PROGRAMMER'S GUIDE TO:

www.andor.com

Version 3.2

PAGE

PAGE

SECTION 2 (continued)

PAGE

SECTION 2 (continued)

PAGE

SECTION 2 (continued)

PAGE

SECTION 1 - INTRODUCTION TO ANDOR BASIC

The **Andor Basic** Programming Language allows you to create your own routines to automate data acquisition and data processing.

- Section 1 of this **Programmer's Guide** explains the underlying syntax of the language and provides an overview of the ready-made functions that are provided with **Andor Basic**. These functions have been created specifically to cater for a broad range of spectroscopic processing tasks. The Function Overview at the start of Section 1 groups Andor Basic functions by topic. Functions listed in the Overview represent the latest implementation of Andor Basic. In many instances, a single function in the Overview may encompass a range of functionality that, in previous releases of Andor Basic, was represented by several separate functions. For the sake of completeness and backward compatibility the earlier functions remain in this Programmer's Guide, and their names are labeled **'superseded by…'**. However, the new functions (which encourage a more intuitive programming style) are strongly recommended.
- In Section 2 the syntax and use of each function is described in detail. A brief program extract is included for each function. Functions whose names are marked 'not PDA' are not available for use with Photodiode Arrays. Functions whose names are marked 'not CCD' are not available for use with CCD detectors
- Finally, Section 3 provides a selection of fully developed sample programs - each of which implements a typical processing task from the field of multichannel spectroscopy.

You can run **Andor Basic** programs by selecting **New Program** from the **File** menu, entering the program in the **Program Editor Window** and clicking the **Run Program** button on the **Main Window**. Please refer to the User's Guide to **Andor Basic** for further details of how to edit, save and run programs.

GROUP FUNCTION OVERVIEW & SPECIAL VARIABLES

Calibration

- copyxcal coefficient$
- xcal xpix

Data Acquisition

- cooler
- run
- SetAccumulate SetAcquisitionType SetDataType SetFastKinetics SetFVB SetGain SetGate SetGateMode SetImage SetKinetics SetMultiTracks SetShutter SetShutterTransferTime SetSingleTrack SetTemperature SetTriggerMode ShowTimings

Data Set Manipulation

- area
- carea cfft copy create cursorx cursory
- detectorx detectory
- fft fwhm
- GetNumberSeries GetNumberTracks
- icfft imagex imagey InfoText
- KineticSlice
- maxpos maxval minpos minval
- SetDataLabel SetXLabel smooth1 smooth2 smooth3
- rayremove2
- zero

Data Window

- ActiveKineticPosition ActiveOverlay ActiveTab ActiveTrack AddOverlay
- ChangeDisplay CloseWindow
- DetachOverlay
- GraphData GraphEscape GraphFont GraphFrame GraphGrid GraphLabel GraphLine GraphNew GraphPrint GraphTextSize GraphXAxis GraphXLabel GraphYAxis GraphYLabel
- MaximizeWindow MinimizeWindow MoveWindow
- PositionCursor
- rescale reset RestoreWindow
- scale ScaleData ScaleToActive scalex scaley
- TopWindow
- update

File Handling

- close
- ExportBMP ExportTiff
- kill
- load LoadAsciiXY
- Poly
- read
- save separator
- write

GPIB

- ibclr ibconfig ibfind ibrd ibrdf ibrdi ibrsc ibsic ibwrt ibwrtf

Hardware

- auxin auxout
- inport inportb
- outport outportb

Keyboard

- input
- key

Mathematical

- abs acos alog asin atan
- cos diff
- exp
- ln
- log
- mod
- sin sqrt
- tan

Miscellaneous

- beep
- date$ delay
- time time$

NETWORK

The following commands are used to control Andor cameras across Ethernet. An example program is shown in **Section 3**.

- abort
- connect
- disconnect
- GetAcquiredData GetStatus
- shutdown

Serial Port

- baud
- comread comwrite
- handshake
- ignore
- newline
- stopbits
- terminator

String Handling

- asc
- chr$
- instr
- left$ len
- mid$
- right$
- str$
- val

Text Screen

- cls
- format
- output
- print
- update

Special Variables

- Background Bg
- Cal
- Iberr ibsta
- live
- ref
- reference
- sig
- Signal

Reserved Words

- and
- else end endif Eor
- gosub
- if input
- or
- Output
- print program
- return
- then
- Wend
- while

MANIPULATING DATA

Accessing Data

A data set is referenced by putting a hash sign # in front of its number. **Andor Basic** recognizes anything starting with # as a data set. For a full explanation of data sets, see the section WORKING WITH DATA in the User's Guide supplied with your system.

#5 refers to data set 5

Because it is necessary to reference the data within a data set, a further convention is introduced here. The name of the data is specified after the data set number, linked to it by an underscore. The allowed names are signal, background, reference, live and cal. Abbreviations of sig, bg and ref are also accepted.

#5_ref refers to the reference data in data set 5

Most manipulations in **Andor Basic** deal with the named data rather than the complete data set. For convenience the default data is signal.

#2=#3 *copies the data from signal data in data set 3 to signal data in data set 2*
Further examples here will use the more complete form - i.e.

#2_sig=#3_sig
Further examples here will use the more complete form - i.e.

#2_sig=#3_sig

At this stage the data referenced by #2_sig possibly consists of 1024 pixels if it is a one dimensional image, or 1024 by 256 for a two dimensional image. (The actual numbers may vary depending on the detector in use.) In order to reference a particular pixel, the data are treated as an array and the individual pixels specified within square brackets [ ]. Square brackets indicate absolute pixel co-ordinates on the CCD.

#2_sig[312]	refers to pixel 312 of the signal data in data set 2
#3_sig[300,125]	refers to a pixel with x-y coordinates of 300,125

Creating a Data Set

Individual pixels within a data set may now be manipulated - i.e.

#2_sig[312]=5

A data set can be created by an acquisition. This is automatically given the reserved data set number, zero. A data set may also be saved to disk and reloaded at a later time. A data set which only exists on disk does not have a data set number associated with it. This number is specified when it is being loaded from disk, either automatically, where the lowest unused number is given, or manually i.e.

load("penray.sif") a data set number will be assigned automatically

load(#100, "penray.sif") load 'penray.sif' to data set 100

If the data set has been loaded from disk it will also have a filename associated with it which will appear at the top of the data set window.

Creating a Subset

Sometimes it is required to construct a new data set based on an existing data set. The simplest form of this is:

#2_sig=#3_sig create data set #2 and copy #3_sig to #2_sig

Using the Range Operator

To specify part of a data set, i.e. a range of pixels in either one or two dimensions use can be made of the range operator <<.

#2_sig[200<<300] specifies all pixels from #2_sig[200] through #2_sig[300]

#2_sig[200<<300,150<<200] specifies an area 101 by 51 pixels

A subset of the original data may be specified and copied as follows:

#3_sig=#2_sig[200<<300,150<<200]	data set #3 is created to receive the subset of #2

The range operator may also be used to specify one complete row of a two dimensional data set as follows:

#5_sig[1<<1024,53] specifies row 53

Kinetics

When operating in kinetics mode each of the named data sets may actually consist of any number of kinetic frames which have been acquired at intervals over a period of time. In order to be able to specify which of the kinetic frames is being referenced another convention is introduced here.

The kinetic frame is specified by putting its number in braces { } after the data set name.

- **#2_ref{53} refers to frame 53 of the reference data of kinetic data set #*2***

If it is required to reference a particular pixel, this again comes at the end using square brackets.

- **#2_ref{53}[512] refers to pixel 512 of the above frame**

To return a value for a particular area within a kinetic series, use the **area** command, and indicate the number of the kinetic frame within braces { }.

- **area**(#2_sig{4},200,400)

An Alternative Syntax

So far we have used a pixel's co-ordinates to access its data value. Square brackets [ ] are used to indicate absolute pixel co-ordinates on the CCD. In some cases, however, there may be more convenient ways to return a data value.

Say you acquire data using Single-Track readout mode, from a track covering rows 20 to 30 on the CCD. For any given column in the track, The system returns a single, binned, data value: the system stores the same data value for each pixel in the column. However, if you are using simple pixel co-ordinates, you need to specify appropriate *x* and *y* values to return a data value for, say, the 300th column in the track.

value=#0_sig[300,20]

Simply stating #0_sig[300] will not work. This translates to #0_sig[300,1], and since the data from the pixel at *x*=300, *y*=1 was not stored (the pixel is outside the track) a value of 0 is returned.

To help simplify this type of situation an alternative syntax is available. This syntax uses ( ) brackets in place of the [ ] brackets. In the alternative syntax, the above example can be written as

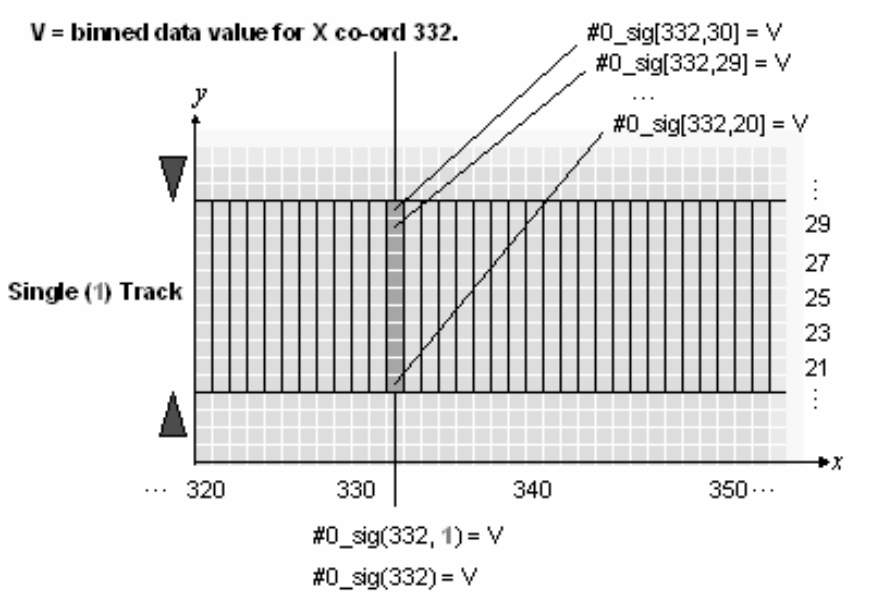
value=#0_sig(300,1)

or

value=#0_sig(300)

In this syntax the "co-ordinates" do not refer to the actual pixel co-ordinates. Rather they refer to stored data values within a track or tracks of data (and, in the case of an image, may refer data values corresponding to superpixels). This syntax is explained in the examples that follow

Single-Track



Let us assume that data have been acquired in Single-Track readout mode. The Track Height is 11, centred on row 25 (thus the track covers rows 20 to 30). To access the 332nd data point in the track (*x*=332) you may use any of the following expressions

#0_sig[332,20], #0_sig[332,21], etc.;

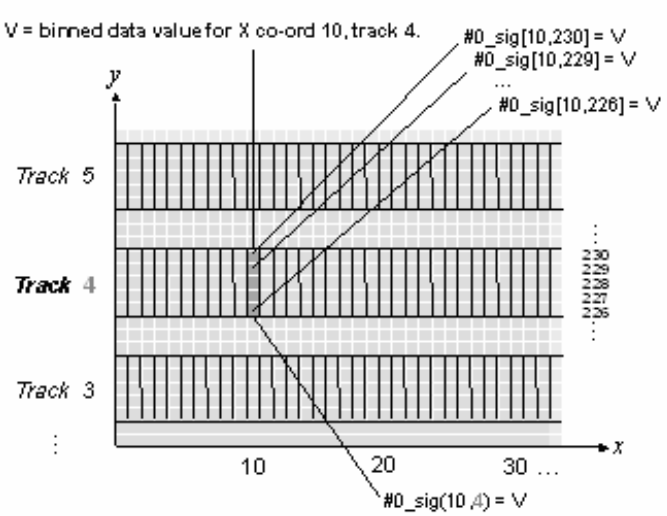
or

either #0_sig(332,1) or #0_sig(332).

In #0_sig(332,1) the optional 1 refers to the fact that data from the single track are being treated by The system as a one-dimensional array of values. Note that if you are using the range operator with the ( ) syntax you must specify the second co-ordinate:

#1 = #0_sig(200<<400) **Wrong !**
#1 = #0_sig(200<<400, 1) **OK !**
#1 = #0_sig(200<<400, 1<<1) **OK !**

Multi-Track



Let us assume next that data have been acquired in Multi-Track readout mode. There are 5 tracks of height 5 rows and an offset of 0. To access the 10th data point (*x*=10) on the 4th track (which in this instance covers rows 226 to 230) you may use any of the following expressions:

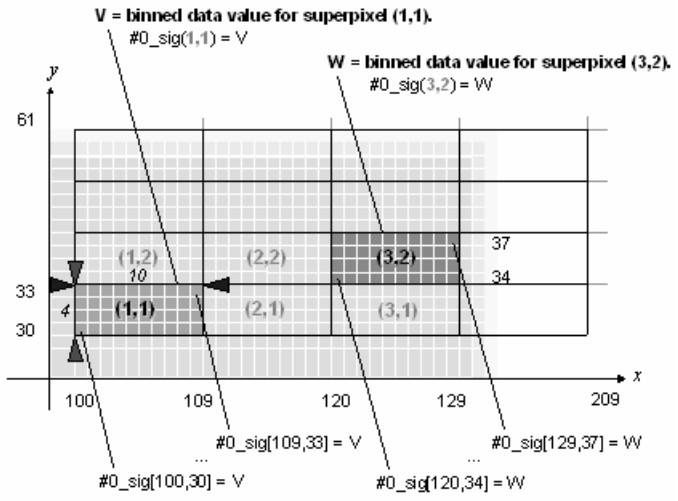
#0_sig[10,226], #0_sig[10,227], etc.;

or

#0_sig(10,4)

The 4 in #0_sig(10,4) references the 4th of the 5 rows in a notional two-dimensional array being used internally by The system to store data values for each track.

Image



In this example a subimage has been acquired with the following co-ordinates:

left (*x1*)	100
right (*x2*)	209
bottom (*y1*)	30
top (*y2*)	61

and with a binning pattern that creates superpixels of width (*x*) 10 pixels and height (*y*) 4 pixels.

Using the [ ] syntax the first data value read can be accessed using any absolute pixel co-ordinates that lie within the first superpixel - e.g.

#0_sig[100,30], #0_sig[101,32], #0_sig[108,33], #0_sig[109,33]
Similarly, a multitude of expressions can be used to return the other data values.
With the ( ) syntax, the expression to access the first data point is

#0_sig(1,1)
while the last value (there is a total of 88 superpixels in this example) is read using:

#0_sig(11,8)

Again, the notation with the ( ) brackets references The system's internal representation of the superpixels' values as a two dimensional data array.

Statements

A statement is an instruction to the computer. Statements are executed sequentially, from the beginning of the program to the end, unless they are redirected by one of the program flow statements. If the <Esc> key is pressed, the program will be terminated.

PROGRAM FLOW

The statements which control the program flow are as follows:

- **While ... wend**
- **if ... then ... else ... endif**
- **Gosub ... label ... return**
- **End**

while ... wend

The **while ... wend** statements allow the program to repeat a number of statements until a test condition is matched, e.g. a loop counter counts down to zero. The syntax is shown below. The indentation of the statements is not essential, but it helps to clarify the section of the statement which is repeated.

```
while (expression)
  statement1
  statement2
          ...
wend
```

statement1 and statement2, etc. will be executed repeatedly while (expression) is true.

if ... then ... else

The **if ... then** statement allows the conditional execution of statements. Statements following the **then** keyword will only be executed if the expression following the **if** keyword is true.

Example 1:

if (expression) **then** statement

Additionally, if the expression following the **if** keyword is false, a different set of statements can be executed following the **else** keyword. The **if ... then** statement can be used in a variety of ways, as demonstrated below:

Example 2:

if (expression) **then** statement1 **else** statement2

The above example is a single line statement. There is also a multi-line version which requires the use of **endif** to define the end of the operation.

Example 3:

```
if (expression) then
  statement1
  statement2
   ....
else
  statement3
  statement4
   .....
endif
```

NOTE: More than one statement can be placed on the same line by separating the statements with a colon.

Example 4:

```
a=1:b=2:c=3                :rem initialize variables

if (a > b) then
  print(a;" is greater than ";b)
else
  print(a;" is less than or equal to ";b)
endif
```

gosub ... label ... return

Subroutines allow different parts of the program to reuse code, e.g. a plotting routine.

The syntax is as follows:-

```
statement1
statement2
gosub .label
statement3
statement4
....
end

.label
  statement0
  statement1
    ....
return
```

NOTE: A period <.> must precede the actual label name

Example:

```
Print("First line")
Print("Second Line")
gosub .funct1
Print("Third line")
gosub .funct2
Print("Forth Line")
end

.funct1
  Print("First line of subroutine Funct1")
  Print("Second line of subroutine Funct1")
return
.funct2
  Print("First line of subroutine Funct2")
  Print("Second line of subroutine Funct2")
return
```

comments

The **rem** keyword allows the inclusion of comments in a program. Any text after a **rem** is ignored until the end of the line.

In order to allow larger sections of a program to be commented out, the use of C type comments is also permitted. The start of a block is marked by /* and the end of the block is marked by */ as in Example 2.

The use of nested comments is permitted. You may also comment out single lines using the C++ type // as in Example 3.

Example 1:

```
a=0          :rem initialize the variable a to 0
```

Example 2:

```
a=1
/* This is a comment
   which may extend
   over more than one line */
print a
```

Example 3:

```
a=2
// Comment out this single line
print a
```

EXPRESSIONS

An expression is part of a statement that has a numeric value. For example, the expression

a > 10

evaluates to either 0 or 1 depending on the value of 'a'. This expression is called a conditional expression and will be evaluated to be a logical true or false value. If the expression is true, then the result will be 1. If the expression is false, then the result will be zero.

NOTE: Expressions can be combinations of operators, variables and constants.

Arithmetic Operators

There are 5 arithmetic operators:

*	multiplication
/	division
+	addition
-	subtraction
^	raise to the power

Example 1:

Multiply the signal data in data set #1 by 5 and assign the answer to the reference data in data set #2

#2_ref = #1_sig * 5

Example 2:

Divide the signal data in #2 by the signal data in #3 and assign the answer to the signal data in #4

#4_sig = #2_sig / #3_sig

Example 3:

'x' is assigned the value of 'y' raised to the power of three.

x = y ^ 3

Bitwise Operators

There are 3 bitwise operators which can be used in an Andor Basic program:

or	Bitwise OR operator
eor	Bitwise exclusive OR operator
and	Bitwise AND operator

These operate only on integers and, if used with a floating point number, the number will first be truncated to be an integer.
'Bitwise' operators act on the individual bits of the variables.

Example 1:
7 **or** 6 is converted to binary form: 0111 **or** 0110
to give an answer of: 0111, i.e. 7

Example 2:
7 **eor** 6 is converted to binary form: 0111 **eor** 0110
to give an answer of: 0001, i.e. 1

Example 3:
5 **and** 3 is converted to binary form: 0101 **and** 0011
to give an answer of: 0001, i.e. 1

and, more generally,

Example 4:
if (a**==**2) **and** (b**==**4) **then ?**"a and b are correct"

i.e. this checks to see if ‘a’ and ‘b’ have the desired values.

Logical Operators

There are three logical operators which can be used in an AndorMCD program.

&&	logical AND operator
||	logical OR operator
!	logical negation operator

These can be used with floating point numbers.

'Logical' operators act on the numerical value of the variables.

Example 1:

if (a**==**12) **&&** (b**==**23) **then ?**a

will print to the screen the value of 'a' if it is equal to 12 and 'b' is equal to 23

Example 2:

if (a**==**12) **||** (b**==**23) **then ?**a

will print to the screen the value of 'a' if it is equal to 12 or 'b' is equal to 23

Example 3:

if !(a**==**12) **then ?**a

will print to the screen the value of 'a' if it is not equal to 12

Relational Operators

The following relational operators can be used:

==	equal to (note that the single '=' CANNOT be used)
<>	not equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Example 1:

This expression will evaluate true if the variable 'a' is equal to 2:

a==2

Example 2:

This conditional expression will return a result of 1 if the variable 'b' is greater than 100:

b>100

Variable Names

Variable names are used to represent values in a program. The following rules apply:-

- Names are case sensitive and may be of any length.
- Only alphanumeric characters and the underscore character may be used.
- The first character must be alphabetic.
- Underscores are permitted in string variable names, but are not permitted in numeric variable names, i.e. help_me$ is permissible but a_b is not.
- The dollar sign at the end of a name indicates a string variable (text).

FILE HANDLING

File Handling is one area in which the Andor programming language differs significantly from other 'basics'. The number of functions has been kept to a minimum and, in fact, when working with text files, most operations can be accomplished by using the read and write functions. However, it is necessary to consider two types of files:

- **Text files**
- **Data files**

Text Files

There is no single command for opening a file, and only one file can be in use at any given time. Changing the file name closes the file that is open, and will then open a new one of that name. There is a close function in case a file is being shared on a network, but it is more common to rely on the program to close all files when it ends.

The command:

write("report.dat",MyData$)

serves several purposes. Firstly, it will check if any file is already open. If there is a file open whose name differs from that which has been entered (in this example "report.dat"), the function will close the old file and open "report.dat". If the names are the same, then no files will be opened or closed, because this file is already open. If there is no file open, then a file of this name ("report.dat") will be opened. If the file that it is being asked to open does not already exist, then this file will be created. The command will then proceed to append 'MyData$' to the end of the file.
The command :

read("report.dat",MyData$)

carries out the same sequence of opening and closing as described above, and then proceeds to read from the file.

NOTE: Because write appends to the end of existing files, it is necessary to have a kill function which will delete files in the current directory. No confirmation is asked for, because this would interrupt the program flow. As a result, this function should be treated with extreme caution.

Data Sets

Individual data sets may be loaded or saved to disk, for example:

load(#1,"trace1.sif")
save(#2,"trace2.sif")

These two commands will load the file "trace1.sif" to data set #1 and save the contents of data set #2 as "trace2.sif", respectively.

PROGRAM EXAMPLES

Example 1

```
rem  This program starts by loading a data set from disk and
rem  assigning it the first free data set number.  This number
rem  will be used to reference the data set in subsequent lines
rem  of the program.

filename$="C:\instaspc\images\penray.sif"   :rem example path
dset=load(filename$)              :rem load a data set
if dset then                      :rem if successful
  mpos=maxpos(#dset,1,1024)       :rem get position of maximum
  mval=maxval(#dset,1,1024)       :rem get maximum value

rem  realistically it would be quicker just to return the value
rem  at #dset[mpos]

rem  A simple print command to show the result.

  print("The maximum value of ";filename$;" is ";mval;" at          position ";mpos)

rem Mpos is the pixel position. The wavelength might be more
rem useful.

  mwav=xcal(#dset,mpos)
  print("The maximum value of ";filename$;" is ";mval;" at          wavelength ";mwav;"nm")

endif
```

Example 2

```
rem  This example assumes that the file "kin50.sif"
rem  is a kinetic data set consisting of 50 traces
rem  acquired at equal intervals over a period of time.
rem  Perform a ‘kinetic slice’ on the signal data
rem  to show how one pixel has changed with time.
rem  Note the use of the {} brackets to define the kinetic frame.
rem  Screen updates have been disabled to speed up
rem  execution of the program. Note also that there is now a
rem  KineticSlice( ) function to automate this process.

update(0)

rem  Load kinetic data set.  Example directory path shown.

load(#5_sig,"C:\instaspc\images\kin50.sif")
create(#10_sig,50)          :rem Make new data set.
i=1
while(i<51)
  #10_sig[i]=#5{i}[350]     :rem Make kinetic slice.
  i=i+1
wend

update()            :rem Force an update
```

SECTION 2 - LIST OF FUNCTIONS

abort

Syntax: abort( )

Description: Aborts an acquisition taking place on a remote machine connected to Ethernet.

See also: connectconnect disconnect

Example: This example program connects to a remote machine with an IP address of 111.222.333.444 and aborts an acquisition taking place:

Connect ("111.222.333.444")
abort ()
disconnect ()

---

abs

Syntax: abs(*x*)

Description: Returns the absolute value of *x*.

Example: x=-0.5
y=**abs**(x)
print(y)

Result: 0.5

---

acos

Syntax: acos(*x*)

Description: Calculates the arc cosine. The function expects an argument between -1.0 and 1.0 and returns a floating point number in the range 0 to pi.

See also: sin cos tan acos asin atan

Example: x=0.5
y=**acos**(x)
print(y)

Result: 1.0472

ActiveKineticPosition

Syntax: ActiveKineticPosition(*#dataset*, *seriesposition*)

Description: Moves to the kinetic series position specified in the parameter *seriesposition*. This function works on the active data set and is applicable when the data is a kinetic series.

See also: ActiveOverlay ActiveTrack AddOverlay PositionCursor

Example: **rem** Acquire a kinetic series and move to the 10th
rem acquisition in the series.

```
SetDataType(1)                  :rem counts
SetAcquisitionMode(3)           :rem setup kinetics
SetKineticNumber(20)            :rem 20 in kinetic series
SetAcquisitionType(0)           :rem signal
run()
ActiveKineticPosition(#0,10)    :rem move to the 10th
                                :rem acquisition in the series.
```

ActiveOverlay

Syntax: ActiveOverlay(*#dataset*,*overlaynumber*)

Description: Data from different data sets can be displayed as different colored traces in the same data window . This function allows you to select the trace you want to manipulate (i.e. the 'active' trace or overlay). The function has the same effect as pressing on one of the colored buttons in the window. *#dataset* is the number on the window displaying the data and *overlaynumber* is a number between 0 and 8, where 0 is the original data set and numbers 1 - 8 are the imported data sets.

See also: AddOverlay DetachOverlay

Example:

```
rem Load three different data sets and display them in the
rem same window. Choose the second data set as the active
rem one.

load(#1,"data1.sif")                    :rem load data
load(#2,"data2.sif")
load(#3,"data3.sif")
overlay1=AddOverlay(#1,#2)              :rem copy data into the same
                                        :rem window
overlay2=AddOverlay(#1,#3)
ActiveOverlay(#1,overlay1)              :rem make #2 the active data
                        :rem set.
scalex(#1,200,400)                      :rem #2 in x.
```

ActiveTab

Syntax: ActiveTab(*#dset*,*tab*)

Description: Makes either signal, reference or background data the active display. *#dset* is the window displaying the data, and *tab* is either *sig*, *ref* or *bg*.

See also: ActiveOverlay ChangeDisplay

Example: **rem** Acquire signal, reference and background.
rem Make signal the active display.

```
SetAcquisitionType(0)      :rem signal
SetAcquisitionMode(1)      :rem single scan
SetReadoutMode(0)          :rem fully vertically binned
SetDataType(1)             :rem counts
run()
SetAcquisitionType(1)      :rem background
run()
SetAcquisitionType(2)      :rem reference
run()
ActiveTab(#0,sig)          :rem make signal the active display
```

ActiveTrack (not PDA)

Syntax: ActiveTrack(*#dset*,*trackposition*)

Description: Moves to the track specified in the parameter *trackposition*. This function works on the active data set and is applicable when the data are multi-track or full image. If the data form a multi-track image that has 5 tracks, for example, then *trackposition* can be any number between 1 and 5. If the data are a full resolution image with, for example, 1024x256 pixels, *trackposition* can be any number between 1 and 256

See also: ActiveKineticPosition ActiveOverlay AddOverlay PositionCursor

Example:
```
load(#1,"image.sif")      :rem load an image
ActiveTrack(#1,100)       :rem move to track 100
```

Result: Returns the overlaynumber. Each Data set that is added to the window will be assigned a number between 1 and 8

AddOverlay

Syntax: AddOverlay(*#dataset1*,*#dataset2*)

Description: This function allows spectra from different data sets to be displayed together in the same window. The function will place *#dataset2* in the same window as *#dataset1*. Each data set will be displayed in a different color. Up to nine different data sets can be displayed in a single window.

See also: ActiveOverlay DetachOverlay

Example:
```
rem Load two spectra and display them in the same window.

load(#1,"data1.sif")                    :rem load data
load(#2,"data2.sif")

overlaynumber=AddOverlay(#1,#2)    :rem copy data into same   :rem window
```

Result: Returns the overlaynumber. Each Data set that is added to the window will be assigned a number between 1 and 8.

---

alog

Syntax: alog(*x*) or alog(*#data*)

Description: Calculates the inverse logarithm to the base 10. The function can be applied to single values or to named data within a data set. If used with named data, an assignment (either to itself or to other named data within a data set) must accompany this function.

See also: log exp ln

Example 1:
```
#2=alog(#1_sig)
```

Example 2:
```
print(alog(2.00))
```

Result: 100

---

area

Syntax: area(*#data*, *start*, *end*) or area(*#data*, *start*, *end, flag)*

Description: Calculates the area under the trace contained in *#data*, from *start* to *end* inclusive, using the trapezoid rule. The function returns this area as a floating point number. The default action is to perform the calculation using pixels as the units on the X axis. The return value in this case is the total number of counts within the defined region. If *flag*=1 then the calculation is performed using calibrated units as the *start* and *end* points. The return value in this case is dependent on the calibration on the X axis. When calculating the area for a radiometric trace, the area function will return the answer in μw/cm²

See also: carea fft smooth1

Example 1:
```
Startpixel=xpix(#1_sig,650)
  rem calculate pixel which corresponds to 650nm
endpixel=xpix(#1_sig,720)
  rem calculate pixel which corresponds to 720nm
a=area(#1_sig,startpixel,endpixel)
  rem calculate area from 650 to 720nm
print("Area = ";a)
```

Example 2:
```
a=area(#1_sig,650,750,1)
  rem The 1 flag indicates the calculation will be performed
  rem with calibrated data.
```

Example 3:
```
a=area(#1_sig{4},200,400)
  rem the braces { } contain the number of a kinetic frame
```

Result:
```
a=area(#1_sig{4},200,400)
  rem the braces { } contain the number of a kinetic frame
```

---

asc

Syntax: asc(*text*)

Description: Returns a numeric value which is the ASCII code for the first character of *text.* The inverse of this function is CHR$

See also: chr$

Example:
```
x$="InstaSpec"
print(asc(x$))
```

Result: 73

---

asin

Syntax: asin(*x*)

Description: Calculates the arc sine. The function expects an argument between -1.0 and 1.0, and returns a floating point number in the range -pi/2 to pi/2.

See also: acos atan sin cos tan

Example:
```
x=0.5
y=asin(x)
print(y)
```

Result: 0.523599

---

atan

Syntax: atan(*x*)

Description: Calculates the arc tangent. The function expects any floating point number as an argument, and will return a floating point number in the range -pi to pi

See also: acos asin sin cos tan

Example:
```
x=0.5
y=atan(x)
print(y)
```

Result: 0.463648

---

auxin

Syntax: auxin(*port*)

Description: Returns the state of the auxiliary input connector specified by *port*. *port* may take a value 1 to 4 corresponding to each of the four inputs available on the InstaSpec card. A TTL High signal on the corresponding pin of the auxiliary connector will result in **auxin** returning a value of 1. A TTL Low signal will return a value of 0.

See also: auxout outportb inportb

Example:
```
a=auxin(1)
if a == 1 then run()
```

---

auxout

Syntax: auxout(*port,value*)

Description: The auxiliary output ports 1,2,3 and 4 are controlled with **auxout** which sets the specified *port* low for a zero *value* and high for a non-zero *value*. A port which is set high has a TTL High signal on the corresponding pin of the auxiliary connector.

See also: auxin outportb inportb

Example:
```
auxout(1,1):rem set port high
auxout(1,0):rem set port low
```

---

baud

Syntax: baud(*comport,baud_rate*)

Description: Sets the *baud_rate* for the specified *comport*. The default baud rate is 9600 and allowed values are between 110 and 56 000. Com ports 1,2,3 and 4 are supported.

See also: comwrite comread handshake stopbits

Example: baud(1,4800)

beep

Syntax: beep( )

Description: This function requires no arguments. When called the function uses the computer speaker to make a noise.

See also: delay

Example:
```
beep()
print("Error"):rem Make a noise to indicate an error has
        :rem occurred
```

carea

Syntax: carea(*#data*,*start_pixel*,*end_pixel*)
carea(*#data,start_pixel,end_pixel,flag)*

Description: Returns the corrected area under *#data* from pixels *start_pixel* to *end_pixel* inclusive, using the trapezoid rule. To correct the area for a sloping baseline, a new baseline is drawn from the *start_pixel* to the *end_pixel* before the area calculation is performed. The function returns this area as a floating point number. The default action is to perform the calculation using pixels as the units on the X axis. The return value in this case is the total number of counts within the defined region. If *flag*=1 then the calculation is performed using calibrated units as the *start* and *end* points. The return value in this case is dependent on the calibration on the X axis.

See also: Area

Example 1: y=**carea**(#12_sig,1,512)

Example 2: **rem** Calc area between 600 and 700nm of signal data in data set #1

y=**carea**(#1_sig,**xpix**(#1_sig,600),**xpix**(#1_sig,700))

Example 3: **rem** Use calibrated units.

y=**carea**(#1_sig,600,700,1).

cfft

Syntax: cfft(*#data*)

Description: Calculates the complex FFT of *#data*, where *#data* has a width corresponding to a power of 2 (typically 512, 1024 or 2048). The real and imaginary parts are held as an array of complex numbers in the data set to which they are assigned. They are held as real / imaginary pairs occupying adjacent locations in the data set. As in the example, an assignment to a data set must accompany this function.

See also: area fft icfft carea smooth1

Example:

```
#2_sig=cfft(#1_sig)          :rem puts the complex FFT spectrum of
                             :rem #1_sig into #2_sig
#2_sig[4]=0
#2_sig[5]=0                  :rem remove a frequency (1st harmonic)
#1_sig=icfft(#2_sig)         :rem change frequency back to spatial
```

ChangeDisplay (not PDA)

Syntax: ChangeDisplay(#*dset*,*mode*)

Description: Changes the display to 2D, 3D or Image. The parameter *mode* can have the following values: 2D=*3*; Image=*2*; 3D=*1*

Example:
```
rem Acquire a single scan, full resolution image
rem and change the display to image

SetDataType(1)            :rem counts
SetAcquisitionType(0)     :rem signal
SetAcquisitionMode(1)     :rem single scan
SetReadoutMode(4)         :rem full resolution image
run()
ChangeDisplay(#0,2)       :rem change display to image
```

chr$

Syntax: chr$(*x*)

Description: Converts an ASCII code to its corresponding character where *x* is any numeric expression in the range 0 to 255.

See also: asc

Example:
```
print(chr$(73))
```

close

Syntax: close( )

Description: Closes any open (text-based) data file (as opposed to data set, for which see **CloseWindow**). This function is not normally required, because Andor Basic will automatically close any file that is open, either when the filename is changed or at the end of the program. The function is provided to avoid network conflicts with file sharing. It returns an error number of zero on success, and a negative number on failure. If the return value is not assigned then a failure will cause an error message to be printed

See also: read write kill CloseWindow

Example 1:
```
write ("report.dat", "hi there ")
rem Append "hi there" to end of file "report.dat".
close()    :rem Close "report.dat".
```

Example 2:
```
error=close()
if error<0 then print("Error in closing file.")
```

CloseWindow

Syntax: CloseWindow(*#dataset*)

Description: This function closes the window holding the data set *#dataset*. It can also be used with the current *Program Editor Window* or the *Program Output Window*: replace the *#dataset* parameter with *program* or *output* respectively.

See also: MoveWindow RestoreWindow MinimizeWindow MaximizeWindow TopWindow

Example:
```
rem Create a data set by loading data from disk
load(#2,"penray.sif")
...
rem now close the window
CloseWindow(#2)
```

---

cls

Syntax: cls( )

Description: This function clears the screen of any text that is being displayed. The cursor position is then set for the top left hand corner of the output window. Any new text will be printed here by using the **print** function.

See also: print

Example:
```
cls()                   :rem Clear the screen
print("Hello world")    :rem Print message in top left hand
                        :rem screen position
```

coefficient$

Syntax: coefficient$(*#data*)

Description: Returns the calibration constants for the x-axis of the data set (*#data*) as a string. The string is of the form "C1, C2, C3, C4", where C1 to C4 are the coefficients. The x-axis calibration is stored as a third order polynomial of the form:

Cal = C1 + C2*P + C3*P*P + C4*P*P*P

where P is the pixel number, starting from 1.

Example:
```
a$=coefficient$(#1_sig):rem Gets the calibration
                        :rem coefficients.
print(a$)               :rem Prints the
                        :rem coefficients on screen
```

comread

Syntax: comread(*comport,a$)*

Description: This function reads text from the serial port specified by *comport* and assigns it to the string variable *a$*.. Text is read line at a time, requiring a separate **comread** for each new line.

See also: handshake comwrite baud str$ val

Example:
```
rem Send a message on com1.
baud(1,4800)   :rem Set up com1 to 4800 baud.
Comwrite (1," Hello World ")

rem Receive a numeric value on com2.
baud(2,9600)   :rem Set up com2 to 9600 baud.
Handshake(2,0) :rem Handshaking turned off.
comread(2,a$)  :rem print the message received from com2
print(a$)
```

comwrite

Syntax: comwrite(*comport,text)*

Description: This function writes *text* to the serial port specified by *comport*

See also: handshake comread baud str$ val terminator newline

Example:
```
rem Send a message on com1.
baud(1,4800)   :rem Set up com1 to 4800 baud.
Comwrite (1," Hello World ")

rem Receive a numeric value on com2.
baud(2,9600)   :rem Set up com2 to 9600 baud.
Handshake(2,0) :rem Handshaking turned off.
comread(2,a$)  :rem print the message received from com2
print(a$)
```

connect (network command)

Syntax: connect(IPNumber)

Description: Makes a connection to a remote machine via Ethernet.

See also: disconnectdisconnect GetAcquiredDataGetAcquiredData GetStatus

Example: This example program connects to a remote machine with an IP address of 111.222.333.444 and acquires a full vertically binned image from a CCD connected to it. Any commands between **connect** and **disconnect** will be sent to the remote machine.

```
connect("111.222.333.444")
SetFVB()
SetSingleScan(0.5)
run()
test = GetStatus()
// The following loop tests to see when the
// acquisition has finished.  If 20072 is returned
// the driver is still acquiring.
while (test == 20072)
  test = GetStatus()
wend
GetAcquiredData(#1)  // Display data in #1
disconnect()
```

cooler

Syntax: cooler(*state*)

Description: If *state* is equal to 1, cooling is switched ON. The rate of temperature change is controlled until the temperature is within 3° of the set value. The desired temperature is set using **SetTemperature**.
If state is equal to 0, cooling is switched OFF. If the detector has previously been cooled, the rate of temperature increase is controlled until the temperature reaches 0°.

See also: SetTemperature

Example: This example shows how to cool a CCD head to -10°C:
SetTemperature(-10)
cooler(1)

---

copy

Syntax: copy(*#data1*, *#data2*)

Description: This function makes an exact copy of *#data1* and places the result in *#data2*. This function is more powerful than the assignment statement #data2=#data1, because the data header, including its calibration and comments, are also copied into the destination data set.

See also: copyxcal

Example: **copy**(#1_sig,#2_sig):**rem** copy the #sig in data set #1 to #2

---

copyxcal

Syntax: copyxcal(*#data1*, *#data2*)

Description: This function sets the X calibration of *#data2* to be the same as that of *#data1*.

See also: copy xpix xcal

Example: **Copyxcal**(#1_sig,#2_sig):**rem** copy the X calibration of #1_sig
:**rem** to #2_sig

---

cos

Syntax: y = cos(*x*)

Description: Calculates the cosine. This function expects an argument, in radians, and returns a floating point number in the range -1.0 to 1.0

See also: acos asin atan sin tan

Example:
```
x=0.5
y=cos(x)
print(y)
```

Result: 0.877583

---

create

Syntax: create(#data,x)
create(#data,x,y)
create(#data,x,y,number_in_series)
create(#data,title,x)
create(#data,title,x,y)
create(#data,title,x,y,number_in_series)

Description: Used to create an empty data set of a known size. #data specifies the number of the data set and the name of the data to be created or modified. x specifies the size of the data set on the x-axis, while y, if present, equals the size on the y-axis (i.e. the number of rows or tracks that will appear in a 2-D or 3-D display). If the data set already exists it will be resized and overwritten by this command. The third form allows the creation of a data set containing several members of a series, similar to that created by a kinetic acquisition. This might be useful if several aquisitions are being made but a separate data window for each is too demanding on system resources. The last three forms of the **create** command allow a title to be given to the data set.

Example: This example assumes that the file “kin50.sif” is a kinetic data set consisting of 50 traces acquired at equal intervals over a period of time. It is desired to perform a ‘kinetic slice’ on the signal data, showing how one pixel has changed with time. Note the use of the {} brackets to define the kinetic frame.

```
load(#5_sig,"kin50.sif")     :rem Load kinetic data set.
create(#10_sig,50)           :rem Make new data set.
i=1
while(i<51)
 #10_sig[i]=#5{i}[350]       :rem Make kinetic slice.
 i=i+1
wend
```

cursorx

Syntax: cursorx(#data)

Description: Returns the position in pixels on the x-axis of the cursor for the specified data.

See also: cursory

Example: x=**cursorx**(#1_sig)
y=**cursory**(#1_sig)
print("Cursor is at x=";x;" y=";y)

---

cursory (not PDA)

Syntax: cursory(#data)

Description: Returns the position in pixels on the y-axis of the cursor for the specified data.

See also: cursorx

Example: x=**cursorx**(#1_sig)
y=**cursory**(#1_sig)
print("Cursor is at x=";x;" y=";y)

---

date$

Syntax: date$()

Description: Returns the date in the format Day:Month:Year.

See also: time$

Example: **print**(**date$**())

---

delay

Syntax: delay(x)

Description: Waits for a delay of x milliseconds.

See also: beep

Example:
```
rem  Send out a beep at 0.5 sec intervals.
i=0
while i<10
  beep()
  delay(500)
  i=i+1
wend
```

DetachOverlay

Syntax: DetachOverlay(#dataset,overlaynumber)

Description: Removes data that has been put into a window by means of the **AddOverlay** function. #dataset refers to the window displaying the data and overlaynumber is the constant between 1 and 8 that is returned by the **AddOverlay** function.

See also: AddOverlay ActiveOverlay

Example: **rem** Display two different sets of data in the same window and **rem** then remove the data that has been added.

```
load(#1,"data1.sif")          :rem load data
load(#2,"data2.sif")
overlaynumber=AddOverlay(#1,#2) :rem copy data into window
DetachOverlay(#1,overlaynumber) :rem remove data just added
```

detectorx

Syntax: detectorx()

Description: Returns the horizontal size, in pixels, of the detector.

See also: detectory

Example: **rem** convert image to black and white

```
threshold=2500
xsize=detectorx()
ysize=detectory()
xpos=1
while(xpos<=xsize)
  ypos=1
  while(ypos<=ysize)
    if(#1_sig[xpos,ypos]<threshold)then
      #1_sig[xpos,ypos]=0
    else
      #1_sig[xpos,ypos]=100
    endif
    ypos=ypos+1
  wend
  xpos=xpos+1
wend
```

detectory (not PDA)

Syntax: detectory()

Description: Returns the vertical size, in pixels, of the detector.

See also: detectorx

Example: **rem** convert image to black and white

```
threshold=2500
xsize=detectorx()
ysize=detectory()
xpos=1
while(xpos<=xsize)
  ypos=1
  while(ypos<=ysize)
    if(#1_sig[xpos,ypos]<threshold) then
      #1_sig[xpos,ypos]=0
    else
      #1_sig[xpos,ypos]=100
    endif
    ypos=ypos+1
  wend
  xpos=xpos+1
wend
```

diff

Syntax: diff(#data)

Description: Performs a 3 point differentiation on #data.

As in the example, an assignment to a data set must accompany this function.

Example: This example differentiates the spectrum in the data set #4 and places the resultant data into a new data set #7.

Result: #7 = **diff**(#4)

Disconnect (network command)

Syntax: disconnect(IPNumber)

Description: Frees a connection from a remote machine made via Ethernet. This would be used if an acquisition was being made from both the local machine and a remote machine.

See also: connectconnect GetAcquiredDataGetAcquiredData GetStatus

Example: This example program connects to a remote machine with an IP address of 111.222.333.444 and acquires a full vertically binned image from a CCD connected to it. Any commands between **connect** and **disconnect** will be sent to the remote machine.

```
connect("111.222.333.444")
SetFVB()
SetSingleScan(0.5)
run()
test = GetStatus()
// The following loop tests to see when the
// acquisition has finished. If 20072 is returned
// the driver is still acquiring.
while (test == 20072)
  test = GetStatus()
wend
GetAcquiredData(#1) // Display data in #1
disconnect()
// acquire an image from the local machine
SetImage()
SetSingleScan(0.1)
run()
```

exp

Syntax: exp(x) or exp(#data)

Description: This function calculates the exponent or inverse natural logarithm. The function can be applied to single values or to named data within a data set. If used with named data, an assignment (either to itself or to other named data within a data set) must accompany this function.

See also: alog log ln

Example:
```
#7_sig=exp(#1_sig)
x=0.5
y=exp(x)
print(y)
```

Result: 1.64872

Export16

Syntax: 1) Export16(SIFfile$, dataFile$)

2) Export16(#source, dataFile$)

Description: Option 1) converts an InstaSpec SIF file into a corresponding 16 bit integer data file (*.DAT).

Option2) saves an InstaSpec dataset into the corresponding 16 bit integer data file (*.DAT).

SIFfile$ = name of an InstaSpec SIF file,
dataFile$ = name of a data file (*.DAT),
#source = InstaSpec SIF dataset.

In both cases, the DAT file comprises data only and has no header information. The original SIF file/dataset remains unchanged.

The *.DAT file format reflects the CCD chip format - e.g. with a CCD sensor of 1024 x 256 pixels then the first 1024 data values (NOTE not bytes, see Reading a *.dat file below) in the *.DAT file correspond to the first row of the CCD, the second 1024 data values correspond to the second row of the CCD, etc.

The range of possible data types available in InstaSpec are shown in the table below:

DATA TYPE	RANGE
16 bit integer	-32,768 to 32,767
32 bit integer	-2,147,483,648 to 2,147,483,647

Export 16 (continued)

Notes:

1. Any data value outside the 16 bit range is automatically truncated at the corresponding limit value.
2. Option 1) of Export16( ) will only export the signal part of the SIF file. Option 2) allows the user to export any of the sig, bg, ref or cal parts of the SIF dataset.
3. Export16( ) is applicable to all binning modes (e.g. Full Vertical Binning or Full Image, etc.) and acquisition modes (e.g. single scans or kinetic series, etc).
4. Reading a *.dat file. It is the user's responsibility to determine how many data bytes to read in when using their own software to read a *.DAT file. Each 16 bit data value requires 2 bytes to store the value. Thus for example, to read in a 16 bit *.DAT file consisting of 1024 data values, the user would have to read in 2048 bytes in total.

See also: ExportGRAMSSpc, Export32, ExportFloat, SaveAsciiXY

Example 1: The InstaSpec file "bluesky.sif" is in the directory c:\instaspc\spectra. This example will export the SIF file straight from disk out to a 16 bit integer *.DAT file

NOTE: Only the sig part of "bluesky.sif" is converted to DAT.

```
SIF$ = "c:\instaspc\spectra\bluesky.sif"
data$ = "c:\bluesky.dat"
Export16(SIF$,data$)
```

Example 2: The InstaSpec dataset #22 contains a kinetic series of 20 spectra. This example will export the complete SIF signal data (i.e. 20 spectra) out to a data file "test.dat".

```
Data$ = "c:\test.dat"
Export16(#22_sig,data$)
```

Export32

Syntax: 1) Export32(*SIFfile$*, *dataFile$*)

2) Export32(*#source*, *dataFile$*)

Description: Option 1) converts an InstaSpec SIF file into the corresponding 32 bit integer data file (*.DAT).

Option2) saves an InstaSpec dataset currently in memory as a 32 bit integer data file (*.DAT).

SIFfile$ = name of an InstaSpec SIF file,
dataFile$ = name of a data file (*.DAT),
#source = InstaSpec SIF dataset.

In both cases, the DAT file comprises data only and has **no header information**. The original SIF file/dataset remains unchanged.

The DAT file format reflects the CCD chip format - e.g. with a CCD sensor of 1024 x 256 pixels the first 1024 data values (NOTE not bytes, see Reading a *.DAT file below) in the *.DAT file correspond to the first row of the CCD, the second 1024 data values correspond to the second row of the CCD, etc.

The range of possible data types available in InstaSpec are shown in the table below:

DATA TYPE	RANGE
16 bit integer	-32,768 to 32,767
32 bit integer	-2,147,483,648 to 2,147,483,647
32 bit float	3.4×10^{-38} to $3.4 \times 10^{+38}$

Export 32 (continued)

Notes:

1. Any data value outside the 32 bit range is automatically truncated at the corresponding limit value.
2. Option 1) of Export32( ) will only export the signal part of the SIF file. Option 2) allows the user to export any of the sig, bg, ref or cal parts of the SIF dataset.
3. Export32( ) is applicable to all binning modes (e.g. Full Vertical Binning or Full Image, etc.) and acquisition modes (e.g. single scans or kinetic series, etc).
4. Reading a *.dat file. It is the user's responsibility to determine how many data bytes to read in when using their own software to read a *.DAT file. Each 32 bit data value requires 4 bytes to store the value. Thus for example, to read in a 32 bit *.DAT file consisting of 1024 data values, the user would have to read in 4096 bytes in total.

Example 1: The InstaSpec file "bluesky.sif" is in the directory c:\instaspc\spectra. This example will export the SIF file straight from disk out to a 32 bit integer *.DAT file.

NOTE: Only the sig part of "bluesky.sif" is converted to DAT.

```
SIF$ = "c:\instaspc\spectra\bluesky.sif"
data$ = "c:\bluesky.dat"
Export32(SIF$,data$)
```

Example 2: The InstaSpec dataset #22 contains a kinetic series of spectra. This example will export the complete SIF signal data out to a data file "test.dat".

```
Data$ = "c:\test.dat"
Export32(#22_sig,data$)
```

ExportBMP (not PDA)

Syntax: ExportBMP(*data*, *file*)

Description: Exports an Andor sif file into the corresponding bitmap file.

data can either be:

a dataset, e.g. #1

or

a file path, e.g. “c:\data\data1.sif”.

file specifies the file path of the bitmap file to be created

e.g. “c:\data\data1.bmp”.

See also: Export16 Export32 ExportFloat ExportGramsSPC ExportTiff

Example: This example exports the data set #2 to a bitmap file data2.bmp using the default settings:

file$ = “c:\data\data2.bmp”

ExportBMP(#2, file$)

ExportFloat

Syntax: 1) ExportFloat(*SIFfile$*, *dataFile$*)

2) ExportFloat(*#source*, *dataFile$*)

Description: Option 1) converts an InstaSpec SIF file into the corresponding 32 bit floating point data file (*.DAT).

Option2) saves an InstaSpec dataset currently in memory as a 32 bit floating point data file (*.DAT).

SIFfile$ = name of an InstaSpec SIF file,
dataFile$ = name of a data file (*.DAT),
#source = InstaSpec SIF dataset.

In both cases, the DAT file comprises data only and has *no header information.* The original SIF file/dataset remains unchanged.

The DAT file format reflects the CCD chip format - e.g. with a CCD sensor of 1024 x 256 pixels the first 1024 data values (NB not bytes, see Reading a *.DAT file below) in the *.DAT file correspond to the first row of the CCD, the second 1024 data values correspond to the second row of the CCD, etc.

The range of possible data types available in InstaSpec are shown in the table below:

DATA TYPE	RANGE
16 bit integer	-32,768 to 32,767
32 bit integer	-2,147,483,648 to 2,147,483,647

ExportfFoat (continued)

Notes:

1. Option 1) of *ExportFloat()* will *only* export the signal part of the SIF file. Option 2) allows the user to export any of the sig, bg, ref or cal parts of the SIF dataset.
2. *ExportFloat()* is applicable to *all* binning modes (e.g. Full Vertical Binning or Full Image, etc.) and acquisition modes (e.g. single scans or kinetic series, etc).
3. Reading a *.dat file .It is the user's responsibility to determine how many data bytes to read in when using their own software to read a *.DAT file. Each 32-bit float data value requires 4 bytes to store the value. Thus for example, to read in a 32 bit *.DAT file consisting of 1024 data values, the user would have to read in 4096 bytes in total.

See also: ExportGRAMSSpc Export16 Export32 SaveAsciiXY

Example 1: The InstaSpec file "bluesky.sif" is in the directory c:\instaspc\spectra. This example will export the SIF file straight from disk out to a 32 bit floating point *.DAT file

NOTE: Only the sig part of "bluesky.sif" is converted to DAT.

```
SIF$ = "c:\instaspc\spectra\bluesky.sif"
data$ = "c:\bluesky.dat"
ExportFloat(SIF$,data$)
```

Example 2: The InstaSpec dataset #22 contains a kinetic series of 20 spectra. This example will export the complete SIF signal data (i.e. 20 spectra) out to a float data file "test.dat".

```
Data$ = "c:\test.dat"
ExportFloat(#22_sig,data$)
```

ExportGRAMSSpc

Syntax: 1) ExportGRAMSSpc(*SIFfile$*, *SPCfile$*)

2) ExportGRAMSSpc(*#source*, *SPCfile$*)

Description: Option 1) converts an InstaSpec SIF file into the corresponding GRAMS/32 file.

SIFfile$ = name of an InstaSpec SIF file.

SPCfile$ = name of a GRAMS/32 file.

Option 2) saves an InstaSpec dataset currently in memory as a GRAMS/32 file.

#source = InstaSpec SIF dataset.

SPCfile$ = name of a GRAMS/32 file.

In both cases the original SIF file/dataset remains unchanged.

Notes:

1. ExportGRAMSSpc( ) is *applicable only* for SIF files/datasets which have a Single Track or a Full Vertically Binned (FVB) binning pattern. To export a row (or track) from a Multi Track or Full Image you should first copy the row into a new window (see **copy**( )).
2. ExportGRAMSSpc( ) can export a single scan or a series of scans i.e. in a Kinetic/Fast Kinetic series.
3. Option 1) of ExportGRAMSSpc( ) will *only* export the signal part of the SIF file (a SIF file can consist of *sig, bg, ref* and *cal* data). Option 2) allows the user to specify either the *sig, bg, ref* or *cal* part of the SIF dataset.
4. ExportGRAMSSpc( ) will attempt to copy over the SIF data units if the corresponding units exist in GRAMS/32, otherwise the units are set to the default unit "Arbitrary units".
5. GRAMS/32 is a registered trademark of Galactic Industries Corporation.

See also: Export16 Export32 ExportFloat SaveAsciiXY Copy

Example 1: The InstaSpec file "bluesky.sif" is in the current directory. This example will export the SIF file straight from disk out to a GRAMS/32 SPC file:

NOTE: Only the sig part of "bluesky.sif" is converted to DAT.

```
SIF$ = "bluesky.sif"
SPC$ = "bluesky.spc"
ExportGRAMSSpc(SIF$,SPC$)
```

Example 2: The InstaSpec dataset #22 contains a kinetic series of 20 spectra. This example will export the complete SIF signal data (i.e. 20 spectra) out to a GRAMS/32 SPC file:

```
SPC$ = "c:\SPCFiles\test.spc"
ExportGRAMSSpc(#22_sig,SPC$)
```

ExportTiff (not PDA)

Syntax: 1) ExportTiff(*data*, *file*)

2) ExportTiff(*data*, *file*, *type*)

3) ExportTiff(*data*, *file*, *type*, *range*)

4) ExportTiff(*data*, *file*, *type*, *range*, *area*)

5) ExportTiff(data, file, type, range, area, position)

Description: Exports an Andor sif file into the corresponding tiff file.

data can either be a data set e.g. #1 or a file path e.g. “c:\data\data1.sif”.

file specifies the file path of the tiff file to be created e.g. “c:\data\data1.tif”.

type specifies whether the tiff file is to be 8 bit, 16 bit or color (0, 1 or 2).

range specifies whether the data range of the tiff file is to be the total range of the image, 0 - 65535 or the current displayed range (0, 1, 2).

area specifies whether the tiff file will incorporate the entire image or just the displayed part (0 or 1).

if the data set is a kinetic series, the image which is to exported can be specified by the parameter *position*.

The default tiff file will use the current palette, range of data values and displayed area.

See also: Export16 Export32 ExportFloat ExportGramsSPC ExportBMP

Example 1: This example exports the data set #2 to a tiff file data2.tif using the default settings:

```
file$ = "c:\data\data2.tif"
ExportTiff(#2, file$)
```

Example 2: This example exports the data set data3.sif to a 16 bit tiff file data3.tif:

```
file1$ = "c:\data\data3.sif"
file2$ = "c:\data\data3.tif"
ExportTiff(file1$, file2$, 1)
```

fft

Syntax: fft(*#data*)

Description: Calculates the FFT power spectrum of *#data*. where *#data* has a width corresponding to a power of 2 (typically 512, 1024 or 2048). As in the example, an assignment to a data set must accompany this function.

See also: area cfft icfft carea smooth1

Example: #2_sig=**fft**(#1_sig) :**rem** puts the FFT power spectrum of
:**rem** #1_sig into #2_sig

---

format

Syntax: format(*x*.*y*)

Description: When this function is used, it affects all subsequent numeric output using the **print**, and **write** functions. *x* and *y* specify the number of digits before and after the decimal point. This allows columns of figures to be displayed with their decimal points aligned and to the required precision. Values of zero for *x* and *y* cause the function to be ignored. **format** retains its value between programs.

See also: print write

Example:
```
format(4.2)
print(3.14159)
print(31.4159)
print(314.159)
print(3141.59)
print(31415.9)
```

Output:
```
3.14
 31.42
 314.16
3141.59
31415.9
```

---

fwhm

Syntax: fwhm(*#data,xstart,xend,flag*)

Description: Returns the full width half maximum value of *#data* between the points *xstart* and *xend.* The final parameter, *flag*, has a value of zero (or may be omitted altogether) if *xstart* and *xend* are given in pixels. In this case the return value will also be in pixels. If *flag* has a non-zero value then *xstart* and *xend* are given in calibration units. In this case the return value will also be in the calibration units applied to the *data*.

See also: area carea

Example:
```
print(fwhm(#1_sig,1,50))       :rem Working in pixels.
print(fwhm(#1_sig,550,560,1))  :rem Working in nm.
print(fwhm(#1_sig,550,560))    :rem Working in pixels.
```

GetAcquired Data (network command)

Syntax: GetAcquiredData(*#dataset*)

Description: Retrieves the data acquired by a remote machine connected to Ethernet and displays the data in a data set chosen by the user.

See also: connect disconnect GetStatus

Example: This example program connects to a remote machine with an IP address of 111.222.333.444 and acquires a full vertically binned image from a CCD connected to it. Any commands between **connect** and **disconnect** will be sent to the remote machine.

```
connect("111.222.333.444")
SetFVB()
SetSingleScan(0.5)
run()
test = GetStatus()
// The following loop tests to see when the
// acquisition has finished.  If 20072 is returned
// the driver is still acquiring.
while (test == 20072)
  test = GetStatus()
wend
print(GetAcquiredData(#1))  // Display data in #1
disconnect()
```

GetNumberSeries

Syntax: GetNumberSeries(*#KineticSeries*)

Description: Returns the number of scans that comprise a Kinetic Series. This function is normally applicable to data sets that are Kinetic / Fast Kinetic Series, although it will return a value of 1 for Single Scan acquisitions.

Example: This example will load in a previously saved Kinetic Series ("HgNe.sif") and determine the number of scans in the series:

Load(#2, "HgNe.sif")
NoSeries = **GetNumberSeries**(#2)

The variable NoSeries will now equal the number of scans in the Kinetic Series.

GetNumberTracks (not PDA)

Syntax: GetNumberTracks(*#DataSet*)

Description: Returns the number of tracks that comprise a data acquisition. For a full description of each acquisition mode refer to the User's Guide

Example 1: This example will load in a previously saved Multi-Track image ("HgNe.sif") and determine the number of tracks in the image.

Load(#2, "HgNe.sif")
NoTracks = **GetNumberTracks**(#2)

The variable NoTracks will now equal the number of tracks in the Multi-Track image.

Example 2: The data set #3 is a full resolution image of dimensions 1024 X 256 pixels. The number of tracks in this instance will equal the number of rows on the CCD chip, i.e. 256.

NoTracks = **GetNumberTracks**(#3)

GetStatus (network command)

Syntax: GetStatus()

Description: Tests to see the current status of the remote machine which is being accessed at the time. One of the following will be returned:

20013 Unable to communicate with card
20018 Computer unable to read the data via the ISA slot at the required rate
20022 Unable to meet Kinetic cycle time
20023 Unable to meet Accumulate cycle time
20072 Acquisition in progress
20073 IDLE waiting on instructions
20074 Executing temperature cycle

NOTE: While an acquisition is in progress no remote settings can be altered. If an attempt is made to do so the program will exit and the data from the acquisition will be lost.

Example: This example program connects to a remote machine with an IP address of 111.222.333.444 and acquires a full vertically binned image from a CCD connected to it. Any commands between **connect** and **disconnect** will be sent to the remote machine.

```
connect("111.222.333.444")
SetFVB()
SetSingleScan(0.5)
run()
test = GetStatus()
// The following loop tests to see when the
// acquisition has finished.  If 20072 is returned
// the driver is still acquiring.
while (test == 20072)
  test = GetStatus()
wend
GetAcquiredData(#1)  // Display data in #1
disconnect()
```

GraphData

Syntax: GraphData(*dataset, color, style*)
GraphData(*dataset, color*)
GraphData(*dataset*)
GraphData(*dataset, 0, style*)

Description: Plots the dataset on to the graphics output window in the style and color set by the user. The default setting is a solid black line. Multiple datasets can be plotted simultaneously provided that the new x and y ranges are defined in each case. Used as parameters in accordance with the syntax above, the values 0 to 6 represent the following **colors** or b.

		Color		Style
0	**represents**	black	**or**	Solid line
1		red		Dash
2		green		Dot
3		blue		Dash-dot
4		yellow		Dash-dot-dot
5		cyan		-
6		violet		-

Example 1:
```
rem plots dataset #1
GraphData(#1)       :rem solid black line,
GraphData(#1, 1)    :rem solid red line,
GraphData(#1, 0, 2) :rem dotted black line,
GraphData(#1, 3, 3) :rem blue dash-dotted line
```

Example 2:
```
rem load file laser.sif as dataset #12
load(#12, "laser.sif")
xstart = 100
xend = 700
ystart = 50
yend = 500
rem delete previous graph and create a new one
GraphNew()
GraphGrid(1)
GraphXLabel("Wavelength nm")
GraphYLabel("Counts")
GraphFont("Times Roman")
GraphTextSize(30)
rem add title Example Graph in 30pt Times Roman
GraphLabel(xstart+20, ystart-30, "Example Graph")
GraphXAxis(700, 900)
GraphYAxis(0, 200)
rem generate layout for graph.
GraphFrame(xstart, ystart, xend, yend)
rem plot dataset 12 with a solid red line
GraphData(#12, 1)
```

GraphEscape

Syntax: GraphEscape(*string*)

Description: Enables the user to send escape sequences to their printer.

Example: **GraphEscape**(chr$(27);chr$(69);) **:rem** command to reset Hewlett
:rem Packard printer
GraphPrint() **:rem** sends command to printer

GraphFont

Syntax: GraphFont(*fontname*)

Description: Sets the name of the text font to be used for subsequent labels when using either **GraphXLabel**, **GraphYLabel** or **GraphLabel**. The default type is 10 point Arial. Multiple font types can be used in a graph. The font size can also be changed using **GraphTextSize**.

See also: GraphXLabel GraphYLabel GraphLabel GraphTextSize

Example:
```
GraphFont("Times Roman")    :rem  future labels use Times
                            :rem  Roman.
GraphXLabel("Wavelength nm") :rem  labels x-axis in 10pt
                            :rem  Times Roman.
GraphFont("Courier")        :rem  future labels use Courier.
GraphTextSize(14)           :rem  font size changed to 14pt.
GraphYLabel("Counts")       :rem  labels y-axis in 14pt
                            :rem  Courier.
```

GraphFrame

Syntax: GraphFrame(*xstart, ystart, xend, yend*)

Description: The purpose of the function **GraphFrame** is to take all the information you have supplied and to display it in a window. It creates a border with top left corner (*xstart*, *ystart*) and bottom right corner (*xend*, *yend*) where (0, 0) is the top left corner of the window.The dataset you wish to display is then plotted inside.

The function **GraphFrame** should be called once you have defined your range of x and y values, labeled the x and y-axes and chosen the appropriate type of grid. If you have not provided the plotting region using the **GraphXAxis** and **GraphYAxis** functions before **GraphFrame** is called an error will be returned. The only graphing functions that should be called after **GraphFrame** are **GraphData** and **GraphPrint**. Any other commands may be ignored.

See also: GraphXAxis GraphYAxis

Example:

```
GraphNew()          :rem  create new graph.
GraphGrid(1)        :rem  generate grid on graph.
GraphXAxis(700, 900)  :rem  plotting area must be defined.
GraphYAxis(0, 200)    :rem  before GraphFrame can be used.
xstart = 10         :rem  top left coordinates
ystart = 20         :rem  of the frame.
xend = 300          :rem  bottom right coordinates.
yend = 400          :rem  of the frame.
GraphFrame(xstart, ystart, xend, yend)
```

GraphGrid

Syntax: GraphGrid(*type*)

Description: The **GraphGrid** function sets the type of grid to be used. It must be called before the **GraphFrame** function as it will be ignored otherwise. The following options are available:

type	grid
0	none
1	solid
2	dashed

Example: **GraphGrid**(2) **:rem** generates a dashed grid on graph.

GraphLabel

Syntax: GraphLabel(*x, y, label*)

Description: Allows you to add multiple comments to a graph. Each comment is positioned at the specified location. The parameters *x* and *y* passed to **GraphLabel** are measured in screen pixels where the origin is the top left corner of the window. The font and text size of each label can be altered using the **GraphFont** and **GraphTextSize** functions. Any number of labels can be used in a graph and the font and text size can be changed on each occasion.

See also: GraphXLabel GraphYLabel GraphFont GraphTextSize

Example:
```
rem place the text "Counts" at coordinates (20, 20)
rem in 14 point Courier
GraphFont("Courier")
GraphTextSize(14)
GraphYLabel(20, 20, "Counts")
```

GraphLine

Syntax: GraphLine(*x1, y1, x2, y2, color, style*)

GraphLine(*x1, y1, x2, y2, color*)

GraphLine(*x1, y1, x2, y2*)

GraphLine(*x1, y1, x2, y2, 0, style*)

Description: Draws a line between 2 user-defined points. Used as parameters in accordance with the syntax above, the values 0 to 6 represent the following **colors** or **styles**.

		Color		**Style**
0	**represents**	black	**or**	solid line
1		red		dash
2		green		dot
3		blue		dash-dot
4		yellow		dash-dot-dot
5		cyan		-
6		violet		-

Example: **rem** connect the points (10, 50) and (20, 200) with a

rem solid black line,

GraphLine(10, 50, 20, 200)

rem connect the points (99, 13) and (119, 25) with a

rem solid red line,

GraphLine(99, 13, 119, 25, 1)

rem connect the points (9, 9) and (22, 22) with a

rem dotted black line,

GraphLine(9, 9, 22, 22, 0, 2)

rem connect the points (0, 7) and (2, 22) with a

rem dash-dotted blue line,

GraphLine(0, 7, 2, 22, 3, 3)

GraphNew

Syntax: GraphNew(*void*)

Description: Creates a new window into which subsequent graph plotting commands are executed. If a graph plot window already exists it is deleted.

Example: **GraphNew**() **:rem** create new graph plot

GraphPrint

Syntax: GraphPrint(*void*)

Description: Prints the completed graph to the default printer

Example: **GraphPrint**() **:rem** prints graph to default printer.

GraphTextSize

Syntax: GraphTextSize(*textsize*)

Description: This sets the text size for any labels. The default size is 10 point Arial. Multiple text sizes can be used in a graph. The font type can also be changed using **GraphFont**.

See also: GraphXLabel GraphYLabel GraphLabel GraphFont

Example: **GraphTextSize**(12) **:rem** font size changed to 12pt.
GraphXLabel("Pixels") **:rem** labels x-axis in 12pt Arial.
GraphFont("Courier") **:rem** future labels use Courier.
GraphTextSize(14) **:rem** font size changed to 14pt.
GraphYLabel("Counts") **:rem** labels y-axis in 14pt Courier.

GraphXAxis

Syntax: GraphXAxis(*xstart, xend*)

Description: Sets the range of x values that will be plotted using the current x-axis calibration. These limits are required for **GraphFrame**.

See also: GraphYAxis GraphFrame

Example: **GraphXAxis**(100, 900) **:rem** range of x values to be plotted

---

GraphXLabel

Syntax: GraphXLabel(*xstart, xend*)

Description: Allows the user to label the x-axis. The font type and size can be set using the **GraphFont** and **GraphTextSize** commands.

See also: GraphLabel GraphYLabel GraphFont GraphTextSize

Example: **GraphFont**("Courier") **:rem** sets font type to Courier.
GraphTextSize(14) **:rem** font size changed to 14pt.
GraphXLabel("Wavelength") **:rem** labels x-axis in 14pt Courier

---

GraphYAxis

Syntax: GraphYAxis(*ystart, yend*)

Description: Sets the range of y values that will be plotted using the current y-axis calibration. These limits are required for **GraphFrame.**

See also: GraphXAxis GraphFrame

Example: **GraphYAxis**(100, 900) **:rem** range of y values to be plotted

---

GraphYLabel

Syntax: GraphYLabel(*label*)

Description: Allows the user to label the y-axis. The font type and size can be set using the **GraphFont** and **GraphTextSize** commands.

See also: GraphLabel GraphYLabel GraphFont GraphTextSize

Example: **GraphFont**("Courier") **:rem** sets font type to Courier.
GraphTextSize(14) **:rem** font size changed to 14pt.
GraphYLabel("Counts") **:rem** labels y-axis in 14pt Courier

---

handshake

Syntax: handshake(*comport,x*)

Description: This function sets the handshaking used on the serial port specified by *comport*. *x* can have one of two possible values:

0 No handshaking. RTS and DTR are asserted continuously.

1 RTS/CTS handshaking. DTR is asserted continuously.

See also: ignore terminator

Example: **rem** receive a numeric value on com2

baud(2,9600) :**rem** Set up com2 to 9600 baud.

handshake(2,0) :**rem** Handshaking turned off.

comread(2**,**input$) :**rem** Print the message received from com2.

print(input$)

ibclr

Syntax: ibclr(*device*)

Description: Clears a device by sending it the GPIB Selected Device Clear (SDC) message.

Status and error codes are returned in the special variables **ibsta** and **iberr**

See also: ibsic

Example: This simple example will establish a connection to an external device (with a GPIB address of 5) and clear the device.

The **ibconfig** command sets the timeout period (in event of error) to 1 sec.

Result:

```
Cls()
GPIBerror = 0x8000
device = ibfind("dev5")
gosub .err
ibconfig(dev,3,11)
gosub .err
ibclr(device)
gosub .err
end

.err
  if (ibsta and GPIBerror) then
    print("ERROR: GPIB error code is ";iberr)
    end         :rem Upon error then exit program
  endif
return
```

ibconfig

Syntax: ibconfig(*device_name,option,value*)

Description: **ibconfig** is a method for specifying most of the software configuration parameters associated with the GPIB interface. Due to the large number of these options they are not detailed in full here. Please refer to the NI Software Reference Manual for a complete list.

Status and error codes are returned in **ibsta** and **iberr**.

See also: ibfind ibrd ibwrt

Example: **rem** Establish connection with an instrument at GPIB address **rem** 15.
rem Query the status of the instrument and monitor all GPIB **rem** errors.

```
GPIBerror = 0x8000
board=ibfind("gpib0")     :rem Find interface board.
ibconfig(board,3,11)      :rem Change timeout to 1 sec.
dev=ibfind("dev15")       :rem Find instrument.
gosub .err
ibwrt(dev,"cl")     :rem Clear instrument.
gosub .err
ibwrt(dev,"is")     :rem Send status command.
gosub .err
ibrd(dev,a$)        :rem Read reply.
gosub .err
print("Status returned = ";a$)
end

.err
  if (ibsta and GPIBerror) then
    print("ERROR: GPIB error code is ";iberr)
    end         :rem Upon error exit program
  endif
return
```

ibfind

Syntax: ibfind(*device_name*)

Description: **ibfind** returns the device number for *device_name.* It is this number which is used by the **ibwrt**, **ibrd** and **ibconfig** commands.
Status and error codes are returned in **ibsta** and **iberr**.

See also: ibconfig ibrd ibwrt

Example: **rem** Establish connection with an instrument at GPIB address **rem** 15.
rem Query the status of the instrument and monitor all GPIB **rem** errors.

```
GPIBerror = 0x8000
board=ibfind("gpib0")   :rem Find interface board.
ibconfig(board,3,11)    :rem Change timeout to 1 sec.
dev=ibfind("dev15")     :rem Find instrument.
gosub .err
ibwrt(dev,"cl")   :rem Clear instrument.
gosub .err
ibwrt(dev,"is")   :rem Send status command.
gosub .err
ibrd(dev,a$)      :rem Read reply.
gosub .err
print("Status returned = ";a$)
end

.err
  if (ibsta and GPIBerror) then
    print("ERROR: GPIB error code is ";iberr)
    end         :rem Upon error exit program
  endif
return
```

ibrd

Syntax: ibrd(*device_number,string_variable*)

Description: **ibrd** is used to receive text from a device on the GPIB interface bus. The received text is placed in *string_variable.*

Status and error codes are returned in **ibsta** and **iberr**.

See also: ibconfig ibfind ibwrt

Example: **rem** Establish connection with an instrument at GPIB address **rem** 15.
rem Query the status of the instrument and monitor all GPIB **rem** errors.

```
GPIBerror = 0x8000
board=ibfind("gpib0")   :rem Find interface board.
ibconfig(board,3,11)    :rem Change timeout to 1 sec.
dev=ibfind("dev15")     :rem Find instrument.
gosub .err
ibwrt(dev,"cl")   :rem Clear instrument.
gosub .err
ibwrt(dev,"is")   :rem Send status command.
gosub .err
ibrd(dev,a$)      :rem Read reply.
gosub .err
print("Status returned = ";a$)
end

.err
  if (ibsta and GPIBerror) then
     print(“ERROR: GPIB error code is “;iberr)
     end          :rem Upon error exit program
  endif
return
```

ibrdf

Syntax: ibrdf(*device, filename*)

Description: Reads data from a device into a file.
Status and error codes are returned in the special variables **ibsta** and **iberr**.

See also: ibwrtf

Example: This example will send a query command to an external device requesting status details and place the device response into a text file

To take a specific example we will query the status details of a Tektronix TDS350™ oscilloscope at a GPIB address of 5. The details (in this instance, of channel 1) include the time-base scaling factor, vertical scaling factor, trigger position, etc. The device response will be sent out to the text file "c:\tek350.txt" where it can later be viewed by InstaSpec's built-in editor.

```
The ibconfig command sets the timeout period (in event of error) to 1 sec.
cls()
GPIBerror = 0x8000
device = ibfind("dev5")
gosub .err
ibconfig(device,3,11)
gosub .err
file$ = "c:\tek350.txt"
ibwrt(device,"CH1?") :rem ask for Channel 1 details
gosub .err
ibrdf(device, file$)
gosub .err
end

.err
  if (ibsta and GPIBerror) then
    print("ERROR: GPIB error code is ";iberr)
    end         :rem Upon error exit program
  endif
return
```

ibrdi

Syntax: ibrdi(*device, #data, NumPts*)

Description: Reads integer (2 bytes) data from a device into a dataset (#data). *NumPts* specifies the number of data points read in from the device.

Status and error codes are NOT returned in the special variables **ibsta** and **iberr** because **ibrdi** performs its own internal error checking.

Example: This example will transfer integer data from an external device and place the data into a dataset (in this instance, #3). To take a specific example we will transfer a data scan (1000 points) from a Tektronix TDS350™ oscilloscope at a GPIB address of 5. To limit the number of device-specific commands it is assumed that the oscilloscope already contains the desired scan and that we need only do the transfer. The device specific "CURVE?" command prompts the TDS350™ to begin the transfer. The **ibconfig** command sets the timeout period (in event of error) to 1 sec.

```
cls()
GPIBerror = 0x8000
device = ibfind("dev5")
gosub .err
board = ibfind("gpib0")
gosub .err
ibconfig(device,3,11)
gosub .err
ibclr(device)  :rem Clear GPIB device communication
gosub .err
ibsic(board)   :rem Clear GPIB board communication
gosub .err
ibwrt(device,"CURVE?") :rem ask for data scan
gosub .err
ibrdi(device, #3, 1000) :rem no need to call .err
end
.err
  if (ibsta and GPIBerror) then
    print("ERROR: GPIB error code is ";iberr)
    end         :rem Upon error exit program
  endif
return
```

ibrsc

Syntax: ibrsc(*board, mode*)

Description: Requests or releases system control.

If *mode* is zero, *board* releases system control and functions requiring System Controller capability are disabled. If *mode* is non-zero, functions requiring System Controller capability are subsequently allowed.

Status and error codes are returned in the special variables **ibsta** and **iberr**.

Example: This example will let the gpib board release system control.

The **ibconfig** command sets the timeout period (in event of error) to 1 sec.

```
cls()
GPIBerror = 0x8000
board = ibfind("gpib0")
gosub .err
ibconfig(board,3,11)
ibrsc(board, 0)
gosub .err
end

.err
  if (ibsta and GPIBerror) then
    print("ERROR: GPIB error code is ";iberr)
    end         :rem Upon error exit program
  endif
return
```

ibsic

Syntax: ibsic(*board*)

Description: Asserts the GPIB interface clear (IFC) line.

The IFC signal resets only the interface functions of the GPIB board and *not* the externally connected device(s). You should consult your device documentation to determine how to reset the internal functions of your device.

Status and error codes are returned in the special variables **ibsta** and **iberr**.

See also: ibclr

Example: This simple example will establish a connection to a GPIB board and assert the GPIB interface clear.

The ibconfig() command sets the timeout period (in event of error) to 1 sec.

```
cls()
GPIBerror = 0x8000
board = ibfind("gpib0")
gosub .err
ibconfig(board,3,11)
gosub .err
ibsic(board)
gosub .err
end

.err
  if (ibsta and GPIBerror) then
    print("ERROR: GPIB error code is ";iberr)
    end         :rem Upon error exit program
  endif
return
```

ibwrt

Syntax: ibwrt(*device,text*)

Description: **ibwrt** sends *text* to the specified *device*. If a reply is expected **ibrd** should be used subsequently to receive it.
Status and error codes are returned in **ibsta** and **iberr**.

See also: ibconfig ibfind ibrd

Example: **rem** Establish connection with an instrument at GPIB address **rem** 15.

rem Query the status of the instrument and monitor all GPIB **rem** errors.

rem Ensure that the gpib.com device driver is installed from **rem** config.sys.

```
GPIBerror = 0x8000
board=ibfind("gpib0")   :rem Find interface board.
ibconfig(board,3,11)    :rem Change timeout to 1 sec.
dev=ibfind("dev15")     :rem Find instrument.
gosub .err
ibwrt(dev,"cl")   :rem Clear instrument.
gosub .err
ibwrt(dev,"is")   :rem Send status command.
gosub .err
ibrd(dev,a$)      :rem Read reply.
gosub .err
print("Status returned = ";a$)
end

.err
  if (ibsta and GPIBerror) then
    print(“ERROR: GPIB error code is “;iberr)
    end         :rem Upon error exit program
  endif
return
```

ibwrtf

Syntax: ibwrtf(*device, filename*)

Description: Writes data from a file to a device.

Status and error codes are returned in the special variables **ibsta** and **iberr**.

See also: ibrdf

Example: This example will send a text file containing three short lines of commands to an external device.

To take a specific example we will configure the acquisition parameters of a Tektronix TDS350™ oscilloscope at a GPIB address of 5. The parameters consist of number of scans to average (10 scans), the time-base (1 microsecond) and the vertical scaling (100 mV/div).

The **ibconfig** command sets the timeout period (in event of error) to 1 sec.

The text file “tek350.txt” (given below) can be created using any text editor or the built-in editor in InstaSpec.

```
HOR:SEC 1E-6
CH1:SCALE 100E-3
ACQUIRE:STATE 10
cls()
GPIBerror = 0x8000
device = ibfind(“dev5”)
gosub .err
ibconfig(device,3,11)
gosub .err
file$ = “c:\tek350.txt”
ibwrtf(device, file$)
gosub .err
end

.err
  if (ibsta and GPIBerror) then
    print(“ERROR: GPIB error code is “;iberr)
    end         :rem Upon error exit program
  endif
return
```

icfft

Syntax: icfft(*#data*)

Description: Calculates the inverse FFT of a complex array in *#data*. The real and imaginary parts are held as an array of complex numbers in the data set to which they are assigned. They are held as real / imaginary pairs occupying adjacent locations in the data set. This array will normally have been created by the **cfft** function. As in the example, an assignment to a data set must accompany this function.

See also: area fft cfft carea diff smooth1

Example:
```
#2_sig=cfft(#1_sig) :rem Puts the complex FFT
                    :rem spectrum of #1_sig into
                    :rem #2_sig.
#2_sig[4]=0
#2_sig[5]=0         :rem Remove a frequency (1st
                    :rem harmonic).
#1_sig=icfft(#2_sig) :rem Change frequency back to
                    :rem spatial.
```

ignore

Syntax: ignore(*value*)

Description: Provides a means of discarding an unwanted character when reading text from the comports. The value 0x0a, corresponding to a line feed, is ignored by default.

See also: baud comread comwrite handshake terminator

Example:
```
rem receive a numeric value on com2
baud(2,9600)     :rem setup com to 9600 baud
terminator(0x0d) :rem carriage return at end of message
ignore(0x0a)     :rem any line feeds
comread(2,a$)    :rem read from com2
```

imagex

Syntax: imagex(#*dataset*)

Description: Returns the horizontal size, in pixels, of the *dataset*.

See also: imagey

Example: **rem** convert image to black and white

```
threshold=2500
xsize=imagex(#1)
ysize=imagey(#1)
xpos=1
while(xpos<=xsize)
 ypos=1
 while(ypos<=ysize)
  if(#1_sig[xpos,ypos]<threshold) then
   #1_sig[xpos,ypos]=0
  else
   #1_sig[xpos,ypos]=100
  endif
  ypos=ypos+1
 wend
 xpos=xpos+1
wend
```

imagey (not PDA)

Syntax: imagey(#*dataset*)

Description: Returns the vertical size, in pixels, of the *dataset*.

See also: imagey

Example: **rem** convert image to black and white

```
threshold=2500
xsize=imagex(#1)
ysize=imagey(#1)
xpos=1
while(xpos<=xsize)
 ypos=1
 while(ypos<=ysize)
  if(#1_sig[xpos,ypos]<threshold) then
   #1_sig[xpos,ypos]=0
  else
   #1_sig[xpos,ypos]=100
  endif
  ypos=ypos+1
 wend
 xpos=xpos+1
wend
```

InfoText

Syntax: infotext(*#Dataset, "text")*

Description: Allows user to enter text into the File Information Window associated with the dataset *#Dataset.* The **File Information Window** provides relevant information on the associated dataset, together with an edit window which allows the user to enter their own comments. This is a useful feature for uniquely identifying a set of experimental conditions (or acquisition parameters) to a particular dataset.

Example: Dataset #6 contains %Transmittance data acquired for a particular color filter FilterA11. #6 was acquired using the remote commands available in InstaSpec Basic™ and we wish to uniquely associate #6 with the corresponding filter. This can be done by writing a text message into the File Information Window associated with #6 and then (re)saving the dataset to disk (to an InstaSpec file called data1.sif off the c: drive).

Infotext(#6, "This dataset corresponds to FilterA11 acquired under a White light source")

Infotext(#6, "This is run 2 of 10 in total")

Thus, the Comments section of the File Information Window associated with data1.sif will now contain the above sentences in "", appended unto any previous messages.

inport

Syntax: inport(*port*)

Description: This function will read a WORD (2 bytes) sized value from the processor *port* address specified. The low byte of *value* is obtained from *port* and the high byte of *value* from *port* + 1. The *port* address may be specified either as a variable, a decimal number or as a hexadecimal number (by prefixing 0x). The processor ports are not to be confused with the serial or parallel ports. **NOTE: This function should be used with caution, as an invalid port address may corrupt the operation of your computer.**

See also: auxin auxout inportb outportb outport

Example: This example reads the WORD value from the address HEX745:
data = **inport**(0x745)

---

inportb

Syntax: inportb(*x*)

Description: This function returns the byte value obtained by reading the computer hardware port address *x*. This function should be used with caution, as an invalid address may upset the operation of your computer. The processor ports are not to be confused with the serial or parallel ports.

See also: outportb auxin auxout

Example: x = **inportb**(0x700):**rem** Read byte value from address 700 HEX

---

input

Syntax: input(*prompt, variable*) or input(*prompt, string_variable*)
input *text x* or input *text a$* [for compatibility with earlier versions]

Description: This function provides a means of inputting information via the keyboard during program execution. A dialog box is displayed with the specified *prompt*. A *variable* or *string_variable* can then be entered. [The syntax with bracketed parameters is preferred.]

See also: key

Example:
```
input("Enter value", x)      :rem Enter a number.
print("the number you have just entered is ";x)
input("Enter filename", f$)  :rem Enter an ASCII string.
write(f$,( "This is file ";f$))
```

instr

Syntax: instr(*text1,text2*)

Description: Searches for the first occurrence of *text2* in *text1* and returns the position at which it is found. A value of 0 is returned if *text2* is not found.

See also: left$ mid$ right$

Example:

```
input("Enter filename", f$)
a = instr(f$,".")    :rem Search for period in filename.
if a then
  name$ = left$(f$,a-1)       :rem Name.
  extension$ = mid$(f$,a+1,3) :rem Extension.
else
  name$=f$
  extension$=""               :rem No extension.
endif

print(name$)
print(extension$)

if (len( name$ ) > 8) then print("Filename too long")
```

key

Syntax: key(*text*)

Description: This function will suspend the execution of the program until a key is pressed. A dialog box is created with *text* as a prompt. The use of *text* is optional and may be omitted. If *text* is present, it may form a multi-line prompt, where a new line is forced whenever the backslash '\' character is encountered. **key** can be used on its own, where the return value is ignored, or it can return the ASCII value of the key pressed

See also: input

Example 1:
```
key()     :rem wait for a keystroke
```

Example 2:
```
key("Any key to continue")   :rem wait for a keystroke
```

Example 3:
```
k=0
while ( k<> 'q' )
  run()
  k=key("Enter data set number or 'q' to quit")
  if ( k>='1') and (k<='9') then
    k=val(chr$(k))
    #k_sig=#0_sig
  endif
wend
```

Example 4:
```
key("Operator instructions ... \\Close the shutter")
```

---

kill

Syntax: kill("*filename*")

Description: Deletes the file *filename* in the current working directory. **kill** should be used with extreme caution, as no warning is given. This function returns an error number of zero on success, and a negative number on failure. If the return value is not assigned then a failure will cause an error message to be printed.

See also: close write read

Example 1:
```
kill("testprog.dat")  :rem testprog.dat has now been deleted
```

Example 2:
```
error=kill("testprog.dat")  :rem Errors handled by program.
if error<0 then
  print("Failed to delete file")
endif
```

KineticSlice

Syntax: KineticSlice(*#Destination, #KineticSeries, xpixel)*
KineticSlice(*#Destination, #KineticSeries, xpixel, ypixel)*

Description: Generates a "kinetic slice" from a Kinetic/Fast Kinetic series dataset (*#KineticSeries)* and puts the data into a new dataset *#Destination: xpixel* specifies the x pixel through which the "slice" is to be taken.

"Kinetic slice" refers to a sequence of data where each data point in *#Destination* is acquired from each of the individual scans that comprise the kinetic series dataset. The slice starts with the first scan in *#KineticSeries* and continues until reaching the last scan in *#KineticSeries.*

If *#KineticSeries* consists of a sequence of non-fully vertically binned scans (e.g. full acquisition images) the user can also specify a y pixel co-ordinate from which to acquire a slice. If *ypixel* is omitted Andor Basic assumes a default value of 1. *xpixel* and *ypixel* must not exceed the width and height of the CCD sensor.

Example: Dataset #3 is a kinetic series consisting of 100 fully vertically binned scans which illustrate the time evolution of a particular fluorescence. An important spectral feature lies on the x pixel = 232 and is seen to decay with time. We can look at how the intensity of this feature changes with time by taking a kinetic slice through each of the 100 scans, i.e.:
KineticSlice(#4, #3, 232)
Dataset #4 should now consist of 100 data points taken (from pixel 232) from each scan in the kinetic series #3

left$

Syntax: left$(*text,x*)

Description: Returns the *x* leftmost characters of the string *text*.

See also: right$ mid$

Example: a$="abracadabra"
print(**left$**(a$,3))
print(**mid$**(a$,3,3))
print(**right$**(a$,3))

Result: abr
rac
bra

---

len

Syntax: len(*text*)

Description: Returns the length of the string text

See also: str$ val

Example: **input**("Enter filename", f$)
if (**len**(f$) > 8) **then print**("Filename too long")

---

ln

Syntax: ln(*x*) or ln(*#data*)

Description: This function returns the natural logarithm. . The function can be applied to single values or to named data within a data set. If used with named data, an assignment (either to itself or to other named data within a data set) must accompany this function.

See also: log alog exp

Example: #1_sig=**ln**(#1_sig)
y=**ln**(x)

---

load

Syntax: load(*#dataset*, *filename*)
load(*filename)*

Description: Loads a data set from disk. If *#dataset* is not specified it will be allocated automatically the lowest unused number. The number allocated is returned by the command. A return value of 0 indicates a failure in loading the data set.

See also: kill save

Example:
```
rem load a file from disk
input("Enter filename", fname$)  :rem Get the filename.
input("Enter data set", dset)    :rem Get the data set
                                 :rem number.
print("Now loading ";fname$;" to data set number ";dset)
load(#dset,fname$)               :rem Load the file
```

LoadAsciiXY

Syntax: LoadAsciiXY(*#Destination, filename)*
LoadAsciiXY(*#Destination, filename, #Calibration)*
LoadAsciiXY(*#Destination, filename, #Calibration, order)*

Description: Imports a standard x, y text (ASCII) file and puts the data into a new data set *#Destination*. The *filename* parameter specifies the file to be imported.
#Calibration is an x-axis calibrated data set (e.g. pixels or wavelength units, etc.). Using this parameter allows the user to take the x-axis calibration of *#Calibration* and apply it to the x-axis of the dataset *#Destination.* Thus the x-axis calibration of *#Destination* becomes exactly the same as *#Calibration*. In fact, each data point of *#Destination* is calculated by interpolating the data contained in *filename* against the x-axis of *#Calibration.* If the parameter *order* is not defined, cubic splines are employed for the interpolation else a polynomial with order *order* is used.
The data file *filename* must contain two columns of ASCII data separated by (e.g.) a space, tab, comma or semicolon, etc. The first column should contain the independent variable (x-axis - e.g. wavelength values) and the second column the dependent variable (y-axis - e.g. Counts), with the same number of points in each column. The minimum number of points is two, but the greater the number of points the greater the degree of accuracy attained in any final calculations.

Example 1: The file “test.dat” contains two columns of data, separated by a ',' delimiter. The file is given below and can be created with any standard editor or (as in this case) created using the built-in editor in InstaSpec. In this example we will import the file test.dat (which resides in a subdirectory “TestData” of the C: drive) into a new dataset, #12.

```
100, 21
200, 24
300, 27
400, 22
500, 25
600, 33
700, 31
800, 29
900, 36
1000, 32
```

LoadAsciiXY(#12, "c:\testdata\test.dat")

Data set #12 should now contain the data curve (10 points) from test.dat.

LoadAsciiXY (continued)

Example 2: Dataset #10 contains x-axis calibrated data (1024 points) ranging from 360 to 830 nm (this is the visible wavelength region for an 'average young observer'). This example will import the data file "test.dat", as above, and interpolate the data of test.dat to match the calibration range of #10.

LoadAsciiXY(#12, "c:\testdata\test.dat", #10)

The data set #12 should now contain the data curve from test.dat and be interpolated to the wavelength range 360 to 830 nm, i.e. the data curve from test.dat now contains 1024 data points interpolated to the x-axis calibration of #10.

log

Syntax: log(*x*) or log(*#data*)

Description: This function takes the log to the base 10. The function can be applied to single values or to named data within a data set. If used with named data, an assignment (either to itself or to other named data within a data set) must accompany this function.

See also: alog ln exp

Example:
```
#1_sig=log(#1_sig):rem Take log of signal data of data set
                   :rem #1.
y=log(x)           :rem Take the log of x and assign it to y.
```

MaximizeWindow

Syntax: MaximizeWindow(*#dataset*)

Description: This function maximizes the window holding the data set *#dataset*. *Data Windows* may be maximized, minimized or normal sized. When maximized they are their largest possible size within the bounds of the application.

When minimized they are reduced to icons where only the name of the data set is shown. When restored they return to their normal size. This normal size may be changed by standard mouse operations or with the **MoveWindow** function.

MaximizeWindow can also be used with the current *Program Editor Window* or the *Program Output Window*: replace the *#dataset* parameter with *program* or *output* respectively.

See also: CloseWindow MoveWindow RestoreWindow MinimizeWindow TopWindow

Example:
```
rem Create a data set by loading data from disk
load(#2,"penray.sif")
rem now maximize the window
MaximizeWindow(#2)
```

---

maxpos

Syntax: maxpos(*#data*, *start_pixel*, *end_pixel*)

Description: Returns the position of the highest value in *#data* within the given range from *start_pixel* to *end_pixel*, inclusive.

See also: minpos maxval minval

Example:
```
rem Find the pixel position of the maximum value in #1_sig.
mpos = maxpos(#1_sig,1,1024)
rem Now find the actual value.
mval = maxval(#1_sig,1,1024)
print("The maximum is ";mval;" at pixel number ";mpos
```

---

maxval

Syntax: maxval(*#data*, *start_pixel*, *end_pixel*)

Description: Returns the highest value in *#data* within the given range from *start_pixel* to *end_pixel*, inclusive.

See also: maxpos minpos minval

Example:
```
rem Find the pixel position of the maximum value in #1.
mpos = maxpos(#1_sig,1,1024)
rem Now find the actual value.
mval = maxval(#1_sig,1,1024)
print("The maximum is ";mval;" at pixel number ";mpos)
```

---

mid$

Syntax: mid$(*text,x,y*)

Description: Returns *y* characters from the string *text* starting at position *x*.

See also: right$ left$

Example:
```
a$="abracadabra"
print(left$(a$,3))
print(mid$(a$,3,4))
print(right$(a$,3))
```

Result:
```
Abr
Raca
Bra
```

MinimizeWindow

Syntax: MinimizeWindow(*#dataset*)

Description: This function minimizes the window holding the data set *#dataset*. *Data Windows* may be maximized, minimized or normal sized. When maximized they are their largest possible size within the bounds of the application. When minimized they are reduced to icons where only the name of the data set is shown. When restored they return to their normal size. This normal size may be changed by standard mouse operations or with the **MoveWindow** function.

MinimizeWindow can also be used with the current *Program Editor Window* or the *Program Output Window*: replace the *#dataset* parameter with *program* or *output* respectively.

See also: CloseWindow MoveWindow RestoreWindow MaximizeWindow TopWindow

Example:
```
rem Create a data set by loading data from disk.
load(#2,"penray.sif")
rem Now minimize the window.
MinimizeWindow(#2)
```

---

minpos

Syntax: minpos(*#data*, *start_pixel*, *end_pixel*)

Description: This function returns the position of the lowest value in *#data*, within the given range from *start_pixel* to *end_pixel*, inclusive

See also: maxpos maxval minval

Example:
```
rem Find the pixel position of the minimum value in #1_sig.
mpos = minpos(#1_sig,1,1024)
rem Now find the actual value.
mval = minval(#1_sig,1,1024)
print("The minimum is ";mval;" at pixel number ";mpos)
```

---

minval

Syntax: minval(*#data*, *start_pixel*, *end_pixel*)

Description: This function returns the lowest value in *#data*, within the given range from *start_pixel* to *end_pixel*, inclusive

See also: maxpos minpos maxval

Example:
```
rem Find the pixel position of the minimum value in #1_sig.
mpos = minpos(#1_sig,1,1024)
rem Now find the actual value.
mval = minval(#1_sig,1,1024)
print("The minimum is ";mval;" at pixel number ";mpos)
```

---

mod

Syntax: mod(x,y)

Description: Returns the remainder of the ratio x/y

See also:

Example 1:
```
x = 9
y = 5
a = mod(x,y)   :rem Calculate the remainder of x/y
```
Result: a = 4

Example 2:
```
x = 5
y = 8
a = mod(x,y)
```
Result: a = 5 (5/8 = 0 and 5 remaining)

MoveWindow

Syntax: MoveWindow(*#dataset*, *left, top, width,height*)

Description: This function moves the window holding the data set *#dataset* and resizes it to the dimensions in the arguments. *Left, top,width* and *height* are given in screen pixels. *Data Windows* may be maximized, minimized or normal sized. When maximized they are their largest possible size within the bounds of the application. When minimized they are reduced to icons where only the name of the data set is shown. When restored they return to their normal size. This normal size may be changed by standard mouse operations or with the **MoveWindow** function.

MoveWindow can also be used with the current *Program Editor Window* or the *Program Output Window*: replace the *#dataset* parameter with *program* or *output* respectively.

See also: CloseWindow RestoreWindow MinimizeWindow MaximizeWindow TopWindow

Example:
```
rem Create a data set by loading data from disk.
load(#2,"penray.sif")
rem Now move the window to the top left
rem of the viewing area.
MoveWindow(#2,0,0,200,150)
```

---

newline

Syntax: newline(*value*)

Description: This command determines which characters are appended to a **comwrite**.

Value = 0 no characters appended

Value = 1 linefeed appended (0x0a)

Value = 2 line feed and carriage return appended (0x0a, 0x0d)

The default value is 2.

See also: comwrite terminator

Example:
```
newline(0)
```

---

outport

Syntax: outport(*port, value*)

Description: This function will send a WORD (2 bytes) sized value to the processor *port* address specified. The low byte of *value* is sent to *port* and the high byte of *value* is sent to *port* + 1. The *port* address and *value* may be specified either as a variable, a decimal number or as a hexadecimal number (by prefixing 0x). The processor ports are not to be confused with the serial or parallel ports.

NOTE: This function should be used with caution, as an inappropriate port address may corrupt the operation of your computer.

See also: auxin auxout inportb outportb inport

Example: This example sends the hexadecimal number E8 to address HEX745.

outport(0x745, 0xE8)

---

outportb

Syntax: outportb(*port,value*)

Description: This function will send a byte sized *value* to the processor *port* address specified. The *port* address and *value* may be specified either as a variable, a decimal number, or as a hexadecimal number (by prefixing 0x). The processor ports are not to be confused with the serial or parallel ports.

See also: auxin auxout inportb

Example: **outportb**(0x745,12) :**rem** Send a character to address HEX745.

---

output

Syntax: output(*left,top,width,height*)

Description: The output window is where text generated by the programming language is placed. The function **output** allows this window to be positioned precisely within the main InstaSpec window. The parameters *left, top, width* and *height* are given in screen pixels. *Left* and *top* are relative to the InstaSpec window. Only one output window is permitted so this function can be used to move the output window if it exists, or create it if it does not exist.

Example: **output**(50,100,200,200) :**rem** Make a square output window.

---

poly

Syntax: poly(Dest, Src, nPoints, order)

Description: This function is designed to take a dataset (*Src*) and create a new dataset (*Dest*) which is a polynomial approximation of order *order* with *nPoints* points.

See also: LoadAsciiXY

Example:
```
rem load file data1.sif which contains a calibrated spectrum
rem import an ascii file and give it the same calibration
rem create new data set which is a polynomial of order 3 with
rem 2000 points

load(#1, "data1.sif")
loadasciixy(#2, "test.txt", #1)
poly(#3, #2, 2000, 3)
```

PositionCursor

Syntax: PositionCursor(*#data*,*xpixel*)

PositionCursor(*#data*,*xpixel*,*ypixel*) - not PDA

Description: Puts the cursor on the desired pixel in the active data set. *xpixel* refers to the horizontal pixel(column) on the detector. *ypixel* refers to vertical pixel (row) on the detector. There is no need to specify the *ypixel* if the data is single track.

See also: ActiveOverlay ActiveTrack AddOverlay ChangeDisplay

Example:
```
load(#1,"image.sif")        :rem load an image
PositionCursor(#1,100,200)  :rem place the cursor on pixel
                            :rem (100,200) on the display.
```

---

print

Syntax: print(*variable_list*)
print *variable_list* [for compatibility with earlier versions]

Description: This function outputs text to the screen. *variable_list* is a list of numeric or string variables, separated by commas or semicolons. If the text is separated by commas, then the output text will be separated by tab spaces. If, however, the text is separated by semicolons, then the output text will be continuous. The placing of a semicolon at the end of the *variable_list* will inhibit the normal action of forcing a new-line. Using the print function on its own will send a new line to the screen. [The syntax with bracketed parameters is preferred.]

See also: cls update

Example:
```
print("The value is " ; x)
print("The value of pixel 3 is ";#1[3];"counts")
```

---

rayremove2

Syntax: rayremove2(*#dataset1*, *#dataset2*, *sensitivity*)

Description: Compares two similar data sets and replaces any values which are significantly larger in one with a scaled version of the pixel value in the other data set. The larger the value of sensitivity the larger the difference has to be before a pixel value is changed.

Example: This example assumes the data sets contain similar values with an occasional large value due to a cosmic ray. The function rayremove2 will attempt to correct these large values.

Result:
load(#1, "cosmic1.sif")
load(#2, "cosmic2.sif")
rayremove2(#1,#2,1

read

Syntax: read(*filename*, *string_variable*)

Description: Reads a line of text from a file and assigns it to the given *string_variable*. As long as the *filename* does not change successive reads will fetch consecutive lines from the file. This function returns an error number of zero on success, and a negative number on failure.

See also: write save close kill

Example 1:
```
f$="header.txt"
 read(f$,a$)
 print("The first line of ";f$;" is ";a$)
```

Example 2:
```
f$="header.txt"
 a=read(f$,a$)
 if a<0 then
  print("Error on read")
 else
  print("The first line of ";f$;" is ";a$)
 endif
```

---

reset

Syntax: reset(*#dataset*)

Description: This function causes InstaSpec to return the active data to its original configuration - undoing any adjustment to scale that you may have performed. Reset is available for all the Display Modes.

See also: ActiveKineticPosition ActiveTrack rescale ScaleData scalex scaley

Example:
```
rem Load an image(1024x256).
rem Move to track 100 and then reset.

load(#1,"image.sif") :rem load data
ActiveTrack(#1,100)  :rem move to a particular track
reset(#1)            :rem reset display to original config.
```

---

rescale

Syntax: rescale(*#dataset*)

Description: This function acts on the active data set. It causes InstaSpec to display against an appropriate data scale (and, in the case of Image Display Mode, in appropriate colors or grayscale tones) all data that falls within your selected range on the x- (and y-) axes. The rescale will be performed according to the current rescale mode. There are three types of rescale mode which can be selected from the menu: these will cause the data to be scaled between minimum and the maximum (min...max); between zero and the maximum (0...max); or between 0 and 65535 (0...65535), the maximum number of counts that can be recorded for a single pixel. Rescale is available for all the Display Modes.

See also: ChangeDisplay reset ScaleData scalex scaley

Example: **rem** Load a data set and rescale.
load(#1,"data1.sif") **:rem** load data
ScaleData(#1,0,1000) **:rem** scale data axis
rescale(#1) **:rem** return to original scaling

RestoreWindow

Syntax: RestoreWindow(*#dataset*)

Description: This function restores the window holding the data set *#dataset*. *Data Windows* may be maximized, minimized or normal sized. When restored they return to their normal size. This normal size may be changed by standard mouse operations or with the **MoveWindow** function. **RestoreWindow** can also be used with the current *Program Editor Window* or the *Program Output Window*: replace the *#dataset* parameter with *program* or *output* respectively.

See also: CloseWindow MoveWindow MinimizeWindow MaximizeWindow TopWindow

Example:
```
rem Create a data set by loading data from disk.
load(#2,"penray.sif")
MaximizeWindow(#2)
...
...
...
rem Now restore the window.
RestoreWindow(#2)
```

right$

Syntax: right$(*text,x*)

Description: Returns the *x* rightmost characters of the string *text*.

See also: left$ mid$

Example: a$="abracadabra"
print(**left$**(a$,3))
print(**mid$**(a$,3,3))
print(**right$**(a$,3))

Result: abr
rac
bra

run

Syntax: run( )

Description: Run, by default, starts a Signal acquisition using the current parameters as defined under the Acquisition menu. Each of the parameters may be changed, however, by using the appropriate function. The effect of any change remains in force for the remainder of that program but is lost when the program ends.

Each of the following functions takes a single integer (0 - 4) as its argument. The table shows the meaning of this value for each of the functions.

	0	1	2	3	4
SetAcquisitionMode	-	Single Scan	Accumulate	Kinetics	-
SetAcquisitionType	Signal	Background	Reference	Cal	-
SetTriggerMode	Internal	External	-	-	-
SetReadoutMode	Full Vertical Binning	Multi-Track	-	Single-Track	Full Res Image

Example 1: **SetAcquisitionMode**(1) :**rem** Single scan
SetReadoutMode(0) :**rem** Full vertical binning

The following function is used to set the data-type. It takes a single integer argument (0 - 9). The table shows the meaning for each of the values.

SetDataType	0	User
	1	Counts
	2	Counts (Background Corrected)
	3	% Absorptance
	4	% Reflectance
	5	% Transmittance
	6	Absorbance Units
	7	Absorption Coefficient
	8	Data * Ref
	9	Log Base 10

Example 2: **SetDataType**(2) :**rem** counts-background.

Each of the following functions takes a single positive value as its argument.

	Minimum	Maximum
SetExposureTime	0	4e6
SetAccumulateCycleTime	0	4e6
SetAccumulateNumber	1	32767
SetKineticCycleTime	0	4e6
SetKineticNumber	1	32767

SetExposureTime(0.6) **:rem** 0.6 Secs exposure.

save

Syntax: save(*#dataset*, *filename*)

Description: This function saves the data in *#dataset* to a file '*filename*'. The title bar for *#dataset* will also be updated to reflect the name used when saving to a file. No warning is given if the file already exists

See also: load kill

Example:
```
input("Enter filename", fname$)
input("Enter data set to save", dset)
print("Now saving data set";dset;" to filename ";fname$)
save(#dset, fname$)
    :rem Save data set #dset to the file called
    :rem fname$.
```

SaveAsciiXY

Syntax: SaveAsciiXY(file$, ascii$)

SaveAsciiXY(file$, ascii$, delimiter)

SaveAsciiXY(#data, ascii$)
SaveAsciiXY(#data, ascii$, delimiter)

file$ = InstaSpec SIF file, ascii$ = ASCII (text) file
#data = InstaSpec SIF dataset.

Delimiter = 1 corresponds to a comma (,)
delimiter = 2 corresponds to a tab ( )
delimiter = 3 corresponds to a semicolon (;)
delimiter = 4 corresponds to a space ( )

Description: Saves an InstaSpec sif file into the corresponding ASCII file. The original sif file remains unchanged.

The delimiter is used to separate the ASCII (text) columns. The delimiter is optional and the default case is the current delimiter specified by InstaSpec in the menu option File/Export…As ASCII.

The first option will save out to an ASCII file the corresponding InstaSpec sif file without loading the sif file into InstaSpec.

The third option will save out to an ASCII file the corresponding sif data set which has already been loaded into (or newly created in) InstaSpec.

SaveAsciiXY() automatically detects the sif file format and will save out to the corresponding ASCII file format. The ASCII format is the same format as that used by the File/Export…As ASCII menu option in the InstaSpec program.

If the sif file is a Full Vertically Binned (FVB) scan the corresponding ASCII file will consist of two columns: the first column consists of the x values and the second column comprises the data values. A Single Track scan will have the same format. A Full Resolution Image (FRI) (and a Multi-Track scan) will consist of N+1 columns, where N = number of rows (or tracks) present on the CCD sensor. Each column corresponds to the data from the associated row of the CCD, i.e. the second column comprises the data from the first row of the CCD, the third column = data from second row, etc.

NOTE: Kinetic Series: SaveAsciiXY() is implemented for FVB kinetic series and Single Track kinetic series only.

SaveAsciiXY (continued)

See also: LoadAsciiXY

Example 1: The file bluesky.sif is in the directory "c:\instaspc\images". This example will save the file into the corresponding ASCII format in the same directory. The columns will be separated by a semicolon (;).

```
file$ = "c:\instaspc\images\bluesky.sif"
ascii$ = "c:\instaspc\images\bluesky.asc"
SaveAsciiXY(file$, ascii$, 3)
```

Example 2: In this example the data set #21, which consists of a Full Resolution Image, is saved in ASCII format to a file called "test.asc" in the current directory.

The column delimiter is the delimiter last used by InstaSpec in the menu option File/Export…As ASCII.

```
SaveAsciiXY(#21,"test.asc")
```

---

scale (not PDA)

Syntax: scale(*#dset*,*xminpixel*,*xmaxpixel*,*yminpixel*,*ymaxpixel*)

Description: Sets the x (horizontal) and y (vertical) limits of the active data set in the window numbered *#dset*. This function is used to zoom in or out in an image

See also: ActiveOverlay AddOverlay ScaleData scalex scaley

Example:

```
rem Take an acquisition of an image
rem and zoom into a particular region.

SetDataType(1)          :rem counts
SetAcquisitionType(0)   :rem signal
SetAcquisitionMode(1)   :rem single scan
SetReadoutMode(4)       :rem full resolution image
run()
ChangeDisplay(#0,2)     :rem display data as image
scale(#0,200,400,50,100) :rem zoom in
```

ScaleData

Syntax: ScaleData(*#data*,*minval*,*maxval*)

Description: Sets the data-axis limits of the active data set in the window numbered *#data*. When the data in an image are scaled, the computer finds the maximum (*maxval*) and minimum (*minval*) data values, and scales the available number of colors to the range, where the range = *maxval* - *minval*.

See also: AddOverlay scale scalex scaley

Example:
```
load(#1,"data1.sif") :rem Load data.
ScaleData(#1,0,1000) :rem Display data in the range
                     :rem 0 to 1000.
```

ScaleToActive

Syntax: ScaleToActive(#1)

Description: This function causes all the data traces displayed in the same data window to be displayed with the same x and data scaling as the active data trace. In order for the function to be fully successful, the other data traces must have the same x-calibration as the active data trace.

See also: ActiveOverlay AddOverlay ScaleData scalex scaley

Example:
```
rem Load three data sets.
rem Choose the second one as the active data set.
rem Set the x and data scales of the second data set
rem and then scale the other data to be the same.

load(#1,"data1.sif")       :rem Load the data.
load(#2,"data2.sif")
load(#3,"data3.sif")
overlay1=AddOverlay(#1,#2) :rem Copy the data
                           :rem into the same window.
overlay2=AddOverlay(#1,#3)
ActiveOverlay(#1,overlay1) :rem Choose #2 to be the active
                           :rem data set.
scalex(#1,200,400)         :rem Scale the x and data axis
                           :rem of #2.
ScaleData(#1,0,1000)
ScaleToActive(#1)          :rem Display #2 and #3
                           :rem with the same scaling as #2.
```

scalex

Syntax: scalex(*#dset*,*minpixel*,*maxpixel*)

Description: Sets the x-axis limits of the active data set in the window numbered *#dset*. This function is used to zoom in or out in x.

See also: ActiveOverlay AddOverlay scale ScaleData scaley

Example:
```
load(#1,"data1.sif")  :rem Load data.
scalex(#1,200,400)    :rem Display the data
                      :rem between pixels 200 and 400.
```

Scaley (not PDA)

Syntax: scaley(*#dset*,*minpixel*,*maxpixel*)

Description: This function is only applicable when the data are displayed as an image. Sets the y-axis (vertical display) limits of the *#dset*.

See also: ChangeDisplay scale ScaleData scalex

Example:
```
load(#1,"data1.sif")  :rem load an image
ChangeDisplay(#1,2)   :rem change display to image
scaley(#1,50,150)     :rem zoom in along the vertical axis
```

---

separator

Syntax: separator(*text*)

Description: All the characters in *text* are treated as separators. They are used by the read function, in order to determine when to stop reading characters from a file. This is particularly useful when reading tables of numbers separated by spaces or commas.

See also: read write kill close

Example 1 :
```
i = 1
separator(" ,")         :rem Spaces & commas are
                        :rem separators.
create(#1_sig,1024)
while i<1025
  read("textfile.dat",a$):rem Read 1024 data points and...
  #1_sig[i] = val(a$)   :rem display in signal data set 1.
  i = i+1
wend
```

Example 2:
```
separator(chr$(10)) :rem Non printing character as separator.
```

Example 3:
```
separator(" ,";chr$(10)):rem Space comma & non-printing
                        :rem character as separator.
```

SetAccumulate

Syntax: SetAccumulate(*expTime*, *numAccums*, *ACT*)

Description: Sets the CCD acquisition mode to Accumulate.

Accumulation is the process by which data that have been acquired from a number of similar scans are added together in computer memory. This results in an improved signal to noise ratio.

expTime The exposure time (in secs).

NumAccums The number of acquisitions to accumulate together to form one composite scan.

ACT The Accumulate Cycle Time (ACT (in secs)) i.e. the delay or periodicity between successive acquisitions.

NOTES:

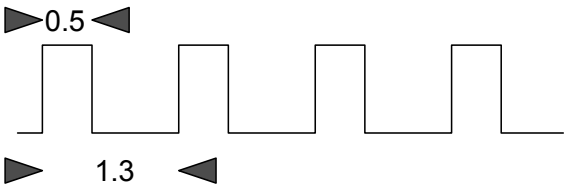
1. **The system will default to a minimum Exposure Time should you attempt to enter too low a value.**
2. **Each timing parameter (e.g expTime, ACT etc) may be automatically rounded up to the nearest rated value. The command ShowTimings( ) prints (to the Output Window) the actual timing sequences used by InstaSpec.**

Replaces: SetAccumulate( ) renders the following commands obsolete:

SetExposureTime( ), SetAccumulateNumber( ), SetAccumulateCycleTime( ) and SetAcquisitionMode(2).

SetSingleScan SetKinetics SetFastKinetics SetFrameTransfer

Example: This example programs the CCD sensor to acquire a Full Vertically Binned (FVB) scan composed of 4 accumulations. Each accumulation is to be obtained with an exposure time of 0.5 sec every 1.3 secs.



SetAccumulate(0.5,4,1.3)
SetFVB()
ShowTimings() :rem Prints actual timings
Run()

SetAccumulateCycleTime (superseded by SetAccumulate, SetKinetics)

Syntax: SetAccumulateCycleTime(*time*)

Description: Sets the accumulate cycle time to *time* (in seconds). This function will only take effect if the acquisition mode has been set to Accumulate or Kinetics and the trigger mode is Internal. The period entered is rounded up to the nearest permissible value. The function is not available if you are using External trigger mode, since an external trigger can occur at any time and with any or no periodicity.

See also: run + full list of acquisition functions

Example: **SetAccumulateCycleTime**(0.5) :**rem** Sets the accumulate cycle
:rem time to 500ms.

SetAccumulateNumber (superseded by SetAccumulate, SetKinetics)

Syntax: SetAccumulateNumber(*count*)

Description: Sets the number of scans to be added together in computer memory to form an 'accumulated' scan. This function only takes effect if the acquisition mode has been set to Accumulate or Kinetics

See also: run + full list of acquisition functions

Example: **SetAccumulateNumber**(5) :**rem** Sets the number of scans
:rem accumulated to 5.

SetAcquisitionMode (superseded by - see Description)

Syntax: SetAcquisitionMode(*mode*)

Description: Sets the acquisition mode to one of the following:
1 = Single Scan
2 = Accumulate
3 = Kinetics.
All other values are reserved. For full description of each mode refer to the User's Guide.
SetAcquisitionMode(1) superseded by SetSingleScan.
SetAcquisitionMode(2) superseded by SetAccumulate.
SetAcquisitionMode(3) superseded by SetKinetics. SetAcquisitionMode(4) superseded by SetFastKinetics

See also: run + full list of acquisition functions

Example: **SetAcquisitionMode**(1) :**rem** Single Scan.

SetAcquisitionType

Syntax: SetAcquisitionType(*type*)

Description: Set the acquisition type to one of the following:
0 = Signal;

1 = Background

2 = Reference

All other values are reserved. For full description of each mode refer to the Users Guide.

See also: run + full list of acquisition functions

Example: **SetAcquisitionType**(1) :**rem** Background.

SetCenterRow (not PDA), superseded by SetSingleTrack

Syntax: SetCenterRow(*row*)

Description: Sets the center row of the track read out when using the single track readout mode.

See also: run + full list of acquisition functions

Example: **SetReadoutMode**(3) :**rem** Set the readout mode to Single Track.

SetCenterRow(50) :**rem** Set the center row to 50.

SetSingleTrackHeight(11) :**rem** Set track height to 11.

:**rem** Reads out tracks 45 to 55.

SetCoefficient

Syntax: SetCoefficient(#*data*,C1, C2, C3, C4)
SetCoefficient(#*data*)

Description: Sets the calibration coefficients for the x-axis of a data set (*#data)*. (NOTE the x-axis label remains unchanged).
If the coefficients C1, C2, C3 and C4 are omitted then InstaSpec will remove the x-axis calibration from #*data* and change the x-axis label to pixels.

InstaSpec stores the x-axis calibration as a third order polynomial of the form:

Calibrated X value = C1 + C2*P + C3*P*P + C4*P*P*P

Where P is the pixel number (e.g. 1 to 512 or 1024, etc.)

See also: Coefficient$

Example: This example will set the calibration coefficients of the data set #11 and amend the x-axis label to “Wavelength um” (using SetXLabel).

SetCoefficient(#11, 0.254,3.4e-4,-4.4e-9,-7.6e-12)

SetXLabel(#11,2,2)

SetDataLabel

Syntax: SetDataLabel(*#DataSet, label)*

Description: Allows you to (re)set the data axis label of the data set #DataSet. The value you assign to *label* must lie in the range 1 to 14. The labels that correspond to each of the possible values are shown below:

1	Counts
2	Counts (Bg Corrected)
3	% Absorptance
4	% Reflectance
5	% Transmittance
6	Absorbance Units
7	Absorption Coefficient (/ m)
8	Attenuation (dB)
9	Data*Ref
10	Log Base 10
11	Irradiance (uW/cm2/nm)
12	Irr. Cal. (uW/cm2/nm)
13	Radiance (uW/cm2/sr/nm)
14	Rad. Cal. (uW/cm2/sr/nm)

See also: SetXLabel

Example: This example shows how to reset the data axis label of the data set #4 to Counts (Bg Corrected):

```
SetDataLabel(#4,2) :rem Data label is now
                   :rem Counts (Bg Corrected)
```

SetDataType

Syntax: SetDataType(*type*)

Description: Sets the data type of a resultant acquisition to one of the types listed below. This option will only take effect when the Acquisition Type is "Signal". Background scan are always in Counts while Reference scans are in Counts(Bg corrected).

1 = Counts
2 = Counts (Background corrected)
3 = % Absorptance
4 = % Reflectance
5 = % Transmittance
6 = Absorbance Units
7 = Absorption coefficient
8 = Attenuation
9 = Data * Ref
10 = Log (Base 10)

All other values are reserved. For full description of each mode refer to the Users Guide.

See also: run + full list of acquisition functions

Example: **SetDataType**(2) :**rem** Set the data type
:rem to Counts (Bg corrected).

SetExposureTime (superseded by SetAccumulate, SetKinetics, SetSingleScan)

Syntax: SetExposureTime(*time*)

Description: Sets the exposure/shutter time of individual scans to *time* in seconds. Rounds up to the nearest permissible time. Exposure time is only applicable while the readout mode is set to Full Vertical Binning.

See also: run + full list of acquisition functions

Example: **SetReadoutMode**(0) :**rem** Set readout mode to Full Vertical
:rem Binning.
SetExposureTime(0.01) :**rem** Set the exposure time to 10 ms.
Run() :**rem** Take acquisition.
SetReadoutMode(1) :**rem** Set readout mode to Multi-Track.
SetExposureTime(0.1) :**rem** Set the shutter time to 100 ms.
Run() :**rem** Take acquisition.

SetFastKinetics (not PDA)

Syntax: SetFastKinetics(*expTime*, *number*, *height*)

Description: Sets the CCD acquisition mode to Fast Kinetics and configures the fast kinetics parameters *expTime*, *number*, *height*. Fast Kinetics allows exposure times on a microsecond timescale. Use Fast Kinetics when you need an exposure time that is smaller than the minimum Kinetic Cycle Time in a standard Kinetic Series, see the User's Guide for more information.

expTime = The exposure time (in micro-secs).
number = The number of spectra you want to acquire.
height = Light-sensitive sub-area height of CCD sensor (in rows)

In Fast Kinetics the spectrum to be recorded is imaged across the top of the CCD (the sub-area). The sub-area forms the light sensitive region of the CCD. The un-illuminated part of the CCD is used for storage of spectra before readout. You must ensure that light does not fall on this storage part of the CCD by using, for example, an imaging spectrograph. The minimum exposure time is a function of the vertical shift speed and the number of rows in the sub-area. Thus, assuming a typical value of 16 microseconds for the vertical shift speed and a sub-area of 8 rows, the minimum exposure time will be 128 microseconds. During binning the "sub-area" can be readout in one of two ways:

- Fully binned in the vertical direction using SetFVB( )
- Binned as an image, using SetImage( )

NOTE: The system will default to a minimum Exposure Time should you attempt to enter too low a value. Use the Andor Basic command ShowTimings( ) to determine the actual exposure time calculated by InstaSpec which, provided it does not go below the minimum exposure time for your system, should correspond with your chosen value, rounded up to the nearest microsecond.

Replaces: SetFastKinetics( ) renders the following commands obsolete:
SetFKExposureTime( ), SetFKHeight( ), SetFKNumber( ) and SetAcquisitionMode(4)

See also: SetSingleScan SetAccumulate SetKinetics SetFrameTransfer SetFVB SetImage

Example: This example will program the CCD sensor to Fast Kinetics, readout mode to Full Vertical Binning (FVB) and specify the exposure time to be 100 microseconds, height=4 rows and number in series = 64.

```
SetFastKinetics(100,64,4)
SetFVB()        :rem Applies to 4 rows in sub-area
ShowTimings()   :rem Prints actual timings
Run()
```

SetFKExposureTime (not PDA, superseded by SetFastKinetics)

Syntax: SetFKExposureTime(*time)*

Description: Sets the Fast Kinetics exposure *time* in microseconds. This function is only applicable if the acquisition mode has been set to Fast Kinetics.

In Fast Kinetics the spectrum to be recorded is imaged across the top of the CCD (the sub-area). The unilluminated part of the CCD is used for storage of spectra before readout. You must ensure that light does not fall on this storage part of the CCD by using, for example, an imaging spectrograph.

The minimum exposure *time* is calculated as vertical shift speed x number of rows in the sub-area. Thus, assuming a typical value of 16 microseconds for the vertical shift speed and a sub-area of 8 rows, the minimum exposure time will be 128 microseconds.

The user is advised to use the Andor Basic command **ShowTimings** to ascertain the actual exposure time calculated by InstaSpec which, provided it does not go below the minimum exposure time for your system, should correspond with your chosen value, rounded up to the nearest microsecond.

See also: SetFKHeight SetFKNumber ShowTimings

Example: This example will set the acquisition mode to Fast Kinetics, readout mode to Full Vertical Binning and specify the exposure time to be 100 microseconds:

SetAcquisitionMode(4) :**rem** Fast Kinetics
SetReadoutMode(0) :**rem** Full Vertical Binning
SetFKNumber(50) :**rem** *number* = 50 scans.
SetFKHeight(3) :**rem** track *height* = 3 CCD rows.
SetFKExposureTime(100) :**rem** *time* = 100 microsecs

SetFKHeight (not PDA, superseded by SetFastKinetics)

Syntax: SetFKHeight(*height)*

Description: This function is only applicable if the acquisition mode has been set to Fast Kinetics. It sets the *height* (in pixels) of the CCD sub-area.

In Fast Kinetics the spectrum to be recorded is imaged across the top of the CCD (the sub-area). The unilluminated part of the CCD is used for storage of spectra before readout. You must ensure that light does not fall on this storage part of the CCD by using, for example, an imaging spectrograph.

The *height* of the sub-area is constrained by the height of the CCD chip and the number of spectra in the series, i.e.

maximum sub-area height <= chip height/(number of spectra + 1)

The +1 appears in the above equation because you cannot use the (illuminated) sub-area of the chip for storing spectra.

See also: SetFKNumber SetFKExposureTime

Example: This example will set the acquisition mode to Fast Kinetics, readout mode to Full Vertical Binning and specify the top 4 CCD rows as the Fast Kinetics sub-area.

SetAcquisitionMode(4) :**rem** Fast Kinetics
SetReadoutMode(0) :**rem** Full Vertical Binning
SetFKExposureTime(88) :**rem** 88 microseconds
SetFKNumber(10) :**rem** 10 spectra in the FK series
SetFKHeight(4) :**rem** sub-area = 4 rows.

SetFKNumber (not PDA, superseded by SetFastKinetics)

Syntax: SetFKNumber(*number)*

Description: Sets the *number* of Fast Kinetic scans to be acquired as members of a kinetic series. This function is only applicable if the acquisition mode has been set to Fast Kinetics.

In Fast Kinetics the spectrum to be recorded is imaged across the top of the CCD (the sub-area). The unilluminated part of the CCD is used for storage of spectra before readout. You must ensure that light does not fall on this storage part of the CCD by using, for example, an imaging spectrograph.

The maximum *number* of spectra is constrained by the height of the CCD chip and the height of the sub-area i.e.:

maximum number spectra <= (chip height/sub-area height) – 1

The -1 appears in the above equation because you cannot use the (illuminated) sub-area of the chip for storing spectra.

See also: SetFKHeight SetFKExposureTime

Example: This example will set the acquisition mode to Fast Kinetics, readout mode to Full Vertical Binning and specify the number of scans to be 50.

SetAcquisitionMode(4) :**rem** Fast Kinetics
SetReadoutMode(0) :**rem** Full Vertical Binning
SetFKExposureTime(88) :**rem** 88 microseconds
SetFKNumber(50) :**rem** 50 spectra in the FK series
SetFKHeight(4) :**rem** sub-area = 4 rows.

SetFVB (not PDA))

Syntax: 1) SetFVB( )
2) SetFVB(*HBin*)

Description: Sets the CCD binning mode to Full Vertical Binning (FVB). See the User's Guide for a full description of the CCD binning modes

Replaces: SetFVB() renders the following commands obsolete:
SetReadoutMode(0)

See also: SetMultiTracks SetSingleTrack SetImage

Example: This example programs the CCD sensor to acquire a single FVB scan of exposure time 0.5 sec with horizontal binning set to be 2:

SetFVB(2)
SetSingleScan(0.5)
Run()

SetGain (not PDA or CCD)

Syntax: SetGain(*gain*)

Description: Allows the user to control the voltage across the microchannel plate of an Andor ICCD. Increasing the gain increases the voltage and so amplifies the signal. Gain values between 0 and 255 are permitted.

See also: SetGateMode

Example: This example shows how to set the gain:

SetGain(10)

SetGate (not PDA or CCD)

Syntax: SetGate(*delay*, *width*, *step*)

Description: Sets the Gater parameters for an ICCD system. The image intensifier of the InstaSpec V system acts as a shutter on nanosecond time-scales using a process known as gating.

delay Sets the delay(>=0) between the T0 and C outputs on the SRS box to 'delay' nanoseconds.

width Sets the width(>=0) of the gate in nanoseconds.

step Sets the amount (<>0, in nanoseconds) by which the gate position is moved in time after each scan in a kinetic series.

For a full description of the SRS and the Gater refer to the Users' Guide

Replaces: SetGate( ) renders the following commands obsolete:

SetGateDelay( ), SetGateWidth( ) and SetGateStep( )

Example: This example will program the ICCD gater with the following parameters: delay=100 ns, width = 15 ns and step = 20 ns:

SetGate(100,15,20)

SetGateDelay (not PDA or CCD, superseded by SetGate)

Syntax: SetGateDelay(*delay*)

Description: Sets the delay between the T0 and C outputs on the SRS box to *delay* seconds. This is only applicable to InstaSpec V systems. For a full description of the SRS and the Gater refer to the User's Guide.

See also: run + full list of acquisition functions

Example: **SetGateDelay**(10e-9) :**rem** Set the delay to 10
:rem nanoseconds.

SetGateMode (not PDA or CCD)

Syntax: **SetGateMode(*mode*)**

Description: Enables software control of the photocathode gating mode.

- ***mode = 0:*** Gating is controlled from FIRE signal ANDed with the gater input. This can be important when the system has to use free-running pulsed lasers which are continuously triggering the gater input. ANDing the FIRE signal and the gater input prevents laser pulses from being accumulated on the CCD during readout.
- ***mode = 1:*** Gating is controlled from the FIRE signal only. The FIRE signal is internal and the gater input is ignored. This mode allows the tube to be used as a very fast shutter suitable for imaging, for example.
- ***mode = 2:*** Gating is controlled from the gater input only.
- ***mode = 3:*** Gating is ON continuously.
- ***mode = 4:*** Gating is OFF continuously.

See also: SetGain SetGate

Example: This example shows how to set the gate mode to Fire Only:

SetGateMode(1)

SetGateStep (not PDA or CCD, superseded by SetGate)

Syntax: SetGateStep(*step*)

Description: Sets the amount by which the gate position is moved in time after each scan in a kinetic series. This function is only applicable to InstaSpec V systems. For a full description of the Gater refer to the User's Guide.

See also: run + full list of acquisition functions

Example: **SetGateDelay**(10e-8) :**rem** Set the delay to 100 nanosecond.

SetGateWidth(15e-9) :**rem** Set the width to 15 nanosecond.

SetGateStep(20e-9) :**rem** Set the step to 20 nanosecond

SetGateWidth (not PDA or CCD, superseded by SetGate)

Syntax: SetGateWidth(*width*)

Description: Sets the width of the gate in seconds. This function is only applicable to InstaSpec V systems. For a full description of the Gater refer to the User's Guide.

See also: run + full list of acquisition functions

Example: **SetGateDelay**(10e-8) :**rem** Set the delay to 100 nanosecond.

SetGateWidth(15e-9) :**rem** Set the width to 15 nanosecond.

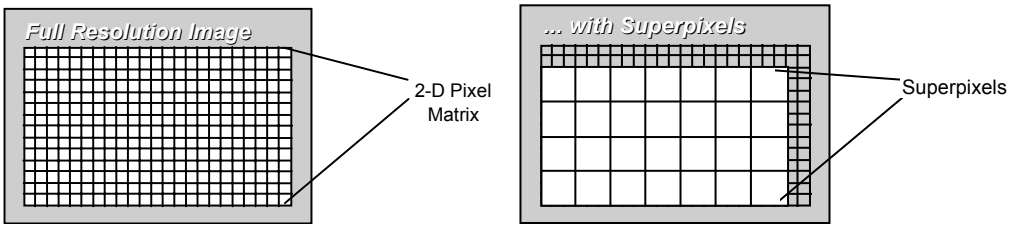
SetGateStep(20e-9) :**rem** Set the step to 20 nanosecond.

SetHBin (not PDA, superseded by SetImage)

Syntax: SetHBin(*NoPixels*)

Description: Changes the horizontal binning pattern associated with the Full Resolution Image readout mode. *NoPixels* is measured in CCD pixels and can range between 1 and 64.

The function, which may be used in conjunction with **SetVBin**, allows the user to configure the chip into a matrix of "superpixels" (each consisting of two or more individual pixels) from which to bin and read out charge.



For example, if you use **SetHBin** to set horizontal binning to 4 pixels, and **SetVBin** to set the vertical binning to 4 pixels, the CCD-chip is notionally divided into a matrix of superpixels, each of which (in this instance) measures 4 x 4 pixels and each of which provides a signal for readout.

To read out a matrix of superpixels, InstaSpec bins charge vertically into the shift register from several rows at a time (4 rows in this case, representing the height of your superpixels), and then bins charge horizontally from several columns of the shift register at a time (here 4 columns, representing the width of the superpixels). The resulting image is more coarsely defined than the default Full Resolution Image - but processing is faster, storage requirements are lower, and potentially the signal to noise ratio is improved.

See also: SetVBin

Example: This example will acquire an image by configuring a CCD-chip of dimensions 1024 x 256 into a 128 x 256 superpixel format. It does this by specifying a horizontal binning pattern of 8 pixels.

SetReadoutMode(4) **:rem** Full Resolution Image

SetHBin(8) **:rem** bin 8 pixels horizontally

Run()

SetImage (not PDA)

Syntax: 1) SetImage( )

2) SetImage(*HStart*, *HEnd*, *HBin*, *VStart*, *VEnd*, *VBin*)

3) SetImage(*HStart, HEnd, HBin, VStart, VEnd, VBin*)

4) SetImage(*HStart, HEnd, HBin, VStart, VEnd, VBin, XFlip, YFlip*)

Description: Sets the CCD binning mode to Image and configures the image parameters *HStart*, *HEnd*, *HBin*, *VStart*, *VEnd*, *VBin*.

The parameters *HStart*, *HEnd*, *VStart* and *VEnd* specify a sub-image region of the CCD sensor i.e.

HStart = horizontal start pixel of sub-image (left)

HEnd = horizontal end pixel of sub-image (right)

HBin = horizontal binning parameter

VStart = vertical start pixel of sub-image (bottom)

VEnd = vertical end pixel of sub-image (top)

VBin = vertical binning parameter

For a full description of Image acquisitions (and binning parameters), see the User's Guide.

- Option 1) reads out data from the full CCD sensor and uses a value of 1 for both the *HBin* and *VBin* parameters.
- Option 2) as Option 1 but also allows you to flip the image about the centre of the x-range (*XFlip* = 1) or the centre of the y-range (*YFlip* = 1).
- Option 3) allows you to specify a sub-image region of the CCD sensor to readout data from and also to specify values for both the *HBin* and *VBin* parameters.
- Option 4) as Option 3 but also allows you to flip the image about the centre of the x-range (*XFlip* = 1) or the centre of the y-range (*YFlip* = 1).The default setting for *XFlip* and *YFlip* is 0 which means the image will not be flipped.

Replaces: SetImage( ) renders the following commands obsolete:

SetHBin( ), SetVBin( ) and SetReadoutMode(4).

See also: SetFVB SetMultiTracks SetSingletrack

Example 1: This example programs the CCD sensor to acquire a single Full Resolution Image of exposure time 0.5 sec.:

SetImage()

SetSingleScan(0.5)

Run()

Example 2: This example programs the CCD sensor to acquire an image of dimensions 100x100 starting at 1,1 and with hbin=2, vbin=2, the exposure time is 0.5 sec.:

SetImage(1,101,2,1,101,2)

SetSingleScan(0.5)

Run()

SetKineticCycleTime (superseded by SetKinetics)

Syntax: SetKineticCycleTime(*time*)

Description: Sets the kinetic cycle time in seconds. This function will only take effect if the acquisition mode has been set to Kinetics and the trigger mode is Internal. The period entered is rounded up to the nearest permissible value. The function is not available if you are using External trigger mode, since an external trigger can occur at any time and with any or no periodicity.

See also: run + full list of acquisition functions

Example: **SetKineticCycleTime**(0.5) :**rem** Sets the kinetic
:rem cycle time to 500ms

SetKineticNumber (superseded by SetKinetics)

Syntax: SetKineticNumber(*count*)

Description: Sets the number of scans to be acquired to memory as members of a kinetic series. This function will only have effect if the acquisition mode has been set to Kinetics.

See also: run + full list of acquisition functions

Example: **SetKineticNumber**(5) :**rem** Sets to 5 the number of
:rem scans to be stored.

SetKinetics

Syntax: 1) SetKinetics(*expTime*, *numKins*, *KCT*)

2) SetKinetics(*expTime*, *numAccums*, *ACT*, *numKins*, *KCT*)

Description: Sets the CCD acquisition mode to Kinetics and specifies the kinetic parameters:

- **expTime:** The exposure time (in secs).
- **numAccums:** The number of accumulations to sum together to form one scan.
- **ACT:** The Accumulate Cycle Time (ACT secs) i.e. the delay or periodicity between successive accumulations. A value of 0 for the ACT forces InstaSpec to calculate the minimum value possible.
- numKins: The number of scans in the series.
- **KCT:** The Kinetic Cycle Time (KCT secs) i.e. the delay or periodicity between successive scans in the series. A value of 0 for the KCT forces InstaSpec to calculate the minimum value possible.

The Kinetic Series acquisition mode allows you to acquire a sequence of single (or accumulated) scans.

NOTES:

1. **Syntax option 1) decribes a simple series of single scans whilst 2) decribes a series of scans where each scan is the result of 1 or more accumulations (i.e. an accumulated Kinetic Series). See the User's Guide for more information. e.g. SetKinetics(2,1,0,10,20) is equivalent to SetKinetics(2,10,20) since each scan in the series is composed of only one accumulation.**
2. **The system will default to a minimum Exposure Time should you attempt to enter too low a value.**
3. **Each timing parameter (e.g ACT, KCT etc) may be automatically rounded up to the nearest rated value. The command ShowTimings( ) prints (to the Output Window) the actual timing sequences used by InstaSpec.**

SetKinetics (continued)

Replaces: SetKinetics( ) renders the following commands obsolete: SetExposureTime( ), SetAccumulateNumber ( ), SetAccumulateCycleTime( ), SetKineticNumber( ), SetKineticCycleTime( ) and SetAcquisitionMode(3).

See also: SetSingleScan SetAccumulate SetFastKinetics SetFrameTransfer

Example 1: When a HgNe lamp is first switched on it glows with a red color which changes to blue after it's warmed up (approximately 60 secs later). To monitor the lamp color changes we could acquire a series of spectra of the HgNe lamp during the warm-up period. To do this the CCD sensor should be configured to acquire a Kinetic Series of Full Vertically Binned (FVB) spectra. In this example, the exposure time per spectra is .1 secs, the number of spectra is 100 and the Kinetic Cycle Time is 0.5 secs:

SetKinetics(0.1,100,0.5)
SetFVB()
ShowTimings() :rem Prints actual timings
Run()

Example 2: This example is based on example 1 above but in this case each scan is now the result of 3 accumulations taken one immediately after another. (Accumulating scans improves the signal-to-noise ratio of the spectra) The CCD sensor is programmed to acquire a Kinetic Series of 100 FVB scans, each composed of 3 accumulations. The exposure time per accumulation is 0.1 secs, the ACT is 0 secs and the KCT is 0.5 secs.

SetKinetics(0.1,3,0,100,0.5)
SetFVB()
ShowTimings() :rem Prints actual timings
Run()

NOTE: The ACT is defined as 0 which tells InstaSpec to calculate the minimum possible value so that the accumulations are acquired one after the other. The function ShowTimings( ) prints to the output window the actual timing sequences used by InstaSpec.

SetMultitrackHeight (not PDA, superseded by SetMultiTracks)

Syntax: SetMultitrackHeight(*height*)

Description: Set the height of each track when using the Multi-Track readout mode. The *height* is in pixels. For a full description of each mode refer to the User's Guide

See also: run + full list of acquisition functions

Example: **SetMultitrackHeight**(4) :**rem** Split the CCD into 10 tracks
SetNumberTracks(10) :**rem** each of 4 pixels height.

SetMultiTracks (not PDA)

Syntax: SetMultiTracks(*numTracks*, *height*, *offset*)

Description: Sets the CCD binning mode to Multi-Tracks and configures the multi-tracks parameters.
numTracks = Number of tracks
height = Height of each track (in pixels)
offset = The offset is in pixels and can be positive or negative. A positive value moves the track height up the CCD (and likewise a negative value moves the track height down the CCD).
For a full description of the MultiTracks mode refer to the Users' Guide

Replaces: SetMultiTracks( ) renders the following commands obsolete:
SetNumberTracks( ), SetMultiTrackHeight( ), SetTrackOffset( ) and SetReadoutMode(1).

See also: SetFVB SetSingleTrack SetImage

Example: This example programs the CCD sensor to acquire a single multi-track scan of exposure time 0.5 sec. The number of tracks is 10, the height of each track is 12 pixels and the offset is 0.
SetMultiTracks(10,12,0)
SetSingleScan(0.5)
Run()

SetNumberTracks (not PDA, superseded by SetMultiTracks)

Syntax: SetNumberTracks(*number*)

Description: Set the number of tracks into which the CCD will be split when using the Multi-Track readout mode. For a full description of each mode refer to the User's Guide.

See also: run + full list of acquisition functions

Example: **SetMultitrackHeight**(4) :**rem** Split the CCD into 10 tracks
SetNumberTracks(10) :**rem** each of 4 pixels height.

SetReadoutMode (not PDA, superseded by...(see Description))

Syntax: SetReadoutMode(*mode*)

Description: Sets the readout mode to one of the following:
0 = Full Vertical Binning (superseded by SetFVB);
1 = Multi-Track (superseded by SetMultiTracks);
2 = Reserved;
3 = Single-Track (superseded by SetSingleTrack);
4 = Full Resolution Image (superseded by SetImage)
All other values are reserved. For full description of each mode refer to the User's Guide.

See also: run + full list of acquisition functions

Example: **SetReadoutMode**(1) :**rem** Set the readout mode to Multi-
SetNumberTracks(10) :**rem** Track with the CCD split into 10
SetMultitrackHeight(4):**rem** tracks, each 4 pixels in height.
SetTrackOffset(5) :**rem** The tracks are spread evenly over
:rem the CCD. The vertical offset
:rem equals 5 pixels.

SetReadoutTime

Syntax: SetReadoutTime(*time*)

Description: Sets the horizontal shift time which is effectively the readout time of each pixel. The *time* is in microseconds. The allowable values are:

- **1**
- **2**
- **16**
- **32**

NOTE: 1 and 2 are available only with 1 MHz systems.

See also: run + full list of acquisition functions

Example: **SetReadoutTime**(16) :**rem** Sets the readout time to 16
:rem microseconds.

SetShutter

Syntax: SetShutter(*mode*, *type*)

Description: Allows the user to control the use of a shutter.

mode:

0	Closed
1	Open
2	Auto

type:

0	Output TTL low signal to open shutter
1	Output TTL high signal to open shutter

See also: SetShutter TransferTime

Example: This example shows how to set the shutter mode to Closed and make it open when the TTL pulse becomes high:

SetShutter(0, 1)
SetShutterTransferTime(0.1) // time to open shutter

SetShutterTransferTime

Syntax: SetShutterTransferTime(time)

Description: Sets the time (in secs) to open or close a shutter.

If you are acquiring data other than Full Vertical Binning (FVB), you should employ a shutter to prevent “smearing” the image (here image is taken to mean either a full resolution image, multi-tracks or single-track image, etc.). Smearing occurs whenever the CCD sensor is *exposed to light* while the charge is shifted down the chip into the shift register, prior to reading into the computer. The proper use of a shutter will block off the CCD sensor from the light source, thus allowing the sensor to transfer the signal free from smearing.

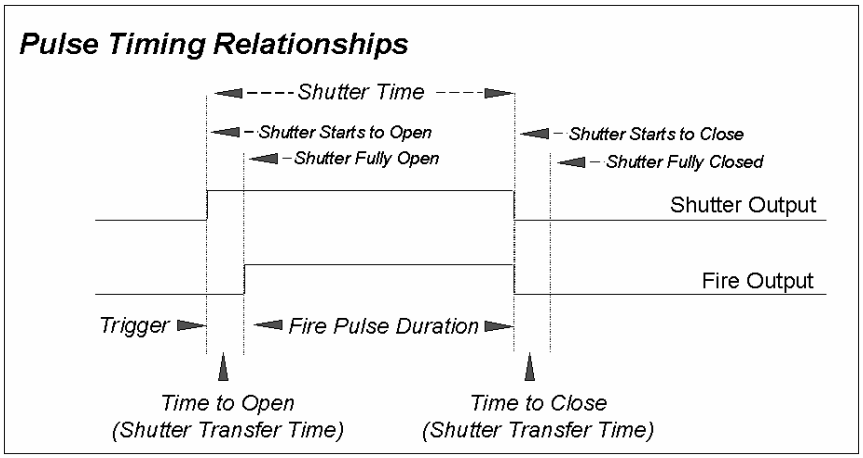
Example: There are several points to consider when using a shutter. Every shutter requires a finite amount of time before it is fully opened and an approximately similar amount of time before it is properly closed, generally known as the **Shutter Transfer Time** (STT). The STT, which for a relatively fast shutter can be ~20 ms, must be taken into account when you try to determine an optimum exposure time for an image. (The STT is normally supplied with the shutter manual.)

For example, a particular application may require that you take a full resolution image with an exposure time of 18 ms and use a shutter with an STT of 25ms. InstaSpec defines the **Shutter Time** as the period for which the shutter is activated - thus the shutter time should be the sum of these two quantities to ensure that the shutter is fully open and the sensor duly exposed for the desired exposure time, i.e.

Shutter time = Desired exposure time + STT
or
Shutter time = 18 + 25 = 43 ms

In situations where the user is using a pulsed light source the source can be pulsed (or triggered) after the shutter has fully opened, or in other words, after a period of time equal to the STT. This ensures that the sensor can be exposed to light the instant the shutter has fully opened and for a period equal to the desired exposure time, after which the shutter will begin to close. To prevent smearing the image, the sensor must wait until the shutter has fully closed (i.e. STT secs) before shifting the signal down the chip into the shift register.

InstaSpec™ allows the user to do all this in a very simple fashion. InstaSpec automatically generates a **Fire pulse** (via the FIRE socket of the Multiple I/O Accessory) which allows a pulsed light source to synchronise the instant the shutter is fully open, see figure on the next page.



The Fire Pulse goes high STT secs after the Shutter has been activated and is of duration: Fire Pulse period = Shutter time – STT. InstaSpec will wait a further STT secs after the shutter is deactivated to allow the shutter to close properly before shifting the charge down the CCD chip. This prevents the signal from smearing.

In the example above the Shutter Time = 43 ms and the STT = 25 ms. After a period of STT ms, the shutter has opened fully and the Fire pulse is generated of duration 43 – 25 = 18 ms. Thus the Fire pulse is effectively the desired exposure time.

SetShutterTransferTime(0.025)

SetExposureTime(0.043)

NOTES:

1. **Andor Basic refers to the Shutter Time command as SetExposureTime(), since although some data readout modes do require a shutter some others do not (namely FVB) and so the same command is used in both instances. To summarise: in the event of acquiring non-FVB scans, the command SetExposureTime() is meant to refer to the Shutter Time, the period for which the shutter is activated. In the event of acquiring FVB scans there is no need to use a shutter to prevent smearing, since the charge from each column of pixels is combined or 'binned' on to the shift register at the bottom of the chip, giving a single total charge per column.**
2. **The Shutter Time must always exceed the STT. If the STT exceeded the shutter time then InstaSpec would attempt to acquire a scan and begin shifting the signal down the CCD chip before the shutter has even fully opened. To avoid this situation InstaSpec always makes the shutter time exceed the STT by 1 ms: in other words, the minimum Fire Pulse is of duration 1 ms.**

SetSingleScan

Syntax: SetSingleScan(*expTime*)

Description: Sets the CCD acquisition mode to Single Scan and configures the exposure time. The exposure time is rounded up to the nearest valid value.

expTime = The exposure time (in secs).

NOTES:

The system will default to a minimum Exposure Time should you attempt to enter too low a value.

The Exposure Time may be automatically rounded up to the nearest rated value. The command ShowTimings( ) prints (to the Output Window) the actual timing sequences used by InstaSpec.

Replaces: SetSingleScan( ) renders the following commands obsolete:

SetExposureTime( ) and SetAcquisitionMode(1).

See also: SetAccumulate SetKinetics SetFastKinetics SetFrameTransfer

Example: This example programs the CCD sensor to acquire a single Full Vertically Binned (FVB) scan of exposure time 0.5 sec.:

SetFVB()

SetSingleScan(0.5)

Run()

SetSingleTrack (not PDA)

Syntax: SetSingleTrack(*row*, *height*)

Description: Sets the CCD binning mode to Single Track and configures the track parameters row and height.

row = The center row of the track to be binned out.

height = Height of the track (in pixels)

For a full description of SingleTrack acquisitions, see the Users' Guide.

Replaces: SetSingleTrack( ) renders the following commands obsolete:

SetCenterRow( ), SetSingleTrackHeight( ) and SetReadoutMode(3).

See also: SetFVB SetMultiTracks SetImage

Example: This example programs the CCD sensor to acquire one single-track acquisition of exposure time 0.5 sec. The track center is located at row 128 and the height of the track is 50 pixes:

SetSingleTrack(128,50)

SetSingleScan(0.5)

Run()

SetSingleTrackHeight (not PDA, superseded by SetSingleTrack)

Syntax: SetSingleTrackHeight(*height*)

Description: Set the height of the track when using the Single-Track readout mode. The *height* is in pixels. For a full description of each mode refer to the User's Guide.

See also: run + full list of acquisition functions

Example: **SetReadoutMode**(3) :**rem** Set the readout mode to

SetSingleTrackHeight(4) :**rem** Single-Track with a track height

:rem of 4 pixels

SetTemperature

Syntax: SetTemperature()

Description: Sets the temperature to which you require the head to cool. For the cooling process to begin you must also use the **cooler** function.

See also: **cooler**

Example: This example shows how to cool a CCD head to -10°C:

SetTemperature(-10)

cooler(1)

SetTrackOffset (not PDA, superseded by SetMultiTracks)

Syntax: SetTrackOffset(*offset*)

Description: Set the offset of each track when using the Multi-Track readout mode. The *offset* is in pixels and can be positive or negative. A positive value moves the track height up the CCD. For a full description of each mode refer to the User's Guide.

See also: run + full list of acquisition functions

Example: **SetNumberTracks**(10) :**rem** Split the CCD into 10

SetMultitrackHeight(4):**rem** tracks, each 4 pixels in height.

SetTrackOffset(5) :**rem** The tracks are spread evenly over

:rem the CCD. The vertical offset

:rem equals 5 pixels

SetTriggerMode

Syntax: SetTriggerMode(*mode*)

Description: Sets the trigger mode to either Internal or External, i.e.:

0 = Internal

1 = External

All other values are reserved. For a full description of each mode refer to the User's Guide.

Example: **SetTriggerMode**(0) :**rem** Set the trigger mode to

:rem Internal. InstaSpec will decide when

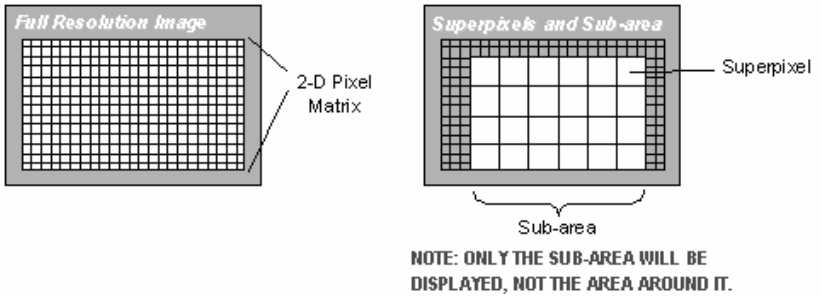
:rem to start the acquisition.

SetVBin (Not PDA, superseded by SetImage)

Syntax: SetVBin(*NoPixels*)

Description: Changes the vertical binning pattern associated with the Full Resolution Image readout mode. *NoPixels* is measured in CCD pixels and can have any value between 1 and the height of the CCD chip in pixels.

The function, which may be used in conjunction with **SetHBin**, allows the user to configure the chip into a matrix of "superpixels" (each consisting of two or more individual pixels) from which to bin and read out charge.



For example, if you use **SetHBin** to set horizontal binning to 4 pixels, and **SetVBin** to set the vertical binning to 4 pixels, the CCD-chip is notionally divided into a matrix of superpixels, each of which (in this instance) measures 4 x 4 pixels and each of which provides a signal for readout.

To read out a matrix of superpixels, InstaSpec bins charge vertically into the shift register from several rows at a time (4 rows in this case, representing the height of your superpixels), and then bins charge horizontally from several columns of the shift register at a time (here 4 columns, representing the width of the superpixels). The resulting image is more coarsely defined than the default Full Resolution Image - but processing is faster, storage requirements are lower, and potentially the signal to noise ratio is improved.

See also: SetHBin

Example: This example will acquire an image by configuring a CCD-chip of dimensions 1024 x 256 into a 1024 x 64 superpixel format. It does this by specifying a vertical binning pattern of 4 pixels.

SetReadoutMode(4) **:rem** Full Resolution Image
SetVBin(4) **:rem** vbin = 4
Run()

SetXLabel

Syntax: SetXLabel(*#DataSet, label ,unit)*
SetXLabel(*#DataSet, label)*

Description: Allows you to (re)set the x-axis label (and unit) of the data set *#DataSet.*

If the *unit* parameter is omitted, InstaSpec will assume a default value of 1, i.e. the first unit corresponding to the particular x-axis label.

X-Axis Label	Units
1 = Pixel	1 = pixels
2 = Wavelength	1 = nm; 2 = um; 3 = cm-1, 4 = eV
3 = Raman Shift	1 = cm-1
4 = Position	1 = um; 2 = mm; 3 cm; 4 = ui(nches), 5 in(ches)
5 = Time	1 = ms; 2 = secs

See also: SetDataLabel

Example: This example shows how to set the x-axis label of the data set #6 to Wavelength and the unit to cm-1.

SetXLabel(#6, 2, 3)

ShowTimings

Syntax: ShowTimings()

Description: If acquisition timings have been specified by the program, these settings cannot always be used exactly entered. **ShowTimings** shows how the timings have been interpreted for the current settings.

See also: run SetExposureTime SetAccumulateCycleTime SetKineticCycleTime

Example: **ShowTimings**()

Result: Exposure time =
Delay time =
Shutter time =
Accumulate cycle time =
Kinetic cycle time =

---

shutdown

Syntax: shutdown()

Description: Shuts down Windows on a remote computer connected via Ethernet to prepare the computer for being switched off. This should be used before turning off a remote computer when no keyboard or mouse is available for closing down Windows.

See also: connect disconnect

Example: This example program connects to a remote machine with an IP address of 111.222.333.444 and shuts down Windows:

connect("111.222.333.444")
shutdown()
disconnect()

---

sin

Syntax: sin(*x*)

Description: Calculates the sine of the variable *x*, which should be entered in radians, and returns the answer in the range -1.0 to 1.0

See also: acos asin atan cos tan

Example: x=0.5
y=**sin**(x) :**rem** Calculate the sin of x.
print(y)

Result: 0.48

---

smooth1

Syntax: smooth1(*#data*)

Description: Performs a 5 point Savitsky-Golay smoothing algorithm on *#data*. As in the example, an assignment to a data set must accompany this function.

See also: smooth2 smooth3 fft

Example: #1_sig=**smooth1**(#1_sig)
rem Perform 5 point smoothing algorithm on the signal data of
rem #1.

---

smooth2

Syntax: smooth2(*#data*)

Description: Performs a 7 point Savitsky-Golay smoothing algorithm on *#data*. As in the example, an assignment to a data set must accompany this function.

See also: smooth1 smooth3 fft

Example: #1_sig=**smooth2**(#1_sig)
rem Perform 7 point smoothing algorithm on the signal data of
rem #1.

---

smooth3

Syntax: smooth3(*#data*)

Description: Performs a 9 point Savitsky-Golay smoothing algorithm on *#data*. As in the example, an assignment to a data set must accompany this function.

See also: smooth1 smooth2 fft

Example: #1_sig=**smooth3**(#1_sig)
rem Perform 9 point smoothing algorithm on the signal data of
rem #1.

---

sqrt

Syntax: sqrt(*x*)
sqrt(*#data*)

Description: Returns the square root of the variable *x* or the square root of *#data*. As in the example, an assignment to data must accompany this function when it is applied to a data set.

See also: alog exp ln log

Example: y=**sqrt**(x) :**rem** take the square root of x
#1_sig=**sqrt**(#1_sig):**rem** take the square root of signal data
:**rem** in data set #1

---

stopbits

Syntax: stopbits(*comport,number_of_stopbits*)

Description: Sets the number of stop bits for the specified comport. The default value is 1. Recognized values are:

1,

1.5

2

Com ports 1, 2, 3 and 4 are supported.

See also: baud comread comwrite handshake

Example: **stopbits**(1,1.5)

---

str$

Syntax: str$(*x*)

Description: Converts a floating point number to a text string

See also: len val

Example: a$=**str$**(#1[320])

rem Convert the value of pixel 320 of 1 into a string.

---

tan

Syntax: tan(*x*)

Description: Calculates the tangent of the angle *x*, which should be entered in radians, and returns the answer in the range -inf to +inf.

See also: acos asin atan cos sin

Example:
```
x=0.5
y=tan(x)    :rem calculate the tan of x
print(y)
```

Result: 0.55

---

terminator

Syntax: terminator(*value*)

Description: Specifies the character which has been used to terminate text being read at the comports. This character has a default value of 0x0d, which corresponds to a carriage return. Equally valid *value*s might be 0x20 for a space, or 0x2c for a comma, or 0x0a for a line feed.

See also: baud comread comwrite handshake ignore

Example:
```
rem receive a numeric value on com2
baud(2,9600)    :rem setup com to 9600 baud
terminator(0x2c) :rem set terminator to be a comma
comread(2,a)    :rem read from com2
print(a)
```

---

time

Syntax: time()

Description: Returns a floating point number which is incremented every second. This function is useful for determining the time elapsed between two events

See also: date$ time$

Example:
```
time1=time()
run
time2=time()
print("Time for acquisition was ";time2-time1;" seconds.")
```

---

time$

Syntax: time$()

Description: Returns the time as a text string. This function returns the time contained in the computer, and this is correct while the computer is correct. The time is printed in twenty-four hour format, with 2 digits each for the hours: minutes: seconds.

See also: date$ time

Example: **print**("The time now is ";**time$**())

TopWindow

Syntax: TopWindow(*#dataset*)

Description: Makes the data window *#dataset* the top window. This is used to bring data windows to the front of the screen so that you can view results without having to click on the window.

See also: ActiveOverlay ActiveTab ChangeDisplay CloseWindow MaximizeWindow MinimizeWindow MoveWindow

Example: **rem** Acquire single scan, fully vertically binned.
rem Load a file. Bring acquisition window to front.

```
SetDataType(1)       :rem Counts.
SetAcquisitionType(0) :rem Signal.
SetAcquisitionMode(1) :rem Single Scan.
SetReadoutMode(0)    :rem Fully Vertically Binned.
run()                :rem Take Acquisition.
load(#1,"data.sif")  :rem Load Data.
TopWindow(#0)        :rem Bring acquisition
                     :rem window to front
```

---

update

Syntax: update(*x)*
update()

Description: When using Andor Basic to make changes to data, the slowest part is often the time taken to redraw the screen after each pixel has been changed. It is often preferable to delay the redrawing until all pixels have been changed and perform an update just once. If ***x*** is set equal to 0 then updating is only carried out by using the **update**() command as in the example. The default action with Andor Basic is to update after each pixel has been modified. This is equivalent to **update**(1).

Example: **rem** without using the update flag and command
rem redrawing may be slow

```
update(0)          :rem will not update until calcs finished
#2=#0              :rem copy Acquisition to a new data set
pixel=300          :rem set up pixel counter
while pixel<400
 #2[pixel]=0       :rem zero this pixel
                   :rem because updating is on by default
                   :rem the data set will be redrawn here
 pixel=pixel+1     :rem increment counter
wend
update(1)          :rem update now that calcs complete
```

val

Syntax: x = val(*a$*)

Description: Converts a valid text string into a floating point number

See also: str$ len

Example:
```
a$="342.68"
a=val(a$)
print(a/2)
```

Result: 171.34

---

write

Syntax: write(*filename*, *variable_list*)

Description: This function appends data or text to an already existing file. If the file does not already exist, a new file, called *filename*, is created. This can be useful for generating reports. *variable_list* is a list of numeric or string variables, separated by commas or semicolons. If the variables are separated by commas, then the output text will be separated by tab spaces. If, however, they are separated by semicolons, the output text will be continuous. The placing of a semicolon at the end of the *variable_list* will inhibit the normal action of forcing a new line. This function returns an error number of zero on success, and a negative number on failure.

See also: read save close kill

Example 1:
```
write("report.dat", "This is a line")
write("report.dat", "This is the next line")
write("report.dat", "Results are";)
write("report.dat", (X;Y;Z))
write("LPT1:", "I am printing out my data")
rem alternative to lprint
write("COM1:", "I am sending out data via the RS-232 port")
```

Example 2:
```
input("Enter filename", f$)
input("Enter string to be saved to file", t$)
a=write(f$,t$)
if a<0 then
  print("Error on write")
endif
```

---

xcal

Syntax: xcal(*#data*, *x*)

Description: Returns the calibrated value (X axis) of a pixel *x* in *#data*.

See also: copyxcal xpix

Example: c=**xcal**(#1_sig,512):**rem** find calibrated value of pixel 512

---

xpix

Syntax: xpix(*#data*, *x)*

Description: Converts calibration units to pixels. Returns the pixel value where *x* is the value on the X-axis in the current calibration units for *#data*.

See also: xcal

Example: **rem** calculate area of signal data in data set #1
rem from 300nm to 500nm
a=**area**(#1_sig,**xpix**(#1_sig,300),**xpix**(#1_sig,500))

rem print the data value at wavelength 445nm of
rem signal data in data set #12
print(#12_sig[**xpix**(#12_sig,445)])

---

zero

Syntax: zero(*#data*)

Description: Set all elements of *#data* to zero

See also: copyxcal xcal

Example: **zero**(#5_sig) :**rem** set all elements of the signal data
:**rem** in data set #5 to zero

SECTION 3 – SAMPLE PROGRAMS

PROGRAM 1 – CLEARING DATA SETS

```
rem         Shows how to set up simple loops to process
rem         sample data sets.

rem         Filenames and other character variables have
rem         the string character ($) appended at the end.

rem         These pathnames are user defined.

NoFiles = 3
file1$ = "c:\instaspc\images\calib.sif"
file2$ = "c:\instaspc\images\vero1.sif"
file3$ = "c:\instaspc\images\vero2.sif"

load(#1,file1$)          rem Loads data into window 1.
load(#2,file2$)
load(#3,file3$)

test = key("Press z to zero the data sets...")
if test == 'z' then
  counter = 1
        while counter <= NoFiles
                zero(#counter)
                counter = counter+1
        wend
endif
```

PROGRAM 2 - TAKING SEVERAL SCANS

```
rem     Takes 10 scans and places them in 10 successive
rem     data sets.
rem     It also demonstrates the use of programming the
rem     detector using Andor Basic.
rem     Place detector in single scan mode

SetAcquisitionMode(1)

rem     Program detector to generate "simple" spectra in
rem     the form of Full Vertical Binning.

SetReadoutMode(0)


counter = 1
while counter < 11
        run()                   :rem acquires a scan
        #counter = #0   :rem save the scan in the
                                :rem window counter
        counter = counter + 1
wend
```

PROGRAM 3 – WORKING WTH KINETICS

```
rem     A HgNe lamp takes about 55 secs to warm
rem     up properly, in which time the light changes
rem     from red to blue.  This example programs the
rem  InstaSpec detector to record the color changes
rem     as it reaches its correct operating temperature.
rem     We do this by recording a kinetic series of (e.g.)
rem     100 scans acquired over 55 seconds.

cls

SetAcquisitionMode(3)                :rem   Kinetics
SetReadoutMode(3)                    :rem   Single Track
SetExposureTime(.03)          :rem   NB must exceed the Transfer
                                     :rem   Shutter Time i.e. the time the
                                     :rem   shutter takes to open
SetTriggerMode(0)                    :rem   Internal triggering
SetKineticCycleTime(0.55)     :rem   100 scans * 0.55 = 55 secs
SetKineticNumber(100)                :rem   100 scans in Kinetic series
SetDataType(2)                       :rem   Counts(Background corrected)
SetAcquisitionType(1)         :rem   Take background first
run()
SetAcquisitionType(0)         :rem   Now acquire the signal
run()
```

PROGRAM 4 – MONITORING GPIB STATUS

```
rem Establish connection with an instrument at GPIB address 15.
rem Query the status of the instrument and monitor all GPIB
rem errors.
rem Ensure that the gpib.com device driver is installed from
rem config.sys.

b=ibfind("gpib0")  :rem find interface board
ibconfig(b,3,11)   :rem change timeout to 1 sec
d=ibfind("dev15")  :rem find instrument
gosub .err
ibwrt(d, "cl")     :rem clear instrument
gosub .err
ibwrt(d,"is")      :rem send status command
gosub .err
ibrd(d,a$)         :rem read reply
gosub .err
print("Status returned = ";a$)
end

.err                    :rem print error if required
  if ibsta > 32767 then  :rem test bit 7
    print("Error code ";iberr)
  endif
return
```