If ddn: and ddm: are the same then the copy is placed on the same volume as the original.

4.4 DATE COMMAND

The DATE command is used to set the current date or to print it on the terminal. To set the date use the following command format:

```
.DATE dd-mmm-yy <ret>
```

Where dd-mmm-yy is the current date in the form day-month-year. For example, .DATE 05-OCT-81 sets the date to October 5th, 1981.

The date must be reset every time you turn the BASYS System on. If the date has been set, the following command will cause it to print on the terminal:

```
.DATE <ret>
```

The date will also print on header lines of I/OBASIC programs and directory listings.

4.5 DELETE COMMAND

The DELETE command is used to delete file names from a storage volume. When you enter the DELETE command, the system will confirm each file to be deleted prior to deleting it. Use this prompt to check your operation before proceeding. If you type Y, the system will delete the specified file. If you type N or anything other than Y, the file specified will not be deleted.

To delete a file, the volume must be write-enabled (Section 3.2). Use the following command format:

```
.DELETE <ret>
```

The system will prompt for the name(s) of the file(s) to be deleted. Enter the file name and the extension following the prompt. More than one file can be specified for deletion by entering the file names and extensions separated by commas, as in the example below. RT-11 confirms each file to be deleted.

```
Files? DEMO.BAS, SPELL.TXT, TRIAL.DOC <ret>

Files deleted:
DK:DEMO  .BAS ? Y <ret>
DK:SPELL .TXT ? Y <ret>
DK:TRIAL .DOC ? Y <ret>
```

The system will not delete a file until you type Y in response to the prompt. If you type N or anything other than Y, the system ignores that file and goes on the next one. Note that the system prints the file name and the device specification as a safeguard against accidental deletion.

4.6 DIRECTORY COMMAND

The DIRECTORY command allows you to find out what files are stored on a volume. The directory is maintained by RT-11; it consists of file names, file lengths, and file creation dates for each volume. The directory will also tell you how much free space is available on the volume.

Your volume does not need to be write-enabled to view the directory since this command does not alter its contents. The following command is used to print the directory on the terminal screen:

```
        .DIRECTORY ddn:  <ret>
```

Where ddn: is the device name. The system will default to DK: if no device name is specified.

A sample directory is shown below and explained in the following paragraph.

```
 10-NOV-86
SWAP  .SYS    26P 17-May-85        RT11SJ.SYS    78P 17-May-85
DX    .SYS     3P 17-May-85        DY    .SYS     4P 17-May-85
DL    .SYS     5P 17-May-85        DD    .SYS     5P 17-May-85
LP    .SYS     2P 17-May-85        LS    .SYS     2P 17-May-85
XM    .SYS     3P 17-May-85        LD    .SYS     8P 17-May-85
IOBAS .SAV    79P 17-May-85        CONFIG.CNF     7P 17-May-85
STARTS.COM     1P 17-May-85        PIP   .SAV    29P 17-May-85
DIR   .SAV    19P 17-May-85        DUP   .SAV    45P 17-May-85
KED   .SAV    59P 17-May-85        LANDER.BAS     3P 17-May-85
CODE  .BAS     5P 17-May-85        TIME  .BAS     4P 17-May-85
ADDRSS.BAS     2P 17-May-85        DU    .SYS     4P 17-May-85
FORMAT.SAV    21P 17-May-85        CONFIG.BAS    25P 17-May-85
 24 Files, 439 Blocks
 549 Free blocks
```

The directory lists the name you gave your file or program followed by the file extension or type. After the file name and extension is the number of blocks of space that the file uses followed by the file's creation date. The final lines of the directory tell you the number of files, the total space used, and the remaining free blocks on the volume. A block on a disk can store 512 characters.

You can also enter the command DIRECTORY/FULL to see the files on a volume and how the available free space is arranged. This information is useful in determining if a volume is near capacity. Section 4.9 describes how to change the arrangement of the available free space on a volume to make it more useful. A sample full directory is shown below:

```
 06-MAY-83
PIP   .SAV     23  02-JAN-83
< UNUSED >     18
CONFIG.BAS     20  16-MAY-82
< UNUSED >     18
TEMP1 .DAT     60  15-MAY-82
< UNUSED >    118
 3 Files, 103 Blocks
 154 Free Blocks
```

In the directory above, the total free space is 154 blocks, but it is broken up into three sections containing 18 blocks, 18 blocks, and 118 blocks respectively.

If there are too many lines in the directory to fit on the screen at one time, the system will scroll the screen to print them all. You may need to stop the scrolling in order to examine the directory. To do this, type the letter S while holding down the CTRL key (CTRL/S). To start the screen scrolling again, type the letter Q while holding down the CTRL key (CTRL/Q).

The DIRECTORY/BRIEF command will print an abbreviated directory containing only the file name and file extension. The DIRECTORY/PRINTER command will cause the directory to be output to a line printer if one is available rather than the terminal. Make sure the line printer is turned on when you issue the DIRECTORY/PRINTER command.

The optional /FULL, /BRIEF, and /PRINTER commands can be used in combination. For example, the command DIRECTORY/FULL/PRINTER will cause a full directory to be output to the line printer.

4.7 INITIALIZE COMMAND

When you initialize a volume, the system will destroy the existing directory and create a new empty directory. The initialize command erases all existing information on the volume.

After initializing a volume, the directory will be empty until you store something on the volume. You must initialize new volumes to prepare them for use, but you can also initialize volumes with unwanted data on them to reclaim the space for use. This command should be used with care because it results in all the information on the volume being deleted.

You must write-enable a volume prior to initializing it because its contents will be deleted (Section 3.2). To initialize a volume use the following command format:

```
.INITIALIZE ddn:  <ret>
```

Where ddn: is the device name which must be specified. The system returns the following prompt.

```
ddn:/INITIALIZE, Are you sure?
```

If you type Y followed by a carriage return, the system will initialize the volume in the specified drive. If you type N, or anything other than Y, the command will be cancelled and nothing will happen to the disk. This is a safeguard to prevent accidental deletion of files.

The system prints the targeted physical device name followed by the operation to be performed. Use this information to double check your command.

4.8 RENAME COMMAND

The RENAME command changes a file name or file type without altering or moving the file itself. When you enter the RENAME command, the device specified for input and output must be the same.

To rename a file, the volume must be write-enabled (Section 3.2). The command format given below renames the file TRIAL.DOC to TEST.TXT.

```
.RENAME <ret>

From?  DK:TRIAL.DOC <ret>

To?  DK:TEST.TXT <ret>
```

4.9 SQUEEZE COMMAND

The SQUEEZE command compresses all the files on a volume into a block at the beginning of the volume leaving a contiguous block of all remaining free space. Squeezing leaves your remaining free space in a more useable form. Squeezing does not alter the information stored in your files, it only alters the placement of the files on the volume.

If you have created or deleted a large number of files, the free space on the volume may be broken up into small unuseable segments. When you try to create a file using I/OBASIC, the following message may appear on the terminal:

```
        ?Not enough room
```

You can check the directory using the command DIRECTORY/FULL on the volume to see the configuration of free space on the disk. This directory will show how the free space is broken up; if many small, unused segments exist, you should squeeze the disk.

To squeeze a volume it must be write-enabled (Section 3.2). Use the following command format:

```
        .SQUEEZE ddn:  <ret>
```

Where ddn: is the device name. The system will assume DK: if no device name is specified. The system will return the following prompt:

```
        ddn:/SQUEEZE, Are you sure?
```

If you type Y followed by a carriage return the volume will be squeezed. If you type N, or anything other than Y, the operation will be cancelled. When a volume has been squeezed, check the directory using the command DIRECTORY/FREE to see if you have created more useable free space. The blocks of free space should now be arranged in one large block.

NOTE

Squeezing only rearranges existing free space into one large free area. It does not increase the total free space.

4.10 TIME COMMAND

The TIME command is used to set or display the current time in 24-hour notation. The system keeps track of the correct time based on the initial time you enter by issuing the following command:

```
        .TIME hh:mm:ss <ret>
```

Where hh:mm:ss is the current time in the form hours:minutes:seconds. For example 17:23:34 is the correct notation for 23 minutes and 34 seconds past 5 PM. You need not enter the seconds or minutes (TIME 17 or TIME 17:23 are sufficient to specify 5 PM or 5:23 PM respectively).

The TIME command must be entered each time you turn the system on to keep the time current. Every time you enter the TIME command, the new time overrides the previous setting.

To check the time once it has been set, use the following command format:

```
        .TIME <ret>
```

4.11 TYPE COMMAND

The TYPE command will list the contents of a file on the console terminal. The command can be used to list an I/OBASIC program, a data file, a command file, or any other file that is in ASCII format. It may not be used to list the contents of a binary file.

The following is the format for the TYPE command:

```
        .TYPE <ret>
```

The system will prompt for the name of the file to be listed, as in:

```
        Files? filename.ext <ret>
```

where filename.ext is any valid BASYS file specification, such as TEST1.DAT, or DL1:PROG1.BAS.

CHAPTER 5

I/OBASIC EXTENSIONS

The I/OBASIC interpreter is an extended implementation of the Digital Equipment Corporation BASIC interpreter. The I/OBASIC version of BASIC is easier and more convenient to use than standard BASIC. I/OBASIC also provides you with powerful statements that give you the ability to perform sophisticated real-time data acquisition and control without having to write lengthy or complicated programs.

The following I/OBASIC features are described in this chapter:

1. Long-variable name support.

2. Upper/lower case support for keywords and variable names.

3. Program formatting support.

4. Real-time control statements.

I/OBASIC features are explained in this chapter. The real-time control statements included in I/OBASIC are summarized in this chapter, with detailed instructions for their use given in Chapter 6.

5.1 LONG VARIABLE NAME SUPPORT

To improve the readability of your programs, I/OBASIC allows you to use long variable names. While standard BASIC limits you to one or two characters for a variable name, I/OBASIC allows each variable name to be up to 32 characters long, so that you can create more descriptive variable names.

I/OBASIC variable names may contain any combination of letters, numbers, and the underscore character. Variable names may also contain embedded I/OBASIC keywords, such as PRINT, or WAIT. The only restriction is that all I/OBASIC variable names must begin with a letter. For example, the following are valid I/OBASIC variable names:

```
        gas_flow,  fortune,  delay_4time,  tank12
```

The following are not valid I/OBASIC variable names:

```
        12pay,  4_pressure,  6temp2,  _tax
```

This feature of I/OBASIC is very useful for creating programs that are easily understood by yourself and others. It also greatly simplifies the task of writing large or complex programs. Long variable names can be selected which are most descriptive of the variable's purpose.

To illustrate this point, two programs are listed below that do exactly the same thing.

```
        100 a = 5
        110 b = 6
        120 c = 10
        130 d = 2
        140 a = a + b * d + .5 * c * t^2
        150 PRINT a
        160 END

        100 dist = 5
        110 speed = 6
        120 accel = 10
        130 time = 2
        140 dist = dist + (speed * time) + (.5 * accel * time^2)
        150 PRINT dist
        160 END
```

It is obvious that the second program uses variable names that are more descriptive and understandable than those in the first program.

It is also helpful to avoid using variable names that are very long. Longer variable names can be more descriptive, but they also result in longer program lines, thus making for more typing and reading. Keeping variable names to less than about twelve characters still permits a name to be quite descriptive.

The underscore character can also be used within a variable name to delimit different words, making the program more understandable. For example, an accounting program might have the following variable names:

```
        employee_name,  employee_pay,  employee_age
        consultant_name,  consultant_pay,  consultant_age
```

It is suggested for the I/OBASIC programmer that a common prefix name be given to all variables that can be classified as a group, such as those listed above in the accounting program example. This greatly simplifies the task of selecting variable names and, at the same time makes the program easier to read and understand.

5.2 UPPER/LOWER CASE SUPPORT

To improve the readability of your programs, I/OBASIC allows you to enter your program in upper or lower case letters. When your program is listed I/OBASIC will always convert variable names to lower case and I/OBASIC keywords, such as statements and functions, to upper case. For example, if you type in the following program:

```
        10 for loop_count = 1 to 10
        20 LET TIME_FRAME = LOOP_COUNT*2
        30 print time_frame
        40 WAIT(TIME_FRAME)
        50 next loop_count
        60 end
```

it will appear as follows when it is listed:

```
>listnh <ret>
10 FOR loop_count = 1 TO 10
20 LET time_frame = loop_count * 2
30 PRINT time_frame
40  WAIT(time_frame)
50 NEXT loop_count
60 END

>
```

Observe that even though the variables "time_frame" and "loop_count" were originally entered in a variety of cases, they are all converted to lower case by I/OBASIC. Thus, the variable TIME_frame and Time_FRAME are the exact same variable to I/OBASIC.

5.3 PROGRAM FORMATTING

I/OBASIC allows you to insert any number of spaces or tabs between the program line number and the first word of the program line. This allows you to use an indented structure when you program to facilitate debugging and improve readability.

It is desirable to use formatting to make programs more readable and understandable. Program statements occurring within FOR/NEXT loops can be indented to make them easier to read. For example:

```
>listnh <ret>
100     FOR count = 1 TO 10
110             PRINT count
120             square = count * count
130             PRINT square
140     NEXT count
150     PRINT "End of loop"
160     END

>
```

Note that the statements within the FOR/NEXT loop can be seen easily due to their indentation.

5.4 I/OBASIC REAL-TIME CONTROL STATEMENTS

The BASYS System allows you to communicate with external devices connected to it under the control of your I/OBASIC programs. Your I/OBASIC programs can collect data, control processes and instruments, and schedule events in a wide variety of applications. To reduce your programming overhead and to make your programs more powerful without being more complicated, the I/OBASIC interpreter includes statements which will perform many of these real-time control operations.

I/OBASIC real-time control statements are listed below:

1. ANALOG_IN reads high-level analog input channels.
2. ANALOG_LOW_IN reads low-level analog input channels.
3. ANALOG_OUT writes to analog output channels.
4. BIT_CLEAR clears a digital output bit.
5. BIT_SET sets a digital output bit.
6. BIT_TEST tests a digital input or output bit.
7. CHAR_IN reads characters from serial channels.
8. CLOCK_OUT operates the 1601GPT Real-Time Clock.
9. CONVERT_OCTAL converts octal values to and from decimal.
10. DIGITAL_IN reads a digital input or output channel.
11. DIGITAL_OUT writes to a digital output channel.
12. GET_DATE and SET_DATE get and set the system date.
13. GET_TIME and SET_TIME get and set the system time.
14. ON ERROR GOTO enables error processing.
15. ON EVENT GOSUB enables event processing.
16. PEEK reads memory locations.
17. POKE writes to memory locations.
18. TEMPERATURE_IN reads a thermocouple temperature.
19. TEST_ADDRESS tests for valid bus addresses.
20. TIME_OUT starts a software timer.
21. WAIT delays the program.

In addition to the above real-time control statements, there are also a number of SET statements that can be used to select particular operating parameters for the above statements. One example is the SET_ANALOG_VOLTS statement, which sets the engineering units to volts for the analog input and output statements.

Chapter 6 tells you how and when to use each statement and gives examples of programs in which the statements are used. Additional programming examples can be found throughout this manual and also the <u>I/OBASIC Language Reference Manual</u>.

CHAPTER 6

I/OBASIC REAL-TIME CONTROL STATEMENTS

I/OBASIC includes real-time control statements which allow you to perform input and output (I/O) of data between your BASYS System and a variety of external devices. These statements are powerful programming aids that give you the ability to monitor and control external hardware using analog and digital I/O devices in the BASYS system. Your programs control the statements and the statements control the I/O devices.

This chapter describes I/OBASIC statements and gives examples of their use. General information about the use of statements is also given. The following are the main features of the I/OBASIC real-time control statements:

1. Analog and digital I/O statements support programmed I/O, interrupt, and DMA modes of operation. These three modes of operation are explained in Section 6.1.

2. All analog input and output statements can be passed numbers in engineering units, or if desired, in binary format (as an octal ASCII string). The engineering units supported are percent full-scale, volts, and degrees centigrade.

3. Analog, digital, and timer I/O statements can operate either synchronously or asynchronously. When operated asynchronously, it means that the I/O can take place concurrently with I/OBASIC program execution. I/O completion is signalled by the setting of a flag. When operated synchronously, these statements behave like all other I/OBASIC statements, in that they complete all required processing before the next program statement is executed. Use of asynchronous operation is enhanced significantly by use of the "ON EVENT GOSUB" statement.

4. Several statements provide time-related support. Setting and reading the system date and time, operating the real-time clock, issuing programmed delays, and queueing timer flags have all been included.

5. Several miscellaneous statements have been included that permit such things as converting decimal numbers to octal numbers, performing memory PEEK and POKE functions, and testing if an address is present on the bus.

6.1 STATEMENT OPERATING MODES

I/OBASIC statements operate like statements in any other version of BASIC, whereby one program statement at a time is executed, and when it is done, the next program statement executes. However, unlike other versions of BASIC, several real-time control statements available in I/OBASIC can also operate asynchronously, so that they perform I/O simultaneously with the execution of other program statements. The real-time control statements that support a flag argument are the ones that can operate asynchronously. The following three modes of operation apply to these statements and are explained in the following sections:

1. Programmed I/O mode.
2. Interrupt mode.
3. Direct Memory Access (DMA) mode.

6.1.1 PROGRAMMED I/O MODE

When an I/OBASIC program executes a real-time control statement that can be passed an optional flag argument, but no flag argument is passed, that statement will operate in programmed I/O mode. When a statement uses programmed I/O, it completes all processing before the next I/OBASIC program statement is executed. Therefore, the statement is said to operate synchronously, since it always completes at a known place within the I/OBASIC program (actually prior to the next program statement).

6.1.2 INTERRUPT MODE

When an I/OBASIC program executes a real-time control statement and a flag argument is specified, that statement will operate in interrupt mode. When the statement operates in interrupt mode it will begin execution, perform as much processing as possible, and then pass control to the next program statement so it can be executed. At some later point, when the appropriate hardware interrupt needed by the real-time control statement occurs, the statement will finish processing, and then set the value of the flag argument to a one. Therefore, the statement is said to operate asynchronously, since it will actually complete at some indeterminate point later in the program.

This mode of operation can be used to improve the real-time response of the system, to overlap program execution and I/O, and to permit multiple I/O to occur simultaneously.

6.1.3 DIRECT MEMORY ACCESS MODE

This mode is identical in operation to interrupt mode, except that it makes use of DMA hardware to perform the actual I/O. DMA mode can only be used with systems that are configured with an ADAC 1622DMA controller board. This mode uses the least amount of computer time to operate. It will, for instance, permit high-level analog input to occur at aggregate rates up to 100,000 samples per second, simultaneously with program execution.

DMA mode is supported by the ANALOG_IN, statements. It will be used automatically when a flag argument is specified in the statement and DMA hardware is configured on the system. If a flag argument is specified but no DMA hardware exists, then interrupt mode will always be used.

6.2 I/OBASIC ARGUMENT TYPES

Arguments for real-time control statements can be of type integer, real, or string. An integer is a number in the range -32768 to +32767. It cannot have a fractional part or a decimal point. Constants and variables of type integer that appear in an I/OBASIC program must be followed by a percent (%) sign, as in the examples listed below:

```
        123%,  count%,  -16%,  employee_age%
```

Integer constants and variables take up less memory than other I/OBASIC variable names so they are useful in applications where memory is scarce. Also, program execution is always faster with integer constants and variables than with real constants and variables.

Real numbers are numbers within the absolute range of approximately 1E+38 to 1E-37. A real number can contain a fractional part and a decimal point. Constants and variables of type real that appear in an I/OBASIC program are not followed by any special character. Some sample real variables and constants are: 3.14159, flow_rate, -123.0, 8.1E+3, tank_volume

Real variables and constants have a larger range than integers and are useful in most applications. However real variables and constants take up more memory than integers, and also require more computer time when used in calculations.

String constants and variables are groups of letters, numbers and symbols. In an I/OBASIC program string constants must be enclosed in either single or double quotes (" or '). String variables must be followed by a dollar sign ($). Some typical string constants are:

```
        "Hello.",  'anything goes',  ":)$*^22"
```

and some typical string variables are:

```
        a$,  name$,  user_password$,  address12$
```

String variables are useful for storing words, names, phrases, and anything that cannot be stored as a number. I/OBASIC supports a wide range of string handling functions that can be used to manipulate strings and perform operations on them. These functions are listed in the _I/OBASIC_ _Language_ _Reference_ _Manual_.

All I/OBASIC real-time control statements support a special use of string variables for expressing octal (base 8) numbers. If the string "123" is an argument in an I/OBASIC real-time control statement, for example, it will be interpreted as the octal number 123 (83 decimal). If a string is specified as the variable where an I/OBASIC statement should return a value, the statement will place the octal representation of the value in the string variable.

6.3 DOCUMENTATION NOTES

The following documentation notes apply to the descriptions of the I/OBASIC real-time control statements found in this chapter:

1. All ranges given for legal argument values are inclusive unless otherwise specified.

2. Arguments passed as strings in the statement must contain valid octal numbers. Example: "0123" or "177777"

3. All optional parameters and arguments are enclosed in brackets "[ ]".

6.4 INTERRUPT SERVICING AND THE USE OF FLAGS

In real-time control applications, it is often desirable to trigger, or start, an action based upon some event happening. The event may be indicated by a contact closure or a signal level change. The BASYS systems are capable of sensing these events, and reacting to them, as dictated by the application programs.

When an event occurs, the CPU is interrupted, and an interrupt service routine is exercised. The action taken is a function of the type of card through which the interrupt occurs and the I/OBASIC statement being executed.

6.4.1 INPUT/OUTPUT STATEMENTS

Each of the analog and digital I/O statements, as well as a few other statements, have an optional FLAG argument. When an I/OBASIC statement is executed with a specified flag, the statement operates in the interrupt, or asynchronous, mode. In this mode it will begin execution, perform as much processing as possible, and then pass control to the next program statement for execution. At some later point, when the appropriate hardware interrupt needed by the real-time statement occurs, the statement being processed will finish, and then the value of the flag argument will be set to a one.

With the DIGITAL_IN and DIGITAL_OUT statements, if the flag is not used, the statements are executed synchronously, which means that the input or output of data occurs when the statement is encountered in the program. Advancing to the next statement occurs only after the statement is completed. If the flag is used, the flag is set to zero when the statement is encountered, and then the program immediately moves on to the next statement. Meanwhile, the DIGITAL_IN or DIGITAL_OUT statement is not executed until the interrupt circuit on the board being acted upon is activated. When the interrupt signal does come, data is transferred through the card, and the flag is set to a one.

If the val argument of the DIGITAL_IN or DIGITAL_OUT statement was specified as an array, then one element of the array would be transferred for each interrupt that occurred. When the last element of the array is transferred, the flag is set to a one.

For the ANALOG_IN, ANALOG_LOW_IN and TEMPERATURE_IN statements, if no flag is used, the A/D converter operates in the programmed I/O mode, and all processing of the input statement is completed before advancing to the next statement. If a flag argument is used, the A/D converter operates in the interrupt mode. In this mode execution is begun and as much processing as possible is performed before the program transfers to the next statement. Meanwhile, the A/D continues its digitizing process. When the A/D finishes, it requests an interrupt and then the processing of the input statement is completed. At this time the flag is set to a one.

If the val argument in the input statement is specified as an array, then the A/D , once started, continues to digitize until the array is filled. Each conversion operates through the interrupt circuit. When the array is full, processing of the input statement is completed, and the flag is set to a one. Depending upon whether SET_AIN_SCAN or SET_AIN_NOSCAN was used, the array will contain data from a number of channels or from one channel. In filling an array, successive triggers for the A/D can come from the software (if SET_AIN_NOTRIGGER is used) or from an external clock such as the 1601GPT if SET_AIN_TRIGGER IS USED.

For input statements, if the flag is used and DMA hardware is present, then the A/D will always operate in DMA mode. (See sec. 6.1.3). In DMA mode, the flag operates the same as interrupt mode with array.

For the ANALOG_OUT statement, use of the flag means that the D/A converter(s) will operate in DMA mode. The val argument would normally specify the name of the array (must be integer) that will be transferred to the D/A. The flag will be set to a one when the array is fully transferred.

6.4.2 ON EVENT GOSUB

The ON EVENT GOSUB statement enables event processing within an I/OBASIC program. When a flag argument from any real-time control statement is set to a one, the program will branch to the line specified in the ON EVENT GOSUB statement.

When event processing is desired in an I/OBASIC program, the ON EVENT GOSUB statement is generally placed at the beginning of the program.

Event processing is very useful for creating powerful real-time control programs that do not have to poll for completion of I/O or timed events.

Although there is only one event processing subroutine possible within an I/OBASIC program, the event processing subroutine can determine what flag caused the event by checking to see if a flag is a zero or a one. Control branches to the subroutine when a flag is set only after the current I/OBASIC statement completes entirely. Therefore, statements such as INPUT should be avoided in a program if real-time response is required. The CHAR_IN statement can be used in its place.

EXAMPLE:

```
10 REM Collect 200 samples from each of 5 channels
       and then output data through digital channel 0
20 ON EVENT GOSUB 1000
30 DIM temp(1000)
40 SET_AIN_SCAN
50 ANALOG_IN(4,temp(),full)
               .
               .
  perform other processing here
               .
               .
1000 REM event processing subroutine starts here
1010 DIGITAL_OUT(0,temp())
1020 EVENT RETURN
               .
    other program statements
               .
               .
2000 END
```

6.5 STATEMENT DESCRIPTIONS

The real-time control statements are described below. They are grouped for easier understanding:

1. Analog input statements
2. Temperature measurement statements
3. Analog output statements
4. Digital I/O statements
5. Time and timing statements
6. Miscellaneous statements

6.5.1 ANALOG INPUT STATEMENTS:

6.5.1.1 ANALOG_IN - Reads high-level analog input channels.

```
FORM:    ANALOG_IN(chan, val [,flag])

         or

         AIN(chan, val [,flag])
```

This statement reads a channel on the high-level analog input board, and places the values in the argument val. If argument flag is specified, the channel is read in interrupt mode, or DMA mode if a 1622DMA controller is present. (Not supported in PICOBASYS).

The chan argument specifies the channel to be read; it can be either a constant or variable, and it can be of type integer, real, or string. Chan must be in the range 0 to 191.

The first of two allowable high-level boards is accessed by channel values 0-63 and the second (if any) is accessed by chan values 128-191. Chan values 0-63 correspond to channels on the first board, and chan values 128-191 correspond to channels on the second board. The boards operate independently with respect to channel scanning and DMA (if present), so it is possible to to have two ANALOG_IN statements with flags specified operating simultaneously. In DX11 & PX11 the high level A/D board is model 1023AD, 4012HLAD, and 4112WRAD.

The val argument is a variable of type integer, real, or string or an array of type integer or real. If argument val is of type real, the values returned are in units of either volts or percent full-scale (see the SET_ANALOG_PERCENT and SET_ANALOG_VOLTS statements). If argument val is of type integer or string the values returned are the unscaled binary readings from the A/D.

When val is an array, the array elements either contain multiple readings of the same channel (argument chan) or contain readings for channels 0 to the value specified by chan (or from chan 128 to the value specified for the 2nd AD). The former is the default and is selected by the SET_AIN_NOSCAN and the latter is selected by SET_AIN_SCAN. If an array is dimensioned for n elements and the array is integer, 2n memory locations are reserved for storage. If the array is real (such as volts or percent full scale), 4n elements are reserved.

The flag argument must be a variable of type integer or real.

NOTE

The ANALOG_IN statement also supports direct transfer of analog input channel data to a virtual array. The virtual array must be located on an XM memory disk, and it must be of type integer. For complete information on this feature, see the chapter Data Files and Virtual Arrays in this manual.

If the ANALOG_IN statement is used with a flag and DMA hardware is present, data is collected via DMA mode. Since the frame trigger for DMA is a hardware trigger, the ANALOG_IN statement must be preceded with a SET_AIN_TRIGGER statement.

EXAMPLE:
The sample program below will read analog input channels 0 to 10 and print their values in units of volts.

```
>listnh <ret>
100 SET_ANALOG_VOLTS
110 FOR channel = 0 TO 10
120     ANALOG_IN(channel,value)
130     PRINT channel,value
140 NEXT channel
150 END
>
```

The following program will read analog input channel one, wait for the interrupt and then print the channel voltage expressed as percent full-scale.

```
>listnh <ret>
100 SET_ANALOG_PERCENT
110 ANALOG_IN(1,value,flag)
120 IF flag = 0 THEN 120
130 REM flag = 1 means ANALOG_IN statement is done
140 PRINT value
150 END
>
```

6.5.1.2 SET_AIN_GAIN - Sets the high-level analog gain.

FORM: SET_AIN_GAIN(val)

This statement will set the hardware programmable gain on the high-level analog input device. Programmable gain can be used to amplify small analog input signals, thereby increasing the resolution of the converted value.

The actual programmable gain that is set by this statement is determined by passing a gain 'code' in the argument val. There is a different code (or number) for each possible gain. Argument val can be either a constant or variable of type integer, real, or string. The following table shows the legal gain codes for argument val, and the corresponding gain that will be set on the analog input device:

val	Gain	
0	8	
1	4	
2	2	
3	1	Default

The default code is 3, which sets the gain to 1. This means that the analog input signal is passed unchanged to the analog input device. For example, if the range of the analog input board is 10 volts and the gain is set to 4, then the new range will be 10/4 = 2.5 volts. Thus, the maximum value read from the board will be 2.5 volts. If a voltage greater than the range is input to the board, the value returned will equal the maximum value. If the analog engineering units are set to percent (using statement SET_ANALOG_PERCENT), analog input values will always be in the range 0 to 100 for unipolar and -100 to 100 for bipolar configurations.

The following program will read 10 voltages from the analog input channels. It is assumed that the maximum voltage to be read is 1.25 volts and that the board is configured with a range of 10 volts.

```
>listnh <ret>
100 SET_ANALOG_VOLTS
110 SET_AIN_GAIN(0) \ REM set gain to 8
120 FOR count = 1 to 10
130      ANALOG_IN(1,value)
140      PRINT value
150 NEXT count
170 END

>
```

6.5.1.3 SET_AIN_SCAN / SET_AIN_NOSCAN - Sets high-level analog input channel scanning.

FORM: SET_AIN_SCAN

FORM: SET_AIN_NOSCAN

These statements will enable or disable channel scanning for the ANALOG_IN statement. When channel scanning is enabled, the chan argument specifies the highest channel to be scanned so that the first channel of a multiplexer connected to the A/D (chan 0 or 64) up to the selected high channel are read in the ANALOG_IN statement. If two A/D converters are used, the SET commands must be established for each converter separately.

The default setting for I/OBASIC is NOSCAN. The default setting is restored each time a run command is issued.

The following program will scan channels 0 - 5 and output their values 10 times.

```
        >listnh <ret>
        100 DIM scan_values(5)
        110 SET_AIN_SCAN
        120 FOR output_counter = 1 to 10
        130     ANALOG_IN(5,scan_values())
        140     FOR print_counter = 0 to 5
        150             PRINT scan_values(print_counter)
        160     NEXT print_counter
        170 NEXT output_counter
        180 END

        >
```

See the ANALOG_IN statement description for a programming example.

6.5.1.4 SET_AIN_TRIGGER / SET_AIN_NOTRIGGER - Sets the high-level analog input triggering mode.

FORM: SET_AIN_TRIGGER

FORM: SET_AIN_NOTRIGGER

These statements will enable or disable external (hardware) triggering for the ANALOG_IN statement.

The default setting for I/OBASIC is NOTRIGGER. The default setting is restored when a RUN command is issued.

If the analog input device is triggered faster than it is capable of operating, I/OBASIC will issue the error message "?A/D trigger too fast".

When using the A/D converter in DMA mode, the ANALOG_IN statement must be preceded by the SET_AIN_TRIGGER statement in order to enable the external clock for channel to channel triggering. The external clock can be the on-board potentiometer controlled clock or it can be generated by the 1601GPT card. The on-board clock has a range of 800Hz to 25 KHZ. Standard jumper position selects on-board clock unless a 1601GPT is ordered with the system. See BASYS Hardware Manual, Sec. 5.2.10.5 for jumper options.

The following program will set the high-level analog input device to external triggering and then wait for the analog input to change.

```
        >listnh <ret>
        100 SET_AIN_TRIGGER
        110 ANALOG_IN(5,old_val)
        120 ANALOG_IN(5,new_val)
        130 IF old_value = new_value THEN 120
        140 PRINT "Input changed from ";old_val;" to ";new_val
        150 END
        >
```

6.5.1.5 ANALOG_LOW_IN - Reads low-level analog input channels.

FORM: ANALOG_LOW_IN(chan, val [,flag])

or

AINL(chan, val [,flag])

This statement reads a channel on the low-level analog input board, and places the values in the argument val. If argument flag is specified, then the channel is read in interrupt mode.

Arguments for this statement are identical to those for the ANALOG_IN statement. The chan argument must be in the range 0 to 1023 for DX11 and PX11. If the cold-junction compensation is hardwired on the ADAC low level analog input board, then each board will contain 128 channels (0-127). (See Appendix A, Sec. 2a). If however, the cold-junction compensation is software programmable, then channels 64-127 will not be present on that board. (See Appendix A, BASYS Configuration Program 1). This will have the effect of creating 'holes' in the channel number range. The val argument must be a variable of type integer, real, or string or an array variable of type integer or real. If argument val is of type real, the values returned are in units of either volts or percent full-scale (see the SET_ANALOG_PERCENT and SET_ANALOG_VOLTS statements). If argument val is of type integer or string the values returned are the unscaled binary readings from the A/D.

EXAMPLE:

The sample program below will read low-level analog input channels zero to ten, and store their values in an array called table:

```
>listnh <ret>
100 DIM table(10)
110 FOR channel = 0 TO 10
120     ANALOG_LOW_IN(channel,voltage)
130     table(channel) = voltage
140 NEXT channel
150 END

>
```

The following program will produce results identical to the program above by using a single ANALOG_LOW_IN statement:

```
>listnh <ret>
100 DIM table(10)
110 SET_AINL_SCAN
120 ANALOG_LOW_IN(10,table())
130 END
>
```

Notice the use of two features, the SET_AINL_SCAN statement to enable low-level channel scanning, and the use of an array as an argument in the ANALOG_LOW_IN statement. When an array is passed, the entire array is filled with analog channel values.

6.5.1.6 SET_AINL GAIN - Sets the low-level analog gain.

FORM: SET_AINL_GAIN(val)

This statement will set the hardware programmable gain on the low-level analog input device. Programmable gain can be used to amplify small analog input signals, thereby increasing the resolution of the converted value.

The actual programmable gain that is set by this statement is determined by passing a gain 'code' in the argument val. There is a different code (or number) for each possible gain. Argument val can be either a constant or variable of type integer, real, or string. The following table shows the legal gain codes for argument val, and the corresponding gain that will be set on the analog input device:

val	Gain	
0	1000	
1	500	
2	200	
3	100	
4	20	
5	10	
6	5	
7	1	Default

The default code is 7, which sets the gain to 1. This means that the full-scale analog input signal range is 10.0 volts/1 or 10.0 volts. Therefore, the maximum signal input to the low-level analog input device should be 10.0 volts.

The program below will read a voltage between -0.02 and +0.02, print it as a percent of full scale (-100 to +100). This assumes that the low-level analog device is configured for bipolar operation and a range of 10 volts.

```
>listnh <ret>
100 SET_ANALOG_PERCENT
110 SET_AINL_GAIN(1) \ REM set gain to 500
130 ANALOG_LOW_IN(4,reading)
140 PRINT "The value is"; reading; " percent."
150 END
>
```

6.5.1.7 SET_AINL SCAN / SET_AINL_NOSCAN - Sets low-level analog input channel scanning.

FORM: SET_AINL SCAN

FORM: SET_AINL_NOSCAN

These statements will enable or disable channel scanning for the ANALOG_LOW_IN statements. When channel scanning is enabled, the chan argument specifies the high channel to be scanned so that the first channel of the multiplexers connected to an 1114AD (chan 0, 128, 256, etc.) to the selected high channel are read in the ANALOG_LOW_IN statement. SET_AINL_SCAN must be specified for each of multiple, low level A/D cards.

The default setting for I/OBASIC is NOSCAN. The default setting is restored when a RUN command is issued.

The program below will repeatedly scan channels 0 to 7 and fill up array scan_values with their values. When the array has been completely filled, the channels will not be read anymore:

```
>listnh <ret>
100 DIM scan_values(200)
120 SET_AINL_SCAN
130 ANALOG_LOW_IN(7,scanvalues())
       .
       .
   [perform other processing here]
       .
       .
300 GO TO 130
>
```

6.5.1.8 SET_AINL TRIGGER / SET_AINL_NOTRIGGER - Sets the low-level analog input triggering mode.

FORM: SET_AINL TRIGGER

FORM: SET_AINL_NOTRIGGER

These statements will enable or disable external (hardware) triggering for the ANALOG_LOW_IN statement.

The default setting for I/OBASIC is NOTRIGGER. The default setting is restored when a RUN command is issued.

See the description of the SET_AIN_TRIGGER statement for an example.

6.5.2 TEMPERATURE MEASUREMENT STATEMENTS:

6.5.2.1 TEMPERATURE_IN - Reads a thermocouple temperature.

FORM: TEMPERATURE_IN(chan, val [,flag])

or

TMPIN(chan, val [,flag])

This statement reads a channel on the low-level analog input board, returning the values in degrees centigrade in the val argument. If argument flag is specified, then the channel is read in interrupt mode, or DMA mode if a 1622DMA controller is present.

The operation of TEMPERATURE_IN is identical to that for the ANALOG_LOW_IN statement, except the val arguments of type real are returned as temperatures in degrees centigrade. If val is specified as an array, dimensioned for n elements, 4n memory locations are reserved for the array.

All of the SET statements that affect the ANALOG_LOW_IN statement will also affect the TEMPERATURE_IN statement. This means, for example, that the SET_AINL_SCAN will enable channel scanning for the TEMPERATURE_IN statement as well. See Appendix A, Sec. 2a for Programmable Cold Junction.

The following sample program reads and displays the temperature readings of the first 16 channels:

```
>listnh <ret>
100 FOR chan = 0 TO 15
110      TEMPERATURE_IN(chan,temp)
120      PRINT chan,temp
130 NEXT chan
140 END
>
```

The program below reads a temperature using interrupt mode. Note that the program fills an array while it is waiting for the TMPIN statement to complete. The TMPIN statement is the short form for the TEMPERATURE_IN statement.

```
>listnh <ret>
100 DIM some_array(10)
110  TMPIN(3,temp,int_flag)
120 FOR count = 0 TO 10 \ REM This operation occurs
130 some_array(count) = 1 \ REM asynchronously with the
140 NEXT count            \ REM TMPIN statement.
150 IF int_flag = 0 THEN 150
160 PRINT temp
170 END
>
```

6.5.2.2 SET_THERMOCOUPLE - Sets the thermocouple type and temperature range.

FORM: SET_THERMOCOUPLE(val)

This statement will set the thermocouple type and its temperature range to be used with the TEMPERATURE_IN statement. The thermocouple types supported are J, K, and T. Only one thermocouple type may be selected for the TEMPERATURE_IN statement, but it may be changed during program execution. Selecting a narrower temperature range will increase the accuracy of the thermocouple readings.

In the syntax above, argument val is a variable or constant that specifies a code for selecting the thermocouple type and temperature range. It must be in the range from one to six. The following table lists the thermocouple types and ranges that are selected for each value of argument val:

val	Type	Range (degrees C)	
1	J	-210 to 870	Default
2	J	-210 to 366	
3	J	-210 to 185	
4	K	0 to 1232	
5	K	0 to 484	
6	T	-200 to 385	

The default value is 1, which selects thermocouple type J and a range of -210 to 870 degrees C. The default value is restored when a RUN command is issued.

The example below reads the temperature of a K type thermocouple and prints it:

```
>listnh <ret>
100 SET_THERMOCOUPLE(4)
110 TEMPERATURE_IN(1,temp)
120 PRINT "Temperature: "; temp; "degrees centigrade."
130 END

>
```

The next example shows how to use the SET_THERMOCOUPLE statement to read different types of thermocouples connected to the same low-level analog input board:

```
>listnh <ret>
100 READ channel,type
110 IF channel = 999 THEN 170
120 SET_THERMOCOUPLE(type)
130 TEMPERATURE_IN(channel,value)
140 PRINT channel,value
150 GO TO 100
160 DATA 0,4,1,1,2,6,7,1,9,2,999,999
170 END

>
```

6.5.3 ANALOG OUTPUT STATEMENTS:

6.5.3.1 ANALOG_OUT - Writes analog output channels.

FORM: ANALOG_OUT(chan, val)

or

AOT(chan, val,[FLAG])

This statement writes to an analog output channel. If argument flag is specified and the channel is part of a 1023AD/1023EX/1622DMA board set, the output will occur using DMA mode.

NOTE

Specifying the optional flag argument requires the 1622DMA board. If the flag argument is specified in the statement and no 1622DMA board is configured in the system a "?Syntax error" is returned.

The flag argument must be a variable of type integer or real.

The chan argument specifies the channel to be updated with the value in argument val. Chan can be a constant or variable of type integer, real, or string. Chan should be in the range 0 to 127 for DX11 and PX11 system.

The val argument contains the value to be output to the channel specified in argument chan. Val can be a constant or variable of type integer, real, or string or an integer array (real arrays are valid only for non-DMA output operations -- those excluding the flag argument).

If argument val is a real array, it should contain numbers in the engineering units currently selected, either volts or percent full-scale.

If argument val is an integer array, it should contain binary numbers within the range of the analog output channel. Engineering units are not applicable to argument val when it is of type integer. These numbers should be in the range:

1. zero to 4,095 for full-scale unipolar 12 bits, or

2. -2,048 to +2,047 for full-scale bipolar 12 bits.

To output analog data in DMA mode, the BASYS System must be configured so that one or two of the analog output channels are DMA channels. This is done by specifying the channel's address to be the same as the address of the DMA board's digital-to-analog converters (see configuration program, Appendix A).

For DMA operations (flag is specified and val is an integer array) the SET_AOT_NOTOGGLE (default) and SET_AOT_TOGGLE statements are used to control the output. When NOTOGGLE setting is in effect, output is to the channel specified by chan. When TOGGLE setting is in effect, the output alternates between the two DMA channels starting with the channel specified in chan.

NOTE

The ANALOG_OUT statement also supports direct transfer data in a virtual array to an analog output channel. The virtual array must be located on an XM memory disk, and it must be of type integer. For complete information on this feature, see the chapter <u>Data Files and Virtual Arrays</u> in this manual.

EXAMPLE:

The sample program below outputs the values from 0 to 99 to analog output channel 3. These values are in units of percent full-scale, which is the default engineering unit when a program is run.

```
>listnh <ret>
100 FOR value = 0 TO 99
110      ANALOG_OUT(3,value)
120 NEXT value
130 END

>
```

Note that these values are in percent full-scale, which is the default engineering unit when a program is run.

The following program outputs the values 0 to 2047 to analog output channel 5 using DMA mode. This assumes DMA hardware is present:

```
>listnh <ret>
100 DIM values%(2047)
110 FOR count = 0 TO 2047 \ REM fill array
120     values%(count) = count
130 NEXT count
140 ANALOG_OUT(5,values%(),flag)
150 IF flag = 0 THEN 150
160 END

>
```

Note that in this example the analog output values are not in units of either percent full-scale or volts, but are binary numbers in the proper range, as discussed above. This requirement is unique to this statement when performing analog output with an integer argument for val. All other analog input and output statements will always operate using engineering units of percent full-scale or volts.

6.5.3.2 SET_AOT_TOGGLE / SET_AOT_NOTOGGLE - Sets DMA analog output toggling.

FORM: SET_AOT_TOGGLE

FORM: SET_AOT_NOTOGGLE

These statements will enable or disable channel toggling for the ANALOG_OUT statement when using DMA mode. This mode can only be used with systems that include the appropriate DMA hardware.

In toggle mode, the data being output will be alternated between the two DMA output channels. In no-toggle mode, the data will be output to only the one DMA channel that is specified in the ANALOG_OUT statement.

The default setting for I/OBASIC is NOTOGGLE. The default setting is restored when a RUN command is issued.

EXAMPLE:

The following program will output the contents of the integer array called data_values using DMA toggle mode. It is assumed that analog output channel 10 has been configured for DMA mode. In this mode data_values%(0) will be output to DAC hardware channel 1, data_values%(1) will be output to DAC channel 2, data_values%(2) will be output to DAC channel 1, and so on.

```
>listnh <ret>
100 DIM data_values%(2047)
110 FOR count = 0 to 2047 \ REM fill array
120     data_values%(count) = count
130 NEXT count
150 SET_AOT_TOGGLE
160 ANALOG_OUT(10,data_values%(),flag)
170 IF flag = 0 THEN 170
180 END

>
```

6.5.3.3 SET_AOT_TRIGGER / SET_AOT_NOTRIGGER - Sets the DMA analog output triggering mode.

FORM: SET_AOT_TRIGGER

FORM: SET_AOT_NOTRIGGER

These statements will enable and disable external triggering for the ANALOG_OUT statement when using DMA mode. This option is only valid for systems with the appropriate DMA hardware.

When external triggering is enabled, the DMA analog output will proceed at the rate specified by an external triggering signal. When external triggering is disabled, the DMA analog output takes place at maximum DMA rate.

The program below will output an array to the analog output device at a rate specified by an external trigger signal.

```
>listnh <ret>
100 DIM data_array%(100)
110 FOR count = 0 to 100 \ REM fill array
120     data_array%(count) = count
130 NEXT count
140 SET_AOT_TRIGGER
150 ANALOG_OUT(3,data_array%(),dma_flag)
>
```

6.5.4 DIGITAL I/O STATEMENTS:

6.5.4.1 BIT_CLEAR - Clears a digital output bit.

FORM: BIT_CLEAR(chan, bit)

or

BIC(chan, bit)

This statement clears a single bit in a digital output channel. This statement operates only in programmed I/O mode.

The chan argument specifies the 16-bit digital channel containing the bit to be cleared. Chan can be a constant or variable of type integer, real, or string. Chan should be in the range 0 to 127 for DX11 and PX11 system.

Argument bit specifies the digital bit within the digital channel to be cleared. Argument bit can be a constant or variable of type integer, real, or string. Its value must be in the range 0 to 15.

EXAMPLE:

The sample program below clears bits 1, 3, and 7 of output channel 1.

```
>listnh <ret>
100 BIT_CLEAR(1,1)
110 BIT_CLEAR(1,3)
120 BIT_CLEAR(1,7)
>
```

6.5.4.2 BIT_SET - Sets a digital output bit.

FORM: BIT_SET(chan, bit)

or

BIS(chan, bit)

This statement sets a single bit in a digital output channel. It operates only in programmed I/O mode.

The chan argument specifies the digital channel containing the bit to be set. Chan can be a constant or variable of type integer real or string. Chan should be in the range 0 to 127 for DX11 and PX11 systems.

Argument bit specifies the digital bit to be set. Argument bit may be a constant or variable of type integer, real, or string. Its value must be in the range 0 to 15.

EXAMPLE:

The sample following below sets bits 1, 3, and 7 of output channel 1.

```
>listnh <ret>
100 BIT_SET(1,1)
110 BIT_SET(1,3)
130 BIT_SET(1,7)
140 END

>
```

6.5.4.3 BIT_TEST - Tests a digital input or output bit.

FORM: BIT_TEST(chan, bit, val)

or

BIT(chan, bit, val)

This statement will test a single bit in a digital input or output channel and place it in argument val. This statement operates only in programmed I/O mode.

The chan argument specifies the channel containing the bit to be tested. Chan can be a constant or variable of type integer, real, or string. Chan should be in the range 0 to 127 for DX11 and PX11 systems.

Argument bit specifies the digital bit to be tested in the channel. Its value must be in the range 0 to 15. Argument bit may be a constant or variable of type integer, real, or string.

The val argument must be a variable of type integer, real, or string. The val argument is returned with the result of the bit test. The value returned is either 0 or 1, depending on whether the bit is cleared or set.

EXAMPLE:

The program below determines the state of each input bit in channel zero and duplicates that state for the corresponding output bit of channel one. The program loops indefinitely.

```
>listnh <ret>
100 FOR bit = 0 TO 15
110     BIT_TEST(0,bit,bitvalue)
120     IF bitvalue = 0 THEN BIT_CLEAR(1,bit)
130     IF bitvalue = 1 THEN BIT_SET(1,bit)
140 NEXT bit
150 GO TO 100
>
```

6.5.4.4 DIGITAL_IN - Reads a digital input or output channel.
FORM: DIGITAL IN(chan, val [,flag])

or

DIN(chan, val [,flag])

This statement reads a digital input channel. If argument flag is specified and the channel supports interrupts, the statement will operate asynchronously.

The chan argument can be a variable or constant of type integer, real, or string in the range 0 to 127 for DX11 and PX11 system.

The val argument can be a variable of type integer, real, or string or an integer or real array. The flag argument can be a variable of type integer or real.

In the case where argument flag is specified in the statement, argument val is restricted to being of type integer.

When val is an array and flag is not specified, the array is filled with input values from the channel. When val is an array and flag is specified, then only one value is read and stored in the first element of the array.

EXAMPLE:

The sample program below duplicates the program presented in Section 6.3.6 (copies the input channel bits to the output channel):

```
>listnh <ret>
100 DIGITAL_IN(0,value)
110 DIGITAL OUT(1,value)
120 GO TO 100
130 END
>
```

The following program reads 256 values from an input channel, and stores the data into an array called datavalues:

```
>listnh <ret>
100 input_channel = 2
110 DIM datavalues(255)
120 DIGITAL_IN(input_channel,datavalues())
>
```

The following program reads channels 3 to 6 and puts their values, as 16-bit decimal numbers, into the real array called d_data:

```
>listnh <ret>
100 DIM d_data(10)
110 FOR in_chan = 3 TO 6
120 DIGITAL_IN(in_chan,d_data(in_chan))
130 NEXT in_chan
>
```

6.5.4.5 DIGITAL_OUT - Writes to a digital output channel.

FORM: DIGITAL OUT(chan, val [,flag])

or

DOT(chan, val [,flag])

This statement writes to a digital output channel. If argument flag is specified and the channel supports interrupts, then the statement will operate asynchronously.

The chan argument can be a variable or constant of type integer, real, or string in the range 0 to 127 for DX11 and PX11 system.

The val argument can be a variable or constant of type integer, real, or string or can be an integer or real array. The flag argument can be a variable of type integer or real.

In the case where argument flag is specified in the statement, argument val is restricted to being of type integer.

When argument val is an array and argument flag is not specified, the entire array is output to the channel. When argument val is an array and argument flag is specified, only the first element of the array is output to the channel.

EXAMPLE:

The following sample program outputs the values 0 to 256 to digital output channel one.

```
>listnh <ret>
100 FOR count = 0 to 256
110      DIGITAL_OUT(1,count)
120 NEXT count
130 END

>
```

The program below outputs the values 0 to 256 to digital output channel 2 using an array.

```
>listnh <ret>
100 DIM values(256)
110 FOR count = 0 to 256
120      values(count) = count
130 NEXT count
140 DIGITAL OUT(2,values())
150 END

>
```

6.5.5 TIME AND TIMING STATEMENTS:

6.5.5.1 CLOCK_OUT - Operates the 1601GPT real-time clock and the on-board PICOBASYS timer.

FORM: CLOCK_OUT(rate, val [,flag])

Argument rate is a constant or variable of type integer, real, or string in the range 0 to 7. Valid values for the argument rate are given in the table below.

Argument val is a constant or variable of type integer, real, or string that specifies the number of time units given by argument rate that are to elapse before the event is triggered. Its value must be in the range 1 to 32767 (2 to 32767 on PICOBASYS).

The flag argument must be a variable of type integer or real.

RATE	MEANING
0	Stop
1	1 MHz
2	100KHz
3	10 KHz
4	1 KHz
5	100 Hz
6	EXT CLK (user supplied)
7	EVENT CLK (60 Hz), (10 Hz on PICOBASYS)

When the optional flag argument is not present, the real-time clock operates in Mode 3 of the 1601GPT (recurring mode) and pulses are generated at the frequency specified by argument rate divided by argument value. The interval is reloaded each time the pulse occurs. When the optional flag argument is present, the real-time clock operates in Mode 2 and a single pulse and an interrupt are generated after the specified time interval. The flag variable is set to indicate completion when the interrupt is fielded by I/OBASIC. In interrupt mode the clock must be reloaded after each interrupt if it is to keep running.

EXAMPLE:
The sample program below will set the clock to run at 2.5 KHz (10 KHz/4) for 10 seconds and then stop.

```
>listnh <ret>
100 CLOCK_OUT(3,4)
110  WAIT(10)
120 CLOCK_OUT(0,0)
130 END
>
```

The following program will set the clock to interrupt after 200 milliseconds and then wait for the interrupt.

```
>listnh <ret>
100 CLOCK_OUT(4,200,flag)
110 IF flag = 0 THEN 110 \ REM Wait for interrupt
120 PRINT "200 milliseconds have elapsed"
130 END

>
```

6.5.5.2 GET_DATE / SET_DATE - Gets and sets the system date.

FORM: GET_DATE(month, day, year)

The GET_DATE statement returns the current date. The month is returned in argument month, the day in argument day, and the year in argument year. The year is returned as two digits only, for example 83 is returned for 1983). All three arguments must be variables of type integer, real, or string.

EXAMPLE:

The following program displays the current date:

```
>listnh <ret>
100 GET_DATE(month,day,year)
110 PRINT "The month is ";month
120 PRINT "The day is ";day
130 PRINT "The year is ";year
140 END
>
```

FORM: SET_DATE(month, day, year)

The SET_DATE statement sets the date to that specified by the arguments month, day, and year. All three arguments must be constants or variables of type integer, real, or string. The argument month is in the range 1 to 12. The argument day is in the range 1 to 31. The argument year is in the range 72 to 103 (for 1972 to 2003).

NOTE

Although each argument must be within the range specified, the date is not checked for validity. For example, the date could be set to 31-FEB-83 by the SET_DATE (2, 31, 83) statement.

EXAMPLE:

The sample program below asks for the month, day, and year and then sets the date to that entered.

```
>listnh <ret>
100 PRINT "Month ";
110 INPUT month
120 PRINT "Day ";
130 INPUT day
130 PRINT "Year ";
140 INPUT year
150 SET_DATE(month,day,year)
160 END

>
```

The following example of the GET_DATE and SET_DATE statement updates by one day (caution:it does not account for leap years):

```
>listnh <ret>
100 GET_DATE(month,day,year)
110 day = day + 1
120 FOR temp = 1 TO month
130   READ days
140 NEXT temp
150 IF day <= days THEN 180
160 day = day - days
170 month = month + 1
180 IF month <= 12 THEN 210
190 month = month - 12
200 year = year + 1
210 SET_DATE(month,day,year)
220 DATA 31,28,31,30,31,30,31,31,30
230 DATA 31,30,31
230 END
>
```

6.5.5.3 GET_TIME / SET_TIME - Gets and sets the system time.

FORM: GET_TIME(val)

This statement returns the system time.

The val argument must be a variable of type integer, real, or string. Val is returned with the number of seconds past midnight (fractional seconds are returned if val is a real variable).

FORM: SET_TIME(val)

This statement sets the time in seconds past midnight.

The val argument can be a constant or variable of type integer, real, or string. The argument contains the number of seconds past midnight (fractional seconds can be specified for reals).

NOTE

If argument val is of type integer or string and the time is greater than 65,535 seconds past midnight a numeric overflow error will result. To avoid this, specify argument val as a real variable or constant.

SPECIAL FORMS for PICOBASYS:

```
GET_TIME(hh, mm, ss)
SET_TIME(hh, mm, ss)
```

These statements return or set system time on the battery backed clock in hours (hh), minutes (mm), and seconds (ss) as indicated. Variables used may be of type integer or real (fractional seconds can be specified for reals).

EXAMPLE:

The sample program below determines the execution time for the statement at line 170 (x = x + 1):

```
>listnh <ret>
100 GET_TIME(start)
110 FOR y = 1 TO 1000
120 NEXT y
130 GET_TIME(finish)
140 overhead = finish - start
150 GET_TIME(start)
160 FOR y = 1 TO 1000
170 x = x + 1
180 NEXT y
190 GET_TIME(finish)
200 time = finish - start - overhead
210 PRINT time;" milliseconds per instruction"
220 END

>
```

6.5.5.4 TIME_OUT - Starts a timer channel.

FORM: TIME_OUT(chan, val [,flag])

The chan argument identifies the timer request (a maximum of 8 timer requests may be outstanding). The chan argument can be a constant or variable of type integer, real, or string. It must be within the range of 0 to 7.

The val argument is a constant or variable of type integer, real, or string. It contains the number of seconds (reals may contain fractional seconds) for the timer request. The minimum resolution is 1/60th of a second.

The flag argument must be a variable of type integer or real. If the flag argument is specified, the I/OBASIC program continues execution and the flag variable is signalled when the time expires. If the flag argument is not supplied, the program is suspended until the timer expires (this mode is identical to using the WAIT statement below).

EXAMPLE:

The example below reads the value of the high-level analog input channel entered every 60 seconds:

```
        >listnh <ret>
        100 PRINT "Enter Analog Input Channel ";
        110 INPUT channel
        120 TIME_OUT(1,60,timer)
        130 ANALOG_IN(channel,value)
        140 PRINT value
        150 IF timer = 0 THEN 150
        160 GO TO 120
        170 END
        >
```

The following program reads high-level analog input channel 1 every 20 seconds and reads digital input channel 4 every 45 seconds:

```
        >listnh <ret>
        100 digtimer = 1 \ antimer = 1
        110 IF digtimer = 1 THEN 140 \ REM wait for timers
        120 IF antimer = 1 THEN 180
        130 GO TO 110
        140 DIGITAL_IN(4,digvalue)
        150 PRINT digvalue
        160 TIME_OUT(1,45,digtimer)
        170 GO TO 110
        180 ANALOG_IN(1,anvalue)
        190 PRINT anvalue
        200 TIME_OUT(2,20,antimer)
        210 GO TO 110
        220 END

        >
```

Note that the digital and analog service routines at lines 140 and 180 respectively use different timer channels (1 and 2) since they are timed at different rates.

6.5.5.5 WAIT - Causes a program wait.

FORM: WAIT(val)

The WAIT statement delays program execution for a specified number of seconds. This serves the same function as SLEEP statements found in some versions of BASIC.

The val argument specifies the number of seconds the program is to wait. Val is a constant or variable of type integer, real, or string and contains the number of seconds to wait. Fractional seconds may be specified with real arguments.

EXAMPLE:

The example below duplicates the previous example using the WAIT statement. It reads an analog input channel every 60 seconds:

```
>listnh <ret>
100 PRINT "Enter Analog Input Channel ";
110 INPUT channel
120 WAIT(60)
130 ANALOG_IN(channel,value)
140 PRINT value
150 GO TO 120
160 END

>
```

6.5.6 MISCELLANEOUS STATEMENTS:

6.5.6.1 CHAR_IN - Reads characters from serial channels.

FORM: CHAR_IN(chan, var)

The CHAR_IN statement will read one or more characters from a serial channel. This statement can be used to read characters already contained in the input buffer for a serial channel. The CHAR_IN statement is different from the INPUT @ and LINPUT @ statements, since it does not wait for input, but rather reads one or more characters already typed at the serial channel specified. It is therefore useful for polling serial channels for input, without causing the program to 'hang' at one spot.

When the CHAR_IN statement is executed, all characters subsequently typed at the serial channel will not be echoed until an INPUT @ or LINPUT @ statement is executed for that serial channel (or INPUT and LINPUT statements for the console serial channel).

Argument chan is a variable or constant of any type that contains the serial channel number. The value of this argument must be supplied by the program. Legal range is 0 to 7 for DX11 and PX11 system. Note that serial channel number zero is the console terminal.

The CHAR_IN statement acts differently depending upon whether the type of argument var is numeric (integer or real) or string.

If argument var is of type string, all characters currently residing in the input buffer for the serial channel are placed in the string. More than one character can therefore be read using a string variable. If no characters are present in the input buffer when the CHAR_IN statement executes a null string will be returned.

If argument var is of type integer or real, only one character will be read from the input buffer. Its ASCII code equivalent will be returned in the numeric variable. If no characters are present in the input buffer when the CHAR_IN statement executes, a -1 will be returned.

EXAMPLE:

```
>listnh <ret>
10 REM poll serial channels and send input to console
20 FOR term_no = 1 to 3
30 CHAR_IN(term_no,a$)
40 IF a$ = "" GO TO 60
50 PRINT "Channel ";term_no;" ";a$
60 NEXT term_no
70 END
>
```

6.5.6.2 CONVERT_OCTAL - Converts octal and decimal values.

FORM: CONVERT_OCTAL(arg1, arg2)

The CONVERT_OCTAL statement converts values from decimal to octal and vice versa. The decimal argument is specified as an integer or real and the octal argument is specified as a string. The input argument arg1 is converted to output argument arg2.

EXAMPLE:

The following example demonstrates decimal to octal conversion:

```
>listnh <ret>
100 PRINT "Enter decimal value ";
110 INPUT value
120 CONVERT_OCTAL(value,octal$)
130 PRINT "Octal equivalent is ";octal$
140 GO TO 100
150 END

>
```

The following example demonstrates octal to decimal conversion:

```
>listnh <ret>
100 PRINT "Enter octal value ";
110 LINPUT octal$
120 CONVERT_OCTAL(octal$,value)
130 PRINT "Decimal equivalent is ";value
140 GO TO 100
150 END

>
```

6.5.6.3 PEEK - Reads a memory location.

FORM: PEEK(address, val)

This statement reads a memory location. It returns the value of the memory location specified in argument address. Argument val contains the value of the location. The argument address must be even (or word aligned). The address argument can be a constant or a variable of type integer, real, or string. Argument val must be a variable of type integer, real, or string.

The statement is useful for access to devices in the I/O page or to access specific monitor or other memory locations.

NOTE

If the address specified does not exist, the "?Hardware not present" error message is generated.

The following program displays the current RT-11 monitor version number and release level (by PEEKing at the appropriate monitor offsets):

```
>listnh <ret>
100 PEEK("54",rmon_base)
110 PEEK(rmon_base + OCT("276"),value)
120 sysver = INT(value / 256)
130 sysupd = value - (sysver * 256)
150 PRINT "RT-11 Version",sysver
160 PRINT "Release Level",sysupd
170 END
```

6.5.6.4 POKE - Writes to a memory location.

FORM: POKE(address, val)

The POKE statement is used to write to specific memory locations. The argument address specifies the address of the location to be written. The val argument specifies the value to be written.

The address and val arguments can be constants or variables of type integer, real, or string.

The example below will prompt for a string and then output the string to the terminal. The string is output without the use of the PRINT statement.

```
>listnh <ret>
100 PRINT "Enter a string";
110 LINPUT a$
120 FOR y% = 1% TO LEN(a$)
130   char% = ASC(SEG$(a$,y%,y%))
140   GOSUB 1000
150 NEXT y%
160 PRINT
170 STOP
1000 PEEK("177564",output_csr%)
1010 temp% = output_csr% / 128%
1020 IF temp%/2% = temp% / 2 THEN 1000
1030 POKE("177566",char%)
1040 RETURN
>
```

6.5.6.5 SET_ANALOG_PERCENT / SET_ANALOG_VOLTS - Sets the analog I/O engineering units.

FORM: SET_ANALOG_PERCENT

FORM: SET_ANALOG_VOLTS

These statements will set the analog input and output engineering units to either percent full-scale, or volts. All subsequent values specified in analog input or output commands will be in volts or percentages accordingly.

In PERCENT mode, the voltages will be expressed as the percentage of the full scale (maximum voltage). The values are 0 - 100 for unipolar operation and -100 to 100 for bipolar operation. In VOLTS mode, the voltages will be expressed in volts.

For example, if the analog input device is set for bipolar operation with a range of 10 volts and the analog input channel is fed -5 volts, then the ANALOG_IN statement will give a value of -50 in PERCENT mode and -5 in volts mode.

The default setting for I/OBASIC is PERCENT. The default setting is restored when a RUN command is issued.

6.5.6.6 TEST_ADDRESS - Tests for a valid bus address.

FORM: TEST_ADDRESS(address, val)

This statement tests whether the address specified by argument address is present on the bus. The argument address is a constant or variable of type integer, real, or string. The val argument must be a variable of type integer, real, or string. Val must be an even number. Val is returned as a zero if the address does not respond and a one if it does.

The following program outputs the addresses that exist in the I/O page (addresses 160000 to 177777 octal).

```
>listnh <ret>
100 address = OCT("160000")
110 stop_address = OCT("177776")
120 TEST_ADDRESS(address,temp)
130 IF temp = 0 THEN 160
140 CONVERT_OCTAL(address,address$)
150 PRINT address$
160 IF address = stop_address THEN 190
170 address = address + 2
180 GO TO 120
190 END
>
```

6.5.7 ERROR PROCESSING

The ON ERROR GOTO statement enables error processing within an I/OBASIC program. When error processing is enabled and a program error occurs (such as division by zero) the program will branch to the line number specified in the ON ERROR GOTO statement. Error processing enables you to correct for some fatal and nonfatal error conditions that could occur within a program.

When error processing is desired in an I/OBASIC program, the ON ERROR GOTO statement is generally placed at the beginning of the program. Specifying a line number of zero will disable error processing. By default, error processing is disabled when a program is run.

The error processing routine that starts at the line number given in the ON ERROR GOTO statement can make use of the ERL and ERR functions to determine the type and location of a program error. In general, the error processing routine is specific to a given program, and it is written to process only certain types of errors that would be likely to occur because of the programs particular application (such a frequent numeric overflows during calculations).

The RESUME statement is used to exit the error processing routine and either retry the program statement that caused the error or resume execution at another line number.

6.6 ADDITIONAL INFORMATION

The following sections provide general information relevant to the statements described above.

6.6.1 DEFAULTS FOR SET STATEMENTS

The following are the default SET configurations when the I/OBASIC RUN command is issued:

SET_AIN_GAIN(3)	- high-level analog gain = 1
SET_AINL_GAIN(7)	- low-level analog gain = 1
SET_AIN_NOSCAN	- channel scanning disabled.
SET_AINL_NOSCAN	- channel scanning disabled.
SET_AIN_NOTRIGGER	- external triggering disabled.
SET_AINL_NOTRIGGER	- external triggering disabled.
SET_AOT_NOTOGGLE	- channel toggling disabled.
SET_AOT_NOTRIGGER	- external triggering disabled.
SET_ANALOG_PERCENT	- analog units set to percent.
SET_THERMOCOUPLE(1)	- thermocouple is type J.

These default settings can be altered using the appropriate SET statement. The default settings are restored each time a RUN command is issued so each program that requires changes to the default must execute the appropriate SET statement. The SET configurations can be changed as many times as desired within a program, but a SET statement should not be issued when I/O is active on the device that the SET statement affects.

6.6.2 NULL ARGUMENTS

I/OBASIC real-time control statements cannot be passed null arguments. If null arguments are passed to any statement, the results will be unpredictable.

The following program statement contains a null argument for the chan argument, and is therefore illegal:

```
        ANALOG_IN( , val)        *will NOT work*
```

6.6.3 FLAG ARGUMENT

The flag argument is optional in all of the real-time control statements that support it. The argument controls whether the statement operates synchronously or asynchronously.

When the flag argument is not specified, the statement will not return control to the next I/OBASIC statement until all required operations have been completed. When the flag argument is specified, the next I/OBASIC statement is executed after the statement has initiated (but not necessarily completed) the operation.

When a statement specifies the flag argument, the value of the variable is set to zero. After the statement has completed its operation (for example, all values have been read from the analog input channels) the flag variable is set to a one. Therefore, a program may execute the statement, continue processing subsequent statements, and then loop until the flag variable is set to a one (using an IF THEN statement).

CAUTION

When real-time control statements are operated asynchronously the values of the arguments that are specified with the statement must not be altered or read until the statement has completed all of its processing. This is indicated by the flag argument being set to a one.

For example, if you issue the statement ANALOG_IN(1,value,flag) in a program, the control is passed back to the next statement in your I/OBASIC program immediately. If the next line of the program uses the variable value, the program will not work because the ANALOG_IN statement has not finished updating argument value from the high-level analog input device. Therefore, your program must wait for the ANALOG_IN statement to complete before reading or modifying the arguments value or flag. The way to determine if the statement has completed is to check the flag variable. Argument flag will be zero if the statement is not completed yet; argument flag will be one when the statement has finished. A simple way to wait for the statement to complete is to use an IF/THEN statement that checks if argument flag has been set to one, as in the following example:

```
>listnh <ret>
100 ANALOG_IN(1,value,flag)
110 IF flag = 0 THEN 110 \ REM wait for flag to be 1
120 REM continue on with rest of program.
>
```

It should be apparent, however, that the program above does not do anything different than the following program lines that operate in programmed I/O mode:

```
>100 ANALOG_IN(1,value)
>110 REM continue on with rest of program.
```

So, when should interrupt mode be used?

Let's assume that we need to write a program that will read the high-level analog input and the digital input at the same time. We write a program as follows:

```
>listnh <ret>
100 ANALOG_IN(1,analog_value)
120 DIGITAL_IN(3,digital_value)
130 REM continue with rest of program
>
```

This does not quite do the trick because the two inputs are not really being read at the same time. What is happening is that the analog input is read and the entire ANALOG_IN routine must complete before the digital input is read. What is needed is a way to start the DIGITAL_IN routine before the ANALOG_IN routine finishes. This can be implemented using interrupt mode.

```
>listnh <ret>
100 ANALOG_IN(1,analog_value,analog_flag)
110 DIGITAL_IN(3,digital_value%,digital_flag)
120 REM wait for both routines to finish
130 IF analog_flag = 0 THEN 130
140 IF digital_flag = 0 THEN 140
150 REM both routines are finished
160 REM proceed with program.
>
```

The above program allows both ANALOG_IN and DIGITAL_IN statements to operate asynchronously at the same time, so that it is not necessary for one real-time statement to complete before starting another one.

Another application of interrupt mode is in time-critical situations. Take the program below:

```
>listnh <ret>
100 FOR channel = 1 TO 10
110     ANALOG_LOW_IN(channel,value)
120     PRINT "The value of" ;channel;" is ";
130     PRINT value
140 NEXT channel
150 END
>
```

Assume that, for some reason, it is necessary to speed up the above program. One way to do this is to use asynchronous operation. Looking at the program, it can be seen that the PRINT statement at line 120 could be executed at the same time as the ANALOG_LOW IN statement. The result of line 120 is not affected in any way by the result of the analog input, thus, there is no reason to wait for the ANALOG_LOW_IN to finish before executing the PRINT statement.

```
>listnh <ret>
100 FOR channel = 1 TO 10
110     ANALOG_LOW_IN(channel,value,flag)
120     PRINT "The value of ";channel;" is ";
130     REM wait for analog input statement to complete
140     IF flag = 0 THEN 140
150     PRINT value
160 NEXT channel
170 END

>
```

CHAPTER 7

DATA FILES AND VIRTUAL ARRAYS

DX11 and PX11 I/OBASIC programs can write data permanently onto a storage volume, as well as read data that has been previously stored. This data is stored in RT-11 formatted files on the storage volume. The size of the storage volume is the only limit to the number of files or the amount of data in each file that can be stored.

There are two different kinds of I/OBASIC data files; sequential files, and virtual array files.

A sequential file is treated in the same way as the console terminal is for INPUT, LINPUT, and PRINT statements. The INPUT # and LINPUT # statements can be used to read data from a sequential file, and the PRINT # statement can be used to write data to a sequential file.

A virtual array file is treated in the same way that I/OBASIC treats arrays in memory. An array is declared to be a virtual array by using the DIM # statement. The only difference between a virtual array and a memory array is that virtual array values are always stored in a file on a storage volume, whereas memory array values are stored directly in computer memory. Virtual arrays can therefore be much larger than memory arrays, but they also will take more time for an I/OBASIC statement using them to execute.

A special feature of BASYS is the support for virtual arrays with the ANALOG_IN, ANALOG_LOW_IN, and ANALOG_OUT statements. This feature allows direct transfer of analog channel values between the analog I/O hardware and an I/OBASIC virtual array. There are certain restrictions in using this feature. See the section 'Virtual Arrays with AIN, AINL, and AOT' in this chapter for more information on this feature.

NOTE

> PX11 & DX11 treats semiconductor memory as a disk, and therefore virtual arrayson the systemsare actually stored in semiconductor memory. Do not confuse this 'disk' memory with the normal system memory used for nonvirtual arrays. There are still virtual and nonvirtual arrays in both systems.

7.1 OPENING A FILE

Before a sequential file or virtual array file can be used, you must open the file by associating the file with a channel number. This is done by using the I/OBASIC OPEN statement. The open statement can either open an existing file or create a new file. The format of the OPEN statement is:

```
OPEN string [FOR INPUT] AS FILE [#]expr

or

OPEN string [FOR OUTPUT] AS FILE [#]expr
```

where:

string is a BASYS file specification. It can be either a string constant, variable, or expression.

FOR INPUT specifies opening an existing file. This means that you can only read (INPUT) from the file.

FOR OUTPUT specifies creating a new file. For sequential files, this means that you can only write (PRINT) to the file. For virtual array files, you can either read to or write from the file.

expr is the channel number of the file. The channel number can have any integer value between 1 and 12.

The following are three examples of using the OPEN statement for sequential and virtual array files:

```
>100 OPEN "DATA1" FOR INPUT AS FILE #1


>100 my_file$ = "JOHN1"
>110 OPEN my_file$ FOR OUTPUT AS FILE #3


>100 OPEN "JUNE.DAT" FOR INPUT AS FILE #file_number
```

7.2 CLOSING A FILE

The CLOSE statement is used to close a file once it has been opened. This is generally done at the end of a program to insure that the data in the file will not be lost, but it can also occur anywhere in a program. For example, a file that has been opened for output and then filled with data can then be closed and reopened for input within the same program.

The format of the CLOSE statement is:

```
        CLOSE [ [#]expr, [#]expr,...]
```

where expr specifies the channel number of a file to be closed. If expr is not specified, then I/OBASIC will close all open files.

The following are several examples of the CLOSE statement:

```
        >100 CLOSE


        >100 CLOSE #1, #3


        >100 CLOSE #file_no
```

If a program opens but does not close files, I/OBASIC will close all files when a CHAIN or END statement executes, or when the highest program line number executes.

7.3 USING SEQUENTIAL FILES

I/OBASIC programs access data stored in sequential files using the INPUT #, LINPUT #, and PRINT # statements. The data is always accessed serially, which means that each data item can only be read or written once while the file is open. The file must be closed and reopened to read a data item again.

The INPUT # and LINPUT # statement will read data from a sequential file opened for input. Data is read from the file in the same way that it is read from the terminal.

The PRINT # statement writes data to a sequential file opened for output. I/OBASIC writes the data in the files in the same format it would use to print it on the console terminal.

The following is a sample program that will write data to an output file:

```
>listnh <ret>
10 OPEN "TEST1" FOR OUTPUT AS FILE #2
20 PRINT #2, "This is a test output file"
30 FOR count = 1 TO 20
40   PRINT #2, count
50 NEXT count
60 CLOSE #2
70 END

>
```

The following sample program will read the data file that was written by the above program:

```
>listnh <ret>
10 OPEN "TEST1" FOR INPUT AS FILE #4
20 INPUT #4, string$
30 PRINT string$
40 FOR loop = 1 TO 20
50   INPUT #4, number
60   PRINT number
70 NEXT loop
80 CLOSE
90 END

>
```

7.4 CHECKING FOR THE END OF AN INPUT FILE

The IF END # statement can be used to test if the end of an input file has been reached. The format of the statement is:

```
IF END [#]expr THEN {statement | linenumber}
```

where expression is the channel number of the file to be tested.

If, in the course of using INPUT # or LINPUT # statements, the end of the input file has been reached, then the IF END # statement can be used to either execute a statement or transfer control to another program line. The following sample program operates the same as the one given in the preceding section to input data from a file. However, this program uses the IF END # statement:

```
>listnh <ret>
10 OPEN "TEST1" FOR INPUT AS FILE #4
20 IF END #4 THEN 80
25 INPUT #4, string$
30 PRINT string$
40 IF END #4 THEN 80
50   INPUT #4, number
60   PRINT number
70 GO TO 40
80 PRINT "End of file reached."
90 CLOSE
100 END

>
```

7.5 RESTORING A FILE TO THE BEGINNING

The RESTORE # statement will reset the data pointer for a specified input file to the beginning. The format of the statement is:

```
RESTORE #expr
```

where expr is the channel number of the file to be restored.

This statement is useful for reading a file a multiple number of times, since the file does not have to be closed and reopened each time. The statement operates in a similar manner to the RESTORE statement that is used to reset the READ pointer for data contained in DATA statements.

7.6 USING VIRTUAL ARRAY FILES

Virtual array files are used when random access of data is desired, or when arrays are too large to fit in memory. Virtual arrays are also supported with the ANALOG_IN, ANALOG_LOW_IN, and ANALOG_OUT statements.

Virtual array files have several advantages over sequential files:

1. They can be accessed randomly, rather than sequentially. Any element in a virtual array is accessed in the same time as any other element.

2. Data stored in virtual array files is in binary format, and is not converted to ASCII as for sequential files. Therefore, there is no loss of precision caused by data conversion.

3. You can simultaneously write to and read from a virtual array file without having to OPEN and CLOSE it.

Virtual array files also have several advantages over arrays stored in memory:

1. Virtual array files can be much larger than arrays stored in memory.

2. Data is permanently stored in a virtual array, whereas data in a memory array is lost whenever a new program is run, or the system is turned off.

Virtual arrays also have several restriction, which do not apply to arrays in memory:

1. Accessing elements in a virtual array is slower than a memory array because the data must be read from the file first.

2. Virtual string arrays cannot have dynamic lengths, but must have a preallocated maximum length, which is specified in the DIM # statement.

3. Only one virtual array can be declared in each DIM # statement.

4. Virtual arrays or virtual array elements cannot be passed as arguments to certain of the real-time control statements. The statements that restrict this usage are:

```
ANALOG_IN
ANALOG_OUT
BIT_CLEAR
BIT_SET
BIT_TEST
CHAR_IN
CLOCK_OUT
CONVERT_OCTAL
DIGITAL_IN
DIGITAL_OUT
GET_DATE
GET_TIME
ON ERROR GOSUB
ON EVENT GOSUB
PEEK
POKE
SET_AIN_GAIN
SET_AINL_GAIN
SET_DATE
SET_TIME
TEMPERATURE_IN
TEST_ADDRESS
TIME_OUT
WAIT
```

Exceptions to the above list are the ANALOG_IN, and ANALOG_OUT statements, which permit virtual arrays to be used for the val argument.

7.7 DIMENSIONING VIRTUAL ARRAYS

To use a virtual array, you should include a DIM # and an OPEN # statement in your program. After the virtual array file is opened, the elements of the array can be used in the same way as elements of an array in memory.

The format of the DIM # statement for declaring a virtual array is:

```
DIM #integer1, array [=integer2]
```

where:

integer1 is a constant that specifies the channel number of the virtual array file to be used.

array is any one or two-dimensional array name. It has the same format as in the DIM statement.

integer2 is an optional constant that specifies the maximum length for elements in a virtual string array. It must be in the range 1 to 255.

To access data in an existing virtual array file, ensure that the DIM # statement specifies the same data type and subscripts that are specified in the program that created the file.

The following is a sample program that creates a virtual array file, and then writes values read from an analog input channel to each array element:

```
>listnh <ret>
10 DIM #1, virt_array(10000)
20 OPEN "DATA" FOR OUTPUT AS FILE #1
30 FOR index = 1 to 10000
40   ANALOG_IN(4,value)
50   virt_array(index) = value
60 NEXT index
70 CLOSE #1
80 END

>
```

7.8 VIRTUAL ARRAYS WITH AIN AND AOT

Virtual arrays can be used for the val argument in the ANALOG_IN, and ANALOG_OUT statements. This feature permits direct transfer of data between a virtual array and an analog input or output device, allowing for large amounts of data to be input or output at high speed. The following restrictions apply when usingvirtualarrays for thevalargumentinthese statements:

1. An ADAC 1622DMA hardware controller must be present in order to make use of this feature. A 1622DMA controller may be used in DX11 systems if the maximum desired memory for real-time virtual array storage is within the first 256 K bytes (18 bit addressing). The device must be configured with the analog input or output hardware.

2. The virtual array must be present in an extended memory disk (such as XM4: or XM1: for PX11 & DX11

3. The virtual array must be of type integer, and cannot have more than 32,767 elements. The array reserves two bytes for each element to be stored.

4. No engineering unit conversion takes place, so that the unscaled binary analog data is stored in the virtual array. It is easy to convert this data into engineering units. Simply multiply it by the full scale engineering value (e.g., 10 volts) and divide by 2048 for bipolar data or 4096 for unipolar data.

5. The highest virtual array element to be input or output must first be referenced (as in a LET statement) before the AIN, or AOT statements are used withthe virtual array.

The following sample program reads analog input channel zero into a virtual array. A total of 10000 points are read and stored in the virtual array.

```
>listnh <ret>
10 DIM #1, virt_array%(10000)
20 OPEN "XM1:DATA" FOR OUTPUT AS FILE #1
30 virt_array%(10000) = dummy%  REM do before AIN
35 SET_AIN_TRIGGER
40 ANALOG_IN(0,virt_array%(),flag)
50 CLOSE #1
60 END

>
```

The next sample program reads analog input channel 4 into a virtual array. A total of 30000 points are read and stored in the virtual array. After the points are collected, the largest value is found and stored in variable max. The data is stored as unscaled binary data (with values between -2048 and +2047 for bipolar operation).

The ANALOG_IN statement causes analog input data to be transferred directly to disk (actually extended memory), using the 1622DMA direct memory access controller.

```
>listnh <ret>
10 DIM #1, virt_array%(30000)
20 OPEN "XM1:VALUES.DAT" FOR OUTPUT AS FILE #1
30 virt_array%(30000) = dummy%  REM do before AIN
35 SET_AIN_TRIGGER
40 ANALOG_IN(0,virt_array%(),flag)
50 max = -2047
60 FOR index = 1 to 30000
70  IF virt_array%(index) < max THEN
80  max = virt_array%(index)
90 NEXT index
100 PRINT "Maximum value is ";max
110 CLOSE #1
120 END

>
```

CHAPTER 8
FORMATTED PRINTING

I/OBASIC supports a variation of the PRINT statement that can be used to format the output sent to a terminal or a file. This variation is the PRINT USING statement.

The following formats for numbers can be controlled with the PRINT USING statement:

1. Number of digits.

2. Location of decimal point.

3. Inclusion of symbols (trailing minus signs, asterisks, dollar signs, commas).

4. Exponential format.

The following formats for strings can be controlled with the PRINT USING statement:

1. Number of characters.

2. Left-justified format.

3. Right-justified format.

4. Centered format.

5. Extended field format.

The format of the PRINT USING statement is:

```
PRINT USING string, list

or

PRINT #channel, USING string, list

or

PRINT @channel, USING string, list
```

where:

string is a coded format image of the line to be printed. The string is called the format string. If it is a string constant, it must be enclosed in double quotation mark.

list contains the items to be printed.

channel is the serial channel number for the PRINT @ form, and the file channel number for the PRINT # form.

8.1 FORMATTING NUMBERS WITH PRINT USING

The pound sign symbol (#) is used to specify the size of a numeric field. The numbers are left justified within the field. For example:

```
>listnh <ret>
10 alpha = 123
20 beta = 1234
30 PRINT USING "####",alpha
40 PRINT USING "####",beta

>runnh <ret>

 123
1234

>
```

Numbers are rounded when printed with PRINT USING. For example:

```
>listnh <ret>
10 alpha = 123.6
20 beta = 1234.2
30 PRINT USING "####",alpha
40 PRINT USING "####",beta

>runnh <ret>

 124
1234

>
```

A decimal point, as well as digits to the right of the decimal point, can also be printed, as in the following example:

```
>listnh <ret>
10 alpha = 123.6
20 beta = 1234.2
30 PRINT USING "####.#",alpha
40 PRINT USING "####.#",beta

>runnh <ret>

 123.6
1234.2

>
```

If you have not reserved enough digits for a number in the PRINT USING statement, I/OBASIC will print a percent sign (%) before the number and ignore the format for the number as specified in the PRINT USING statement. Also, be sure to include an extra a pound sign (#) in the format specification for the minus sign of negative numbers.

The following special formatting can also be done with the PRINT USING statement:

1. Trailing minus signs can replace preceding minus signs for negative numbers by following the last pound sign with a minus sign, as in: "###.##-".

2. Preceding spaces can be asterisk filled by starting the format field with two astericks, as in: "**###.##".

3. A floating dollar sign can be placed in front of the number by starting the format field with two dollars signs, as in: "$$###.##".

4. Commas can be printed as part of the number by placing a comma in the format field, as in: "#,###,###.##".

5. Numbers can be printed in exponential format by placing four circumflexes at the end of the format field, as in: "###.##^^^^".

8.2 FORMATTING STRINGS WITH PRINT USING

The PRINT USING statement can also format strings. String fields are specified with a starting single quotation mark ('), and optionally followed by a contiguous series of the uppercase letters L, R, C, or E, representing left-justified, right-justified, centered, and extended string fields, respectively.

If a string is larger than its specified string field, I/OBASIC will print as much of the string as will fit, and the rest will be ignored. The only exception is that for extended fields I/OBASIC will print the entire string.

The following are some examples of the PRINT USING statement with strings:

```
>listnh <ret>
10 REM single character field
20 string$ = "ABCDE"
30 PRINT USING " ' ",string$

>runnh <ret>

 A

>old prog1 <ret>

>listnh <ret>
10 REM right-justified field
20 string1$ = "ABCDE"
30 string2$ = "ABC"
40 PRINT USING "'RRRRR",string1$
50 PRINT USING "'RRRRR",string2$

>runnh <ret>

 ABCDE
   ABC

>old prog2 <ret>

>listnh <ret>
10 field$ = "++'CCCC++'EEEE++'LLLL++"
20 INPUT any_string$
30 IF any_string$ = "STOP" THEN 100
40 PRINT USING field$,any_string$
50 GO TO 20
100 END

>runnh <ret>

?ABCD <ret>
++ABCD ++ABCD ++ABCD ++
?ABCDEFG <ret>
++ABCDE++ABCDEFG++ABCDE++
?STOP <ret>

>
```

APPENDIX A

BASYS CONFIGURATION PROGRAMS

A program is provided with PX11 and DX11 systems that can be used to redefine the software control blocks (use with I/O call) to match the hardware configuration. This program, called CONFIG.BAS, allows you to establish what hardware options are present, and what their individual configurations are. Many of the I/OBASIC real-time control statements directly manipulate hardware I/O devices on the system, and therefore they require information on how the hardware is configured.

NOTE

For PX11 and DX11 systems, the CONFIG.BAS program is supplied on the system disk (device SY: XM0: for PX11 and DL0: for DX11).

A.1 PX11 AND DX11 PROGRAM

All PX11 and DX11 systems are shipped with hardware configuration which the I/OBASIC interpreter is compatible with. The information on this configuration is contained within the file CONFIG.CNF. If the hardware configuration is changed, the corresponding information in the CONFIG.CNF file must be modified. The CONFIG.BAS program can be used to modify this configuration file so that it will contain the correct hardware information. The program can also be used to show the current configuration information from the configuration file.

The I/OBASIC interpreter attempts to load the hardware configuration information from the file CNF:CONFIG.CNF, where CNF: is a logical device name. If this file cannot be found, the I/OBASIC interpreter uses the standard, or default, configuration contained within itself (CONFIG.CNF file). Both the distributed CONFIG.CNF file and the I/OBASIC interpreter have the standard hardware configuration. The appendix that follows lists the default, BASYS System hardware configuration.

On PX11 systems the CONFIG.CNF file is located in PROM on device XM0:, and therefore the file cannot be modified. On DX11 systems this file is located on the read/write system disk, and the file can be modified directly.

In order to change the configuration on a PX11 system the following steps are necessary:

1. Use the following RT-11 COPY command to copy the CONFIG.CNF file to the read/write disk XM4:

   ```
   .COPY XM0:CONFIG.CNF XM4:
   ```

2. Create a startup file called START.COM on XM4:. Include the following line in this file:

   ```
   ASSIGN XM4:  CNF:
   ```

   This file can be created using the RT-11 KED editor if you are using a VT-100 terminal, or you can write an I/OBASIC program to create this file using the OPEN and PRINT # statements.

3. Use the OPEN command when running the CONFIG.BAS program to open file XM4:CONFIG.CNF. See instructions on the OPEN command below.

Operation of the CONFIG program is easy and straight forward. It is written in I/OBASIC, and will support the following commands:

1. MODIFY - used to modify the configuration of the configuration file.

2. SHOW - used to display the current configuration of the configuration file.

3. EXIT - used to exit the configuration program. Any open files are automatically closed.

4. HELP - gives a brief list and description of the available commands.

5. SAVE - used to save responses from the MODIFY command. The user is prompted for a file name.

6. OPEN - used to open a configuration file. The active (if any) configuration file is closed and the user is prompted for the file name of another configuration file.

7. @file - used to retrieve commands and responses from a previously saved file.

All commands must be entered in upper case. Comments may be entered on a command line since all inputs preceeded by either a semicolon (;) or an exclamation mark (!) are ignored.

When CONFIG is started, it automatically attempts to open the configuration file CNF:CONFIG.CNF. If this file does not exist the "?File not found" error results and CONFIG cannot be used. To configure a file other than CNF:CONFIG.CNF, use the OPEN command (see below).

A.2 MODIFY

The MODIFY command is used to modify the current configuration of the open configuration file.

The user is asked for the new hardware configuration for the following devices:

1. High-Level Analog Input:

   Base Address
   Vector Address
   DMA Base Address
   DMA Vector Address
   UNIPOLAR or BIPOLAR Operation
   Voltage Range (5 or 10 Volts)

   Up to two ADAC 1023AD High-Level Analog Input cards are supported.

2. Low-Level Analog Input:

   Base Address
   Vector Address
   DMA Base Address
   DMA Vector Address
   UNIPOLAR or BIPOLAR Operation
   Programmable Cold-Junction Enabled

   Up to eight ADAC 4112XXAD Low-Level Analog Input cards can be supported.

2a. Programmable Cold Junction

   When the cold junction compensation option is purchased onthe4112XXAD, softwareselectionofthethermocouple type tobemeasuredisprovided onachanneltochannel basis.

(Caution: A change of thermocouple type or range must be made prior to a conversion request by using the SET_THERMOCOUPLE statement).

To provide even further flexibility, the user is allowed, by jumper selection and the Configuration Program, the ability to make full use of 128 input channels for each 4112XXAD used. The other choice is to software control the cold junction circuit on a channel to channel basis. (See Sec. 5.5 of BASYS Hardware Manual). When the answer is "Yes" to the query "Programmable Cold-Junction Enabled?" the second choice is made.

This choice allows the user to operate the board as a straight millivolt digitizer (with choice of 8 gains) or as a thermocouple digitizer (with choice of 6 thermocouple types and ranges). The configuration is determined by the choice of input statement used - ANALOG_LOW_IN allows straight millivolt digitizing while TEMPERATURE_IN allows thermocouple digitizing. With Programmable Cold-Junction Enabled, only 64 channels of multiplexing can be addressed with one Model 4112XXAD.

If it desired to operate with 128 channels actively connectedto one 4112XXAD,the answer to the query "Programmable Cold-Junction Enabled?" should be NO. See BASYS Hardware Manual, Sec. 5.5, for how to reconfigure the jumpers to accommodate this mode. In this mode, hardware jumpers determine whether the board is to be used as a straight millivolt digitizer (and use the ANALOG_LOW_IN statement) or as a (Thermocouple digitizer and use the TEMPERATURE_IN) statement.

The unit as shipped is configured to allow software control of cold junction and 64 channels of multiplexing per 4112XXAD.

3. Analog Output:

    Channel Address
    UNIPOLAR or BIPOLAR Operation
    Voltage Range (2.5, 5, or 10 Volts)

A maximum of 128 Analog Output channels may be configured.

4. Real-time Clock Output:

    Base Address
    Vector Address

Only one ADAC 1601GPT real-time clock is supported.

5. Digital Input and Output:
   Channel Address
   CSR Address
   Vector Address
   Board Type (INPUT or OUTPUT)

   A maximum of 128 Digital I/O channels may be configured.

NOTE

The digital I/O channel address is the address of the 16-bit data buffer register associated with the board. The CSR address is the control and status register address for the board. Some boards will not have a CSR address (specify it as zero), in which case there is no vector address also. If a board supports more than one 16-bit data buffer, a separate digital channel should be configured for each buffer.

The program will prompt for each of the above parameters. The current value for each parameter is printed in square brackets (such as [170010] for a CSR address). This current value can be selected as the default by simply pressing the return key. If the current value is not appropriate, a new parameter may be entered. All input must be in upper case.

The program checks all responses for validity and prompts only for required parameters (for example, the Vector Address prompt does not appear if no CSR Address has been provided).

Parameters are displayed and entered as follows:

1. Device, Channel, and CSR addresses are in octal.

2. Vectors are in octal.

3. Special fields (such as voltages and polarity) are entered as text strings. UNIPOLAR or BIPOLAR are valid for polarity prompts; 2.5, 5, 10, 2.5 VOLTS, 5 VOLTS, or 10 VOLTS are valid for voltage ranges (unless otherwise restricted); and INPUT or OUTPUT are valid for direction (board type) prompts.

The following warnings may appear during the MODIFY operation:

1. ?CONFIG-F-Invalid xx Address. This occurs when an entered address is not even or within the valid range 160010 to 177776 (octal) or 0. The prompt for an address reappears.

2. ?CONFIG-W-Nonexistent xx Address. This occurs when an entered address if not on the bus. This is only a warning and does not cause the prompt to reappear.

3. ?CONFIG-F-Invalid VECTOR Address. This occurs when an entered VECTOR address is not a multiple of 4 and within the valid range 0 to 774 (octal). The prompt for a vector reappears.

4. ?CONFIG-F-Invalid response. This occurs when an entered text parameter is invalid. The prompt for the parameter reappears.

A.3 SHOW

The SHOW command is used to display the current configuration of the open configuration file.

The SHOW command will display the current values for all parameters described for the MODIFY command above. The SHOW command will only display the active information; nonexistent channels and vectors are not displayed.

The information is displayed on the user's terminal.

NOTE

The message '(Nonexistent)' may appear next to CSR, Base, and DMA addresses to indicate that the hardware is not currently present on the system. This is only an informational message.

A.4 EXIT

The EXIT command is used to exit the configuration program and return control to I/OBASIC. All open files are closed.

WARNING

The in-memory I/OBASIC interpreter is not modified. To use the new configuration, you must either bootstrap your BASYS System or exit and restart the interpreter.

A.5 HELP

The HELP command will display a list and brief description of each CONFIG command. This list and brief description is also displayed when CONFIG is started.

A.6 SAVE

The SAVE command can be used to save the CONFIG dialog. All commands and other input entered by the user are saved in a file. This saved file may later be used as an indirect command file for the CONFIG program.

CAUTION

The SAVEd answer file will probably have to be edited to remove unwanted commands before it can be used as an indirect command file for the CONFIG program.

A.7 OPEN

The OPEN command is used to open the configuration file to be configured. CONFIG.BAS automatically opens the file CNF:CONFIG.CNF for you; therefore, you need not use this command unless you wish to configure a different configuration file.

The command causes the configuration file that is currently open to be closed. Then, the user is prompted for the name of the configuration file to be used in subsequent commands.

Please note:

1. The configuration files opened must exist. CONFIG.BAS checks that the file exists by first opening the file FOR INPUT, closing it, and finally opening if for read/write access. If the file does not exist, the "?File not found" error will occur and CONFIG.BAS must be restarted using the RUN command.

2. CONFIG does not provide a facility for creating new configuration files. The CONFIG.CNF file must be copied (using the RT-11 COPY command) to create new configuration files. These may then be opened using the OPEN command.

3. The only configuration file read by the I/OBASIC interpreter is the CNF:CONFIG.CNF file. All other configuration files must be renamed or copied to CNF:CONFIG.CNF before they can be used by the I/OBASIC interpreter.

A.8 INDIRECT COMMAND FILES

CONFIG supports indirect command files. A command file may be specified using the @file convention and is valid at any prompt. These command files may contain any input that would normally come from the terminal.

All data read from the indirect command file is displayed at the terminal as it is processed. The output is almost identical to what it would be if the user entered the information directly from the terminal.

NOTE

Indirect command files used by CONFIG may NOT be nested. That is, a file cannot invoke another file using '@file'.

A.9 ADDITIONAL INFORMATION

Multiple Control/C's may be used to abort any of the CONFIG commands during operation. CONFIG will return to the CONFIG> prompt to await a new command.

CONFIG automatically enables terminal output before displaying the CONFIG> prompt if it was disabled by typing Control/O.

APPENDIX B

DEFAULT BASYS SYSTEM CONFIGURATION

The suggested default BASYS System hardware configuration is listed below. The software on every BASYS system that is distributed by ADAC Corporation is set to match all I/O hardware purchase. All other hardware options are set to the suggested defaults. The software settings can be changed by the use of a configuration program described in the previous appendix.

NOTE

The suggested default hardware configuration listed below is not the maximum hardware configuration possible. The maximum hardware configuration is outlined in the <u>Introduction</u> chapter of this manual. Additional hardware can be configured by running the CONFIG.BAS or CNFIGM.BAS program as described in the previous appendix.

HIGH-LEVEL ANALOG INPUT #1:	
Base address	17~~6770~~ 4000
Vector address	~~130~~ 70
DMA base address	~~172410~~ 0
DMA vector address	~~134~~
UNIPOLAR or BIPOLAR	BIPOLAR
Range (5 or 10 Volts)	10 VOLTS
HIGH-LEVEL ANALOG INPUTBOARD #2:	
Base address	1740~~00~~ 30
Vector address	370
DMA base address	~~172400~~ 0
DMA vector address	~~374~~
UNIPOLAR or BIPOLAR	BIPOLAR
Range (5 or 10 Volts)	10 VOLTS

LOW -LEVEL ANALOG INPUT #1:

Base address	174010
Vector address	170
DMA base address	~~172420~~
DMA vector address	~~174~~
UNIPOLAR or BIPOLAR	BIPOLAR
Programmable Cold-junction Enabled	~~YES~~

LOW -LEVEL ANALOG INPUT BOARD #2:

Base address	174020
Vector address	370
DMA base address	~~172430~~
DMA vector address	~~74~~
UNIPOLAR or BIPOLAR	BIPOLAR
Programmable Cold-junction Enabled	~~YES~~

ANALOG OUTPUT:

Channel number 0:	
Channel address	4100 17~~2000~~
UNIPOLAR or BIPOLAR	BIPOLAR
Range (2.5, 5, or 10 Volts)	10 VOLTS
Channel number 1:	
Channel address	4102 17~~2002~~
UNIPOLAR or BIPOLAR	BIPOLAR
Range (2.5, 5, or 10 Volts)	10 VOLTS
Channel number 2:	
Channel address	4104 17~~2004~~
UNIPOLAR or BIPOLAR	BIPOLAR
Range (2.5, 5, or 10 Volts)	10 VOLTS

.
.
.

Channel number 31:	
Channel address	172076
UNIPOLAR or BIPOLAR	BIPOLAR
Range (2.5, 5, or 10 Volts)	10 VOLTS

CLOCK OUTPUT:

Base address	170420
Vector address	104

DIGITAL INPUT AND OUTPUT:

Channel number 0:	
Channel address	17~~0010~~ 4242
CSR address	~~0~~ 174240
Vector address	~~200~~ 210
Direction (INPUT or OUTPUT)	OUTPUT
Channel number 1:	
Channel address	17~~0012~~ 4246
CSR address	~~0~~ 174244
Vector address	~~204~~ 214
Direction (INPUT or OUTPUT)	OUTPUT
Channel number 2:	
Channel address	170014
CSR address	0
Vector address	210
Direction (INPUT or OUTPUT)	OUTPUT
Channel number 3:	
Channel address	170016
CSR address	0
Vector address	214
Direction (INPUT or OUTPUT)	OUTPUT
.	
.	
.	
Channel number 15:	
Channel address	170046
CSR address	0
Vector address	274
Direction (INPUT or OUTPUT)	OUTPUT

NOTE

The digital I/O channels are assigned consecutive vectors from 200 to 274, except those for channels 9, 10, and 13. These channels are assigned vectors 144, 150, and 164 respectively. This is done to avoid conflicts with already preassigned system vectors.

NOTE

The vector for digital I/O channel zero (vector 200) conflicts with the RT-11 parallel line printer device (LP). If a DISKBASYS or PROMBASYS system is configured with this device, then the vector for digital channel zero should be reassigned to vector 120, if that digital channel is used for interrupts.

The following are the serial channel addresses for the PX11 & DX11 systems. Although these addresses cannot be changed by a configuration program, they are listed here for reference.

The DX11 and PX11 serial channel addresses are identical and are listed below:

Channel	Base Address	Vector Address
0 (console)	177560	60
1	176500	300
2	176510	310
3	176520	320
4	176530	330
5	176540	340
6	176550	350
7	176560	360

APPENDIX C

I/OBASIC INSTRUCTION TIMES

The following chart lists several I/OBASIC program instruction times. The times listed are in milliseconds. The DX11 and PX11 systems used both a DEC LSI-11/23 CPU with FPU installed, and a DEC LSI-11/73 CPU. The times for the LSI-11/23 CPU are listed on the left under the DX11 and PX11 column, and the times for the LSI-11/73 CPU are listed on the right. The instruction times are only approximate and should not be considered exact.

	DX11 & PX11		
	1123CPU		1173CPU
integer addition	0.63	/	0.32
integer multiplication	0.75	/	0.38
integer division	0.77	/	0.40
real addition	1.13	/	0.58
real multiplication	1.38	/	0.65
real division	1.31	/	0.65
sine function	2.76	/	0.77
GO TO line number	0.18	/	0.08
IF i%=0% THEN line number			
condition true:	0.68	/	0.37
condition false	0.65	/	0.35
CHAR_IN(0%,i%)			
no characters waiting	3.62	/	1.68
CHAR_IN(0%,a$)			
no characters waiting	3.99	/	1.80

APPENDIX D

IMMEDIATE MODE COMMANDS

I/OBASIC allows commands to be entered in immediate mode. This means that statements need not be entered as part of programs; they can be given directly to the I/OBASIC interpreter which will process them immediately.

Most I/OBASIC commands can be used in immediate mode. The commands are entered at I/OBASIC command level, when the prompt (>) is displayed.

Some examples of immediate mode operation are:

```
>print "hello" <ret>
hello

>print 5+6 <ret>
11

>x=2 <ret>
>print x <ret>
2

>ain(1,val) <ret>
>print val <ret>
13.62

>
```

Immediate mode is good for doing quick calculations, trying out commands, and debugging.

One feature of immediate mode is that it allows you to examine the values of your program variables after running your program. After your program finishes execution, simply type PRINT followed by the name of the variable that you are interested in and I/OBASIC will type the value of that variable. For example:

```
>listnh <ret>
10 alpha = 2
20 FOR beta = 1 TO 5
30 alpha = alpha + 2
40 NEXT beta
50 END

>runnh <ret>

>print alpha <ret>
12

>
```

Another useful feature of immediate mode is that the GOTO statement can be used to start your program at any line number you wish. If, instead of typing RUN, you type GOTO <line number>, your program will begin execution at that line number. Note that when you type RUN, all the variables in your program are automatically set to zero. This does not happen when executing a program using an immediate mode GOTO statement. The variables and the defaults do not get reset.

One typical debugging method using immediate mode is to insert STOP statements after parts of your program that you are having trouble with. When the program hits the STOP statement, control will be returned to you. You may then examine any variables using the PRINT statement in immediate mode to see if they are what you expect them to be. The values of any variables in your program may be changed in immediate mode as well. Your program may then be continued by issuing an immediate mode GOTO statement using the line number after the STOP statement in your program.

APPENDIX E

MEMORY CONSIDERATIONS

The BASYS System provides a fixed amount a physical memory for a user-written program to run in. The amount of memory available depends on the exact hardware and processor configuration of the system. Because there is a fixed amount of memory available, you should be aware of several techniques that can be used to gain more usable memory for either additional program statements or data. The following is a list of some techniques that can be used to obtain more usable memory:

1. Eliminate or reduce unnecessary items in a program, such as REM statements and optional keywords, such as LET.

2. If possible, consolidate the variables used in your program so that they are used for multiple purposes.

3. Make maximum use of multiple statement lines.

4. Use the short form of the real-time control statements.

5. Make efficient use of program loops, subroutines, and user-defined functions.

6. Split up large programs into several smaller programs by using the CHAIN or OVERLAY statements. When doing this, you can make use of the memory disk handler so that the CHAINed or OVERLAYed programs are brought into memory quickly.

7. Reduce the size of arrays in memory to the size required (using the DIM statement).

8. Use virtual array files for arrays that are too large to fit into memory. If you wish, you can make use of the memory disk handler, described in another section of this manual, so that virtual array access will be fast.

9. Reduce the number of simultaneously open files by opening a file just before you need it and closing it immediately after the last use.

10. After you delete program lines, store the program with the SAVE command and restore it with the OLD command to further optimize program memory requirements.

11. Use integer arrays, variables, and constants, wherever possible.

12. Use short or single-letter variable names. Each character in a variable name requires one byte of storage for the first occurrence of that variable name within a program. Keeping variable names short will reduce program memory requirements.

APPENDIX F

PROGRAMMING PROMS FOR PX11 SYSTEMS

This appendix describes how to program PROMs containing user developed software for PX11 systems. PX11 systems are supplied with a program, called EPROM.BAS, that is designed to interface with a PROM programmer. The PROM programmer is an optional hardware item available from ADAC Corporation. A user can develop I/OBASIC programs and have them permanently programmed into a PROM memory disk, rather than residing in a battery-backed up CMOS memory disk. The following steps are required to do this:

1. Create a disk image of the software that is to be placed into PROM. For example, if two I/OBASIC programs, called TEST1.BAS and TEST2.BAS are to be placed into PROM, then they should be the only two programs that reside on the disk. Typically, the XM1 or XM4 disk can be used for this.

2. Use the RT-11 SQUEEZE command to move all files on this disk to the physical beginning of the disk.

3. Connect the PROM programmer to a serial line on the BASYS system. This is usually serial line 1. The line must be configured for 1200 baud and SPACE parity.

4. Press KEY on the programmer to turn it on. If you just plugged it in it will say "POWER FAIL". Then press CLR, and the programmer will do a self test. Press CLR again to get it to command level.

5. Select function '0' on the programmer. Press NEXT to see the next value for the current parameter (successive NEXTs can be used to see all possible values for parameters). Press ENTR to store the displayed option and move on to the next parameter. You should set the programmer for 1200 baud with SPACE parity. The remainder of the parameters can have their default values.

6. Press ENTR on the programmer until "SELECT FUNCTION" is displayed to get to command level. Then press REM to enter remote mode. This completes the interaction with the programmer.

7. Run program EPROM.BAS on the PX11 system. The following questions will need to be answered:

   1. Enter the device name to be burned into PROM. You should enter the name of the disk that was created and contains the programs to be placed into PROM (for example XM1:).

   2. Bad directory, please enter device length in blocks. This question is only asked if a non RT-11 structured device is being programmed.

   3. Enter the serial line number of PROM programmer. Usually this is serial line 1. Serial line 0 is the console terminal and cannot be used.

   4. Enter the PROM type (2716, 2732, 2764, 27128, 27256). Select one of the PROM types. The program will then the tell you how many PROM chips will be needed to program the disk.

   5. Enter the socket number you wish to burn (zero to stop). The socket number corresponds to the silk-screen socket numbers (1 to 16) on the ADAC 1822PROM board. The program will then tell you to insert a PROM chip into the programmer. It will then begin programming the PROM chip.

      When programming of a particular PROM chip is complete the console terminal bell will ring and the following message will be printed by EPROM.BAS:

      ```
      Programming complete, remove the PROM
      ```

      Remove the PROM chip. The program will then ask for the socket number of the next chip to be programmed.

The following are the possible error messages that can be produced by program EPROM.BAS:

```
?ERROR - Unknown PROM type
?ERROR - Bad PROM code
?ERROR - Starting address outside device
?ERROR - PROM not erased
?ERROR - Programmer not responding
?ERROR - Down line loading aborted
?ERROR - Programming failure
```

APPENDIX G

AUTOSTARTING DX11 AND PX11

DX11 and PX11 systems can be made to automatically execute a user-written I/OBASIC program upon system power-up. Normally, when the system powers up the I/OBASIC interpreter is waiting for keyboard input commands from the user (the '>' prompt appears).

To set up a PX11 system for executing a user-written I/OBASIC program on power-up, simply create the file START.COM on either XM1:, XM2:, XM3:, or XM4: containing the following lines:

```
RUN SY0:IOBAS
RUN filespec
```

Where filespec is the I/OBASIC program name that should be executed.

To set up a DX11 system for executing a user-written I/OBASIC program on power-up, include the two lines given above in the existing file STARTS.COM located on the system disk.

Adding the above lines can be done using the RT-11 KED editor if you have a VT-100 terminal, or an I/OBASIC program can be written that treats the file as a data file using the I/OBASIC PRINT # and OPEN statements as in the following example:

```
10 OPEN "XM1:START.COM" FOR OUTPUT AS FILE #1
20 PRINT #1, "RUN SY:IOBAS"
30 PRINT #1, "RUN TEST1.BAS"
40 CLOSE #1
50 END
```

APPENDIX H

SUMMARY OF LANGUAGE ELEMENTS

This appendix gives a brief summary of all of the various commands, statements, and functions that are supported in DX11, & PX11, systems. Refer to sections in this manual, the I/OBASIC Language Reference Manual, and the Digital Equipment Corporation RT-11 and RT-11 BASIC manuals for further information on each of these elements.

H.1 RT-11 COMMANDS

RT-11 commands are used to interact with the DISKBASYS and PROMBASYS operating system, called RT-11. The following are the RT-11 commands that are available on DX11 and PX11 systems.

ASSIGN	Assigns a logical device name to a physical device.
COPY	Makes a copy of a file.
DATE	Sets or displays the current system date.
DELETE	Deletes the files you specify.
DIRECTORY	Lists the files stored on a device.
FORMAT	Formats floppy diskettes (DISKBASYS only)
INITIALIZE	Clears and initializes a device directory.
RENAME	Assigns a new name to an existing file.
SQUEEZE	Consolidates free space on a volume.
TIME	Sets or displays the current time of day.
TYPE	Lists a file on the terminal.

H.2 REAL-TIME CONTROL STATEMENTS

Real-time control statements are used within I/OBASIC programs for interacting with external devices. Listed below are the real-time control statements that have been added to the I/OBASIC interpreter for this purpose. Some of these statements have short forms that can also be used. The short forms are indicated in parenthesis:

ANALOG_IN	(AIN) Reads high-level analog input channels.
ANALOG_LOW_IN	(AINL) Reads low-level analog input channels.
ANALOG_OUT	(AOT) Writes to an analog output channel.
BIT_CLEAR	(BIC) Clears a digital output bit.
BIT_SET	(BIS) Sets a digital output bit.
BIT_TEST	(BIT) Tests a digital input or output bit.
CHAR_IN	Reads characters from serial channels.
CLOCK_OUT	Operates the 1601GPT real-time clock.
CONVERT_OCTAL	Converts octal and decimal values.
DIGITAL IN	(DIN) Reads a digital input or output channel.
DIGITAL_OUT	(DOT) Writes to a digital output channel.
EVENT RETURN	Returns from an event processing subroutine.
GET_DATE	Gets the current system date.
GET_TIME	Gets the current system time.
ON EVENT GOSUB	Declares an event processing subroutine.
PEEK	Reads a memory location.
POKE	Writes to a memory location.
TEMPERATURE_IN	(TMPIN) Reads thermocouple temperatures.
TEST_ADDRESS	Tests for a valid bus address.
TIME OUT	Starts a software timer.
WAIT	Causes a program delay.

SET Statements:

SET_AIN_GAIN	Sets the high-level analog gain.
SET_AINL_GAIN	Sets the low-level analog gain.
SET_AIN_NOSCAN	Disables high-level analog channel scanning.
SET_AIN_SCAN	Enables high-level analog channel scanning.
SET_AINL_NOSCAN	Disables low-level analog channel scanning.
SET_AINL_SCAN	Enables low-level analog channel scanning.
SET_AIN_NOTRIGGER	Disables hardware triggering for high-level analog channels.
SET_AIN_TRIGGER	Enables hardware triggering for high-level analog channels.
SET_AINL_NOTRIGGER	Disables hardware triggering for low-level analog channels.
SET_AINL_TRIGGER	Enables hardware triggering for low-level analog channels.
SET_ANALOG_PERCENT	Sets analog units to percent full-scale.
SET_ANALOG_VOLTS	Sets analog units to volts.
SET_AOT_NOTOGGLE	Disables DMA analog output channel toggling.
SET_AOT_TOGGLE	Enables DMA analog output channel toggling.
SET_AOT_NOTRIGGER	Disables hardware triggering for DMA analog output channels.
SET_AOT_TRIGGER	Enables hardware triggering for DMA analog output channels.
SET_DATE	Sets the current system date.
SET_THERMOCOUPLE	Sets the thermocouple type and range.
SET_TIME	Sets the current system time.

H.3 STANDARD BASIC STATEMENTS

The following are the programming statements that are supported by I/OBASIC. These statements are found in the standard Digital Equipment Corporation RT-11 BASIC interpreter:

CHAIN	Loads and executes the program specified.
CLOSE	Closes the file associated with a channel number.
COMMON	Preserves variables when CHAINing between programs.
DATA	Contains data for READ statements.
DEF FN	Define a one-line user function.
DIM	Declares arrays.
DIM #	Declares virtual arrays.
END	Defines the physical end of the program.
FOR NEXT	Sets up a loop to be executed a number of times.
GOSUB	Transfers control to a subroutine.
GOTO	Transfers control to a specified line number.
IF END	Detects end-of-file condition for sequential files.
IF THEN	Conditional branch or execution of a statement.
INPUT	Reads data from the console terminal.
INPUT #	Reads data from the console terminal or a file.
INPUT @	Read data from a serial line.
KILL	Deletes the file.
LET	Assigns the value of an expression to a variable.
LINPUT	Reads an entire record from the console terminal.
LINPUT #	Reads a record from the console terminal or a file.
LINPUT @	Reads a record from a serial line.
NAME	Renames a file.
ON ERROR GOTO	Transfers control after a program error.
ON GOSUB	Transfers control to a subroutine using an index
ON GOTO	Transfers control to a statement using an index.
OPEN	Opens a file and associates it with a channel number.
OVERLAY	Merges/replaces program lines with those from a file.
PRINT	Prints items on the console terminal.
PRINT #	Prints items on the console terminal or a file.
PRINT @	Prints items on a serial line.
PRINT USING	Prints items using special formatting.
RANDOMIZE	Starts a random sequence for the RND function.
READ	Reads data contained in DATA statements.
REM	Inserts comments into the program.
RESTORE	Rewinds the READ/DATA pointer.
RESTORE #	Rewinds a sequential file.
RESUME	Continues execution after on ON ERROR condition.
RETURN	Returns from a subroutine (complement of GOSUB).
STOP	Terminates program execution.

H.4 TERMINAL CONTROL COMMANDS

Terminal control commands are used for controlling the input and output at the console terminal. They are useful for correcting typing mistakes, and regulate the flow of the display at the terminal. All of these commands are single keystroke commands that do not need to be followed by a carriage return.

CTRL/C	Stops program execution.
CTRL/O	Discards further terminal output.
CTRL/Q	Cancels effect of CTRL/S.
CTRL/S	Suspends output to the terminal.
CTRL/U	Deletes the current input line.
DELETE	Erases the last character typed.

H.5 I/OBASIC COMMANDS

The following commands are used when running the I/OBASIC interpreter. They are executed as soon as they are typed, and therefore cannot be part of a program:

APPEND	Merges/replaces program lines with those from a file.
BYE	Returns control to the RT-11 monitor.
CLEAR	Sets variables to zero, deletes strings and arrays.
COMPILE	Saves a compiled version of the program.
DEL	Deletes program lines.
LENGTH	Displays used and available memory in 16-bit words.
LIST	Prints the specified program lines on the terminal.
LISTNH	Same as LIST with header message suppressed.
NEW	Erases the current program and variables from memory.
OLD	Loads a SAVEd program into memory.
RENAME	Changes the current program name.
REPLACE	Writes the program to an existing program file.
RESEQ	Resequences program line numbers.
RUN	Executes the program.
RUNNH	Same as RUN with header message suppressed.
SAVE	Writes the program to a new program file.
SUB	Used to edit program lines.
UNSAVE	Deletes the specified file.

H.6 ARITHMETIC FUNCTIONS

Arithmetic functions return a value when executed in a program line within an I/OBASIC program. The value returned is either of type integer or real, depending upon the function.

ABS	Absolute value.
ATN	Arctangent of angle in radians.
COS	Cosine of angle in radians.
EXP	Exponential; e raised to a power.
INT	Integer value.
LOG	Natural logarithm.
LOG10	Base 10 logarithm.
PI	Value of pi (approximately 3.141593).
RND	Random number between 0 and 1.
SGN	Sign of an expression.
SIN	Sine of angle in radians.
SQR	Square root.
TAB	Horizontal print tab (only for PRINT statement).

H.7 STRING FUNCTIONS

String functions are used for manipulating strings. Some functions return string values, while other functions simply operate on strings and return numeric values. They are very useful when working with character input and output.

ASC	Returns the ASCII value for the 1-character string.
BIN	Converts binary number string to decimal value.
CHR$	Generates a 1-character string of the ASCII value.
CLK$	Returns the time as a string in the form hh:mm:ss.
DAT$	Returns the date as a string in the form dd-mmm-yy.
LEN	Returns the number of characters in the string.
OCT	Converts octal number string to decimal value.
POS	Returns position of a substring.
SEG$	Extracts a substring from a string.
STR$	Returns string representation for the numeric value.
TRM$	Removes trailing blanks from a string.
VAL	Converts decimal number string to decimal value.

H.8 MISCELLANEOUS FUNCTIONS AND STATEMENTS

The following functions and statements perform miscellaneous operations. In general, most programs do not need to make use of these functions, but they are documented here as a reference for the advanced programmer.

ABORT	Terminates a program and removes it from memory.
CANCEL_CTLO	Restores terminal output after a CTRL/O.
CTLC	Checks for CTRL/C typed at console terminal.
DISABLE_CTLC	Disables CTRL/C from interrupting program.
ENABLE_CTLC	Allows CTRL/C to interrupt program execution.
ERL	Returns the line number of a program error.
ERR	Returns the error code of an ON ERROR condition.
SET_WIDTH	Used to set the terminal's margin (width).