HEWLETT PACKARD

HP 98645A Measurement Library User's Library

HP 98645A

Measurement Library

User's Manual

HEWLETT PACKARD

Update 3 (July 1988)
Manual Part Number 98645-90001
E0684
Printed in U. S. A.
June 1984

HEWLETT-PACKARD COMPANY
Roseville Networks Division
8000 Foothills Boulevard
Roseville, California 95678

PRINTING HISTORY

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past updates; however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with the user-inserted update information. New editions of this manual will contain new information, as well as updates.

98645-90001

First Edition	June 1984
Update 1	November 1985
Update 2	December 1987
Update 3	July 1988

MANUAL UPDATE

MANUAL IDENTIFICATION

Title: HP 98645A Measurement Library User's Manual

Part Number: 98645-90001

UPDATE IDENTIFICATION

Update Number: 3 (July 1988)

This update also includes:
Update 2 (December 1987)
Update 1 (November 1985)

This Update Goes With: First Edition (June 1984)

THE PURPOSE OF THIS MANUAL UPDATE
is to provide new information for your manual to bring it up to date. This is important because it ensures that your manual accurately documents the current version of the product.

THIS UPDATE CONSISTS OF
this cover sheet, a printing history page (if any), any replacement pages, and write-in instructions (if any). Replacement pages are identified by the update number at the bottom of the page. A vertical line (change bar) in the outside margin indicates new or changed text material. The change bar is not used for typographical or editorial changes that do not affect the content of the text.

TO UPDATE YOUR MANUAL
identify the latest update (if any) already contained in your manual by referring to the printing history page. Incorporate only the updates from this packet not already included in your manual. Following the instructions on the back of this page, replace existing pages with the update pages and insert new pages as indicated. If any page is changed in two or more updates, such as the printing history page which is furnished new for each update, only the latest page will be included in the update package. Destroy all replaced pages. If write-in instructions are included they are listed on the back of this page.

HEWLETT-PACKARD COMPANY
Roseville Networks Division
8000 Foothills Boulevard
Roseville, California 95678

98645-90001
U0788
Update 3
July 1988

TECHNICAL MANUAL UPDATE

(98645-90001)

Note that "*" indicates a changed page.

UPDATE	DESCRIPTION
3	Replace the following pages with the new pages attached: Title*/ii* All of Section 1.
1 & 2	Replace the following pages with the new pages attached: iii/iv* 2-3/2-4* index-1*/index-2* index-3*/index-4

USING THE LIBRARY

INTRODUCTION

The HP 98645A Measurement Library provides a set of easy-to-use subroutines for taking readings from the HP 98640A Analog-to-Digital Converter (ADC) card. These subroutines can be used from the BASIC or Pascal language systems on HP 9000 Series 200 or Series 300 computers. The subroutines are written in Pascal, and are adapted to the BASIC language with the CSUB utility package. The Measurement Library is compatible with BASIC 2.0, 2.1, 3.0, 4.0, 5.0, 5.1, and Pascal 2.0, 2.1, 3.0, 3.1, 3.2.

The Measurement Library subroutine calls are a superset of the "HP 14751A Computer Aided Test Programming Package for the Model 6944A". BASIC programs written using the HP 14751A routines should be able to use the Measurement Library software with very little modification.

Features

The HP 98645A Measurement Library allows you to:

Take a single reading from any of 8 channels at any of 4 gains.

Take readings by scanning across 1 to 8 channels, any number of times.

Take readings from channels in random order as specified in an address array. Optionally, you can specify the gain and pace interval for each reading, and the readings can be repeated any number of times.

Express readings in three different units:
Base units: binary integer returned from the ADC.
Standard units: base units adjusted for gain and calibration, expressed as real numbers.
User units: standard units times a user multiplier plus a user offset.

Take calibration (zero) readings on a specified channel, and apply that calibration adjustment to all readings.

Re-set gain or units at any time.

Take readings at the full 55 kHz sampling speed of the ADC card from either BASIC or Pascal.

Take readings under interrupt mode in BASIC.

Software Provided

The HP 98645A Measurement Library includes these subroutine packages:
MEAS__LIB for use with BASIC 2.0
MEAS__LIB3 for use with BASIC 3.0
MEAS__LIB4 for use with BASIC 4.0
MEAS__LIB5 for use with BASIC 5.0
MEASLIB42 for use with BASIC 4.0 on a Model 320 computer
MEASLIB52 for use with BASIC 5.0 or 5.1 on a Model 320 computer
INTR2__1 for use with interrupt mode in BASIC 2.1
MEAS__LIB.CODE for use with Pascal 2.0/2.1
MEAS__LIB3.CODE for use with Pascal 3.0/3.1/3.2

MEASLIB32.CODE for use with Pascal 3.1/3.2 on a Model 320 computer.

The software is provided on the following media:
Option #630: 3-1/2" floppy disc
Option #655: 5-1/4" floppy disc

MEAS__LIB and MEAS__LIB.CODE will not use floating point hardware. MEAS__LIB3, MEAS__LIB4, MEAS__LIB5, and MEAS__LIB3.CODE will use a Floating Point Math card if it is installed in the system; otherwise they will use the Pascal floating point library routines. MEASLIB42, MEASLIB52, and MEASLIB32.CODE will use the built-in floating point hardware in the Model 320 computer; these routines are not compatible with any other processor.

THE GENERAL APPROACH

The way you write programs using the Measurement Library is pretty much the same whether you use the BASIC or Pascal language system. There are, however, significant differences in the way you set up your system environment. We will discuss these differences in the next few paragraphs.

Using BASIC 2.1

If you are using the BASIC 2.1 system, take the following steps to get your application up and running:

1) **Boot up BASIC 2.0.**

2) **Load the BASIC 2.1 extensions.** The 2.1 extensions are located on the Extended BASIC 2.1 disc. Insert that disc into the master drive and issue the command LOAD BIN "AP2__1".

3) **Load the interrupt processing package** if you will be taking readings in interrupt mode. (Interrup mode readings are discussed later in this section.) The interrupt processing package is located on th Measurement Library disc. Insert that disc into the master drive and issue the command LOAI BIN "INTR2__1".

4) **Load any other BASIC extensions** that you need for your application. For example, this would b the time to load Graphics 2.1.

5) **Write your BASIC program** or load a previously written program into memory. In the paragraph below we will describe how to write your application program using the Measurement Library.

6) **Load the Measurement Library subroutines** if they are not already part of the program you wrote in the pr-vious step. The subroutines are located on the Measurement Library disc. Insert that disc into the master drive and issue the command LOADSUB ALL FROM "MEAS__LIB".

7) **Run your program.** Debug as necessary (repeating steps 5 through 7).

Using BASIC 3.0, 4.0, 5.0, or 5.1

If you are using the BASIC 3.0, BASIC 4.0, BASIC 5.0, or BASIC 5.1 system, take the following steps to get your application up and running:

1) **Boot up BASIC 3.0, 4.0, 5.0, or 5.1.**

2) **Load the BASIC 3.0, 4.0, 5.0, or 5.1 IO binary** if you will be taking readings in interrupt mode. (Interrupt mode readings are discussed later in this section.) The IO binary is located on the BASIC Language Binary disc. Insert that disc into the master drive and issue the command LOAD BIN "IO".

3) **Load any other BASIC binaries** that you need for your application. For example, this would be the time to load graphics routines.

4) **Write your BASIC program** or load a previously written program into memory. In the paragraphs below we will describe how to write your application program using the Measurement Library.

5) **Load the Measurement Library subroutines** if they are not already part of the program you wrote in the previous step. The subroutines are located on the Measurement Library disc. Insert that disc into the master drive and issue the command:

```
LOADSUB ALL FROM "MEAS_LIB3"  or
LOADSUB ALL FROM "MEAS_LIB4"  or
LOADSUB ALL FROM "MEAS_LIB5"  or
LOADSUB ALL FROM "MEASLIB42"  or
LOADSUB ALL FROM "MEASLIB52"
```

as appropriate for your system. (Refer to the paragraphs on "Software Provided", above, for information on which software goes with which system.)

6) **Run your program.** Debug as necessary (repeating steps 4 through 6).

General BASIC Programming

The Measurement Library subroutines add approximately 23,700 bytes to your BASIC program. The INTR2__1 binary adds approximately 1200 bytes.

Note that integer parameters used in the Measurement Library subroutine calls must be explicitly typed as INTEGER. (You can find out which parameters are integers by looking at the parameter descriptions in the subroutine call listings in Section 2 of this manual.) Real parameters and string parameters (those ending in $) need not be explicitly typed. Literal constants of any type (integer, real, or string) may be used. Note that integers must not contain a decimal point.

You can invoke Measurement Library routines by calling them (CALL statement) or simply by entering them by name. When you use them in an IF . . THEN statement or an ON . . statement, the "CALL" must be explicit.

Using Pascal 2.0, 2.1, 3.0, 3.1, or 3.2

You can call the Measurement Library subroutines from the Pascal language by importing the Measurement Library and using the library subroutines as procedure calls with the syntax described in Section 2 of this manual. Typically, you import the Measurement Library with a compiler directive of

```
    $SEARCH 'MEAS_LIB'$ or
    $SEARCH 'MEAS_LIB3'$ or
    $SEARCH 'MEASLIB32'$
```

and an import statement of

```
    IMPORT measurement_lib;
```

in your code. Importing the Measurement Library adds about 17600 bytes to your Pascal program.

If the Pascal system modules INTERFACE and IO have not been merged into the system library file, you will also have to include the compiler directive

```
    $SEARCH 'INTERFACE.','IO.'$
```

Note that the "." after each file name is significant.

The procedure calls for the Measurement Library are all exported from the file MEAS__LIB.CODE (or MEAS__LIB3.CODE), along with the following types:

```
    TYPE  shortint = -32768..32767;
          byte = 0..255;
          str255 = string[255];
          iarraytype = ARRAY[0..maxint] OF shortint;
          rarraytype = ARRAY[0..maxint] OF real;
          rarraypt = ^rarraytype;
          iarraypt = ^iarraytype;
```

Due to the rigorous structure of the Pascal language, you can't default parameters in the procedure calls. However, to save you the bother of declaring real and integer arrays for the pace and gain array parameters of the random__scan procedure, you can use the default pace or gain value (established by a call to Config__0 or Set__gain) by specifying a 0 for the array size and a NIL for the array pointer. All other parameters for all procedure calls must be explicitly provided in the procedure call as real, integer (or shortint), or string variables, or as constants or literal constants. For all array parameters, make sure that the array elements are of the correct type, real or shortint; do not substitute integer for shortint. And take care that the size parameter you pass for an array does not exceed the actual size you declared for that array. (If you exceed the declared array size, you can write all over the other variables in your program, and cause yourself much anguish.)

Once your Pascal program has been written and compiled, it must be merged or linked to the Measurement Library using the Pascal system librarian program. Be sure to transfer ALL the modules in MEAS__LIB or MEAS__LIB3. If IO is not in your system library file you will also have to transfer the module IOCOMASM from the file IO (found on your LIB: disc).

Interrupt mode operation is not supported in the Pascal environment. (That means we don't guarantee that it will work. If you try it and it doesn't work, you can purchase consulting services from the nearest HP sales and service office. See the back section of this manual for a list of sales and service offices.) An interrupt service routine (ISR) is required for interrupt mode to work in Pascal, and we do not provide a Pascal ISR with the Measurement Library. If you try to use interrupt mode in Pascal without a proper ISR, you will probably crash your system. If you're an experienced Pascal programmer, you may be able to write your own ISR. For more information on ISRs, refer to the Pascal 2.0 System Designer's Guide, part number 09826-90074.

WRITING THE PROGRAM

In both BASIC and Pascal, writing your application program involves two major activities: setting up the card to take readings, and taking the readings. In addition, BASIC programs may take readings in interrupt mode. We will cover these subjects in the paragraphs that follow. We will also say a few words about externally paced readings.

All of the subroutine calls referred to below are described in detail in Section 2 of this manual.

Setting Up

Setting up an ADC card for readings requires allocation of a common area, as well as calls to at least three subroutines: Meas_lib_init, Config_0, and Init.

The common area serves as the heap space for the subroutines in the Measurement Library. It is allocated automatically in Pascal; in BASIC you must allocate it explicitly at the beginning of your program. Reserve this area by including the following statement in your program:

```
    20      COM/Heapcom/ INTEGER Heaparea(1:n)
```

where n is the size of the Heaparea array. The size of Heaparea is determined by the number of configured names for ADC cards (more about that later) and the number of readings taken for calibration (ditto). Use 53 integers for each ADC card configuration and 4 integers for each reading used in calibration. We recommend using Heaparea(1:1300); this allows all 16 possible ADC card configurations and a calibration run of 100 readings.

In both BASIC and Pascal, the subroutine calls to Meas_lib_init, Config_0, and Init do the following:

Meas_lib_init initializes the Measurement Library, and must be called before any other subroutines in the library are called. Meas_lib_init needs to be called only once in your program.

Config_0 sets up an ADC card for taking readings. At a minimum, you specify a name by which you will call the card and the model number of the card. In addition, you can specify the select code of the card, its gain, a pace rate for taking readings, an error reporting parameter for normal mode overrange errors, and the units (base, standard, or user) in which the readings will be reported. (Reporting units are discussed below.) If you do not supply these optional parameters, Config_0 will supply default values.

Init resets an individual card, disables interrupts for that card, and sets the calibration array for that card to its default values. Init must be used before any other calls except Meas_lib_init, Config_0 and System_init. System_init is the same as Init, except that it initializes all cards that have been configured.

The set-up portion of a typical BASIC program might look like this:

```
          .
          .
20    COM/Heapcom/ INTEGER Heaparea(1:1300)
30    INTEGER Select_code, Gain
40    Name$="ADC"
50    Model$="98640A"
60    Select_code=18
70    Gain=1
80    Pace=0.01
90    Error$="No"
100   Unit$="Standard"
          .
          .
220   Meas_lib_init
230   Config_0(Name$,Model$,Select_code,Gain,Pace,Error$,Unit$)
240   Init(Name$)
          .
          .
```

The analogous Pascal code would look like this:

```
          .
CONST  name = 'ADC';
       model = '98640A';
       select_code = 18;
       gain = 1;
       pace = 0.01;
       error = 'NO';
       units = 'STANDARD';
       multiplier = 1.0;
       offset = 0.0;

BEGIN
   meas_lib_init;
   config_0(name,model,select_code,gain,pace,error,units,multiplier,offset);
   init(name);
          .
          .
```

The most frequently used configuration parameters can be reset without reconfiguring the card; these parameters are gain, pace interval, and units. The gain can be reset with a call to the Set__gain subroutine, or a new gain can be specified as a parameter to the Input or Random__scan subroutine. (The Input and Random__scan subroutines are used to take voltage readings from the ADC; they are described later in this section.) A new pacing interval can be specified as a parameter to the Input, Sequential__scan, or Random__scan subroutine. And the units can be reset with a call to the Set__units subroutine. (Note that if you specify pace or gain parameters in an Input, Sequential__scan, or Random__scan call, the specified pace or gain value holds only for the duration of the call; it reverts to its previous value after the call completes.)

Calibration

Calibration gives you a way of compensating for offsets that are inherent to the ADC card. To use the calibration feature, you must first reserve one of the channels on the card and short the + Input and - Input terminals on that channel to card ground. Then use the Calibrate subroutine to take a specified number of readings from that channel at a specified pace rate. The readings are taken at each of the gain settings and the average at each gain is saved. These average readings are then used to calculate correction values for positive and negative readings at each gain setting. When a subsequent reading is taken on any of the other channels, the appropriate correction value is subtracted from the raw reading before conversion to standard or user units.

Reporting Units

Reporting units come in three flavors: base, standard, and user; you specify one of these with the Config__0 or Set__units command. The units are:

<u>Base units.</u> Base units are in the form of a 16-bit binary integer, of which twelve bits represent the magnitude of the reading. Readings reported in base units are raw readings; gain factors and calibration corrections are not applied to base units. The format of a base unit reading is:

```
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| B | W | O | S | D | D | D | D | D | D | D | D | D | D | D | D |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

where:

B = BUSY. If bit 15 = 1, the ADC is busy. The reading is not taken and all other bits are invalid. If bit 15 = 0, a valid reading is returned.

W = WAIT. If bit 14 = 1, the ADC card was in the wait state at the time of the reading. This means that the card was not read within the interval specified in the pacing timer -- that is, a paced read was not made at the correct time. (Generally you will not see this bit set, since the ADC Library software reports an incorrectly paced read as an error and will not return a value for the reading.)

O = OVERRANGE. If bit 13 = 0, a common mode overrange condition occurred during this reading, and the reading is invalid. (Common mode overrange errors are discussed later in this section.) If bit 13 = 1, no common mode overrange condition occurred during this reading. Note that the sense of this bit is negative true.

S = SIGN. If bit 12 = 0, the value returned for the reading is positive. If bit 12 = 1, the value returned for the reading is negative.

D = DATA. The data bits give the 12-bit binary magnitude of the voltage read from the ADC. (The sign of the voltage is given by the S bit, bit 12.)

Note that all readings taken from the ADC card by the ADC Library software are returned to your program through real number parameters. This includes readings in base units. Thus, while the base unit readings have integer values, they look like real numbers to your program until you explicitly convert them to integers. Assigning them to integer variables in BASIC, or using the trunc or round function in Pascal, will make the conversion.

Standard units. Standard units are base units adjusted for gain and calibration, expressed as rea numbers. They are, in other words, volts.

User units. User units are standard units to which a user-specified multiplier and offset have beer applied, expressed as real numbers. You specify the values for the multiplier and offset in a Config__(or Set__units subroutine call. (The default values for multiplier and offset yield standard unit values. You might use user units to change the units of your readings or to compensate for a known offset ir your readings, or both.

For example, say you were taking readings from a 4-to-20 mA current loop transmitter connected to a flow meter. Say further that the range of the flow meter was from 0 to 50 gallons per minute, and that you were making your voltage readings across a 250-ohm resistor. That would mean that a reading of 1.0 volts corresponded to a flow rate of 0 gpm and that 5.0 volts corresponded to 50 gpm. Using y=mx+b, you can derive a multiplier of 12.5 and an offset of -12.5, and specify these as parameters to a Config__0 call.

```
        .
        .
180   Config_0("Flow","98640A",18,1,.01,"No","User",12.5,-12.5)
        .
        .
```

Then, whenever you take a reading from that current loop, the result is expressed directly in gallons per minute. That's a lot easier than making a conversion from standard units every time you take a voltage reading.

Error Reporting and Handling

The Measurement Library reports errors for a variety of reasons. Typical errors include configuration errors, pacing errors, and overrange errors. When such an error occurs, the Measurement Library forces a system error and returns the error number. Your application program can trap and handle these errors using the ON ERROR mechanism (in BASIC) or the Try-Recover mechanism (in Pascal). In BASIC, you can get the error number with the ERRN function; in Pascal, use the ESCAPECODE function. (Certain run time errors may be reported in BASIC as the Pascal error number plus 400. These errors are listed in Appendix A.) If the errors are not trapped, your program will abort and the system will report the error.

The errors that can be returned by the Measurement Library are listed in Appendix A.

Note that one of the parameters of the Config__0 subroutine determines whether normal overranges are reported as errors or not. Note also that if you are using base units, no overrange errors -- either normal or common mode -- are reported. (You can detect overrange conditions from the bits returned in base unit format.) Overrange errors and pacing errors are discussed in more detail later in this chapter.

Multiple Configurations

The Measurement Library allows you to have up to 16 different ADC card configurations at any one time. Each configuration requires a separate call to Config__0, and each call specifies a unique name for a card. You can assign multiple names, and thus multiple configurations, to a single card if you wish. This would allow you to take readings from different voltage sources on different channels of the same card without reconfiguring the card all the time. For example, say you had flow meters connected to channels 1, 2, and 3 of the card and thermocouples connected to channels 4, 5, 6, and 7. You could specify one name for a flow meter configuration and another name for a thermocouple configuration:

```
        .
        .
180   Config_0("Flow","98640A",18,1,.01,"No","User",12.5,-12.5)
190   Config_0("Thermo","98640A",18,64,.01,"No","Standard")
        .
        .
```

When you want to take a reading from either type of voltage source, just specify the name of the appropriate configuration in your reading call:

```
        .
        .
420   Input("Thermo",5,Tvolt)
430   Input("Flow",2,Gpm)
        .
        .
```

If 16 different ADC configurations are not enough for your application, you can get more by re-using existing names. Do this by making a call to Config_0 and specifying an existing name; the old configuration parameters for that name will be erased and the new parameters (or their default values) will replace them. You will then have to re-initialize the name with a call to the Init subroutine before you can use the new configuration.

Note that the use of different names for the same ADC card will not work in interrupt mode. DO NOT ATTEMPT TO ACCESS AN ADC BY A DIFFERENT NAME DURING INTERRUPT MODE DATA TRANSFERS.

Taking Readings

Taking readings is the whole reason for having an ADC card. Now that you've got your system configured, it's time to start taking those readings. All readings from the ADC card are taken by three subroutines: Input/Read_channel, Sequential_scan, and Random_scan. Here's how you use them:

Input/read_channel. Use the Input or Read_channel subroutine for taking a single reading from a channel on the ADC card. Optionally, you can specify a gain and a pace interval in the subroutine call.

A call to Input would look like this in BASIC:

```
340   Input("ADC",Chan,Volts)
```

The analogous call to Read_channel would look like this in Pascal:

```
read_channel('ADC',chan,volts,gain,pace);
```

Input is the name of the routine as used in a BASIC program; in a Pascal program, use Read_channel. Input was chosen for BASIC for compatibility with the HP 14751A software. Note that you must be very specific when you call the Input subroutine: the I must be upper case and all the other letters must be lower case; otherwise there will be a conflict with the BASIC keyword INPUT. The name Input doesn't work at all with Pascal (another keyword conflict), so Read_channel was chosen instead. Whatever the name, the subroutine works the same way in either language.

Note that if you specify the optional parameters for gain and/or pace interval, they override the existing values only for the duration of the subroutine call. After the call has completed, the gain and pace interval parameters revert to their previous values.

The operation of the Input subroutine in interrupt mode is different from its normal operation. Refer to the discussion of interrupt mode, later in this section, for more details.

Sequential scan. Use the Sequential_scan subroutine to take readings on all channels in sequence from a starting channel to an ending channel. These readings are all taken at the same pace rate (which you specify) and the same gain (specified by the most recent call to Config_0 or Set_gain), and the values are returned to a data array. Optionally, you can repeat the readings as many times as you want. For example, if you wanted to take readings from channels 2 through 7 on an ADC card, at the same gain and pace rate, Sequential_scan would be the appropriate subroutine to use.

In BASIC:

```
            .
            .
100    INTEGER Start, Stop, Repeat
110    REAL Data(1:6)
            .
            .
230    Name$="ADC"
240    Start=2
250    Stop=7
260    Pace=0.01
270    Rept=1
            .
            .
460    Sequential_scan(Name$,Start,Stop,Pace,Data(*),Rept)
            .
            .
```

In Pascal:

```
    .
    .
CONST  name = 'ADC';
       pace = 0.01;
       start = 2;
       stop = 7;
       rept = 1
       d_size = 6;

TYPE   d_array = ARRAY [1..6] OF real;
       d_ptr = ^d_array;

VAR    data: d_ptr;

    .
    .
new(data);
sequential_scan(name,start,stop,pace,d_size,data,rept);
    .
    .
```

You must make sure that your data array is large enough to hold all of the readings that the Sequential__scan call will generate. Note that if the call to Sequential__scan aborts, the contents of the array will be undefined. (This is because the Sequential__scan subroutine uses the array space as temporary storage for a variety of nasty, messy variables; it doesn't fill the array with nice, clean data until just before it returns to your program. If the subroutine aborts while the array space is filled with garbage and your program tries to interpret the garbage as data, you may not be pleased with the results.)

The pace interval that you specify when you call Sequential__scan will be maintained only for the duration of that call. After the readings have been taken, the pace interval will revert to its previous value.

Random scan. Use Random__scan when you need lots of flexibility. Random__scan lets you read from the channels on a card in any order, and you can assign an individual pace interval and gain for each reading. Additionally, you can repeat the set of readings as many times as you want.

The readings are controlled by a set of arrays. A channel array lists the order of the channels to be read. A gain array lists the gains for the readings. A pace array lists the pace intervals that will elapse between readings. And a data array stores the results. The sizes of the channel, pace, and gain arrays need not be the same. The Random__scan subroutine simply starts at the beginning of each array and uses the values in sequence. After Random__scan uses the last element in an array, it goes back to the beginning of the array for the next value. (Note that the gain and pace values do not start over just because the channel array repeats.)

For example, consider an ADC card that has flowmeters attached to channels 2, 3, 4, and 5, and thermocouples attached to channels 6 and 7. Say that you wanted to take the following sets of readings:

Channel	2	3	6	4	5	7
Pace	.02	.02	.02	.02	.02	.02
Gain	1	1	64	1	1	64

To take these readings, you could set up the following arrays:

Channel	2	3	6	4	5	7
Pace	.02					
Gain	1	1	64			

In taking readings from the channels in the channel array, the Random__scan subroutine will use the pace array six times and the gain array twice.

The call sequence to take those readings once would be, in BASIC:

```
          .
          .
110   INTEGER Channel(1:6)
120   REAL Pace(1:1)
130   INTEGER Gain(1:3)
140   REAL Data(1:6)
150   DATA 2,3,6,4,5,7
160   READ Channel(*)
170   DATA .02
180   READ Pace(*)
190   DATA 1,1,64
200   READ Gain(*)
          .
          .
320   Repeat=1
330   Random_scan("ADC",Channel(*),Data(*),Repeat,Pace(*),Gain(*))
          .
          .
```

In Pascal the sequence would be:

```
       .
       .
       .
 CONST  name = 'ADC';
        start = 2;
        stop = 7;
        rept = 1;
        d_size = 6;
        p_size = 1;
        g_size = 3;
        c_size = 6;

 TYPE   r_array = ARRAY [1..6] OF real;
        r_ptr = ^r_array;
        i_array = ARRAY [1..6] OF shortint;
        i_ptr = ^i_array;

 VAR    data: r_ptr;
        channel: i_ptr;
        pace: r_ptr;
        gain: i_ptr;
       .
       .
       .
    new(channel);
    channel^[1] := 2;
    channel^[2] := 3;
    channel^[3] := 6;
    channel^[4] := 4;
    channel^[5] := 5;
    channel^[6] := 7;
    new(pace);
    pace^[1] := 0.02;
    new(gain);
    gain^[1] := 1;
    gain^[2] := 1;
    gain^[3] := 64;

    new(data);
    random_scan (name,
                 c_size,channel,
                 d_size,data,
                 rept,
                 p_size,pace,
                 g_size,gain);
       .
       .
       .
```

In the general case, the *i*th reading is taken using the following array elements:

```
Channel:    chan_array[i mod size_of(chan_array)]
Pace:       pace_array[i mod size_of(pace_array)]
Gain:       gain_array[i mod size_of(gain_array)]
Data:       data[i]
```

Make sure that the data array is large enough to hold all of the readings that will be generated by the Random_scan call. (Don't forget to account for repeats.) As with Sequential_scan, if the call to Random_scan aborts, the contents of the array will be undefined.

The channel, pace, and gain arrays must be dimensioned as arrays, even if they are only single-valued. Scalar variables can not be used.

The pace and gain values specified in Random_scan are used only for the duration of the Random_scan call. After the readings have been taken, pace and gain revert to their previous values.

Special Considerations in Taking Readings

Timing of readings. Even though the Measurement Library can take readings at the full 55 kHz sampling speed of the ADC card, it can't return the results to your program that fast. The reason for this is the system overhead of BASIC or Pascal and the overhead of the Measurement Library itself. For any given set of readings the Measurement Library goes through the following steps:

a) Set up the card.
b) Take the reading(s) at the specified pace rate.
c) Convert the readings to the requested data format. (This includes checking the WAIT bit to make sure there wasn't a pacing error.)

The values below indicate the time required to process a reading sequence. The total time required is the sum of item (a), item (b), and the appropriate value from item (c). Note that these are worst-case values. You would get these values from Series 200 computers using an 8 MHz MC68000 processor chip and no floating point math card. Processing times will be shorter for computers with later (faster) processor chips and/or floating point math cards.

```
BASIC

  Input
     a) Set-up time:            2.0 msec
     b) Read time:              (pace interval) * (one reading)†
     c) Data conversion time
                  BASE units:   1.0 msec
              STANDARD units:   1.9 msec
                  USER units:   2.5 msec

  Sequential scan
     a) Set-up time:            3.5 msec + 0.1 msec per reading
     b) Read time:              (pace interval) * (number of readings)
     c) Data conversion time
                  BASE units:   0.3 msec per reading
              STANDARD units:   1.2 msec per reading
                  USER units:   1.5 msec per reading
```

Random scan

a) Set-up time:	3.0 msec + 0.4 msec per reading
b) Read time:	(pace interval) * (number of readings)
c) Data conversion time:	
BASE units:	1.3 msec per reading
STANDARD units:	2.2 msec per reading
USER units:	2.4 msec per reading

PASCAL

Read channel

a) Set-up time:	2.0 msec
b) Read time:	(pace interval) * (one reading)†
c) Data conversion time	
BASE units:	1.0 msec
STANDARD units:	1.9 msec
USER units:	2.2 msec

Sequential scan

a) Set-up time:	3.4 msec + 0.1 msec per reading
b) Read time:	(pace interval) * (number of readings)
c) Data conversion time	
BASE units:	0.3 msec per reading
STANDARD units:	1.0 msec per reading
USER units:	1.5 msec per reading

Random scan

a) Set-up time:	1.2 msec + 0.5 msec per reading
b) Read time:	(pace interval) * (number of readings)
c) Data conversion time	
BASE units:	1.6 msec per reading
STANDARD units:	2.2 msec per reading
USER units:	2.5 msec per reading

†NOTE: "Input" (BASIC) and "Read Channel" (Pascal) take one reading each time they are called. Refer to page 2-9 of this manual.

Array size limits. The Measurement Library limits your maximum array size to 16,777,215 bytes. That's really a hardware limit, imposed by the width of the address bus on HP 9000 Series 200 and Series 300 computers. At 8 bytes per reading that works out to a maximum of 2,097,150 readings from any one call to the Measurement Library, hardly a severe restriction. In practical terms, you will be limited by the size of your physical memory long before you run into the Measurement Library limit.

How your system lets you access that memory can be a different story. It's no problem in a Pascal system, since you can easily allocate an array large enough to take up all of your physical memory. Things are a bit more subtle in BASIC, however.

At first BASIC appears to limit you to 32767 readings from any single call to the Measurement Library, since that's the largest number you can specify as an array dimension. But you can exceed that number of readings by using a multi-dimensional array. You can easily fill up all the memory you have using a two-dimensional array. (BASIC allows you up to six dimensions in your arrays, so you can arrange your data in whatever format is convenient.) The Measurement Library doesn't care if your array is multi-dimensional; all it wants is the starting address of the array (which you supply by passing the name of the array in the subroutine call). The only thing you have to take care of is reading your data out of the multi-dimensional array in the correct order.

(Note that the ability to specify large data arrays does NOT constitute a continuous data acquisition (CDA) scheme. The amount of data you can collect with the Measurement Library subroutines is limited by the amount of memory in your computer. The Measurement Library has no provision for, say, logging high-speed data to a disc for indefinite periods without missing readings.)

The Pipeline

The ADC requires three operations to produce a reading:

1) provide the channel address for the reading
2) latch the voltage and convert it to a digital value
3) return the value to the host computer

For any given reading, these three operations must be done serially:

```
+---------+---------+---------+
| address | convert | return  |
+---------+---------+---------+

 time ------->
```

Figure 1-1. Analog input operation

However, to maximize throughput, the ADC card "pipelines" the readings. That is, while the value for one reading is being returned, the voltage for the next reading is being latched and converted, and the channel address is being provided for the reading after that. For example, during time period t3 in the figure below the first reading is taken from the card while the second reading is being converted and the third address is being supplied.

```
+-----------+-----------+-----------+
| address 1 | convert 1 | return 1  |
+-----------+-----------+-----------+-----------+
            | address 2 | convert 2 | return 2  |
            +-----------+-----------+-----------+-----------+
                        | address 3 | convert 3 | return 3  |
                        +-----------+-----------+-----------+-----------+
                                    | address 4 | convert 4 | return 4  |
                                    +-----------+-----------+-----------+
     t1          t2          t3          t4          t5          t6

   time ------------------------------------------------>
```

Figure 1-2. Analog Input Pipeline

To start the flow of readings, the Measurement Library software primes the pipeline by taking two "garbage" readings (at times t1 and t2 in the figure above); these two readings are thrown away. (Their only purpose was to start pulling valid readings through the pipeline.) The third reading taken is the first valid reading, since it is the first reading that has gone through all three stages of the pipeline; it is written into the data array as the first reading.

For all readings taken in normal mode, the Measurement Library software takes care of priming and emptying the pipeline; it does this by taking two more readings than are requested and throwing away the two extra garbage values. This happens for each subroutine call; you never have to pay any attention to it, since the software takes care of it all.

(Note that since each subroutine call incurs the extra time required for two readings, it is difficult (if not impossible) to maintain accurate and even pacing of readings between one subroutine call and the next. If your application requires accurate pacing for a block of readings, we suggest that you make all of those readings with one subroutine call. Use Sequential_scan or Random_scan, as appropriate to your application.)

For readings taken in interrupt mode, the Measurement Library software does not take care of the pipeline for you. You must keep track of which readings are which (not a very taxing operation) and throw out the garbage. More information on interrupt mode programming is contained later in this section.

Overrange Errors

You can encounter two kinds of overrange conditions with the ADC card: normal mode overrange and common mode overrange. Normal mode overrange occurs when the input voltage exceeds the range of the analog-to-digital converter. Common mode overrange occurs when either side of the differential input voltage exceeds the maximum input voltage of its input amplifier. The next several paragraphs explain how these overrange conditions can affect your readings.

The voltage measured by the ADC card is the differential input voltage between the + Input and - Input terminals of a channel on the card. The two sides of the input signal pass through separate input amplifiers (op amps), and are then sent to an analog-to-digital (A-to-D) converter for conversion to a numeric value. (The figures below show this circuit configured for a gain of 1.)

There are a couple of limitations that apply to this measurement circuit:

1) The voltage output from an input op amp can not exceed ±10 volts, relative to system ground. For a gain of 1, this also means that the input voltage applied to the op amp can not exceed ±10 volts, again relative to system ground. (The situation gets rather more complicated for gains greater than one; the formula for figuring the maximum input voltage is somewhat abstruse, involving various voltages, gains, and a couple of 2s. We won't get into the mathematics of it, but figure 1-6 shows an example of the results that you may see.) Exceeding this input limit causes a common mode overrange: the output of the op amp is clipped at its limit (+10 volts or -10 volts) and the overrange flag (the O bit in a base unit reading) is set to 1.

2) The A-to-D converter, which compares the outputs of the op amps, can not measure a difference of more than 10 volts. If the difference between those outputs is more than 10 volts, the A-to-D converter clips its output value to 10 volts; this situation is defined as a normal mode overrange.

The next few figures show various combinations of input voltages and the outputs they produce. In the figures, + Input and - Input voltages (relative to system ground) are shown in "stick" type, like this:

+4

The differential input voltages are shown in Roman type, like this:

+6

Figure 1-3 shows a typical reading that causes no problems. The input voltages propagate through the op amps with no clipping, the differential voltage is well within the range of the A-to-D converter, and the converter comes up with the correct value.

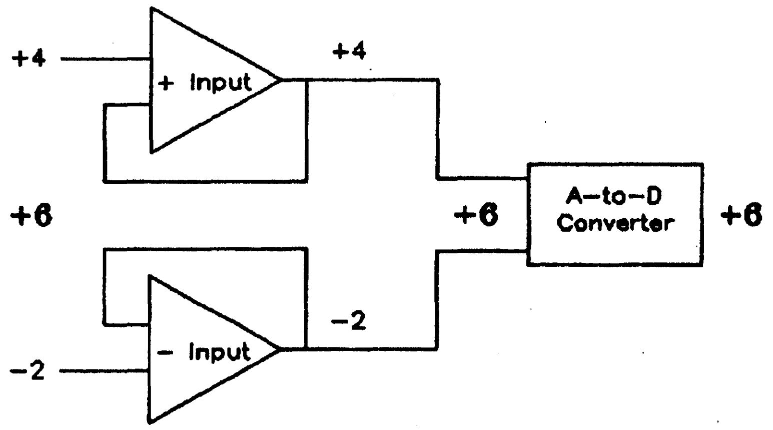


Figure 1-3. Reading OK

Figure 1-4 shows a normal mode overrange condition. The + Input and - Input voltages are within the range of their respective op amps, but the differential input voltage (+12 volts) is too great for the A-to-D converter. The result is a normal mode overrange condition, yielding a full-scale (and incorrect) reading from the A-to-D converter.

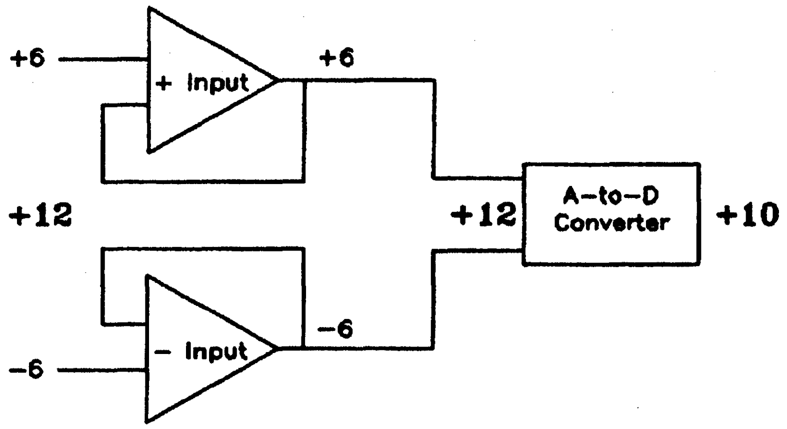


Figure 1-4. Normal mode overrange

Figure 1-5 shows a common mode overrange condition. The + Input voltage of +12 volts is clipped to +10 volts and the overrange flag (O bit) is set to 1. The differential voltage presented to the A-to-D converter is within the range of the converter, so it converts the voltage correctly and comes up with the wrong answer.

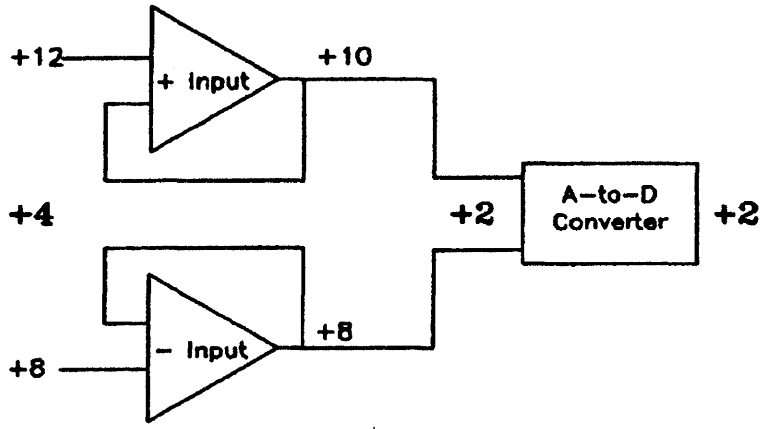


Figure 1-5. Common mode overrange

Figure 1-6 shows a subtler form of common mode overrange that you may encounter at gains greater than 1. This is because the programmable gain amplifier amplifies the difference between the + Input and - Input voltages before sending the result to the A-to-D converter. Even though the input voltages appear to be acceptable, the amplifier may try to boost them out of the acceptable range. In this case, the programmable gain circuit tries to boost the + Input voltage to 11.5 volts, but the output limit of the op amp keeps the voltage from exceeding +10 volts. The overrange flag (O bit) is set to one and the clipped voltage is sent to the A-to-D converter. The resulting value is incorrect.

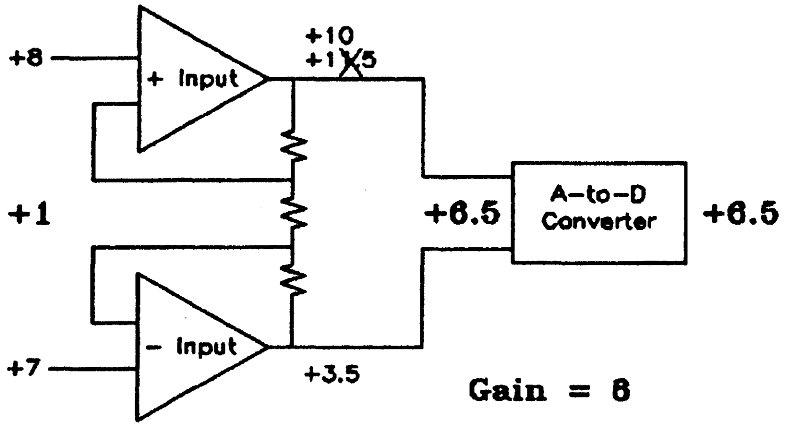


Figure 1-6. Common mode overrange at gain greater than 1

A normal mode overrange is indicated when a reading returns the maximum possible magnitude value. (This is the same as "clipping".) The maximum magnitude value depends on the units in use, as follows:

Base: 4095 (all D bits set to 1)
Standard: (4095 * lsb - calibrate) / gain
User: ((4095 * lsb - calibrate) / gain) * multiplier + offset
    where: lsb = 10 / 4095 volts/bit

Note that it is not possible to tell the difference between a full scale reading and a normal mode overrange reading.

By default, a normal mode overrange condition does not generate an error. However, by setting a parameter in the Config_0 call you can cause an error to be generated when a normal mode overrange occurs.

Common mode overranges are harder to detect than normal mode overranges, since the value of the reading may appear to be correct even though an overrange has occurred. For this reason, common mode overranges are trapped as errors.

Note that the Measurement Library reports errors for normal mode and common mode overranges only when you are operating in standard or user units. If you are operating in base units, no error will be reported. To detect a normal mode overrange in base units, check the D bits for a full scale reading; to detect a common mode overrange, check the O bit.

Pacing Errors

The pace counter on the ADC card is used to determine the duration of the sample portion of the sample and hold cycle. The hold portion is always 9 microseconds, and the minimum sample portion is 9 microseconds. The Measurement Library lets you specify a pace interval that is the sum of these two time periods. Thus you can set the pace at which readings are taken for ease in making accurate time domain measurements of time-varying quantities.

If, due to outside factors (concurrent I/O transfers, keyboard interrupts, and so on), the Measurement Library software is unable to read from the ADC card fast enough to keep up with a programmed pace time, a pacing error will occur. This gives you the assurance that, in the absence of such errors, the time domain measurements are being accurately paced.

While the ADC card and the Measurement Library are fully capable of taking readings every 18 microseconds, the variable gain input amplifiers on the card are not capable of slewing from maximum positive to maximum negative during the 9 microsecond sample period that this pace rate requires. This puts an upper limit on the signal frequency component that the ADC can measure accurately at the 18 microsecond sample rate. The following table shows that maximum frequency component for each gain, for readings to within 1 lsb on a single channel.

Gain	Maximum Signal Frequency Component
1	27 kHz
8	27 kHz
64	15 kHz
512	3.5 kHz

When more than one channel is being sampled (as in Sequential__scan and Random__scan operations) the speed of accurate sampling by the ADC is limited as follows:

Gain	Minimum Pace Time for Multichannel Scans	Equivalent Maximum Sampling Speed
1	50 microseconds	20000 readings per second
8	50 microseconds	20000 readings per second
64	71 microseconds	14000 readings per second
512	1000 microseconds	1000 readings per second

Interrupt Mode

Interrupt mode operation is supported only in the BASIC language system. (It is NOT supported in Pascal.) Interrupt mode is useful when you want your program to continue execution between readings and still maintain an accurate or externally controlled pace rate. There are two subroutines associated specifically with interrupt mode: Enable__intr and Disable__intr. Appropriately enough, interrupt mode is enabled by a call to Enable__intr and is disabled by a call to Disable__intr.

Only a limited subset of Measurement Library subroutine calls are allowed after you have entered interrupt mode:

Input
Config__0
Init
System__init
Disable__intr

Use of any other Measurement Library calls in interrupt mode will result in an error.

When you are in interrupt mode, the Measurement Library does not automatically take care of setting up and clearing out the input pipeline. (Refer to the description of the analog input pipeline earlier in this section.) Thus, when you take a reading with the Input subroutine, the result you get is the value of the reading taken two readings ago. You should discard the data returned from the first two Input calls.

Interrupt mode does not handle multiple configurations of the same card cleanly. To avoid taking erroneous readings, do not take readings from different configurations (names) for the same card while in interrupt mode.

The shortest pacing interval usable in an interrupt mode application is dependent upon many factors. The main factors are the speed of the CPU executing the BASIC program, and the type of BASIC program instructions that are being executed while the ADC is taking readings. To properly understand these factors it is important to understand how the BASIC operating system services interrupts. When BASIC has been enabled to service interrupts for a specific select code with an "ON INTR sc, priority GOSUB label" statement and an interrupt occurs on that select code, BASIC logs the fact that the interrupt has occurred, but does not execute the GOSUB until BASIC has completed executing the current BASIC program line.

When using the ADC library in interrupt mode and an ADC interrupt has occurred, if the time to complete the current BASIC program line, plus the time to execute the GOSUB, plus the time to execute all the BASIC lines until the ADC library "Input" routine actually takes the reading from the ADC card exceeds the pace interval time, the ADC library will return an error 857 indicating that a reading was missed. Therefore the time to service the interrupt depends upon the BASIC program line that is executing when the interrupt occurs as well as the code path to the ADC library "Input" routine.

For the faster interrupt servicing in BASIC, the following tips are offered:

1) Make the ADC library "Input" routine the first statement in the interrupt service routine.

2) Keep the interrupt service routine short. Remember that the pace interval period starts with the "Input" routine, but cannot be serviced until the interrupt service routing "RETURN" statement has been executed.

3) Avoid BASIC instructions which take long times to execute like input/output operations or matrix operations on large arrays.

4) Avoid other interrupt processing at a higher priority than the ADC interrupt service routine.

5) Set the 98640A ADC card at the highest physical interrupt level possible (in this case 6). See the 98640A Reference Manual, HP part number 98640-90001, for details.

When using the interrupt mode it is important to determine experimentally that the pace interval being used is compatible with the BASIC program instructions being executed while waiting for interrupts on the particular computer family on which the program is executing.

The following example shows a BASIC program that takes readings in interrupt mode. Its purpose is to take 8 voltage readings; to do that it takes 10 readings and ignores the first 2 (invalid) readings.

```
       .
       .
40    REAL Volts (-1:8)
50    I=2
       .
       .
110   Config_O("ADC","98640A",18,1,.036)
120   Init("ADC")
130   Enable_intr("ADC")
140   ON INTR 18 GOSUB Service
150   Input("ADC",5,Volts(-1))
       .
       .
340   Service:        !
350      Input("ADC",5,Volts(I-2))
360      I=I+1
370      IF I>10 THEN
380         OFF INTR 18
390         Disable_intr("ADC")
400         FOR J=1 TO 8
410            PRINT Volts (J)
420         NEXT J
430         STOP
440      END IF
450      RETURN
460   END
```

Note that the order of the Enable_intr call and the ON INTR statement is not critical. Enable_intr does not physically enable interrupts on the ADC card; it only sets flags in the Measurement Library. The card interrupts are physically enabled by the first Input call after Enable_intr (line 150 in this example).

External Pacing

You might use external pacing for ADC readings if:

-- you want to use a pace interval longer than that allowed by the Measurement Library software (0.0393336 second)

-- you want the readings to be controlled by an external event, rather than by time

External pacing is primarily a hardware operation. It is largely controlled by two hardware control lines, IPACDA (internal pace disable) and EPCON (external pace control). There's not a lot of software involvement, other than making the read requests that you would normally make for an internally paced read. The timing of the execution of those read requests is controlled by the hardware. (There's no provision in the software for controlling IPACDA and EPCON directly; you'll have to build your own circuits to control them.)

In the next several paragraphs we will look at some of the features of the hardware and software that affect external pacing, and then we will see how they fit together in external pacing applications. In this manual we'll limit our discussion of the hardware to telling you when the IPACDA and EPCON control lines must be set low or high; we won't give you instructions for building the circuits that control those lines. You can, however, get more information about those control lines from the ADC hardware manual, part number 98640-90001.

Hardware Considerations

There are two control lines of interest for external pacing:

IPACDA determines whether the readings are paced by the internal pacing timer on the ADC card. If IPACDA is low, the internal pacing timer of the card is used; if IPACDA is high, the internal pacing timer is bypassed and readings are taken at the free run speed of the card (one reading every 18 microseconds). Note that IPACDA must be high when readings start in order for the timing of the first reading of a series to be accurately known. (IPACDA can be set low after the start of readings if you want the readings to be paced by the internal pacing timer.)

EPCON controls whether or not any readings are taken. If EPCON is low, readings are taken whenever they are requested. If EPCON is high, requested readings are held off; a read request will not complete until EPCON goes low again.

In summary, when EPCON is low, readings are taken at the free run speed of the card (if IPACDA is high) or at the time programmed into the internal pacing timer (if IPACDA is low). When EPCON is high, readings stop.

Software Considerations

When making externally paced readings, you will have to allow for the software set-up time of the various subroutines.

The set-up times in the BASIC language for the reading subroutines are:

```
Input              2.0 milliseconds
Sequential_scan    3.5 milliseconds + 0.1 milliseconds per reading
Random_scan        3.0 milliseconds + 0.4 milliseconds per reading
```

Set-up times in Pascal are:

```
Read_channel        2.0 milliseconds
Sequential_scan    3.4 milliseconds + 0.1 milliseconds per reading
Random_scan        1.2 milliseconds + 0.5 milliseconds per reading
```

Applications

External pacing applications divide into two general types: single readings and bursts of readings.

Single readings. The idea behind taking single externally paced readings is that you keep EPCON high until you want to take a reading, set it low only long enough to take the reading, and then set it high again. The steps in taking a single reading are:

1) Set IPACDA high. IPACDA will remain high for the duration of externally paced readings.

2) Set EPCON high. This holds off all readings.

3) Issue a call to Input/read__channel, Sequential__scan, or Random__scan.

4) Wait. The length of time you wait should be at least the set-up time.

5) When it is time to take a reading, set EPCON low. Keep it low for 1 to 15 microseconds, then set it high again. This will allow one (and only one) reading to be taken.

6) Repeat step 5 until you have taken all the readings that you requested with the subroutine call in step 3. The subroutine will return to your application program only after all requested readings have been taken.

As indicated in step 4, each subroutine call you make requires that you wait the set-up time before pulsing the EPCON line to take the first reading. For Input (or Read__channel) calls made in normal mode, that means that you must wait the set-up time before each reading. If you're using Input in interrupt mode, the set-up time is required only before the first reading. Keep in mind, however, that the EPCON pulses should be at least 36 milliseconds apart if you're operating in interrupt mode.

Bursts of Readings. The idea behind taking readings in bursts is that you request multiple readings with a subroutine call, and then take those readings in one burst by setting EPCON low until all of the readings have been taken. These readings can be taken at the free run speed of the card, or they can be paced by the card's internal pacing timer. The following steps are for triggering burst readings that are paced by the internal pacing timer.

1) Set IPACDA and EPCON high.

2) Make a read request by issuing a call to Sequential__scan or Random__scan.

3) Wait. You should wait for at least the set-up time plus the pace interval.

4) Set the EPCON line low. The analog-to-digital conversion for the first reading will start in approximately 3 microseconds.

5) Set the IPACDA line low. This must happen 1 to 15 microseconds after you set EPCON low.

6) Hold EPCON and IPACDA low until all of the requested readings have been taken. (The subroutine call will return to your application program after all of the readings have completed.)

The requirement (in step 3) that you wait the set-up time plus the pace interval assures that the first reading occurs at a more-or-less known time (within approximately 3 microseconds after EPCON is set low), and that the voltage has been sampled for at least the prescribed sample time (pace interval minus 9 microseconds).

Combinations. You can combine the above two methods of external pacing if your application requires. We won't go into those combinations here; we leave that as an exercise for the interested reader. The methods above should give you enough information to make your combination work.

The following pages are replacement pages from the previous update. Pages superseded by the current update are not included.

PREFACE

Purpose: This manual explains how to use the HP 98645A Measurement Library. It assumes that you have a working knowledge of the BASIC or Pascal language system on the HP 9000 Series 200 or Series 300 computers. It also assumes that you are generally familiar with the HP 98640A Analog-to-Digital Converter card. (Refer to the manual for that card, HP part number 98640-90001, for more information.)

Organization: This manual is organized as follows:

Section 1:	How to use the HP 98645A Measurement Library.
Section 2:	Alphabetical listing of Measurement Library subroutine calls.
Appendix A:	Error messages.
Appendix B:	Quick reference guide to Measurement Library subroutine call syntax.

CONTENTS

Config__0 sets up an HP 98640A ADC card for access by the Measurement Library subroutines.

Syntax

```
   BASIC:  Config_0(name,model[,select_code[,gain[,
                    pace[,report_error[,units[,
                    multiplier[,offset]]]]]]])

  Pascal:  PROCEDURE config_0(name: str255;
                              model: str255;
                              select_code: shortint;
                              gain: shortint;
                              pace: real;
                              report_error: str255;
                              units: str255;
                              multiplier: real;
                              offset: real);
```

Parameters

name:	a string or string literal specifying the name used by the Measurement Library software to refer to a particular ADC configuration.
model:	a string or string literal identifying the ADC card model number ("98640A").
select_code:	an INTEGER giving the physical select code (address) of the ADC card. This number is between 8 and 31, and is set by hardware switches on the card (SW1, switches 1 through 5).
gain:	an INTEGER specifying the default ADC hardware gain. The value must be 1, 8, 64, or 512.
pace:	a REAL number defining the default pace time loaded into the pace counter. This value can be from 0.000018 to 0.0393336 seconds, with a resolution of 600 nanoseconds.
report_error:	a string or string literal enabling an error condition on normal mode overrange readings. The value can be either yes or no. (Only the first character is significant; only "y" and "Y" are taken as yes, all others indicate no.)
units:	a string or string literal specifying the units to used to return ADC data. The units can be base, standard, or user. (Only the first character is significant.)

```
base = binary data read directly from the ADC
standard = (base * ADClsb - calibrate) / gain
user = standard * multiplier + offset
```

multiplier:	a REAL number specifying the multiplier used with user units.

offset: a REAL number specifying the offset used with user units.

Default values:

select_code	18
gain	1
pace	.001 second
report_error	no
units	standard
multiplier	1.0
offset	0.0

Discussion

Config__0 establishes a link between a name (which you supply) and an ADC card, and specifies operating parameters for that name and card. Each ADC card used must be configured with a unique name. You can configure the same card with several different names and parameter sets, and everything will work except interrupt mode data transfers. DO NOT ATTEMPT TO ACCESS AN ADC BY ANOTHER NAME DURING INTERRUPT MODE DATA TRANSFERS.

A maximum of 16 names may be configured into the Measurement Library software. If you need more configurations, names may be re-used. If a name is identical to an already used name, all configuration parameters for the old name will be erased and the new configuration parameters or defaults will be used. The name will then have to be reinitialized with Init before it is accessed.

All readings taken by the ADC are reported in one of three reporting units: base, standard, or user. Base units are in the form of a 16-bit binary integer, with the following format:

```
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| B | W | O | S | D | D | D | D | D | D | D | D | D | D | D | D |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                 MSB                                         LSB
```

where:

B = BUSY. If bit 15 = 1, the ADC is busy. The reading is not taken and all other bits are invalid. If bit 15 = 0, a valid reading is returned.

W = WAIT. If bit 14 = 1, the ADC card was in the wait state at the time of the reading. This means that the card was not read within the interval specified in the pacing timer -- that is, a paced read was not made at the correct time. (Generally you will not see this bit set, since the ADC Library software reports an incorrectly paced read as an error and will not return a value for the reading.)

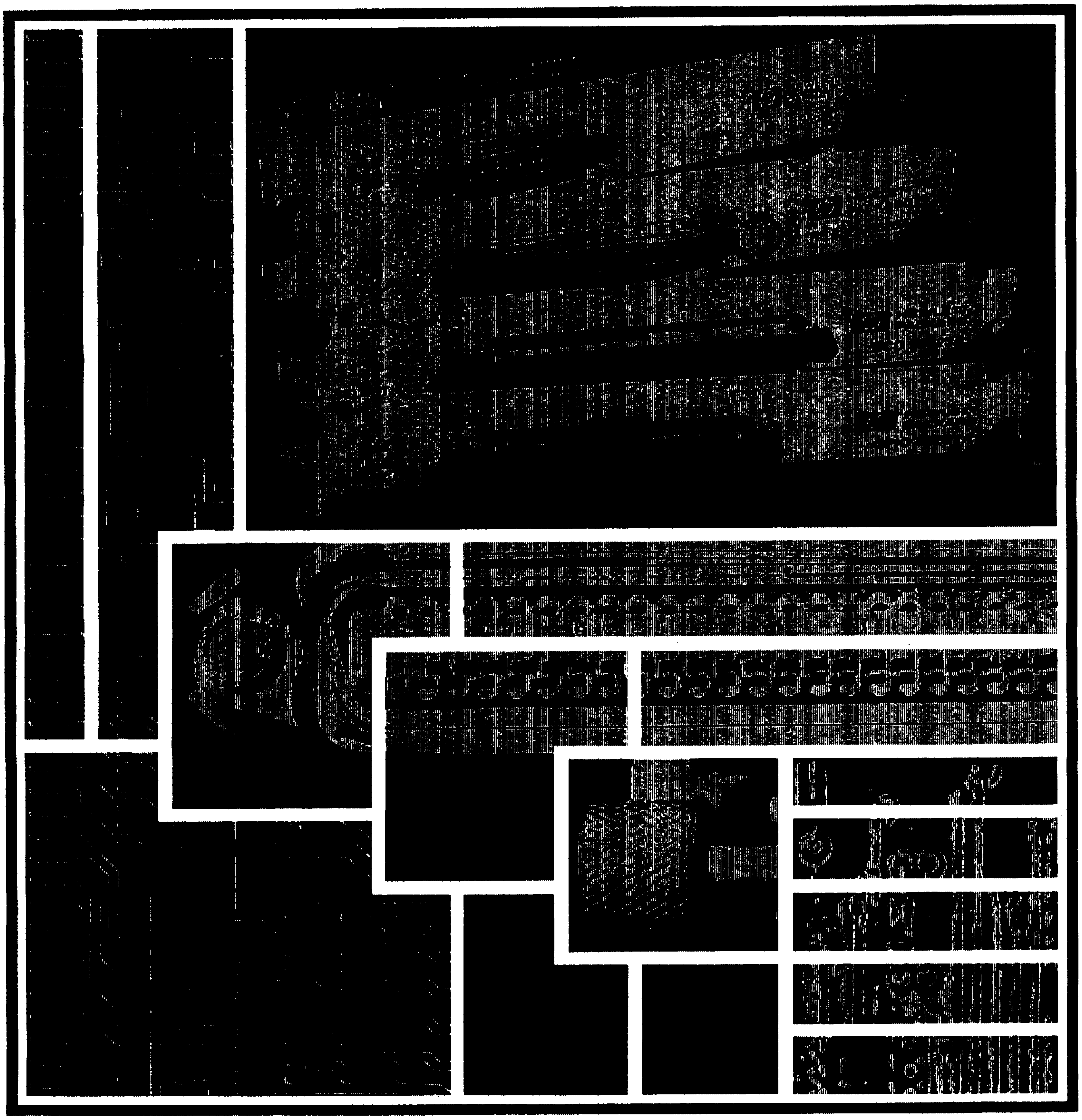
O = OVERRANGE. If bit 13 = 0, a common mode overrange condition occurred during this reading, and the reading is invalid. If bit 13 = 1, no common mode overrange condition occurred during this reading. Note that the sense of this bit is negative true.

S = SIGN. If bit 12 = 0, the value returned for the reading is positive. If bit 12 = 1, the value returned for the reading is negative.

INDEX

HP Computer Systems

HP 98645A Measurement Library User's Manual



HEWLETT PACKARD

HEWLETT PACKARD

Warranty Statement

Hewlett-Packard products are warranted against defects in materials and workmanship. For Hewlett-Packard Desktop Computer Division products sold in the U.S.A. and Canada, this warranty applies for ninety (90) days from the date of delivery.* Hewlett-Packard will, at its option, repair or replace equipment which proves to be defective during the warranty period. This warranty includes labor, parts, and surface travel costs, if any. Equipment returned to Hewlett-Packard for repair must be shipped freight prepaid. Repairs necessitated by misuse of the equipment, or by hardware, software, or interfacing not provided by Hewlett-Packard are not covered by this warranty.

HP warrants that its software and firmware designated by HP for use with a CPU will execute its programming instructions when properly installed on that CPU. HP does not warrant that the operation of the CPU, software, or firmware will be uninterrupted or error free.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR CONSEQUENTIAL DAMAGES.

*For other countries, contact your local Sales and Service Office to determine warranty terms.

HP 98645A Measurement Library

User's Manual

HEWLETT PACKARD

HEWLETT-PACKARD COMPANY
Roseville Networks Division
8000 Foothills Boulevard
Roseville, California 95678

MANUAL PART NO. 98645-90001
E0684
Printed in U. S. A.
June 1984

PRINTING HISTORY

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past updates; however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with the user-inserted update information. New editions of this manual will contain new information, as well as updates.

First Edition .... June 1984

PREFACE

Purpose: This manual explains how to use the HP 98645A Measurement Library. It assumes that you have a working knowledge of the BASIC or Pascal language system on the HP 9000 Series 200 computers. It also assumes that you are generally familiar with the HP 98640A Analog-to-Digital Converter card. (Refer to the manual for that card, HP part number 98640-90001, for more information.)

Organization: This manual is organized as follows:

Section 1: How to use the HP 98645A Measurement Library.

Section 2: Alphabetical listing of Measurement Library subroutine calls.

Appendix A: Error messages.

Appendix B: Quick reference guide to Measurement Library subroutine call syntax.

CONTENTS

USING THE LIBRARY

INTRODUCTION

The HP 98645A Measurement Library provides a set of easy-to-use subroutines for taking readings from the HP 98640A Analog-to-Digital Converter (ADC) card. These subroutines can be used from the BASIC or Pascal language systems on the HP 9000 Series 200 computer. The subroutines are written in Pascal, and are adapted to the BASIC language with the CSUB utility package. The Measurement Library is compatible with BASIC 2.1, BASIC 3.0, Pascal 2.0, Pascal 2.1, and Pascal 3.0.

The Measurement Library subroutine calls are a superset of the "HP 14751A Computer Aided Test Programming Package for the Model 6944A". BASIC programs written using the HP 14751A routines should be able to use the Measurement Library software with very little modification.

Features

The HP 98645A Measurement Library allows you to:

Take a single reading from any of 8 channels at any of 4 gains.

Take readings by scanning across 1 to 8 channels, any number of times.

Take readings from channels in random order as specified in an address array. Optionally, you can specify the gain and pace interval for each reading, and the readings can be repeated any number of times.

Express readings in three different units:

Base units: binary integer returned from the ADC.
Standard units: base units adjusted for gain and calibration, expressed as real numbers.
User units: standard units times a user multiplier plus a user offset.

Take calibration (zero) readings on a specified channel, and apply that calibration adjustment to all readings.

Re-set gain or units at any time.

Take readings at the full 55 kHz sampling speed of the ADC card from either BASIC or Pascal.

Take readings under interrupt mode in BASIC.

Software Provided

The HP 98645A Measurement Library includes these subroutine packages:

MEAS_LIB for use with BASIC 2.0
MEAS_LIB3 for use with BASIC 3.0
INTR2_1 for use with interrupt mode in BASIC 2.1
MEAS_LIB.CODE for use with Pascal 2.0/2.1
MEAS_LIB3.CODE for use with Pascal 3.0

The software is provided on the following media:

Option #630: 3-1/2" floppy disc

Option #655: 5-1/4" floppy disc

THE GENERAL APPROACH

The way you write programs using the Measurement Library is pretty much the same whether you use the BASIC or Pascal language system. There are, however, significant differences in the way you set up your system environment. We will discuss these differences in the next few paragraphs.

Using BASIC 2.1

If you are using the BASIC 2.1 system, take the following steps to get your application up and running:

1) **Boot up BASIC 2.0.**

2) **Load the BASIC 2.1 extensions.** The 2.1 extensions are located on the Extended BASIC 2.1 disc. Insert that disc into the master drive and issue the command LOAD BIN "AP2_1".

3) **Load the interrupt processing package** if you will be taking readings in interrupt mode. (Interrupt mode readings are discussed later in this section.) The interrupt processing package is located on the Measurement Library disc. Insert that disc into the master drive and issue the command LOAD BIN "INTR2_1".

4) **Load any other BASIC extensions** that you need for your application. For example, this would be the time to load Graphics 2.1.

5) **Write your BASIC program** or load a previously written program into memory. In the paragraphs below we will describe how to write your application program using the Measurement Library.

6) **Load the Measurement Library subroutines** if they are not already part of the program you wrote in the previous step. The subroutines are located on the Measurement Library disc. Insert that disc into the master drive and issue the command LOADSUB ALL FROM "MEAS_LIB".

7) **Run your program.** Debug as necessary (repeating steps 5 through 7).

Using BASIC 3.0

If you are using the BASIC 3.0 system, take the following steps to get your application up and running:

1) **Boot up BASIC 3.0.**

2) **Load the BASIC 3.0 IO binary** if you will be taking readings in interrupt mode. (Interrupt mode readings are discussed later in this section.) The IO binary is located on the BASIC 3.0 Language Binary disc. Insert that disc into the master drive and issue the command LOAD BIN "IO".

3) **Load any other BASIC binaries** that you need for your application. For example, this would be the time to load graphics routines.

4) **Write your BASIC program** or load a previously written program into memory. In the paragraphs below we will describe how to write your application program using the Measurement Library.

5) **Load the Measurement Library subroutines** if they are not already part of the program you wrote in the previous step. The subroutines are located on the Measurement Library disc. Insert that disc into the master drive and issue the command LOADSUB ALL FROM "MEAS_LIB3".

6) **Run your program.** Debug as necessary (repeating steps 4 through 6).

General BASIC Programming

The Measurement Library subroutines add approximately 23,700 bytes to your BASIC program. The INTR2_1 binary adds approximately 1200 bytes.

Note that integer parameters used in the Measurement Library subroutine calls must be explicitly typed as INTEGER. (You can find out which parameters are integers by looking at the parameter descriptions in the subroutine call listings in Section 2 of this manual.) Real parameters and string parameters (those ending in $) need not be explicitly typed. Literal constants of any type (integer, real, or string) may be used. Note that integers must not contain a decimal point.

You can invoke Measurement Library routines by calling them (CALL statement) or simply by entering them by name. When you use them in an IF . . THEN statement or an ON . . statement, the "CALL" must be explicit.

Using Pascal 2.0, 2.1, or 3.0

You can call the Measurement Library subroutines from the Pascal language by importing the Measurement Library and using the library subroutines as procedure calls with the syntax described in Section 2 of this manual. Typically, you import the Measurement Library with a compiler directive of

```
$SEARCH 'MEAS_LIB'$
```

or

```
$SEARCH 'MEAS_LIB3'$
```

and an import statement of

```
  IMPORT measurement_lib;
```

in your code. Importing the Measurement Library adds about 17600 bytes to your Pascal program.

If the Pascal system modules INTERFACE and IO have not been merged into the system library file, you will also have to include the compiler directive

```
 $SEARCH 'INTERFACE.','IO.'$
```

Note that the "." after each file name is significant.

The procedure calls for the Measurement Library are all exported from the file MEAS_LIB.CODE (or MEAS_LIB3.CODE), along with the following types:

```
    TYPE  shortint = -32768..32767;
          byte = 0..255;
          str255 = string[255];
          iarraytype = ARRAY[0..maxint] OF shortint;
          rarraytype = ARRAY[0..maxint] OF real;
          rarraypt = ^rarraytype;
          iarraypt = ^iarraytype;
```

Due to the rigorous structure of the Pascal language, you can't default parameters in the procedure calls. However, to save you the bother of declaring real and integer arrays for the pace and gain array parameters of the random_scan procedure, you can use the default pace or gain value (established by a call to Config_0 or Set_gain) by specifying a 0 for the array size and a NIL for the array pointer. All other parameters for all procedure calls must be explicitly provided in the procedure call as real, integer (or shortint), or string variables, or as constants or literal constants. For all array parameters, make sure that the array elements are of the correct type, real or shortint; do not substitute integer for shortint. And take care that the size parameter you pass for an array does not exceed the actual size you declared for that array. (If you exceed the declared array size, you can write all over the other variables in your program, and cause yourself much anguish.)

Once your Pascal program has been written and compiled, it must be merged or linked to the Measurement Library using the Pascal system librarian program. Be sure to transfer ALL the modules in MEAS_LIB or MEAS_LIB3. If IO is not in your system library file you will also have to transfer the module IOCOMASM from the file IO (found on your LIB: disc).

Interrupt mode operation is not supported in the Pascal environment. (That means we don't guarantee that it will work. If you try it and it doesn't work, you can purchase consulting services from the nearest HP sales and service office. See the back section of this manual for a list of sales and service offices.) An interrupt service routine (ISR) is required for interrupt mode to work in Pascal, and we do not provide a Pascal ISR with the Measurement Library. If you try to use interrupt mode in Pascal without a proper ISR, you will probably crash your system. If you're an experienced Pascal programmer, you may be able to write your own ISR. For more information on ISRs, refer to the Pascal 2.0 System Designer's Guide, part number 09826-90074.

WRITING THE PROGRAM

In both BASIC and Pascal, writing your application program involves two major activities: setting up the card to take readings, and taking the readings. In addition, BASIC programs may take readings in interrupt mode. We will cover these subjects in the paragraphs that follow. We will also say a few words about externally paced readings.

All of the subroutine calls referred to below are described in detail in Section 2 of this manual.

Setting Up

Setting up an ADC card for readings requires allocation of a common area, as well as calls to at least three subroutines: Meas__lib__init, Config__0, and Init.

The common area serves as the heap space for the subroutines in the Measurement Library. It is allocated automatically in Pascal; in BASIC you must allocate it explicitly at the beginning of your program. Reserve this area by including the following statement in your program:

```
20    COM/Heapcom/ INTEGER Heaparea(1:n)
```

where n is the size of the Heaparea array. The size of Heaparea is determined by the number of configured names for ADC cards (more about that later) and the number of readings taken for calibration (ditto). Use 53 integers for each ADC card configuration and 4 integers for each reading used in calibration. We recommend using Heaparea(1:1300); this allows all 16 possible ADC card configurations and a calibration run of 100 readings.

In both BASIC and Pascal, the subroutine calls to Meas__lib__init, Config__0, and Init do the following:

Meas__lib__init initializes the Measurement Library, and must be called before any other subroutines in the library are called. Meas__lib__init needs to be called only once in your program.

Config__0 sets up an ADC card for taking readings. At a minimum, you specify a name by which you will call the card and the model number of the card. In addition, you can specify the select code of the card, its gain, a pace rate for taking readings, an error reporting parameter for normal mode overrange errors, and the units (base, standard, or user) in which the readings will be reported. (Reporting units are discussed below.) If you do not supply these optional parameters, Config__0 will supply default values.

Init resets an individual card, disables interrupts for that card, and sets the calibration array for that card to its default values. Init must be used before any other calls except Meas__lib__init, Config__0 and System__init. System__init is the same as Init, except that it initializes all cards that have been configured.

The set-up portion of a typical BASIC program might look like this:

```
         .
         .
20     COM/Heapcom/ INTEGER Heaparea(1:1300)
30     INTEGER Select_code, Gain
40     Name$="ADC"
50     Model$="98640A"
60     Select_code=18
70     Gain=1
80     Pace=0.01
90     Error$="No"
100    Unit$="Standard"
         .
         .
220    Meas_lib_init
230    Config_0(Name$,Model$,Select_code,Gain,Pace,Error$,Unit$)
240    Init(Name$)
         .
         .
```

The analogous Pascal code would look like this:

```
           .
           .
CONST   name = 'ADC';
        model = '98640A';
        select_code = 18;
        gain = 1;
        pace = 0.01;
        error = 'NO';
        units = 'STANDARD';
        multiplier = 1.0;
        offset = 0.0;

BEGIN
   meas_lib_init;
   config_0(name,model,select_code,gain,pace,error,units,multiplier,offset);
   init(name);
           .
           .
```

The most frequently used configuration parameters can be reset without reconfiguring the card; these parameters are gain, pace interval, and units. The gain can be reset with a call to the Set__gain subroutine, or a new gain can be specified as a parameter to the Input or Random__scan subroutine. (The Input and Random__scan subroutines are used to take voltage readings from the ADC; they are described later in this section.) A new pacing interval can be specified as a parameter to the Input, Sequential__scan, or Random__scan subroutine. And the units can be reset with a call to the Set__units subroutine. (Note that if you specify pace or gain parameters in an Input, Sequential__scan, or Random__scan call, the specified pace or gain value holds only for the duration of the call; it reverts to its previous value after the call completes.)

Calibration

Calibration gives you a way of compensating for offsets that are inherent to the ADC card. To use the calibration feature, you must first reserve one of the channels on the card and short the + Input and - Input terminals on that channel to card ground. Then use the Calibrate subroutine to take a specified number of readings from that channel at a specified pace rate. The readings are taken at each of the gain settings and the average at each gain is saved. These average readings are then used to calculate correction values for positive and negative readings at each gain setting. When a subsequent reading is taken on any of the other channels, the appropriate correction value is subtracted from the raw reading before conversion to standard or user units.

Reporting Units

Reporting units come in three flavors: base, standard, and user; you specify one of these with the Config_0 or Set_units command. The units are:

Base units. Base units are in the form of a 16-bit binary integer, of which twelve bits represent the magnitude of the reading. Readings reported in base units are raw readings; gain factors and calibration corrections are not applied to base units. The format of a base unit reading is:

```
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| B | W | O | S | D | D | D | D | D | D | D | D | D | D | D | D |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

where:

B = BUSY. If bit 15 = 1, the ADC is busy. The reading is not taken and all other bits are invalid. If bit 15 = 0, a valid reading is returned.

W = WAIT. If bit 14 = 1, the ADC card was in the wait state at the time of the reading. This means that the card was not read within the interval specified in the pacing timer -- that is, a paced read was not made at the correct time. (You should never see this bit set, since the ADC Library software reports an incorrectly paced read as an error and will not return a value for the reading.)

O = OVERRANGE. If bit 13 = 0, a common mode overrange condition occurred during this reading, and the reading is invalid. (Common mode overrange errors are discussed later in this section.) If bit 13 = 1, no common mode overrange condition occurred during this reading. Note that the sense of this bit is negative true.

S = SIGN. If bit 12 = 0, the value returned for the reading is positive. If bit 12 = 1, the value returned for the reading is negative.

D = DATA. The data bits give the 12-bit binary magnitude of the voltage read from the ADC. (The sign of the voltage is given by the S bit, bit 12.)

Note that all readings taken from the ADC card by the ADC Library software are returned to your program through real number parameters. This includes readings in base units. Thus, while the base unit readings have integer values, they look like real numbers to your program until you explicitly convert them to integers. Assigning them to integer variables in BASIC, or using the trunc or round function in Pascal, will make the conversion.

Standard units. Standard units are base units adjusted for gain and calibration, expressed as real numbers. They are, in other words, volts.

User units. User units are standard units to which a user-specified multiplier and offset have been applied, expressed as real numbers. You specify the values for the multiplier and offset in a Config__0 or Set__units subroutine call. (The default values for multiplier and offset yield standard unit values.) You might use user units to change the units of your readings or to compensate for a known offset in your readings, or both.

For example, say you were taking readings from a 4-to-20 mA current loop transmitter connected to a flow meter. Say further that the range of the flow meter was from 0 to 50 gallons per minute, and that you were making your voltage readings across a 250-ohm resistor. That would mean that a reading of 1.0 volts corresponded to a flow rate of 0 gpm and that 5.0 volts corresponded to 50 gpm. Using y=mx+b, you can derive a multiplier of 12.5 and an offset of -12.5, and specify these as parameters to a Config__0 call.

```
            .
            .
180   Config_0("Flow","98640A",18,1,.01,"No","User",12.5,-12.5)
            .
            .
```

Then, whenever you take a reading from that current loop, the result is expressed directly in gallons per minute. That's a lot easier than making a conversion from standard units every time you take a voltage reading.

Error Reporting and Handling

The Measurement Library reports errors for a variety of reasons. Typical errors include configuration errors, pacing errors, and overrange errors. When such an error occurs, the Measurement Library forces a system error and returns the error number. Your application program can trap and handle these errors using the ON ERROR mechanism (in BASIC) or the Try-Recover mechanism (in Pascal). In BASIC, you can get the error number with the ERRN function; in Pascal, use the ESCAPECODE function. (Certain run time errors may be reported in BASIC as the Pascal error number plus 400. These errors are listed in Appendix A.) If the errors are not trapped, your program will abort and the system will report the error.

The errors that can be returned by the Measurement Library are listed in Appendix A.

Note that one of the parameters of the Config__0 subroutine determines whether normal overranges are reported as errors or not. Note also that if you are using base units, no overrange errors -- either normal or common mode -- are reported. (You can detect overrange conditions from the bits returned in base unit format.) Overrange errors and pacing errors are discussed in more detail later in this chapter.

Multiple Configurations

The Measurement Library allows you to have up to 16 different ADC card configurations at any one time. Each configuration requires a separate call to Config__0, and each call specifies a unique name for a card. You can assign multiple names, and thus multiple configurations, to a single card if you wish. This would allow you to take readings from different voltage sources on different channels of the same card without reconfiguring the card all the time. For example, say you had flow meters connected to channels 1, 2, and 3 of the card and thermocouples connected to channels 4, 5, 6, and 7. You could specify one name for a flow meter configuration and another name for a thermocouple configuration:

```
            .
            .
180    Config_0("Flow","98640A",18,1,.01,"No","User",12.5,-12.5)
190    Config_0("Thermo","98640A",18,64,.01,"No","Standard")
            .
            .
```

When you want to take a reading from either type of voltage source, just specify the name of the appropriate configuration in your reading call:

```
            .
            .
420    Input("Thermo",5,Tvolt)
430    Input("Flow",2,Gpm)
            .
            .
```

If 16 different ADC configurations are not enough for your application, you can get more by re-using existing names. Do this by making a call to Config_0 and specifying an existing name; the old configuration parameters for that name will be erased and the new parameters (or their default values) will replace them. You will then have to re-initialize the name with a call to the Init subroutine before you can use the new configuration.

Note that the use of different names for the same ADC card will not work in interrupt mode. DO NOT ATTEMPT TO ACCESS AN ADC BY A DIFFERENT NAME DURING INTERRUPT MODE DATA TRANSFERS.

Taking Readings

Taking readings is the whole reason for having an ADC card. Now that you've got your system configured, it's time to start taking those readings. All readings from the ADC card are taken by three subroutines: Input/Read_channel, Sequential_scan, and Random_scan. Here's how you use them:

<u>Input/read channel.</u> Use the Input or Read_channel subroutine for taking a single reading from a channel on the ADC card. Optionally, you can specify a gain and a pace interval in the subroutine call.

A call to Input would look like this in BASIC:

```
340    Input("ADC",Chan,Volts)
```

The analogous call to Read_channel would look like this in Pascal:

```
read_channel('ADC',chan,volts,gain,pace);
```

Input is the name of the routine as used in a BASIC program; in a Pascal program, use Read_channel. Input was chosen for BASIC for compatibility with the HP 14751A software. Note that you must be very specific when you call the Input subroutine: the I must be upper case and all the other letters must be lower case; otherwise there will be a conflict with the BASIC keyword INPUT. The name Input doesn't work at all with Pascal (another keyword conflict), so Read_channel was chosen instead. Whatever the name, the subroutine works the same way in either language.

Note that if you specify the optional parameters for gain and/or pace interval, they override the existing values only for the duration of the subroutine call. After the call has completed, the gain and pace interval parameters revert to their previous values.

The operation of the Input subroutine in interrupt mode is different from its normal operation. Refer to the discussion of interrupt mode, later in this section, for more details.

Sequential scan. Use the Sequential_scan subroutine to take readings on all channels in sequence from a starting channel to an ending channel. These readings are all taken at the same pace rate (which you specify) and the same gain (specified by the most recent call to Config_0 or Set_gain), and the values are returned to a data array. Optionally, you can repeat the readings as many times as you want. For example, if you wanted to take readings from channels 2 through 7 on an ADC card, at the same gain and pace rate, Sequential_scan would be the appropriate subroutine to use. In BASIC:

```
         .
         .
100   INTEGER Start, Stop, Repeat
110   REAL Data(1:6)
         .
         .
230   Name$="ADC"
240   Start=2
250   Stop=7
260   Pace=0.01
270   Rept=1
         .
         .
460   Sequential_scan(Name$,Start,Stop,Pace,Data(*),Rept)
         .
         .
```

In Pascal:

```
   .
   .
CONST  name = 'ADC';
       pace = 0.01;
       start = 2;
       stop = 7;
       rept = 1
       d_size = 6;

TYPE   d_array = ARRAY [1..6] OF real;
       d_ptr = ^d_array;

VAR    data: d_ptr;

   .
   .
new(data);
sequential_scan(name,start,stop,pace,d_size,data,rept);
   .
   .
```

You must make sure that your data array is large enough to hold all of the readings that the Sequential_scan call will generate. Note that if the call to Sequential_scan aborts, the contents of the array will be undefined. (This is because the Sequential_scan subroutine uses the array space as temporary storage for a variety of nasty, messy variables; it doesn't fill the array with nice, clean data until just before it returns to your program. If the subroutine aborts while the array space is filled with garbage and your program tries to interpret the garbage as data, you may not be pleased with the results.)

The pace interval that you specify when you call Sequential_scan will be maintained only for the duration of that call. After the readings have been taken, the pace interval will revert to its previous value.

Random scan. Use Random_scan when you need lots of flexibility. Random_scan lets you read from the channels on a card in any order, and you can assign an individual pace interval and gain for each reading. Additionally, you can repeat the set of readings as many times as you want.

The readings are controlled by a set of arrays. A channel array lists the order of the channels to be read. A gain array lists the gains for the readings. A pace array lists the pace intervals that will elapse between readings. And a data array stores the results. The sizes of the channel, pace, and gain arrays need not be the same. The Random_scan subroutine simply starts at the beginning of each array and uses the values in sequence. After Random_scan uses the last element in an array, it goes back to the beginning of the array for the next value. (Note that the gain and pace values do not start over just because the channel array repeats.)

For example, consider an ADC card that has flowmeters attached to channels 2, 3, 4, and 5, and thermocouples attached to channels 6 and 7. Say that you wanted to take the following sets of readings:

Channel	2	3	6	4	5	7
Pace	.02	.02	.02	.02	.02	.02
Gain	1	1	64	1	1	64

To take these readings, you could set up the following arrays:

Channel	2	3	6	4	5	7
Pace	.02					
Gain	1	1	64			

In taking readings from the channels in the channel array, the Random_scan subroutine will use the pace array six times and the gain array twice. The call sequence to take those readings once would be, in BASIC:

```
          .
          .
110    INTEGER Channel(1:6)
120    REAL Pace(1:1)
130    INTEGER Gain(1:3)
140    REAL Data(1:6)
150    DATA 2,3,4,5,6,7
160    READ Channel(*)
170    DATA .02
180    READ Pace(*)
190    DATA 1,1,64
200    READ Gain(*)
          .
          .
320    Repeat=1
330    Random_scan("ADC",Channel(*),Data(*),Repeat,Pace(*),Gain(*))
          .
          .
```

In Pascal the sequence would be:

```
        .
        .
        .
 CONST  name = 'ADC';
        start = 2;
        stop = 7;
        rept = 1;
        d_size = 6;
        p_size = 1;
        g_size = 3;
        c_size = 6;

 TYPE   r_array = ARRAY [1..6] OF real;
        r_ptr = ^r_array;
        i_array = ARRAY [1..6] OF shortint;
        i_ptr = ^i_array;

 VAR    data: r_ptr;
        channel: i_ptr;
        pace: r_ptr;
        gain: i_ptr;
        .
        .
        .
    new(channel);
    channel^[1] := 2;
    channel^[2] := 3;
    channel^[3] := 4;
    channel^[4] := 5;
    channel^[5] := 6;
    channel^[6] := 7;
    new(pace);
    pace^[1] := 0.02;
    new(gain);
    gain^[1] := 1;
    gain^[2] := 1;
    gain^[3] := 64;

    new(data);
    random_scan (name,
                 c_size,channel,
                 d_size,data,
                 rept,
                 p_size,pace,
                 g_size,gain);
        .
        .
        .
```

In the general case, the *i*th reading is taken using the following array elements:

```
Channel:    chan_array[i mod size_of(chan_array)]
Pace:       pace_array[i mod size_of(pace_array)]
Gain:       gain_array[i mod size_of(gain_array)]
Data:       data[i]
```

Make sure that the data array is large enough to hold all of the readings that will be generated by the Random__scan call. (Don't forget to account for repeats.) As with Sequential__scan, if the call to Random__scan aborts, the contents of the array will be undefined.

The channel, pace, and gain arrays must be dimensioned as arrays, even if they are only single-valued. Scalar variables can not be used.

The pace and gain values specified in Random__scan are used only for the duration of the Random__scan call. After the readings have been taken, pace and gain revert to their previous values.

The Pipeline

The ADC requires three operations to produce a reading:

1) provide the channel address for the reading
2) latch the voltage and convert it to a digital value
3) return the value to the host computer

For any given reading, these three operations must be done serially:

```
+---------+---------+---------+
| address | convert | return  |
+---------+---------+---------+

  time ------->
```

Figure 1-1. Analog input operation

However, to maximize throughput, the ADC card "pipelines" the readings. That is, while the value for one reading is being returned, the voltage for the next reading is being latched and converted, and the channel address is being provided for the reading after that. For example, during time period t3 in the figure below the first reading is taken from the card while the second reading is being converted and the third address is being supplied.

```
+-----------+-----------+-----------+
| address 1 | convert 1 | return 1  |
+-----------+-----------+-----------+-----------+
            | address 2 | convert 2 | return 2  |
            +-----------+-----------+-----------+-----------+
                        | address 3 | convert 3 | return 3  |
                        +-----------+-----------+-----------+----------+
                                    | address 4 | convert 4 | return 4 |
                                    +-----------+-----------+----------+
     t1          t2          t3          t4          t5          t6

  time ----------------------------------------------------->
```

Figure 1-2. Analog Input Pipeline

To start the flow of readings, the Measurement Library software primes the pipeline by taking two "garbage" readings (at times t1 and t2 in the figure above); these two readings are thrown away. (Their only purpose was to start pulling valid readings through the pipeline.) The third reading taken is the first valid reading, since it is the first reading that has gone through all three stages of the pipeline; it is written into the data array as the first reading.

For all readings taken in normal mode, the Measurement Library software takes care of priming and emptying the pipeline; it does this by taking two more readings than are requested and throwing away the two extra garbage values. This happens for each subroutine call; you never have to pay any attention to it, since the software takes care of it all.

(Note that since each subroutine call incurs the extra time required for two readings, it is difficult (if not impossible) to maintain accurate and even pacing of readings between one subroutine call and the next. If your application requires accurate pacing for a block of readings, we suggest that you make all of those readings with one subroutine call. Use Sequential__scan or Random__scan, as appropriate to your application.)

For readings taken in interrupt mode, the Measurement Library software does not take care of the pipeline for you. You must keep track of which readings are which (not a very taxing operation) and throw out the garbage. More information on interrupt mode programming is contained later in this section.

Overrange Errors

You can encounter two kinds of overrange conditions with the ADC card: normal mode overrange and common mode overrange. Normal mode overrange occurs when the input voltage exceeds the range of the analog-to-digital converter. Common mode overrange occurs when either side of the differential input voltage exceeds the maximum input voltage of its input amplifier. The next several paragraphs explain how these overrange conditions can affect your readings.

The voltage measured by the ADC card is the differential input voltage between the + Input and - Input terminals of a channel on the card. The two sides of the input signal pass through separate input amplifiers (op amps), and are then sent to an analog-to-digital (A-to-D) converter for conversion to a numeric value. (The figures below show this circuit configured for a gain of 1.)

There are a couple of limitations that apply to this measurement circuit:

1) The voltage output from an input op amp can not exceed ±10 volts, relative to system ground. For a gain of 1, this also means that the input voltage applied to the op amp can not exceed ±10 volts, again relative to system ground. (The situation gets rather more complicated for gains greater than one; the formula for figuring the maximum input voltage is somewhat abstruse, involving various voltages, gains, and a couple of 2s. We won't get into the mathematics of it, but figure 1-6 shows an example of the results that you may see.) Exceeding this input limit causes a common mode overrange: the output of the op amp is clipped at its limit (+10 volts or -10 volts) and the overrange flag (the O bit in a base unit reading) is set to 1.

2) The A-to-D converter, which compares the outputs of the op amps, can not measure a difference of more than 10 volts. If the difference between those outputs is more than 10 volts, the A-to-D converter clips its output value to 10 volts; this situation is defined as a normal mode overrange.

The next few figures show various combinations of input voltages and the outputs they produce. In the figures, + Input and - Input voltages (relative to system ground) are shown in "stick" type, like this:

+4

The differential input voltages are shown in Roman type, like this:

+6

Figure 1-3 shows a typical reading that causes no problems. The input voltages propagate through the op amps with no clipping, the differential voltage is well within the range of the A-to-D converter, and the converter comes up with the correct value.

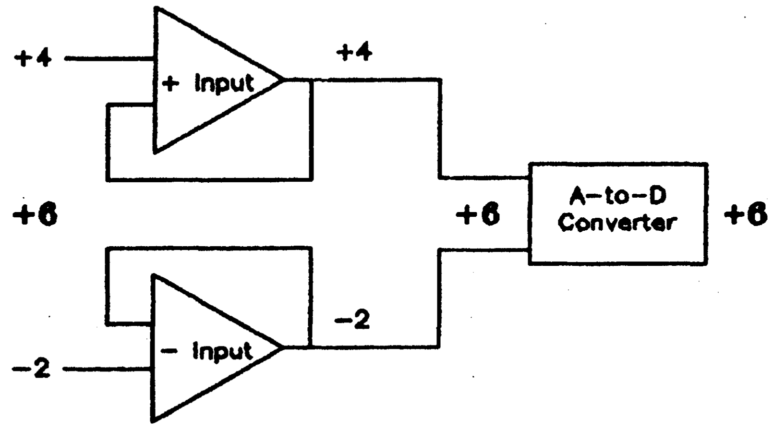


Figure 1-3. Reading OK

Figure 1-4 shows a normal mode overrange condition. The + Input and - Input voltages are within the range of their respective op amps, but the differential input voltage (+12 volts) is too great for the A-to-D converter. The result is a normal mode overrange condition, yielding a full-scale (and incorrect) reading from the A-to-D converter.

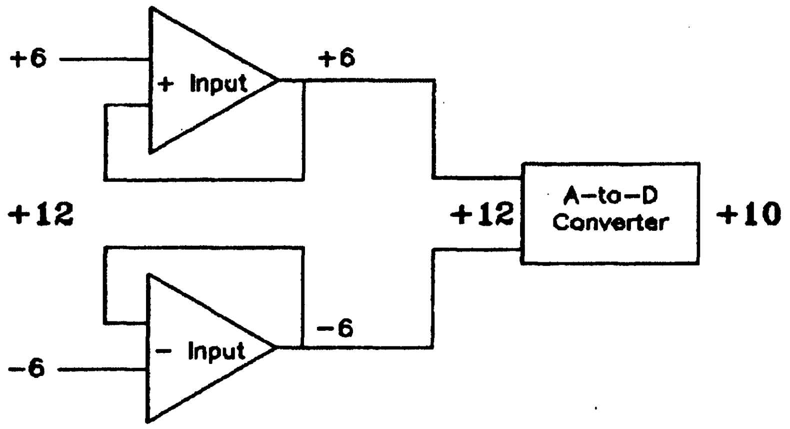


Figure 1-4. Normal mode overrange

Figure 1-5 shows a common mode overrange condition. The + Input voltage of +12 volts is clipped to +10 volts and the overrange flag (O bit) is set to 1. The differential voltage presented to the A-to-D converter is within the range of the converter, so it converts the voltage correctly and comes up with the wrong answer.

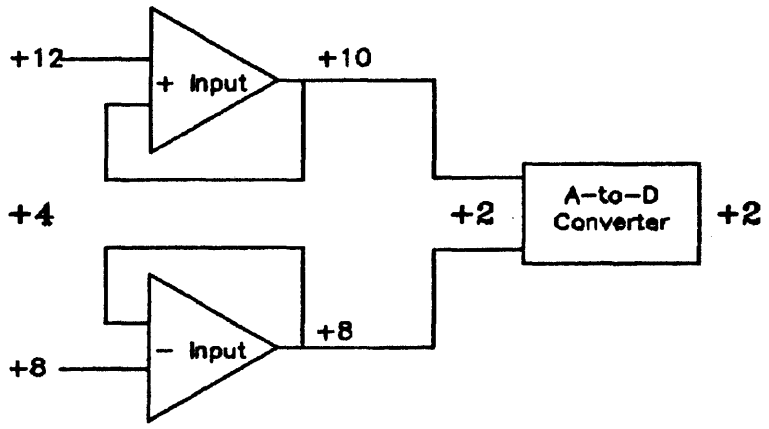


Figure 1-5. Common mode overrange

Figure 1-6 shows a subtler form of common mode overrange that you may encounter at gains greater than 1. This is because the programmable gain amplifier amplifies the difference between the + Input and - Input voltages before sending the result to the A-to-D converter. Even though the input voltages appear to be acceptable, the amplifier may try to boost them out of the acceptable range. In this case, the programmable gain circuit tries to boost the + Input voltage to 11.5 volts, but the output limit of the op amp keeps the voltage from exceeding +10 volts. The overrange flag (O bit) is set to one and the clipped voltage is sent to the A-to-D converter. The resulting value is incorrect.

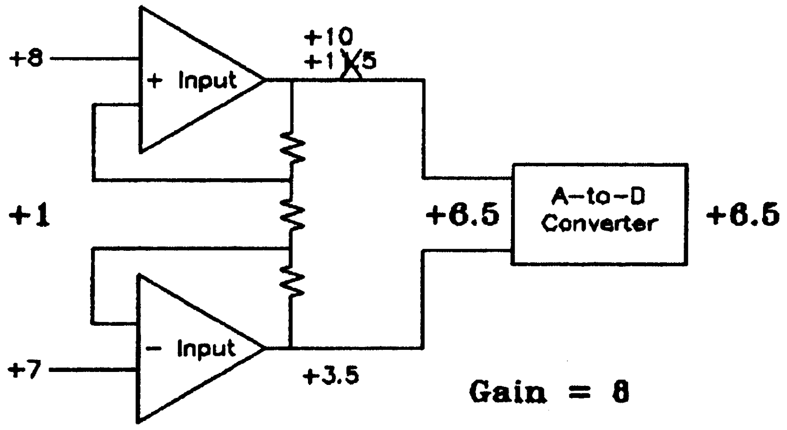


Figure 1-6. Common mode overrange at gain greater than 1

A normal mode overrange is indicated when a reading returns the maximum possible magnitude value. (This is the same as "clipping".) The maximum magnitude value depends on the units in use, as follows:

Base: 4095 (all D bits set to 1)
Standard: (4095 * lsb - calibrate) / gain
User: ((4095 * lsb - calibrate) / gain) * mulitplier + offset

where: lsb = 10 / 4095 volts/bit

Note that it is not possible to tell the difference between a full scale reading and a normal mode overrange reading.

By default, a normal mode overrange condition does not generate an error. However, by setting a parameter in the Config__0 call you can cause an error to be generated when a normal mode overrange occurs.

Common mode overranges are harder to detect than normal mode overranges, since the value of the reading may appear to be correct even though an overrange has occurred. For this reason, common mode overranges are trapped as errors.

Note that the Measurement Library reports errors for normal mode and common mode overranges only when you are operating in standard or user units. If you are operating in base units, no error will be reported. To detect a normal mode overrange in base units, check the D bits for a full scale reading; to detect a common mode overrange, check the O bit.

Pacing Errors

The pace counter on the ADC card is used to determine the duration of the sample portion of the sample and hold cycle. The hold portion is always 9 microseconds, and the minimum sample portion is 9 microseconds. The Measurement Library lets you specify a pace interval that is the sum of these two time periods. Thus you can set the pace at which readings are taken for ease in making accurate time domain measurements of time-varying quantities.

If, due to outside factors (concurrent I/O transfers, keyboard interrupts, and so on), the Measurement Library software is unable to read from the ADC card fast enough to keep up with a programmed pace time, a pacing error will occur. This gives you the assurance that, in the absence of such errors, the time domain measurements are being accurately paced.

While the ADC card and the Measurement Library are fully capable of taking readings every 18 microseconds, the variable gain input amplifiers on the card are not capable of slewing from maximum positive to maximum negative during the 9 microsecond sample period that this pace rate requires. This puts an upper limit on the signal frequency component that the ADC can measure accurately at the 18 microsecond sample rate. The following table shows that maximum frequency component for each gain, for readings to within 1 lsb on a single channel.

Gain	Maximum Signal Frequency Component
1	27 kHz
8	27 kHz
64	15 kHz
512	3.5 kHz

When more than one channel is being sampled (as in Sequential_scan and Random_scan operations) the speed of accurate sampling by the ADC is limited as follows:

Gain	Minimum Pace Time for Multichannel Scans	Equivalent Maximum Sampling Speed
1	50 microseconds	20000 readings per second
8	50 microseconds	20000 readings per second
64	71 microseconds	14000 readings per second
512	1000 microseconds	1000 readings per second

Interrupt Mode

Interrupt mode operation is supported only in the BASIC language system. (It is NOT supported in Pascal.) Interrupt mode is useful when you want your program to continue execution between readings and still maintain an accurate or externally controlled pace rate. There are two subroutines associated specifically with interrupt mode: Enable_intr and Disable_intr. Appropriately enough, interrupt mode is enabled by a call to Enable_intr and is disabled by a call to Disable_intr.

Only a limited subset of Measurement Library subroutine calls are allowed after you have entered interrupt mode:

Input
Config_0
Init
System_init
Disable_intr

Use of any other Measurement Library calls in interrupt mode will result in an error.

When you are in interrupt mode, the Measurement Library does not automatically take care of setting up and clearing out the input pipeline. (Refer to the description of the analog input pipeline earlier in this section.) Thus, when you take a reading with the Input subroutine, the result you get is the value of the reading taken two readings ago. You should discard the data returned from the first two Input calls.

Interrupt mode does not handle multiple configurations of the same card cleanly. To avoid taking erroneous readings, do not take readings from different configurations (names) for the same card while in interrupt mode.

The shortest recommended pacing interval in interrupt mode is 36 milliseconds. This is very close to the longest pacing interval available from the ADC card (39.3336 milliseconds). You can get longer pacing intervals by using external pacing. (External pacing is discussed later in this section.)

The following example shows a BASIC program that takes readings in interrupt mode. Its purpose is to take 8 voltage readings; to do that it takes 10 readings and ignores the first 2 (invalid) readings.

```
          .
          .
40    REAL Volts (-1:8)
50    I=2
          .
          .
110   Config_0("ADC","98640A",18,1,.036)
120   Init("ADC")
130   Enable_intr("ADC")
140   ON INTR 18 GOSUB Service
150   Input("ADC",5,Volts(-1))
          .
          .
340   Service:        !
350      Input("ADC",5,Volts(I-2))
360      I=I+1
370      IF I>10 THEN
380         OFF INTR 18
390         Disable_intr("ADC")
400         PRINT Volts(1:8)
410         STOP
420      END IF
430      RETURN
440   END
```

Note that the order of the Enable_intr call and the ON INTR statement is not critical. Enable_intr does not physically enable interrupts on the ADC card; it only sets flags in the Measurement Library. The card interrupts are physically enabled by the first Input call after Enable_intr (line 150 in this example).

External Pacing

You might use external pacing for ADC readings if:

-- you want to use a pace interval longer than that allowed by the Measurement Library software (0.0393336 second)

-- you want the readings to be controlled by an external event, rather than by time

External pacing is primarily a hardware operation. It is largely controlled by two hardware control lines, IPACDA (internal pace disable) and EPCON (external pace control). There's not a lot of software involvement, other than making the read requests that you would normally make for an internally paced read. The timing of the execution of those read requests is controlled by the hardware. (There's no provision in the software for controlling IPACDA and EPCON directly; you'll have to build your own circuits to control them.)

In the next several paragraphs we will look at some of the features of the hardware and software that affect external pacing, and then we will see how they fit together in external pacing applications. In this manual we'll limit our discussion of the hardware to telling you when the IPACDA and EPCON control lines must be set low or high; we won't give you instructions for building the circuits that control those lines. You can, however, get more information about those control lines from the ADC hardware manual, part number 98640-90001.

Hardware Considerations

There are two control lines of interest for external pacing:

IPACDA determines whether the readings are paced by the internal pacing timer on the ADC card. If IPACDA is low, the internal pacing timer of the card is used; if IPACDA is high, the internal pacing timer is bypassed and readings are taken at the free run speed of the card (one reading every 18 microseconds). Note that IPACDA must be high when readings start in order for the timing of the first reading of a series to be accurately known. (IPACDA can be set low after the start of readings if you want the readings to be paced by the internal pacing timer.)

EPCON controls whether or not any readings are taken. If EPCON is low, readings are taken whenever they are requested. If EPCON is high, requested readings are held off; a read request will not complete until EPCON goes low again.

In summary, when EPCON is low, readings are taken at the free run speed of the card (if IPACDA is high) or at the time programmed into the internal pacing timer (if IPACDA is low). When EPCON is high, readings stop.

Software Considerations

When making externally paced readings, you will have to allow for the software set-up time of the various subroutines. The set-up times in the BASIC language for the reading subroutines are:

```
Input             2.0 milliseconds
Sequential_scan   3.5 milliseconds + 0.1 milliseconds per reading
```

```
Random_scan       3.0 milliseconds + 0.4 milliseconds per reading
```

You can use these set-up times for Pascal programming as well. Pascal set-up times are shorter than those in BASIC, so the times listed above will give you plenty of margin in your Pascal applications.

Applications

External pacing applications divide into two general types: single readings and bursts of readings.

Single readings. The idea behind taking single externally paced readings is that you keep EPCON high until you want to take a reading, set it low only long enough to take the reading, and then set it high again. The steps in taking a single reading are:

1) Set IPACDA high. IPACDA will remain high for the duration of externally paced readings.

2) Set EPCON high. This holds off all readings.

3) Issue a call to Input/read__channel, Sequential__scan, or Random__scan.

4) Wait. The length of time you wait should be at least the set-up time.

5) When it is time to take a reading, set EPCON low. Keep it low for 1 to 15 microseconds, then set it high again. This will allow one (and only one) reading to be taken.

6) Repeat step 5 until you have taken all the readings that you requested with the subroutine call in step 3. The subroutine will return to your application program only after all requested readings have been taken.

As indicated in step 4, each subroutine call you make requires that you wait the set-up time before pulsing the EPCON line to take the first reading. For Input (or Read__channel) calls made in normal mode, that means that you must wait the set-up time before each reading. If you're using Input in interrupt mode, the set-up time is required only before the first reading. Keep in mind, however, that the EPCON pulses should be at least 36 milliseconds apart if you're operating in interrupt mode.

Bursts of Readings. The idea behind taking readings in bursts is that you request multiple readings with a subroutine call, and then take those readings in one burst by setting EPCON low until all of the readings have been taken. These readings can be taken at the free run speed of the card, or they can be paced by the card's internal pacing timer. The following steps are for triggering burst readings that are paced by the internal pacing timer.

1) Set IPACDA and EPCON high.

2) Make a read request by issuing a call to Sequential__scan or Random__scan.

3) Wait. You should wait for at least the set-up time plus the pace interval.

4) Set the EPCON line low. The analog-to-digital conversion for the first reading will start in approximately 3 microseconds.

5) Set the IPACDA line low. This must happen 1 to 15 microseconds after you set EPCON low.

6) Hold EPCON and IPACDA low until all of the requested readings have been taken. (The subroutine call will return to your application program after all of the readings have completed.)

The requirement (in step 3) that you wait the set-up time plus the pace interval assures that the first reading occurs at a more-or-less known time (within approximately 3 microseconds after EPCON is set low), and that the voltage has been sampled for at least the prescribed sample time (pace interval minus 9 microseconds).

Combinations. You can combine the above two methods of external pacing if your application requires. We won't go into those combinations here; we leave that as an exercise for the interested reader. The methods above should give you enough information to make your combination work.

SUBROUTINE INFORMATION

This section gives the subroutine call syntax for the subroutines in the HP 98645A Measurement Library. The subroutine calls supported by the library are:

Calibrate **
Config__0 **
Disable__intr
Enable__intr *
Init
Input *
Meas__lib__init **
Random__scan *
Read__channel
Sequential__scan
Set__gain **
Set__units **
System__init

* These calls incorporate optional extensions beyond the HP 14751A Computer Aided Test Programming Package for the Model 6944A).

** These calls do not exist in the HP 14751A package.

In the following subroutine descriptions, these conventions apply:

-- The parameters list for the subroutine appears in parentheses: ( ). These parentheses must be included in the subroutine call.

-- Optional parameters (BASIC only) are contained within square brackets: [ ].

String parameters for name and model number are case sensitive. (That is, don't use lower case characters in place of upper case, and vice versa.) All other string parameters are case insensitive.

Note that none of the parameters in Pascal calls are optional.

Pascal data types exported by the Measurement Library are as follows:

```
TYPE  shortint = -32768..32767;
      byte = 0..255;
      str255 = string[255];
      iarraytype = ARRAY[0..maxint] OF shortint;
      rarraytype = ARRAY[0..maxint] OF real;
      rarraypt = ^rarraytype;
      iarraypt = ^iarraytype;
```

The remainder of this section gives the subroutine call syntax, arranged by subroutine in alphabetical order. Note that parameters identified as INTEGER are of type INTEGER in BASIC, but of type shortint in Pascal.

CALIBRATE

Calibrate allows you to measure and compensate for the various offsets in the ADC card. To do this, Calibrate dedicates one channel on the card to making reference readings; the offsets derived from the reference readings are used to adjust the readings taken on the remaining channels of the card.

Syntax

```
    BASIC: Calibrate(name,channel,pace,number)

   Pascal: PROCEDURE calibrate(name: str255;
                               channel: shortint;
                               pace: real;
                               number: shortint);
```

Parameters

name:	a string or string literal specifying the ADC name from the Config_0 call.
channel:	an INTEGER specifying the reference channel (from 0 to 7) to be used for calibration.
pace:	a REAL number specifying the calibration pace rate, from 0.000018 to 0.0393336 seconds with a resolution of 600 nanoseconds.
number:	an INTEGER specifying the number of readings to be taken for this calibration. This number must be from 1 to 32767.

Discussion

To use the calibration feature, you must first short the + Input and - Input terminals of one of the channels on the card to card ground; this gives a 0 volt input for that channel. Then you specify that channel in the call to the Calibrate subroutine. When the Calibrate call is executed, the specified number of readings are taken at all gain settings, and the average for each gain setting is saved. The offsets are then used to calculate the proper correction values for positive and negative readings at each gain. When subsequent readings are taken on other channels, the correction value is subtracted from the reading prior to conversion to standard or user units. (No correction is applied to a reading expressed in base units.)

Note that occasionally a Calibrate call will abort with an error 860. This may be caused by temporary transient electrical noise, especially on calibration calls with small numbers of readings. 860 errors from Calibrate calls should routinely be re-tried several times, and the connections of the shorting wires at the calibration channel checked, before you assume that the ADC card is defective.

Note that Calibrate temporarily requires 8 bytes of memory for each reading specified in the number parameter. Large numbers of readings may cause errors due to not enough memory.

Config_0 sets up an HP 98640A ADC card for access by the Measurement Library subroutines.

Syntax

```
    BASIC:  Config_0(name,model[,select_code[,gain[,
                     pace[,report_error[,units[,
                     multiplier[,offset]]]]]]])

   Pascal:  PROCEDURE config_0(name: str255;
                               model: str255;
                               select_code: shortint;
                               gain: shortint;
                               pace: real;
                               report_error: str255;
                               units: str255;
                               multiplier: real;
                               offset: real);
```

Parameters

name:	a string or string literal specifying the name used by the Measurement Library software to refer to a particular ADC configuration.
model:	a string or string literal identifying the ADC card model number ("98640A").
select_code:	an INTEGER giving the physical select code (address) of the ADC card. This number is between 8 and 31, and is set by hardware switches on the card (SW1, switches 1 through 5).
gain:	an INTEGER specifying the default ADC hardware gain. The value must be 1, 8, 64, or 512.
pace:	a REAL number defining the default pace time loaded into the pace counter. This value can be from 0.000018 to 0.0393336 seconds, with a resolution of 600 nanoseconds.
report_error:	a string or string literal enabling an error condition on normal mode overrange readings. The value can be either yes or no. (Only the first character is significant; only "y" and "Y" are taken as yes, all others indicate no.)
units:	a string or string literal specifying the units to used to return ADC data. The units can be base, standard, or user. (Only the first character is significant.)

```
base = binary data read directly from the ADC
standard = (base * ADClsb - calibrate) / gain
user = standard * multiplier + offset
```

multiplier:	a REAL number specifying the multiplier used with user units.

offset: a REAL number specifying the offset used with user units.

Default values:

select_code	18
gain	1
pace	.001 second
report_error	no
units	standard
multiplier	1.0
offset	0.0

Discussion

Config_0 establishes a link between a name (which you supply) and an ADC card, and specifies operating parameters for that name and card. Each ADC card used must be configured with a unique name. You can configure the same card with several different names and parameter sets, and everything will work except interrupt mode data transfers. DO NOT ATTEMPT TO ACCESS AN ADC BY ANOTHER NAME DURING INTERRUPT MODE DATA TRANSFERS.

A maximum of 16 names may be configured into the Measurement Library software. If you need more configurations, names may be re-used. If a name is identical to an already used name, all configuration parameters for the old name will be erased and the new configuration parameters or defaults will be used. The name will then have to be reinitialized with Init before it is accessed.

All readings taken by the ADC are reported in one of three reporting units: base, standard, or user. Base units are in the form of a 16-bit binary integer, with the following format:

```
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| B | W | O | S | D | D | D | D | D | D | D | D | D | D | D | D |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                MSB                                         LSB
```

where:

B = BUSY. If bit 15 = 1, the ADC is busy. The reading is not taken and all other bits are invalid. If bit 15 = 0, a valid reading is returned.

W = WAIT. If bit 14 = 1, the ADC card was in the wait state at the time of the reading. This means that the card was not read within the interval specified in the pacing timer -- that is, a paced read was not made at the correct time. (You should never see this bit set, since the ADC Library software reports an incorrectly paced read as an error and will not return a value for the reading.)

O = OVERRANGE. If bit 13 = 0, a common mode overrange condition occurred during this reading, and the reading is invalid. If bit 13 = 1, no common mode overrange condition occurred during this reading. Note that the sense of this bit is negative true.

S = SIGN. If bit 12 = 0, the value returned for the reading is positive. If bit 12 = 1, the value returned for the reading is negative.

D = DATA. The data bits give the 12-bit binary magnitude of the voltage read from the ADC. (The sign of the voltage is given by the S bit, bit 12.)

MSB = most significant bit.

LSB = least significant bit.

Base unit readings are raw readings; no gain factors or calibration corrections are applied.

The value used by the Measurement Library for the ADC card least significant bit (ADClsb) is the 64-bit floating point value of

10 volts / 4095 bits

or

2.442002442002442 millivolts per bit

The least significant bit (LSB) values used in each gain range are:

LSB = ADClsb / gain

Thus, the approximate LSB values in each gain range are:

Gain	LSB	
1	2.4420	millivolts
8	305.25	microvolts
64	38.156	microvolts
512	4.7695	microvolts

Standard units are real numbers representing true volts. They are equivalent to base unit values corrected for gain and calibration (if any). User units are real numbers equivalent to standard units times a multiplier plus an offset.

ADC readings are always returned from the Measurement Library calls as real numbers or real array elements (IEEE 64 bit floating point binary representation in both BASIC and Pascal). Readings in standard or user units should be stored and manipulated as real numbers. Readings in base units must be converted into integer format by the user program prior to any manipulation of the data. Assignment to an integer variable in BASIC, or using the trunc or round function in Pascal, will suffice.

Refer to "Setting Up" in Section 1 of this manual for a full discussion of ADC card configuration.

DISABLE__INTR

Disable__intr configures the ADC card for normal, non-interrupt mode operation.

Syntax

```
   BASIC: Disable_intr(name)

  Pascal: PROCEDURE disable_intr(name: str255);
```

Parameter

```
name:   a string or string literal specifying the ADC name assigned
        by the Config_0 call.
```

Discussion

Interrupt mode operation is supported only for BASIC environments. Use of interrupt mode in Pascal is not supported. Please refer to the discussion of the Enable__intr subroutine (next page) for further information.

Enable__intr configures an ADC card for interrupt mode operation.

Syntax

```
 BASIC: Enable_intr(name)

Pascal: PROCEDURE enable_intr(name: str255);
```

Parameter

```
name:   a string or string literal specifying the ADC name assigned
        by the Config_0 call.
```

Discussion

Interrupt mode operation is not supported in the Pascal environment. (That means you're on your own if you use it. If you have trouble making it work, you can purchase HP consulting, on a time and materials basis, from your local HP sales and service office. HP sales and service offices are listed in the back of this manual.) For interrupt mode to work in Pascal, you need to have an appropriate interrupt service routine (ISR). If you use interrupt mode without one, you will probably crash your system. We don't provide an ISR as part of the Measurement Library, but if you're a skilled Pascal programmer you may be able to write one of your own. Refer to the Pascal 2.0 System Designer's Guide, part number 09826-90074, for more information on ISRs.

Interrupt mode does work in BASIC. There are a few things you should be aware of:

1) The only Measurement Library calls allowed after an Enable__interrupt call are:

   Input
   Config__0
   Init
   System__init
   Disable__interrupt

2) The Input subroutine functions differently in interrupt mode. Refer to the description of that subroutine later in this section for more information.

3) Interrupt mode does not handle multiple configurations of the same ADC card well. To prevent erroneous readings, do not try to take readings from different configurations (names) of the same ADC card while in interrupt mode.

For a more complete explanation of interrupt mode programming, refer to Section 1 of this manual.

INIT

Init resets and disables interrupt mode on an ADC card, and sets the calibration array to its default values.

Syntax

```
 BASIC:  Init(name)

Pascal:  PROCEDURE init(name: str255);
```

Parameter

name:	a string or string literal specifying an ADC name assigned by the Config_0 call.

Discussion

The Init (initialize) call must be used prior to any other calls except Config__0 and System__init. A single call to System__init may be substituted for individual Init calls for all currently configured cards.

The **Input** or **Read__channel** subroutine takes one reading from a specified channel on an ADC card. Input is used in BASIC programs; Read__channel is used in Pascal programs.

Syntax

```
    BASIC:  Input(name,channel,datum[,gain[,pace]])

   Pascal:  PROCEDURE read_channel(name: str255;
                                   channel: shortint;
                                   VAR datum: real;
                                   gain: shortint;
                                   pace: real);
```

Parameters

name:	a string or string literal specifying an ADC name assigned by the Config_0 call.
channel:	an INTEGER specifying the channel number (from 0 to 7) to be read.
datum:	a REAL variable to hold the value of a reading.
gain:	an INTEGER specifying the hardware gain. The value must be 1, 8, 64, or 512. If a value is not given for the gain, the value specified in a Config_0 or Set_gain call is used.
pace:	a REAL variable specifying the pace interval that elapses before the reading. This value must be from 0.000018 to 0.0393336 seconds, with a resolution of 600 nanoseconds.

Discussion

The reading returned by a call to Input or Read__channel will be formatted according to the units specified in a previous call to Config__0 or Set__units. If you specify values for gain or pace, those values will be used only for the duration of this Input (Read__channel) call.

The Input subroutine operates differently in interrupt mode. This involves the analog input pipeline. (Section 1 of this manual has more information on the pipeline.) For any reading, it takes 3 read operations to get that reading all the way through the pipeline. In normal mode, the Input subroutine performs all 3 of these readings and returns the 1 valid reading. In interrupt mode, Input performs only 1 read operation and returns the value that was requested two operations before; it is up to your program to keep track of the progress of your readings through the pipeline. (For more information on interrupt mode programming, refer to Section 1 of this manual.)

Be careful of how you call the Input subroutine from BASIC: use "Input" (not "INPUT" or "input") to avoid conflict with the BASIC keyword "INPUT". In Pascal, use "Read__channel".

The pace interval comprises the sample time and the analog-to-digital conversion time for the reading. Conversion takes 9 microseconds; thus, the sample time is the pace time minus 9 microseconds.

MEAS_LIB_INIT

Meas_lib_init initializes the global variables in the Measurement Library. In a BASIC environment, it also initializes the heap area.

Syntax

```
    BASIC:  Meas_lib_init

   Pascal:  PROCEDURE meas_lib_init;
```

Discussion

Your application program must call Meas_lib_init before it calls any other Measurement Library subroutines.

Random_scan takes readings from channels in any order that you specify, with whatever pace and gain value that you specify for each individual reading.

Syntax

```
    BASIC: Random_scan(name,chan_array(*),data_array(*)[,rept[,
                     pace_array(*)[,gain_array(*)]]])

   Pascal: PROCEDURE random_scan(name: str255;
                                chan_size: integer;
                                chan_array: anyptr;
                                data_size: integer;
                                data_array: anyptr;
                                rept: shortint;
                                pace_size: integer;
                                pace_array: anyptr;
                                gain_size: integer;
                                gain_array: anyptr);
```

Parameters

name:	a string or string literal specifying the ADC name assigned by the Config_0 call.
chan_size:	(Pascal only) an integer giving the size of the array of channel numbers
chan_array:	in BASIC, the name of an INTEGER array of channel numbers. In Pascal, this is a pointer to the shortint array of channel numbers. The channel numbers can range from 0 to 7.
data_size:	(Pascal only) an integer giving the size of the array of readings.
data_array:	in BASIC, the name of a REAL array to hold readings from the ADC card. In Pascal, this is a pointer to the real array of readings.
rept:	an INTEGER number of times to scan the channel array. This number can be from 1 to 32767; the default value is 1.
pace_size:	(Pascal only) an integer giving the size of the array of pace interval values. Specify 0 if you want to use the default pace value.
pace_array:	in BASIC, the name of a REAL array of pace interval values. In Pascal, this is a pointer to the real array of pace interval values. The values in the array must be from 0.000018 to 0.0393336 seconds, with a resolution of 600 nanoseconds. The default pace value is the value specified in the Config_0 call. In Pascal, you must specify NIL if you want to use the default value.

gain_size:	(Pascal only) an integer giving the size of the array of gain values. Specify 0 if you want to use the default gain value.
gain_array:	in BASIC, the name of an INTEGER array of gain values. In Pascal, a pointer to the shortint array of gain values. The gain values in the array must be 1, 8, 64, or 512. The default gain value is the value specified in the Config_0 or Set_gain call. In Pascal, specify NIL if you want to use the default value.

Discussion

The sizes of the channel, pace, and gain arrays need not be the same. The Random__scan subroutine simply starts at the beginning of each array and uses the values in sequence. After Random__scan uses the last value in an array, it goes back to the beginning of the array for the next value. (Gain and pace values do not start over just because the channel array repeats.) In the general case, the *i*th reading is taken using the following array elements:

```
Channel:    chan_array(i mod size_of(chan_array))
Pace:       pace_array(i mod size_of(pace_array))
Gain:       gain_array(i mod size_of(gain_array))
Data:       data(i)
```

Note that the data array must be large enough to hold all of the readings that will be generated by the Random__scan call (including repeats).

Gain and pace values specified for a Random__scan call are valid only for the duration of that call. After the call has completed, the gain and pace values revert to their default values.

If you are programming in Pascal and you want to use the default pace interval value (the value that was specified in the Config__0 call for the card), you must specify a value of 0 for the pace__size parameter and a value of NIL for the pace__array parameter. Similarly, if you want to use the default gain value, specify a value of 0 for the gain__size parameter and a value of NIL for the gain__array parameter.

The channel, pace, and gain arrays must be dimensioned as arrays, even if they are only single-valued. Scalar variables can not be used.

Read__channel is the Pascal equivalent of the Input subroutine. Refer to the discussion of Input earlier in this section for information on Read__channel.

SEQUENTIAL__SCAN

Sequential__scan takes readings from sequential channels on an ADC card.

Syntax

```
   BASIC:  Sequential_scan(name,start,stop,pace,data_array(*)[,rept])

  Pascal:  PROCEDURE sequential_scan(name: str255;
                                     start: shortint;
                                     stop: shortint;
                                     pace: real;
                                     data_size: integer;
                                     data_array: anyptr;
                                     rept: shortint);
```

Parameters

name:	a string or string literal specifying the ADC name assigned by the Config_0 call.
start:	an INTEGER specifying the number of the first channel to be read. This number must be from 0 to 7.
stop:	an INTEGER specifying the number of the last channel to be read. This number must be from 0 to 7 and must not be less than start.
pace:	a REAL number specifying the pace interval. This interval must be from 0.000018 to 0.0393336 seconds, with a resolution of 600 nanoseconds.
data_size:	(Pascal only) an integer giving the size of the data array.
data_array:	in BASIC, the name of a REAL array to hold the readings taken from the ADC card. In Pascal, this is a pointer to the real array that holds the readings.
rept:	an INTEGER giving the number of scans. This number must be from 1 to 32767; the default value is 1.

Discussion

Sequential__scan scans (reads) the channels on the ADC card sequentially, from the start channel to the stop channel. You can repeat the scans as many times as you want (up to 32767 total scans). The data array must be large enough to hold the total number of readings (including repeats) that you request.

One pace interval is used for all readings taken with the Sequential__scan routine. This pace interval is valid only for the duration of the Sequential__scan call; after this call has completed, the pace interval value reverts to the default value established with the Config__0 call.

The gain value used by Sequential__scan is the default gain value set by the Config__0 or Set__gain call.

Set__gain sets the hardware gain used in taking readings from the ADC card.

Syntax

```
   BASIC: Set_gain(name,gain)

  Pascal: PROCEDURE set_gain(name: str255;
                             gain: shortint);
```

Parameters

name:	a string or string literal specifying the ADC name assigned by the Config_0 call.
gain:	an INTEGER specifying the gain to be used in making readings. The gain must have a value of 1, 8, 64, or 512.

Discussion

The gain set by the Set__gain subroutine permanently overrides the gain set by a previous call to Config__0 or Set__gain.

The description of the Config__0 subroutine in this section discusses the lsb values for each gain setting.

SET__UNITS

Set__units sets the reporting units for readings taken by the Measurement Library subroutines. These units are base, standard, or user units.

Syntax

```
    BASIC: set_units(name,units[,multiplier[,offset]]

   Pascal: PROCEDURE set_units(name: str255;
                               units: str255;
                               multiplier: real;
                               offset: real);
```

Parameters

name:	a string or string literal specifying the ADC name assigned by the Config_0 call.
units:	a string or string literal specifying the units to be used to return ADC data. The units must be base, standard, or user. (Only the first character is significant.)
multiplier:	(user units only) a REAL number specifying the multiplier to be used. (See the discussion, below.) This parameter is not used with base or standard units.
offset:	(user units only) a REAL number specifying the offset to be used. (See the discussion, below.) This parameter is not used with base or standard units.

Discussion

The three types of units are defined as:

```
      base = raw ADC reading returned as a binary integer
  standard = base unit reading adjusted for gain and calibration,
             expressed as a real number
      user = standard * multiplier + offset
```

Set__units permanently overrides any previous units specification made by a Config__0 or Set__units call.

For more information on reporting units, see the description of the Config__0 call in this section and the discussion of reporting units in Section 1 of this manual.

SYSTEM_INIT

System_init initializes all configured cards. For each card, it performs the same functions as the Init subroutine.

Syntax

```
   BASIC:  System_init

  Pascal:  PROCEDURE system_init;
```

MESSAGES

APPENDIX

A

The Measurement Library reports errors with the messages listed on the next page. The list gives the message number used, the meaning of the message, and the calls which can return the message.

On the page after that are listed the Pascal-related error messages that may be returned to your BASIC program as a result of a Measurement Library subroutine call.

Measurement Library Messages

Message Number	Meaning	Reporting Calls
801	Unsupported Model	Config_0
804	Array Too Small	Sequential_scan, Random_scan
812	Name Not Configured	All except System_init and Config_0
815	Use of uninitialized name	All except Init, System_init, and Config_0
835	Illegal select code	Config_0
837	Specified card not at select code	Init
838	Illegal name	Config_0
850	Unsupported Gain	Input, Random_scan, Set_gain, and Config_0
851	Pace out of Range	Input, Sequential_scan, Random_scan, Calibrate, and Config_0
852	Repeat Specification Error	Sequential_scan, Random_scan, and Calibrate
853	Illegal Channel Number	Input, Random_scan, Sequential_scan, and Calibrate
854	Not allowed in Interrupt Mode	Random_scan, Sequential_scan, Set_gain, Set_units, and Calibrate
855	Common mode overrange	Input, Sequential_scan, Random_scan, and Calibrate
856	Normal ADC overrange (must be enabled by Config_0)	Input, Sequential_scan, Random_scan, and Calibrate
857	Pace timing error	Input (interrupt mode only), Sequential_scan, Random_scan, and Calibrate
858	Unsupported units	Set_units, Config_0
859	Max number of names exceeded	Config_0
860	Offsets out of range (card defective or calibration channel not shorted)	Calibrate

Pascal-related Messages

BASIC Message Number	Pascal Message Number	Meaning
400	0	Normal termination
399	-1	Abnormal termination
398	-2	Not enough memory
397	-3	Reference to NIL pointer
396	-4	Integer overflow
395	-5	Divide by zero
394	-6	Real math overflow; number too large
393	-7	Real math underflow; number too small
392	-8	Value range error
391	-9	Case value range error
390	-10	Non-zero I/O result
389	-11	CPU word access to odd address
388*	-12	CPU bus error
387	-13	Illegal CPU instruction
386	-14	CPU privilege violation
385	-15	Bad argument - SIN/COS
384	-16	Bad argument - Natural Log
383	-17	Bad argument - SQRT (square root)
382	-18	Bad argument - real/BCD conversion
381	-19	Bad argument - BCD/real conversion
380	-20	Stopped by user
379	-21	Unassigned CPU trap
378	-22	Reserved
377	-23	Reserved
376	-24	Macro parameter not 0..9 or a..z
375	-25	Undefined macro parameter
374	-26	Error in I/O subsystem
373	-27	Graphics error

*** In a BASIC 2.0 system you may get this error if you try to initialize an ADC card for a select code that contains no card.**

QUICK REFERENCE

This appendix is a quick reference guide to the Measurement Library subroutine calls. We've squeezed the call summaries into small type so that they fit onto the next page; you can take that page out of this manual and hang it on your wall for quick reference.

BASIC programmers please note that the parameters that you pass to the Measurement Library subroutines must be properly typed (integer, real, or string). If you don't know the type of a parameter, you can look it up in Section 2 of this manual.

CALIBRATE

```
BASIC: Calibrate(name,channel,pace,number)

Pascal: PROCEDURE calibrate(name: str255;
                            chan: shortint;
                            pace: real;
                            number: shortint);
```

CONFIG__0

```
BASIC: Config_0(name,model[,select_code[,gain[,
               pace[,report_error[,units[,
               multiplier[,offset]]]]]]])

Pascal: PROCEDURE config_0(name: str255;
                           model: str255;
                           select_code: shortint;
                           gain: shortint;
                           pace: real;
                           report_error: str255;
                           units: str255;
                           multiplier: real;
                           offset: real);
```

DISABLE__INTR

```
BASIC: Disable_intr(name)

Pascal: PROCEDURE disable_intr(name: str255);
```

ENABLE__INTR

```
BASIC: Enable_intr(name)

Pascal: PROCEDURE enable_intr(name: str255);
```

INIT

```
BASIC: Init(name)

Pascal: PROCEDURE init(name: str255);
```

INPUT / READ__CHANNEL

```
BASIC: Input(name,channel,datum[,gain[,pace]])

Pascal: PROCEDURE read_channel(name: str255;
                               channel: shortint;
                               VAR datum: real;
                               gain: shortint;
                               pace: real);
```

MEAS__LIB__INIT

```
BASIC: Meas_lib_init

Pascal: PROCEDURE meas_lib_init;
```

RANDOM__SCAN

```
BASIC: Random_scan(name,chan_array(*),data_array(*)[,
                   rept[,pace_array(*)[,
                   gain_array(*)]]])

Pascal: PROCEDURE random_scan(name: str255;
                              chan_size: integer;
                              chan_array: anyptr;
                              data_size: integer;
                              data_array: anyptr;
                              rept: shortint;
                              pace_size: integer;
                              pace_array: anyptr;
                              gain_size: integer;
                              gain_array: anyptr);
```

SEQUENTIAL__SCAN

```
BASIC: Sequential_scan(name,start,stop,pace,
                       data_array(*)[,rept])

Pascal: PROCEDURE sequential_scan(name: str255;
                                  start: shortint;
                                  stop: shortint;
                                  pace: real;
                                  data_size: integer;
                                  data_array: anyptr;
                                  rept: shortint);
```

SET__GAIN

```
BASIC: Set_gain(name,gain)

Pascal: PROCEDURE set_gain(name: str255;
                           gain: shortint);
```

SET__UNITS

```
BASIC: Set_units(name,units[,multiplier[,offset]])

Pascal: PROCEDURE set_units(name: str255;
                            units: str255;
                            multiplier: real;
                            offset: real);
```

SYSTEM__INIT

```
BASIC: System_init

Pascal. PROCEDURE system_init;
```

INDEX

HEWLETT
PACKARD

Reorder No. or
Manual Part No.

98645-90001-E0684-U0788