HP VEE-Engine and HP VEE-Test Reference

HP Part No. E2100-90013
Printed in USA November 1992

Edition 2

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. HP MAKES NO WARRANTIES OF ANY KIND WITH REGARD TO THIS DOCUMENT, WHETHER EXPRESS OR IMPLIED. HP SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DoD agencies, Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Use of this manual and magnetic media supplied for this product are restricted. Additional copies of the software can be made for security and backup purposes only. Resale of the software in its present form or with alterations is expressly prohibited.

Printing History

Edition 1 - April 1991
Edition 2 - November 1992

Preface

This manual is written for engineers and scientists who have minimal programming experience. It assumes some knowledge of UNIX™, (UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and in other countries), as well as an entry-level knowledge of how to use HP VEE to build models.

About This Manual. This manual contains detailed reference information regarding all the features included in HP VEE. It contains information for both the HP VEE-Engine and the HP VEE-Test products. Any differences between the use of the products are noted in the text.

This manual is structured as follows:

Content of this Manual

Chapter or Appendix	Summary
1: "Menu Overview"	An overview of all the menus in HP VEE.
2: "General Reference"	A reference of the features in all the HP VEE menus, except for those features contained in the `Math` and `AdvMath` menus.
3: "Formula (Math and AdvMath) Reference"	A reference of the features in the `Math` and `AdvMath` menus.
A: "Data Type Conversions"	Reference information about the data type conversions which occur on input terminals in HP VEE.
B: "ASCII Table"	Reference tables of ASCII 7-bit codes.
Glossary	
Index	

For more information on interacting with and using HP VEE, including techniques you can use to build and modify HP VEE models, please see *Getting Started with HP VEE-Engine and HP VEE-Test* and *Using HP VEE-Engine and HP VEE-Test*.

Conventions Used in this Manual

This manual uses the following typographical conventions:

Example	Represents
Installing HP VEE	Italicized words are used for book titles and for emphasis.
`File`	Computer font represents text you will see on the screen, including menu names, features, or text you have to enter.
`cat` *filename*	In this context, the word in computer font represents text you type exactly as shown, and the italicized word represents an argument that you must replace with an actual value.
`File` ⟹ `Open`	Features separated with the arrow indicate the order of selection from a menu.
`Zoom Out | In 2x | In 5x`	Choices in computer font, separated with a bar (|), indicates that you should choose one of the options.
Return	The keycap font graphically represents a key on the workstation's keyboard.
Press CTRL-O	Dash-separated keys represent a combination of keys on the workstation's keyboard that you should press at the same time.

Contents

Figures

Tables

1

Menu Overview

This chapter provides a summary of all the menus in HP VEE.

HP VEE has ten menus that provide you with quick access to its many features. Selecting a feature will result in either some action (such as opening a file) or some object (a control, display, or function). The following sections summarize the HP VEE menus in the order that they appear, from left to right, on the menu bar in the HP VEE window.

File Menu

The `File` menu features allow you to perform actions that affect the entire model.

Figure 1-1. The `File` Menu

Edit Menu

The `Edit` menu features allow you to perform actions that affect multiple objects and lines, as well as debug the model.

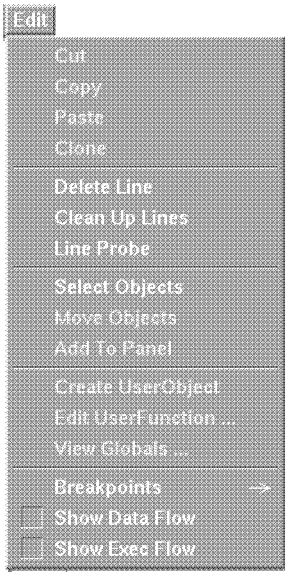

Figure 1-2. The `Edit` Menu

Flow Menu

The `Flow` menu features are objects that affect the way the model runs.

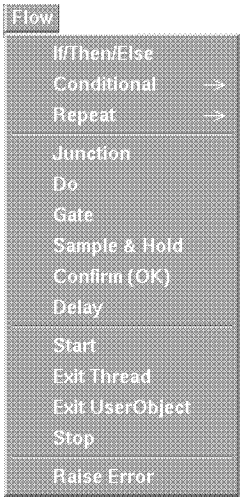

Figure 1-3. The `Flow` Menu

Device Menu

The `Device` menu features are objects that allow you to generate and process data.

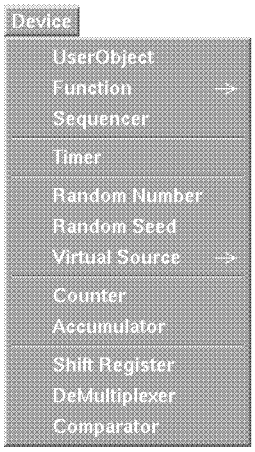

Figure 1-4. The `Device` Menu

I/O Menu

The `I/O` (Input/Output) menu features are objects that allow you to get data from files, programs, and instruments (HP VEE-Test only).

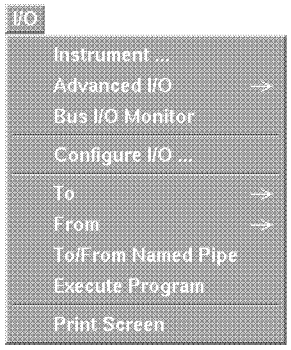

Figure 1-5. The `I/O` Menu

Data Menu

The `Data` menu features are objects that allow you to specify data of specific types and shapes.

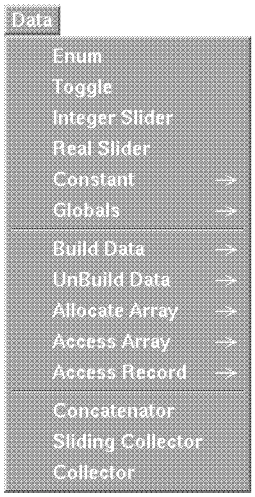

Figure 1-6. The `Data` Menu

Math Menu

The `Math` menu features are objects that specify common mathematical functions.

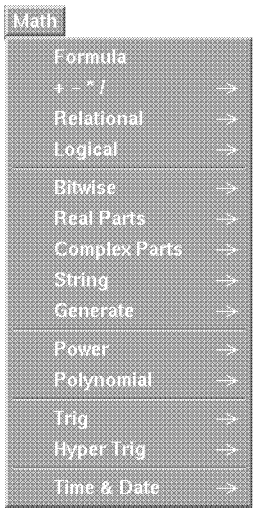

Figure 1-7. The `Math` Menu

AdvMath Menu

The `AdvMath` (Advanced Math) menu features are objects that specify more specialized mathematical functions for engineering.

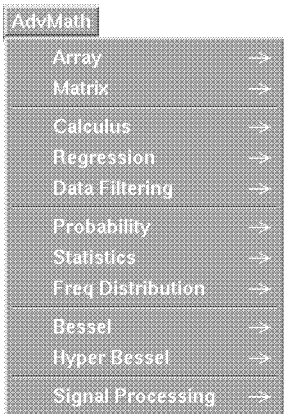

Figure 1-8. The `AdvMath` Menu

Display Menu

The `Display` menu features are objects that allow you to view data in alphanumeric or graphical forms.

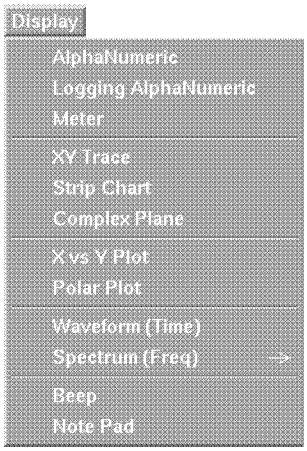

Figure 1-9. The `Display` Menu

Help Menu

The `Help` menu features allow you to find information about HP VEE.

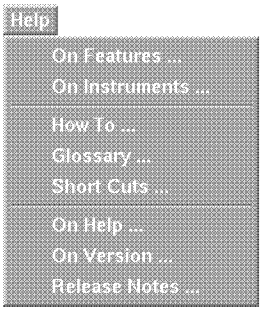

Figure 1-10. The `Help` Menu

2

General Reference

This chapter contains detailed reference information about the features in all the HP VEE menus, except for those features contained in the `Math` and `AdvMath` menus. (For information about features in `Math` and `AdvMath`, please refer to chapter 3.) This chapter is ordered alphabetically by feature name to help you find information about each feature quickly and easily.

Access Array

A menu item.

Use

Use `Access Array` to access the following array modification objects:

- `Set Values`
- `Get Values`
- `Set Mappings`
- `Get Mappings`

Location

`Data` ⟹ `Access Array` ⟹

Notes

These objects are usually used after the `Allocate Array` objects to modify the elements or add mappings to a newly created array.

See Also

`Allocate Array`, `Get Mappings`, `Set Mappings`, `Get Values`, and `Set Values`.

Access Record

A menu item.

Use

Use `Access Record` to access the following record modification objects:

- `Merge Record`
- `SubRecord`
- `Set Field`
- `Get Field`

Location

`Data` ⟹ `Access Record` ⟹

Notes

These objects are usually used after the `Build Record` object to modify the elements, merge records or access portions of the record.

See Also

`Build Record`, `Merge Record`, `SubRecord`, `Set Field`, and `Get Field`.

Accumulator

An object that displays and outputs a running sum of its input values.

Use

Use `Accumulator` to keep a running total.

Location

`Device` ⟹ `Accumulator`

Object Menu

- `Clear` - Clears the contents of the `Accumulator`.
- `Clear at PreRun` - Clears the contents of the `Accumulator` at PreRun. Default is on (checked).
- `Clear at Activate` - Clears the contents of the `Accumulator` at Activate. Default is on (checked).

Note

The type of the accumulated output data is the same as the first data input value since the last clear. Thus, if the `Accumulator` is input a Real and an Int32, it outputs a Real.

If the `Accumulator` data input is activated with a Real then later is activated by a Complex, the current accumulated data is Real; the object returns an error because it cannot convert the Complex data to Real.

See Also

`Counter`.

Activate Breakpoints

When checked, enables all set breakpoints. When not checked, allows the model to run normally.

Use

Use `Activate Breakpoints` to toggle between debugging and normal model running. When `Activate Breakpoints` is on, the menu selection is checked. When a breakpoint is encountered, the models stops running.

Location

`Edit` ⟹ `Breakpoints` ⟹ `Activate Breakpoints`

Notes

The black border surrounding the objects with set breakpoints is still present, even when `Activate Breakpoints` is off. The object menu shows `Set Breakpoint` as being checked regardless of the state of `Activate Breakpoints`.

When a breakpoint is set on an object in a `UserObject` (that is displayed as an icon and `Show Panel on Exec` is off), `Step` ignores the breakpoint.

See Also

`Breakpoint (Object Menu)`, `Clear All Breakpoints`, `Clear Breakpoint`, and `Set Breakpoint`.

Add Control Input (Object Menu)

Adds an asynchronous control input terminal to the object.

Use

Use `Add Control Input` to add asynchronous inputs, such as `Reset` or `Clear`, to objects.

After selecting `Add Control Input`, choose the control to add from a dialog box. The terminal for a control input is a different color than for a data input.

Location

On each object menu ⟹ `Terminals` ⟹ `Add Control Input`

Notes

Control inputs are not available on all objects.

Activation of a control input pin *does not* cause the object to operate, it only forces that particular action to happen (for example, `Clear`).

Often the actions performed by a control input may also be done from a selection on the object menu or by a button on the open view.

See Also

`Add Data Output`, `Add XEQ Input`, `Delete Input`, `Object Menu`, and `Terminals`.

Add Data Input (Object Menu)

Adds a data input terminal to the object.

Use

Use `Add Data Input` to increase the number of data input lines the object processes. On some objects additional data inputs are automatically named A, B, C, and so on. You can change the names of most terminals to names that are more meaningful to you.

Although you cannot change the data type or shape of most input terminals, you can change these constraints on some objects, such as `If/Then`, `Formula`, `Shift Register`, and objects under the `I/O` menu.

Generally, you should not change the data type or shape of an input terminal; if a constraint is changed from `Any`, the input data must be converted to the terminal constraint before the object processes the data. If the conversion can't take place, the object returns an error.

Location

On each object menu ⟹ `Terminals` ⟹ `Add Data Input`

Notes

`Add Data Input` is only available from objects that permit additional data inputs. `Add Data Input` is not available from preconfigured objects such as most `Math` and `AdvMath` objects and all `Conditional` objects.

Short Cuts

You can quickly add a data input terminal by placing the cursor over the input terminal display area and then pressing CTRL A. Each press of CTRL A adds an additional data input terminal.

See Also

Add Data Output, Add Control Input, Add XEQ Input, Conditional, Delete Input, Math, Output Terminals, Object Menu, Show Terminals, and Terminals.

Add Data Output (Object Menu)

Adds a data output terminal to the object.

Use

Use `Add Data Output` to increase the number of data output lines. On some objects, additional data inputs are automatically named X, Y, Z, W, V and so on; you can change these data output names to names that are more meaningful to you.

The output terminal container information (data type and shape) is not displayed in the terminal information dialog box until after the model is run.

Location

On each object menu ⟹ `Terminals` ⟹ `Add Data Output`

Notes

`Add Data Output` is only available from objects that permit additional data outputs. `Add Data Output` is not available from preconfigured objects such as most `Math` and `AdvMath` objects and all `Conditional` objects.

Short Cuts

You can quickly add a data output terminal by placing the cursor over the output terminal display area and then pressing CTRL A. Each press of CTRL A adds an additional data output terminal.

See Also

`Add Data Input`, `Add Error Output`, `Delete Output`, `Instrument`, `Object Menu`, and `Terminals`.

"Formula Reference" chapter.

Add Error Output (Object Menu)

Adds an error output terminal to the object.

Use

Use `Add Error Output` to allow your model to trap and process HP VEE errors. If an `Error` pin is on an object and the object encounters an error, no error dialog box is displayed, the error number is output to the `Error` pin.

If an error is trapped, the error output pin and the sequence output pin are activated; no data is output on the data output pins.

The error output pin outputs a Scalar Int32 container with the value of the error number that was generated.

Location

On the object menu ⟹ `Terminals` ⟹ `Add Error Output`

Notes

`Add Error Output` is available from almost all objects. Some objects, such as `Bus Monitor` and `Note Pad`, never return an error, therefore, they do not allow the addition of an error output pin.

An error output terminal outputs the error number of the HP VEE error. It does not output the meaning of the number or any recovery information. Error information is available under the `Error Code` topic under `Help` ⟹ `How To`

An `Escape` object in a `UserObject` can provide the error number that is output by the `Error` pin.

The error pin on a `UserObject` traps any errors for any object in the `UserObject`.

See Also

`Escape`, `Object Menu`, `Terminals`, and `UserObject`.

Add To Panel

Puts a copy of selected objects on the panel view of the model or a `UserObject`.

Use

Use `Add To Panel` to place a set of selected objects on the panel view to create a user interface to your model. For example, the panel view could contain only the input and display objects for a model; the internal processes of the model are not displayed.

When a panel view exists and the model is not secured, there are two buttons on the left side of the HP VEE or `UserObject`'s title bar: `Detail` and `Panel`. Press these buttons to see the desired view.

If you select `Add To Panel` and a panel view does not exist, it is created.

Location

`Edit` ⟹ `Add To Panel` or
`UserObject` object menu `Edit` ⟹ `Add To Panel`

Notes

The `Add To Panel` feature is not available if no objects are selected.

The panel view of a `UserObject` may be added to the main panel view or the panel view of a `UserObject` that is a surrounding context.

See Also

`Create UserObject`, `Secure`, and `Select Objects`.

Add XEQ Input (Object Menu)

Adds an `XEQ` input pin to the object.

Use

Use `Add XEQ Input` to add an XEQ pin to the object. When this pin is activated, the object operates using the data that is currently on its inputs (even if some inputs are nil or contain old data).

The terminal for XEQ is a different color than that of a data input.

Location

On each object menu $\Longrightarrow$ `Terminals` $\Longrightarrow$ `Add XEQ Input`.

Notes

`Add XEQ Input` is only available from `Confirm (OK)` and `UserObject` only.

`Set Values` and `Collector` already have XEQ inputs.

See Also

`Add Control Input`, `Add Data Input`, `Add Data Output`, `Collector`, `Delete Input`, `Object Menu`, `OK`, `Set Values`, `Terminals` and `UserObject`.

Advanced HP-IB

This menu item has been replaced with `Advanced I/O`.

Advanced I/O

A menu item. This feature is available in HP VEE-Test only.

Use

Use `Advanced I/O` to access the following objects for low-level control of the HP-IB and VXI interfaces (such as bus commands, polling, and service requests):

- `Interface Operations`
- `Device Event`
- `Interface Event`

Location

`I/O` ⟹ `Advanced I/O` ⟹

See Also

`Bus I/O Monitor`, `Device Event`, `Interface Event`, and `Interface Operations`.

Allocate Array

A menu item.

Use

Use `Allocate Array` to access the following array building and initialization objects:

- `Alloc Text`
- `Alloc Integer`
- `Alloc Real`
- `Alloc Coord`
- `Alloc Complex`
- `Alloc PComplex`

Location

`Data` $\Longrightarrow$ `Allocate Array` $\Longrightarrow$

Notes

Use `Set Values` to modify arrays already built.

See Also

`Alloc Coord`, `Alloc Complex`, `Alloc Integer`, `Alloc PComplex`, `Alloc Text`, `Alloc Real`, `Get Values`, `Set Mappings`, and `Set Values`.

Alloc Complex

An object that creates a Complex array.

Use

Use `Alloc Complex` to create an array of Complex elements.

Location

`Data` ⟹ `Allocate Array` ⟹ `Complex`

Open View Parameters

- `Num Dims` - The number of dimensions in the output array. `Num Dimensions` must be an integer. Minimum is 1. Maximum is 10. Default is 1.
- `Init Value` - The value to place in each array element. Default is (0, 0).
- `Size` - The number of elements in the dimension. There is one field per dimension. Default is 10.

`Init Value` and `Size` may be set on the object or from data input pins.

Notes

To change the values of individual elements, use `Set Values`.

If the `Num Dims` value is changed, the view automatically creates or deletes fields on the open view so that the number of dimension size fields matches the value of the `Num Dims` value.

The output array is not mapped. Use `Set Mappings` to map the dimensions of the array.

See Also

`Complex Constant`, `Get Values`, `Set Mappings`, and `Set Values`.

Alloc Coord

An object that creates a Coord array.

Use

Use `Alloc Coord` to create an array of Coord elements.

Location

`Data` ⟹ `Allocate Array` ⟹ `Coord`

Open View Parameters

- `Num Dims` - The number of dimensions in the output array. `Num Dimensions` can only be `1` since Coord arrays of greater than `1` dimension are not allowed in HP VEE.
- `Num Coord Dims` - The number of values in each Coord.
- `Init Value` - The value to place in each array element. Default is (0, 0), (0,0,0), or (0,0,0,0).
- `Size` - The number of elements in the dimension. Default is 10.

`Init Value` and `Size` may be set on the object or from data input pins.

Notes

To change the values of individual elements, use `Set Values`.

See Also

`Coord`, `Get Values`, and `Set Values`.

Alloc Integer

An object that creates an integer array.

Use

Use `Alloc Integer` to create an array of Integer elements.

Location

`Data` ⟹ `Allocate Array` ⟹ `Integer`

Open View Parameters

- `Num Dims` - The number of dimensions in the output array. `Num Dimensions` must be an integer number. Minimum is 1. Maximum is 10. Default is 1.
- `Init Value` | `Lin Ramp` | `Log Ramp`
  - `Init Value` - The values to place in each array element. This value is interpreted as an integer. Default is 0.
  - `Lin Ramp` - A linear ramp of integer values that are as equally spaced as possible between the ramp initial value (inclusive) and the ramp ending value (inclusive). This option is only available if `Num Dims` = 1.
  - `Log Ramp` - A ramp of integer values that are logarithmically spaced as possible between the ramp initial value (inclusive) and the ramp ending value (inclusive). This option is only available if `Num Dims` = 1.
- `Size` - The number of elements in the dimension. There is one field per dimension. Default is 10.

`Init Value` and `Size` may be set on the object or from data input pins.

Notes

To change the values of individual elements, use `Set Values`.

If the `Num Dims` value is changed, the view automatically creates or deletes fields on the open view so that the number of dimension size fields matches the value of the `Num Dims` value.

The output array is not mapped. Use `Set Mappings` to map the dimensions of the array.

See Also

`Integer`, `Get Values`, `Set Mappings`, and `Set Values`.

Alloc PComplex

An object that creates a PComplex array.

Use

Use `Alloc PComplex` to create an array of PComplex elements.

Location

`Data` ⟹ `Allocate Array` ⟹ `PComplex`

Open View Parameters

- `Num Dims` - The number of dimensions in the output array. `Num Dimensions` must be an integer number. Minimum is 1. Maximum is 10. Default is 1.
- `Init Value` - The value to place in each array element. The initial value's phase is normalized to fall between pi and -pi radians; the number's magnitude is converted to a positive if it was negative and the phase angle was shifted 180 degrees. Default is (0, @0).
- `Size` - The number of elements in the dimension. There is one field per dimension. Default is 10.

`Init Value` and `Size` may be set on the object or from data input pins.

Notes

`Set Values` changes the values of individual elements.

If the `Num Dims` value is changed, the view automatically creates or deletes fields on the open view so that the number of dimension-sized fields matches the `Num Dims` value.

The output array is not mapped. Use `Set Mappings` to map the dimensions of the array.

See Also

`Get Values`, `PComplex`, `Set Mappings`, and `Set Values`.

Alloc Real

An object that creates a real array.

Use

Use `Alloc Real` to create an array of Real elements.

Location

`Data` ⟹ `Allocate Array` ⟹ `Real`

Open View Parameters

- `Num Dims` - The number of dimensions in the output array. `Num Dimensions` must be an integer number. Minimum is 1. Maximum is 10. Default is 1.
- `Init Value` | `Lin Ramp` | `Log Ramp`
  - `Init Value` - The value to place in each array element. Default is 0.
  - `Lin Ramp` - A linear ramp of real values that are as equally spaced as possible between the ramp initial value (inclusive) and the ramp ending value (inclusive). This option is only available if `Num Dims` = 1.
  - `Log Ramp` - A ramp of real values that are logarithmically spaced between the ramp initial value (inclusive) and the ramp ending value (inclusive). This option is only available if `Num Dims` = 1.
- `Size` - The number of elements in the dimension. There is one field per dimension. Default is 10.

`Init Value` and `Size` may be set on the object or from data input pins.

Notes

To change the values of individual elements, use `Set Values`.

If the `Num Dims` value is changed, the view automatically creates or deletes fields on the open view so that the number of dimension-sized fields matches the `Num Dims` value.

The output array is not mapped. Use `Set Mappings` to map the dimensions of the array.

See Also

`Get Values`, `Real`, `Set Mappings`, and `Set Values`.

Alloc Text

An object that creates a Text array.

Use

Use `Alloc Text` to create an array of Text elements.

Location

`Data` ⟹ `Allocate Array` ⟹ `Text`

Open View Parameters

- `Num Dims` - The number of dimensions in the output array. `Num Dimensions` must be an integer number. Minimum is 1. Maximum is 10. Default is 1.
- `Init Value` - The value to place in each array element. This value is interpreted as a string. Default is null (an empty string).
- `Size` - The number of elements in the dimension. Default is 10.

`Init Value` and `Size` may be set on the object or from data input pins.

Notes

To change the values of individual elements, use `Set Values`.

If the `Num Dims` value is changed, the view automatically creates or deletes fields on the open view so that the number of dimension-sized fields matches the `Num Dims` value.

The output array is not mapped. Use `Set Mappings` to map the dimensions of the array.

See Also

`Get Values`, `Set Mappings`, `Set Values`, and `Text`.

AlphaNumeric

An object that displays alphanumeric data.

Use

Use `AlphaNumeric` to display any of the data types as a single value, an Array 1D, or a Array 2D. To view an array, scroll the display with the scroll bars.

Location

`Display` ⟹ `AlphaNumeric`

Object Menu

- `Clear At PreRun` - Clear the display at PreRun. Default is on (checked).
- `Clear At Activate` - Clear the display at UserObject Activation. Default is on.
- `Number Formats` - Specifies a different display format for numbers.

Notes

A row of asterisks "***" is displayed if the current width of the `AlphaNumeric` object is too small to display the default precision of the numeric type. Resize the object to display the data.

If there is a 2D Array, you will see a scroll bar.

Arrays that are three-dimensional and larger are displayed as the string `3D Array` and so forth.

ASCII 0-30 are dependent on font and may not even be readable.

See Also

`Logging AlphaNumeric` and `Number Formats`.

Auto Line Routing

Enables/disables automatic line routing preference.

Use

Set `Auto Line Routing` to have HP VEE draw only horizontal and vertical lines with lines routed around other objects. When `Auto Line Routing` is set, a checkmark appears next to it in the menu.

Location

`File` ⟹ `Preferences` ⟹ `Line Routing`

Notes

When `Auto Line Routing` is set, the lines connected to an object are rerouted when that object is moved or sized. It may take several moments to redraw the lines for a complicated model.

If `Auto Line Routing` is off, connecting lines are straight from pin to pin (unless tacked). Choose `Clean Up Lines` to reroute lines around objects.

When a particular routing seems complex, move some of the objects apart to give the `Auto Line Routing` feature more room to find a shorter path.

After using `Clean up Lines`, sometimes it is difficult to determine whether two lines cross each other or if one line is connected to three places. Use `Line Probe` to display the endpoints of the lines.

To set the default mode for `Auto Line Routing`, choose `Save Preferences`.

See Also

`Clean Up Lines`, `Preferences`, and `Save Preferences`.

Beep

An object that generates an audible tone.

Use

Use `Beep` to generate a tone of a specified frequency, duration, and volume.

Location

`Display` ⟹ `Beep`

Open View Parameters

- `Frequency (Hz)` - The frequency of the tone in Hertz. Default is 1000 Hz.
- `Duration (sec)` - The time in seconds for the tone to last. Default is 0.1 sec.
- `Volume (0-100)` - The relative volume of the tone, where 0 is silent and 100 is the loudest. Default is 80.

All of the three open view parameters can be added as data input terminals.

Notes

A `Frequency` or `Volume` value less than or equal to zero will pause for the specified duration.

The `Beep` object may complete execution (activate its sequence output pin) before a long duration tone has completed. Execution of subsequent `Beep` objects, however will pause model execution until the previous tone has completed.

The functionality of the `Beep` object is hardware dependent. In some cases, one or more of the open-view parameters may not affect the actual tone generated. Also, some computers do not have internal speakers and may require an external speaker module. Refer to your computer owner's manual for information about its audio output capabilities.

Break

An object that terminates the current iteration loop.

Use

Use `Break` to stop the loop and continue execution through the sequence output pin of the iterator.

Location

`Flow` ⟹ `Repeat` ⟹ `Break`

Note

`Break` is usually used with `Until Break` objects. It may also be used with other iteration objects and is often used with an `If/Then/Else` object to conditionally stop iteration.

See Also

`For Count`, `For Range`, `For Log Range`, `Next`, `On Cycle`, and `Until Break`.

Breakpoint (Object Menu)

Enables the breakpoint on the object when checked; disables the breakpoint on the object when not checked.

Use

Use `Breakpoint` as a debugging tool to stop the execution of the model before this object operates. An arrow points to the object that operates next. To continue execution, press the `Cont` or `Step` button in the upper right corner of the HP VEE window on the title bar.

When a breakpoint is set by the object menu or selected from the `Edit` menu, the `Breakpoint` selection is checked and the object is highlighted with a black (default) border.

When a breakpoint is set on an object in a `UserObject` (that is displayed as an icon and `Show Panel on Exec` is off), `Step` ignores the breakpoint.

Location

On each object menu $\Longrightarrow$ `Breakpoint`

Notes

To set breakpoints on multiple objects at the same time, use the `Set Breakpoints` feature from the `Edit` menu.

`Breakpoints` are saved with the model or `UserObject`.

See Also

`Activate Breakpoints`, `Clear All Breakpoints`, `Clear Breakpoints`, `Cont`, `Object Menu`, `Set Breakpoints`, and `Step`.

Breakpoints

A menu item.

Use

Use `Breakpoints` to access the following debugging options.

- `Set Breakpoints`
- `Clear Breakpoints`
- `Clear All Breakpoints`
- `Activate Breakpoints`

Location

`Edit` ⟹ `Breakpoints` ⟹

Notes

When a breakpoint is set on an object in a `UserObject` (that is displayed as an icon and `Show Panel on Exec` is off), `Step` ignores the breakpoint.

See Also

`Breakpoint (Object Menu)`, `Line Probe`, `Show Data Flow`, `Show Exec Flow`, and `Step`.

Build Arb Waveform

An object that creates a waveform from a set of coordinate data.

Use

Use `Build Arb Waveform` to create a uniformly sampled waveform from an Array 1D of Coord values.

Location

`Data` ⟹ `Build Data` ⟹ `Arb Waveform`

Open View Parameters

- `Num Points` - The number of points in the output waveform. Default is 256 and is set in `Waveform Defaults`.

`Num Points` may be set from the icon or added as a data input.

Notes

`Build Arb Waveform` uses the following formula:

```
dx = (Xmax-Xmin)/Num Points

Xvalue = Xmin + (point * dx)
```

Therefore, the last data point is not the last independent value of Coord.

The independent values `(x)` of each Coord must be greater than, or equal to, the independent value of the previous Coord in the array.

The `Input Coord` array must be (x,y) pairs, not (x,y,z).

See Also

`Build Waveform`, `Comparator`, `Coord`, `Function Generator`, `Noise Generator`, `Pulse Generator`, and `Waveform Defaults`.

Build Complex

An object that creates a Complex number.

Use

Use `Build Complex` to create a Complex number from real and imaginary components.

Location

`Data` ⟹ `Build Data` ⟹ `Complex`

Notes

If one of the inputs is an array, the other input must be either a Scalar or an array of the same size and shape. Both inputs are converted to type Real. The output is the same size and shape as the inputs. The input values' mappings are ignored and the output Complex value is not mapped.

See Also

`Build Data` and `UnBuild Complex`.

Build Coord

An object that creates a Coord from Real numbers.

Use

Use `Build Coord` to build a Coord from Real components.

Location

`Data` ⟹ `Build Data` ⟹ `Coord`

Notes

The number of data inputs determines the number of coordinate fields. The default (and minimum) is 2 (x,y). The maximum number of coordinate fields is 10.

If one of the inputs is an array, the other input(s) must be either a Scalar or an array of the same size and shape. All inputs are converted to type Real. The output is of the same shape as the inputs. The input values' mappings are ignored.

The data input shape must be a Scalar or Array 1D.

Note

For `Coordinate` data type, the field names will *always* be `x`,`y`, `z`,`w`, `v`,`u`, `t`,`s`, `r`,`q` even though you may change the input terminal names.

See Also

`Build Data`, `Coord`, `UnBuild Coord`, and `UnBuild Data`.

Build Data

A menu item.

Use

Use `Build Data` to access the following objects that create containers of particular data types from separate components:

- `Coord`
- `Complex`
- `PComplex`
- `Waveform`
- `Arb Waveform`
- `Spectrum`
- `Record`

Location

`Data` ⟹ `Build Data` ⟹

See Also

`Allocate Array`, `Build Arb Waveform`, `Build Complex`, `Build Coord`, `Build PComplex`, `Build Record`, `Build Spectrum`, `Build Waveform`, `Set Mappings`, and `Set Values`.

Build PComplex

An object that creates a PComplex (polar complex) number from separate components.

Use

Use `Build PComplex` to create a PComplex number from the magnitude and phase components. Units of the phase component are determined by the `Trig Mode` setting under `Preferences`.

Location

`Data` ⟹ `Build Data` ⟹ `PComplex`

Notes

If one of the inputs is an array, the other input must be either a Scalar or an array of the same size and shape. Both inputs are converted to type Real. The output is of the same shape as the inputs. The input values' mappings are ignored and the output PComplex value is not mapped.

The phase of the initial value is normalized to fall between pi and -pi radians; the magnitude of the number is converted positive if it was negative and the phase angle was shifted 180 degrees.

See Also

`Alloc PComplex`, `Build`, `Build Complex`, `PComplex`, `Trig Mode`, and `UnBuild PComplex`.

Build Record

An object that creates a record from separate components

Use

Use `Build Record` to create a record from scalar or array components. The input data components become *fields* in the output record. The output record can be either a scalar or a one-dimensional array.

Location

`Data` ⟹ `Build Data` ⟹ `Record`

Open View Parameters

- `Output Shape`—You can set the shape of the output record to either `Scalar` or `Array 1D`.

Notes

The fields in the record are named from the names of the corresponding input terminals. The field names are not case sensitive (lowercase and uppercase letters are equivalent). Field names may not be duplicated within a single record. Additional data inputs may be added to create a record with any number of fields. Records of records are allowed (where a field in a record is itself a record). There is no limit on recursion (except for the available memory).

If all inputs are scalar quantities, the record on the `Record` output pin will be a scalar record consisting of one field for each input, regardless of the specified `Output Shape`.

If one or more of the inputs are array, the shape of the resulting record depends on the specified `Output Shape`:

- If `Scalar` is specified as the `Output Shape`, the output record will be a scalar that consists of one field for each input. Each field will be either a scalar or an array, the same shape as the corresponding input.

- If `Array 1D` is specified as the `Output Shape`, the output record will be a one-dimensional array of records with fields for each input. If an input is a scalar or one-dimensional array, the corresponding field in the record will be a scalar. If an input is an `n`-dimensional array, the corresponding field will be an array of `n-1` dimensions. If more than one of the inputs are arrays, they must all have the same size and shape. Let's look at some examples:
  - If input `A` is a one-dimensional array 10 elements long and input `B` is a scalar, the output will be a one-dimensional record array of 10 elements with two fields, `A` and `B`. The individual elements of the `A` input array will appear in the individual array elements of field `A` in the output record. Input `B` will be duplicated 10 times in the `B` field of the output record.
  - If an input is an array of two or more dimensions, the output record will still be an Array 1D. For example, if an input field is a two-dimensional array, 10-by-3 in size, the output record is a one-dimensional array with a length of 10, but with a field which is a one-dimensional array of 3 elements.

See Also

`From DataSet`, `Get Field`, `Merge Record`, `Record Constant`, `Set Field`, `SubRecord`, `To DataSet`, and `UnBuild Record`.

Build Spectrum

An object that creates a Spectrum from separate components.

Use

Use `Build Spectrum` to build spectrum values from an array of PComplex numbers and the frequency interval information.

Location

`Data` ⟹ `Build Data` ⟹ `Spectrum`

Example

If you have an Array 1D of `[5]`, `PComplex (0,@1), (0,@2)`, and so forth, and frequency start equals `11` and stop equals `16`, then frequency sample equal `11, 12, 13, 14, 15`.

Open View Parameters

- `Start/Stop Freq` |`Center/Span Freq` - Determines whether the two type-in values represent `Start/Stop` or `Center/Span`. Either `Start/Stop` or `Center/Span` values can be added as inputs.
- `Freq Sampling` - Specifies whether the frequency sampling values are sampled linearly or logarithmically. Default is `Linear`.

Notes

The input `PComplex` values' mappings are ignored.

HP VEE does not track dB-scaled data. When you perform operations on dB-scaled data, you must ensure that they are done correctly.

Refer to *Using HP VEE*, for more information on the dB scaling of Spectrums.

To convert between db- and linearly-scaled data, use the library models provided with HP VEE in the `/usr/lib/veeengine/lib/` or `/usr/lib/veetest/lib/` directory.

There are the same number of points as sampling intervals. Each point is at the beginning of the sampling interval.

See Also

`Build Data`, `Build Waveform`, and `UnBuild Spectrum`.

`fft(x)` in the "Formula Reference" chapter.

Build Waveform

An object that creates a Waveform from separate components.

Use

Use `Build Waveform` to create a Waveform from a Real array of amplitude values and time span information.

Location

`Data` ⟹ `Build Data` ⟹ `Waveform`

Example

If you have an Array 1D of `[5]`, `Complex (0,1), (0,2)`, and so forth, and frequency start equals `11` and stop equals `16`, then frequency sample equal `11, 12, 13, 14, 15`.

Open View Parameters

`Time Span` - The length of time in seconds over which the y data was sampled. `Time Span` may be added as a data input. Default is 20m seconds set in `Waveform Defaults`.

Notes

The input values' mappings are ignored. `Build Waveform` assumes the `y` data was sampled linearly at a constant rate over the given interval.

There are the same number of points as sampling intervals. Each point is at the beginning of the sampling interval.

See Also

`Build Arb Waveform`, `Build Data`, `Build Spectrum`, and `UnBuild Waveform`.

`ifft(x)` in the "Formula Reference" chapter.

Bus I/O Monitor

An object that logs the messages sent over an I/O interface path (HP-IB, GPIO, VXI, or RS-232). This feature is available in HP VEE-Test only.

Use

Use `Bus I/O Monitor` to record each byte (command or data) transferred between HP VEE and any instruments it is controlling by way of a `State Drivers`, `Component Drivers`, `Direct I/O`, and the `Advanced I/O` devices. The information that is recorded can optionally be sent to a log file.

The display area of `Bus I/O Monitor` contains five columns. For HP-IB, GPIO and RS-232 the columns indicate:

- Column 1 - Line number.
- Column 2 - HP-IB command (`*`), data output (`>`), or data input (`<`).
- Column 3 - Hexadecimal value of the byte transmitted.
- Column 4 - 7-bit ASCII character corresponding to the byte transmitted.
- Column 5 - Bus command mnemonic (HP-IB bus commands only).

Only columns 1 through 3 are used when monitoring GPIO communications.

Only columns 1 through 4 are used when monitoring RS-232 communications.

For VXI the columns indicate:

- Column 1 - Line number.
- Column 2 - VXI command (`*`), data output (`>`), or data input (`<`).
- Column 3 - Memory space (`A16`, `A24`, or `A32`), symbolic name for register or location, offset from named location.
- Column 4 - Hexadecimal value of the byte transmitted.
- Column 5 - Logical address of the VXI instrument.

Location

I/O ⟹ Bus I/O Monitor

Object Menu

- Config - Determines the maximum number of lines retained in the monitor. Once the monitor is full, lines at the beginning of monitor are discarded to make room for newly-logged lines.
- Clear - Clears the contents of the Bus I/O Monitor. Clear may be added as a control input pin.
- Clear at PreRun - Clears the contents of the Bus I/O Monitor at PreRun. Note that if the Bus Monitor is not connected to a thread and Start is used (not Run), PreRun does not happen on the object. This is because only the objects on the Start thread are PreRun.
- Clear at Activate - Clears the contents of the Bus I/O Monitor at activate.
- Logging Enabled - Records activity in the list area of the Bus I/O Monitor. Log Enable and Log Disable control input pins may be added.
- Log To File - Sends data to the specified file as it is recorded by the Bus I/O Monitor. Logging Enabled must also be selected before any monitor data will be saved to the file. Monitor data is always appended to the end of the specified file. The Log File Name parameter can be added as a control input.
- Clear Log File - Deletes the contents of the specified log file. This parameter can be added as a control input.

Notes

The Bus I/O Monitor allows execution to proceed much faster when displayed as an icon rather than an open view.

Each Bus I/O Monitor object is capable of monitoring one physical bus. For example, to monitor HP VEE's activity on one HP-IB card and one RS-232 card, the user would need two Bus I/O Monitors.

Bus I/O Monitor is not a general purpose hardware bus monitor since it only shows the bytes input and output by HP VEE. Any other programs that

input/output to the bus, for example, HP BASIC/UX or shell escape programs that perform I/O, do not log their I/O to the HP VEE monitor.

See Also

`Advanced I/O`, `Configure I/O`, and `Instrument`.

Bus Operations

This menu item has been replaced with `Interface Operations`.

Call Function

An object that executes a previously defined User Function, Compiled Function, or Remote Function.

Use

Use `Call Function` to execute (call) a previously defined User Function, Compiled Function, or Remote Function. Each input and output on the `Call Function` object corresponds to the respective input/output on the called function. (For a User Function, the inputs and outputs correspond directly to the inputs and outputs on the original UserObject that was made into the User Function.) `Call Function` inputs are passed to the called function by position, not by name. For example, the `Call Function` object may have as its inputs, `A` and `B`. But the called function may have as its inputs `x` and `y`. No matter, the first input (`A`) on `Call Function` is passed as the first input (`x`) of the called function. Only position is significant.

You may also call a User Function, Compiled Function, or Remote Function by including its name in any expression whose evaluation is delayed until run time. These include expressions in `Formula`, `If/Then/Else`, `Set Field`, `Get Field`, `Get Values`, or `From DataSet` devices, or expressions in `Sequencer` or `I/O` transactions. For example, you can call the function `someUsrFunc` by including the expression `2+someUsrFunc(a)` in a `Formula` object.

You can select a function to call in two ways. You can execute `Select Function` in the object menu and select a name from a list of available functions, or you can type in the name directly in the `Function Name` field. In either case, `Call Function` will automatically configure its input and output pins for the called function, if that function is recognized by HP VEE.

Since `Call Function` can call User Functions (either locally created or imported from a library), Compiled Functions, or Remote Functions, it is possible to have two functions with the same name. Naming conflicts are resolved according to the following rule:

- The libraries are searched in the following order: 1) local User Functions, 2) imported ("external") User Functions, 3) Compiled Functions, and 4) Remote Functions.

This rule also applies to all objects capable of calling such functions from an expression. In all cases, the first (top-most) output pin of the defined function is the returned value for the expression.

If a function is defined with the same name as a built-in HP VEE function, such as `COS`, the user-defined function will override the built-in function.

Location

`Device` ⟹ `Function` ⟹ `Call Function`

Open View Parameters

- `Function name` - The name of the User Function, Compiled Function, or Remote Function to be called.

Object Menu

- `Select Function` - Lists the functions currently known to HP VEE. Locally created User Functions are listed first, followed by an alphabetically arranged list of the imported library functions (User Functions, Compiled Functions, and Remote Functions). When you select a function, the input and output pins on this object are automatically configured as defined by the called function. (This selection will be grayed out if no functions are known to the HP VEE model.)
- `Configure Pinout` - Automatically configures the input and output pins on this object as defined by the called function. (This selection will be grayed out if the named function is not recognized.)

Notes

The `Call Function` object acts much like a math function call. Any inputs act like the formal arguments of an internal HP VEE function, while any outputs act like return values from that function.

The actual execution of `Call Function` is synchronous—a call to a User Function, Compiled Function, or Remote Function will suspend the execution of parallel threads until the function is complete.

See Also

Delete Library, Import Library, Edit UserFunction, User Function, and UserObject.

Clean Up Lines

Redraws all lines in the model or UserObject to route around objects using only horizontal and vertical line segments.

Use

If you disable Automatic Line Routing, connecting lines are straight from pin to pin (unless tacked). Select Clean Up Lines from time to time to tidy up the lines in your model.

Location

Edit ⟹ Clean Up Lines
or
UserObject (Object Menu) ⟹ Edit ⟹ Clean Up Lines

Notes

Clean Up Lines is context sensitive; when it is selected from the UserObject Edit menu, only the lines within the UserObject are affected. When selected from the menu bar Edit, only the lines in the root context (not in any UserObjects) are affected.

It may take several moments to completely re-route the lines in a complex model.

When a particular routing seems complex, move some of the objects apart to give the Auto Line Routing feature more room to find a shorter path.

After using Clean Up Lines, sometimes it is difficult to determine if two lines cross each other or if one line is connected to three places. Use Line Probe to display the endpoints of the lines.

See Also

Auto Line Routing, Line Probe, Save Preferences, and UserObject.

Clear All Breakpoints

Clears execution breakpoints.

Use

Use `Clear All Breakpoints` to delete all breakpoints when you are finished debugging.

Location

`Edit` ⟹ `Breakpoints` ⟹ `Clear All Breakpoints`

See Also

`Activate Breakpoints`, `Breakpoint (Object Menu)`, `Clear Breakpoints`, and `Set Breakpoints`.

Clear Breakpoints

Clears the breakpoints from the selected objects.

Use

Use `Clear Breakpoints` to clear breakpoints from selected objects when debugging them is no longer necessary. The black border surrounding the objects is removed.

Location

`Edit` ⟹ `Breakpoints` ⟹ `Clear Breakpoints`

Notes

To clear the breakpoint of a single object, you can also clear the `Breakpoint` selection from the object menu.

If no objects are selected, `Clear Breakpoints` is not available.

See Also

`Activate Breakpoints`, `Breakpoint (Object Menu)`. `Clear All Breakpoints`, `Set Breakpoint`, and `Select Objects`.

Clone

Duplicates selected objects and their interconnections, and places a copy of them in the `Paste` buffer.

Use

Use `Clone` when you want an immediate copy of a set of selected objects. `Clone` copies all the attributes of the cloned object including pins, parameters, and size.

Location

`Edit` ⟹ `Clone`

Notes

`Clone` is only available after objects are selected.

To `Clone` a single object, use the `Clone` selection from its object menu.

Each `Clone` overwrites the previous content of the `Paste` buffer.

Each `Clone` of a `State Driver` reflects a different state of the same instrument (HP VEE-Test only).

Each `Clone` of a `UserObject` is an independent copy of all objects in it. Changing one copy *does not* affect others.

See Also

`Cut`, `Clone (Object Menu)`, `Copy`, `Object Menu`, `Paste`, and `Select Objects`.

Clone (Object Menu)

Duplicates this object and places a copy of the object in the `Paste` buffer.

Use

Use `Clone` to create a copy of the object. `Clone` makes a duplicate object immediately available and puts a copy of the object in the `Paste` buffer. `Clone` copies all the attributes of the cloned object including terminals, parameters, and size.

Location

On each object menu $\Longrightarrow$ `Clone`

Notes

To copy multiple objects, use the `Clone` selection from the `Edit` menu.

Each `Clone` overwrites the previous content of the `Paste` buffer.

Each `Clone` of a `UserObject` is an independent copy of all objects in it. Changing one copy *does not* affect any others.

Each `Clone` of a `State Driver` reflects a different state of the same instrument (HP VEE-Test only).

See Also

`Object Menu`, `Cut (Object Menu)`, `Copy`, `Clone`, and `Paste`.

Collector

An object that collects data and outputs an array.

Use

Use `Collector` to create a one-dimensional array from scalar input data, or to create an output array from collected input arrays. If the input arrays have "n" dimensions, the collector can either output an array of n+1 dimensions, or it can output a one-dimensional array (with the input data "flattened"). When all data has been collected, activate the `XEQ` pin to output the resulting array.

Location

`Data` ⟹ `Collector`

Open View Parameters

- `Output Shape` - Choose one of two selections:
  - `n+1 Dim Array` - The input signals (arrays of n dimensions) are collected and output as an array of n+1 dimensions.
  - `1 Dim Array` - The input signals, regardless of shape, are "flattened" by the collector and output as an Array 1D.

  (For scalar input data, the output shape will be "Array 1D" regardless of the selection.)

Object Menu

- `Clear` - Clears the contents of the `Collector`.
- `Clear at PreRun` - Clears the contents of the `Collector` at PreRun. Default is on (checked).
- `Clear at Activate` - Clears the contents of the `Collector` at activate. Default is on (checked).

Notes

Because XEQ is required on the `Collector` you cannot add another trigger pin or delete it.

The `Collector` continues to collect the input data until the XEQ pin is triggered.

The `Collector` has two inputs; one for the data coming in (any shape and type); the other tells the device when to execute. The device accepts multiple input data hits on the first data input pin, and collects the data until the XEQ pin is hit. Then the device copies the full set of data to its output pin and propagates its output. The output data pin is never propagated until the XEQ pin is hit. Once the data propagates, the `Collector` clears its internal buffer and again starts collecting data. The *type* of the output signal is determined by the type of the first data container coming in after each `XEQ` or `Clear`. All subsequent containers are coerced to the type of the first one. If coercion fails, an error is reported.

The *shape* of the output signal depends on the `Output Shape` selection:

- `n+1 Dim Array` operation - If you select `n+1 Dim Array` for the `Output Shape`, the `Collector` will collect the data coming into its `Data` pin, and will output an array of one dimension higher than the number of dimensions of the input data. Each input signal must be the same size and shape as the first input signal, until the `XEQ` pin is triggered. No mapping information is copied—the output signal is not mapped.

  For example, if the first input data is a Real 1D array, 10 long, each subsequent input signal must be a 1D array, 10 long, of a type that can be coerced into Real, until the `XEQ` pin is triggered. The output signal will be a Real 2D array with 10 columns and n rows, where "n" is the number of times data was sent to the `Data` input pin before `XEQ` was triggered.

- `1 Dim Array` operation - If you select `1 Dim Array` for the `Output Shape`, the `Collector` collects the data coming into its `Data` pin and outputs an Array 1D, regardless of the shape of the input data. Scalar or Array 1D data is appended to the end; multi-dimensional arrays are flattened into an Array 1D and then appended.

  For example, if the data in is a scalar Real value `1.2`, then a scalar Int32 value 34, then `XEQ` is hit, it will output a Real Array 1D, two long, with the

values [1.2 34.0]. Then, if it is hit with several Complex data containers, it will output a Complex Array 1D.

Note that for scalar inputs of any type, the Collector will generate the same output signal regardless of the Output Shape selection.

Since Waveform and Spectrum signals do not support more than one dimension, they are not allowed as inputs when the Collector is operating in the n+1 Dim Array mode. To collect Waveform or Spectrum data into a 2D array, unbuild the data first and then collect it into a 2D array. Note that the mapping information will be lost.

Since Record and Coord containers only support up to 1D arrays, only *scalar* Record or Coord containers are allowed as inputs when the Collector is operating in the n+1 Dim Array mode.

Enum inputs are converted to type Text before being placed into an array.

For Record inputs, the field names, types and sizes must match.

The Collector begins with its internal buffer cleared. It clears its internal buffer of collected data each time the XEQ pin is hit. It also optionally clears the buffer at PreRun and Activate. You also may force this buffer to be cleared at other times. Choose the Clear option from the object menu, or add a control input for Clear and programmatically clear the Collector data before XEQ is hit. Subsequent hits on the data input would start the collecting over again. The output is not propagated until XEQ is hit.

If XEQ is hit, and the data input has not been hit with non-nil data since the last clear, the Collector will output a nil signal.

See Also

Alloc Array, Concatenator, Get Values, Set Values, and Sliding Collector.

Comparator

An object that compares two values then outputs the coordinates where the comparison fails.

Use

Use the `Comparator` to compare a numeric test value, such as a Waveform or Coord, with a reference value. If one or more of the values in `Test Value` fails the comparison, a `Coord Array 1D` is output on the `Failures` output pin and the `Failed` output pin activates. If all values pass the comparison, an empty `Coord` is output on the `Failures` output pin and the `Passed` output pin activates.

The independent field (x) in the `Failures` output Coord Array 1D contains the index or mapping of each failure point. The dependent field (y) contains the `Test Value` that failed.

Location

`Device` ⟹ `Comparator`

Open View Parameters

`Test Value == Ref Value` - Change the comparison operator by clicking on it and selecting the function from the list that is displayed. Default is `==`.

Notes

The `Comparator` compares only numeric values, not text strings.

The shape of the `Ref Value` is treated as the same shape as the `Test Value`. For example, a Scalar `Ref Value` of `3` is treated as an Array 1D (with all elements equal to `3`) when the `Test Value` is an Array 1D. If arrays are compared, their size and number of dimensions must match.

If `Test Value` is a scalar, `Ref Value` must be a scalar.

If `Test Value` is an unmapped array, any mappings on `Ref Value` are ignored.

Comparator

If `Test Value` is a mapped array, the independent variable of the `Failures` output Coord (x) is the mapping of the element that failed.

If `Test Value` is a Coord, only the dependent variable of the Coord is used in the comparison. The `Failures` output Coord contains the Coord value that failed.

All tests use an "almost equal" algorithm that checks equality or inequality to at least six significant digits. If the `Ref Value` is an array with more than one element, a value for "virtual zero" is derived from the reference data's magnitude. Any data points with a value of "`0`" must be matched with this "virtual zero" value.

See Also

`Alloc Array`, `Build Arb Waveform`, `Coord`, `If/Then`, and `Set Values`

Component Driver

Selects an I/O object to control an instrument using an Instrument Driver where all the panels are unavailable.

Use

Click on `I/O` ⟹ `Instrument` and examine the list of configured instruments in the Select an I/O Device dialog box.

If the instrument you want is in the list and is properly configured:

1. Click once on the desired instrument to highlight it.

If the instrument is not configured with an ID Filename, the `State Driver` and `Component Driver` buttons are flat (grayed).

2. Click the button at the bottom of the dialog box labeled `Component Driver`.

If the instrument you want is not in the list or is not properly configured, use `Configure I/O` to add or change an existing one, then:

1. Click on `Add` to add a new instrument.

2. Complete the resulting dialog boxes. Refer to the `Configure I/O` entry for details about how to complete these dialog boxes.

Location

`I/O` ⟹ `Instrument` ⟹ `Component Driver`

Notes

All instruments must be configured before they can be accessed through the `Instruments` menu selection. The best way to configure instruments is to use the `Configure I/O` menu selection.

Note that Component Drivers may be operated with or without live instruments connected to the computer. If you wish to control a live instrument, you must set a correct, non-zero address and enable Live Mode. The address and Live Mode setting are controlled by way of the `I/O` ⟹ `Configure I/O` menu selection. If the address is zero or if Live Mode

is off, the instrument object operates but does not attempt to communicate with a physical instrument.

Your system administrator must properly configure your computer before it is possible to communicate between HP VEE and any hardware interface. If you believe that you have properly followed all HP VEE procedures properly and you still cannot achieve any level of communication with an instrument, the problem may be with your computer configuration. Ask your system administrator to read this explanation and verify proper configuration of your system's interface drivers. (These interface drivers are different from the instrument driver files included with HP VEE).

To be useful, at least one input or one output must be added. More than one input or output can be added and both inputs and outputs may appear on the same object. An input performs the set actions for the component. An output performs the get actions.

See Also

`Instrument`, `State Driver`, and `Terminals`.

Using HP VEE, chapter 5.

Complex

An object that outputs constant Complex scalars or an Array 1D.

Use

Use `Complex` to set a Complex Constant or to get user input. To input an array, press tab to enter the next value.

Location

`Data` ⟹ `Constant` ⟹ `Complex`

Example

To use `Complex` as a prompt on a panel, change the name of the `Complex` object to a prompt such as `Enter the AC voltage:`. The user fills in the requested information in the entry field.

Type in `(2,10/27)` to have the `Complex` constant evaluate the formula and display the answer.

Object Menu

- `Auto Execute` - If set, the object operates whenever the values in the field are edited.
- `Config` - Sets the initial number of values to be output with this object. A value of `0` sets the container to a scalar, otherwise the container is an array of the length given.
- `Initialize` - Used to set this object to a particular value at PreRun and/or Activate time.
  - `Initial Value` - A dialog box that specifies the value to be set. Default value is the zero value of that container type.
  - `Initialize At PreRun` - Whether to set the `Initial Value` at PreRun time. Default is off.
  - `Initialize At Activate` - Whether to set the `Initial Value` at Activate time. Default is off.

- `Number Formats` - Specifies a different display format.

Notes

`Initialize` is most often used for initializing values inside a `UserObject`.

The other method for setting initial values is the `Default Value` control pin. The `Default Value` pin allows you to programmatically change the current value.

Note that the `Initial Value` field is always a scalar, even if `Complex` is configured to be an array. The `Default Value` input pin, however, requires its input container to match the shape of the `Complex`.

You can enter just the `Real` and `Imaginary` values, separate them with a comma, and HP VEE formats for you.

See Also

`Alloc Complex`, `Build Complex`, `Number Formats`, `Constant`, `Coord`, `Date/Time`, `Enum`, `Integer`, `PComplex`, `Real`, `Text`, and `Toggle`.

Complex Plane

An object that displays continuously-generated complex data as a Cartesian plot.

Use

Use `Complex Plane` to display Complex, PComplex, or Coord data values on a Real versus Imaginary axis.

Location

`Display` ⟹ `Complex Plane`

Open View Parameters

- `Auto Scale` - Automatically scales the display to show the entire trace.
- `Imag` - The name of the Y (Imaginary) axis.
- `Trace1` - The name of the first trace.
- `Real` - The name of the X (Real) axis.

Object Menu

- `Auto Scale` ⟹ - Automatically scales the display to show the entire trace.
  - `Auto Scale` - Automatically scales both axes.
  - `Auto Scale X` - Automatically scales the X axis.
  - `Auto Scale Y` - Automatically scales the Y axis.

  These parameters may be added as control inputs.
- `Clear Control` ⟹ - Parameters that specify when to clear the display.
  - `Clear` - Clears the displayed trace(s). This parameter may be added as a control input.
  - `Clear At PreRun` - Clears the displayed trace(s) when the model or thread is PreRun.

- `Clear At Activate` - Clears the displayed trace(s) when the User Object is activated.
- `Next Curve` - Resets the pen to display the next curve in a family of curves (data from the next time the display operates) *without* clearing the previous curve. `Next Curve` must be selected (or activated) before each new curve in the family. This parameter may be added as a control input.

- `Zoom` ⟹ - Scales the display.
  - `In` - Magnifies the display to contain only the rectangular region that you selected with the pointer. You select the region after selecting this feature by dragging on the graph area. This action outlines the information with a "rubber band" box.
  - `Out 2|5|10|20|50|100 x` - Expands the scales of the display by a factor in both the X and Y directions about the center.
- `Markers` ⟹ - Allows you to find the exact value of a data point on the displayed curve. If the data is plotted on a log scale, the values shown on the marker are the linear data points.

  To move markers to a different trace, click on the button to the left of the marker values near the bottom of the display. The button cycles through the different line types and colors of the different traces.
  - `Off` - No markers are shown.
  - `One On` - One marker is available.
  - `Two On` - Two markers are available.
  - `Delta On` - Two markers are available and the x and y differences between them are displayed.
  - `Interpolate` - When checked, you can place markers between the displayed data points. The marker values displayed are calculated by linear interpolation. Default is off.
  - `Center` - If markers are available, brings them to the center of the visible part of the trace. This is useful if you've scrolled the display and markers are not visible.

- `Grid Type` ⟹ - Sets the type of grid marks. The value of the major x division is shown below the maximum x value (to the right of the x scale name). The value of the major y division is shown below the y scale name.
  - `No Grid` - No grid lines are shown.
  - `Tic Marks` - Shows tic marks at the major and minor divisions on all four sides of the graph.
  - `Axis` - Shows tic marks at the major and minor divisions on the X and Y axes of the graph. If the actual axes are scrolled off the graph area, axis lines are drawn on the edge closest to the axes.
  - `Lines` - Shows lines at the major divisions and tic marks at the minor divisions. The X and Y axes are shown as thick lines.
- `Panel Layout` ⟹ - Sets the appearance of the open view.
  - `Graph Only` - The open view shows only the graph area and the marker information (if it exists). No buttons, scales, scale names, or traces names are shown. This layout redraws quickest and provides the largest display area.
  - `Scales` - The open view shows the graph area, the scales, the scale names, and trace names. These fields are not recessed and may not be edited. This layout is useful when recessed fields might be distracting, such as printing graphs, or when the fields should be protected from editing on a User Panel.
  - `Scales & Sliders` - The open view shows all information about the graph. It includes the most information and allows you to modify the most elements interactively. It is the default.
- `Traces & Scales` - A control panel that allows you to specify values such as the names, colors, line and point characteristics, minimum values, and maximum values for the traces and/or scales.

  Traces:
  - `Name:` - The name of the trace that is displayed to the left of the graph area and the name of the corresponding input terminal.
  - `Scale:` - If you have multiple Y scales, selects which Y scale is to be used for this trace.

- `Color:` - The color of the trace. Each added trace has a different color than the existing traces. Default is `Pen 4` (yellow).
- `Lines:` - The format of the line connecting data points. Default is a continuous line.
- `Points:` - The symbol that marks each data point. To show unconnected data points, select the single dot `Lines` format and the desired `Points` symbol. Default is a dot.

You can add a `Traces` control input pin to control the trace parameters listed above. The control input data must be a record with the following fields: 1) A `TraceNum` field with an Integer value (`1` is the top trace), *and* 2) one or more of the following fields: `Name`, `Pen`, `LineType`, `PointType`. (The `Pen`, `LineType`, and `PointType` values are integers from `0` to *n*, where `0` draws nothing.) Refer to "Records and DataSets" in *Using HP VEE* for further information.

Scales:

- `Show Scale:` - If you have multiple Y scales, a selection (using a check box) to specify if the end points and an axis of each additional right scale is displayed to the right of the graph area.
- `Scale Name:` - The names of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Maximum:` - The maximum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Minimum:` - The minimum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Mapping:` - The way the x and y data is mapped to the display. The mapping may be linear or log. To make a log-log plot, change both `X` and `Y` axes to `Log`. Default is `Linear`.
- `Log Cycles:` - The maximum number of decades shown (counting down from the maximum x and y values) when `AutoScale` is activated. `Log Cycles` is only used when the `Mapping` is `Log`. `Log Cycles` is useful when a trace contains a large dynamic range in the data. Default is 4.
- `Scale Colors` - The color of any background grid or tic marks. Default is `Gray`.

You can add a Scales control input pin to control the scales parameters listed above. The control input data must be a record with the following fields: 1) A Text field Scale with a value X, Y (or Y1), Y2, or Y3, *and* 2) one or more of the following fields: Name, Min, Max, and Mapping. (The Mapping text value may be Linear or Log). Refer to "Records and DataSets" in *Using HP VEE* for further information.

- Add Right Scale - Adds up to two additional scales to permit traces to have different scale ranges. After adding a right scale, use Traces & Scales to assign a trace to the scale.
- Plot - Presents the Plotter Configuration control panel. When OK is pressed, a copy of the device's entire display is plotted on the selected plotter. This parameter may be added as a control input. If the current Plotter Configuration is in Plot to File mode, you may specify the destination file or directory name as string data on the Plot control input. If no control input value is given, the file or directory name specified in Plotter Configuration will be used. See Plotter Config for more information.

Notes

Inputs must be Scalar or Array 1D of type Complex, PComplex, or Coord.

Add traces with the Terminals ⟹ Add Data Input object menu selection. Up to twelve traces are allowed.

Input data of type Coord is plotted by simply using its x and y values without first being converted to type Complex.

A Title control input may be added, which sets the title bar name to the specified text value. This allows programmatic control over the title shown when the display is printed or plotted.

See Also

Magnitude Spectrum, Polar Plot, Strip Chart, Waveform (Time), X vs Y Plot, XY Trace, and Plotter Config.

Concatenator

An object that outputs the concatenation of all its input data into an Array 1D.

Use

Use `Concatenator` to combine two or more data elements into the elements of an Array 1D. The size of the resultant array is the sum of the sizes of the inputs. Data inputs can be added to allow three or more containers to be concatenated.

`Enum` inputs are converted to type `Text` before concatenation occurs.

Location

`Data` ⟹ `Concatenator`

Notes

The output data type is the highest type of all the input data types. If any input is an array of more than one dimension, the array is flattened (row major) into an Array 1D for the concatenation.

To concatenate multiple lines of Text into one line, use `Accumulator`.

Nil inputs are ignored. This is different than the `concat(x,y)` function which operates within the formula box. In the formula box, all nil inputs are automatically converted to Real scalars with value `0`. Thus, `concat(a,b)`, where `a` is nil and `b` is an Int32 scalar value `5`, results in the Array 1D, two long of Real with values [0 5]. The `Concatenator` object with two inputs with nil and an Int32 scalar value `5` outputs an Int32 array one long with value [5].

If the first value (first data input) is mapped, the other input values may be either mapped with the same point spacing, or unmapped. The result will be mapped from the `Xmin` to `Xmin + dx * N` where "Xmin" is the minimum value of the mapping of the first array value, "dx" is the distance between two consecutive points in the first value, and "N" is the size of the result array.

See Also

`Accumulator`, `Alloc Array`, `Collector`, `Set Values`, and `Sliding Collector`.

`concat(x,y)` in the "Formula Reference" chapter.

Conditional

A menu item.

Use

Use `Conditional` to access the following conditional branches:

- `If A == B`
- `If A != B`
- `If A < B`
- `If A > B`
- `If A <= B`
- `If A >= B`

Use `Conditional` objects to decide which subthread to run.

Location

`Flow` ⟹ `Conditional` ⟹

Notes

`Conditional` objects are preconfigured `If/Then/Else` objects. You cannot modify the condition or change the number of, type, or shape of data inputs or outputs.

Note the difference between menu items under `Relational` and under `Conditional`. `Relationals` are formulas with output `0` or `1`; `Conditionals` are `If/Then/Else` and have two outputs of which one fires.

Note that the return value of `AND, OR, XOR`, and `NOT` is of type Int32 and is the same shape as the operands. This is different than the conditionals such as `==` that always return a scalar.

The logical operators are defined for type text only in the sense of whether the string is null or not. That is, `"0"` and `"1"` are logically true since they are non-null strings, while `""` is false. Therefore, `"zoo" AND ""` (null string) is logically false since the second string is null. Remember that when comparing a Text type to a non-string type, the latter is promoted to a Text type. This means that `"zoo" AND "0"` is true since the Real `0` is promoted to the string

"0" and, since both strings are non-null, the AND expression is true, returning 1.

See Also

`If A == B`, `If A != B`, `If A < B`, `If A > B`, `If A <= B`, `If A >= B`, and `If/Then/Else`.

`Relational` in the "Formula Reference" chapter.

Configure I/O

Allows you to create and edit configurations for instruments and selected hardware interfaces. This feature is available in HP VEE-Test only.

Use

Use `Configure I/O` to add or delete an instrument, or edit an existing configuration. This section contains a very brief overview on using `Configure I/O`. If you have not already done so, you should read the following information:

In `Configure I/O`, you must decide whether you wish to configure an instrument for use with or without a driver. If you are not sure whether a driver is available, follow the configuration procedure up to the point where a driver file is specified.

If an instrument is configured without a driver, it can only be used to create `Direct I/O` objects. If an instrument is configured with a driver, it can be used to create `State Drivers`, `Component Drivers`, and `Direct I/O` objects.

The configuration is Read/Saved to `.veeio` file in $HOME directory, so `Config` information is saved across executions of HP VEE.

See also "Configuring Instruments" and "Details of Configure I/O Dialog Boxes" in *Using HP VEE*, chapter 5.

Configuring an Instrument With A Driver

1. Click on `I/O` ⟹ `Configure I/O`. The Configure I/O Devices dialog box appears.

2. Click on `Add` or `Edit` in the Configure I/O Devices dialog box to add a new instrument configuration or edit an existing one. The Device Configuration dialog box appears.

3. Click on the `Instrument Driver Config` button. The Instrument Driver Configuration dialog box appears.

4. Click on the `ID Filename` field. A list of driver files appears.

5. Scroll through the list of driver files and double-click on the one that matches your instrument.

6. Complete the remaining fields in the Instrument Driver Configuration dialog box, then click on OK.

7. Complete the fields in the Device Configuration dialog box, then click on OK. (Note that the Interface field defaults to HP-IB. Click on the field to display a list of available interfaces.)

8. Click on Save in the Configure I/O Devices dialog box to add the newly configured instrument to the Configure I/O Devices list.

Configuring an Instrument Without A Driver

1. Click on I/O ⟹ Configure I/O. The Configure I/O Devices dialog box appears.

2. Click on Add or Edit in the Configure I/O Devices dialog box to add a new instrument configuration or edit an existing one. The Device Configuration dialog box appears.

3. Click on the Direct I/O Config button. The Direct I/O Configuration dialog box appears.

4. Complete the fields in the Direct I/O Configuration dialog box, then click on OK.

5. Complete the fields in the Device Configuration dialog box, then click on OK. (Note that the Interface field defaults to HP-IB. Click on the field to display a list of available interfaces.)

6. Click on Save in the Configure I/O Devices dialog box to save the newly configured instrument.

Location

I/O ⟹ Configure I/O

See Also

Advanced I/O, Bus I/O Monitor, and Instrument.

Using HP VEE, chapter 5.

Confirm (OK)

See `OK`.

Constant

A menu item.

Use

Use `Constant` to access the following input constants:

- `Text`
- `Integer`
- `Real`
- `Coord`
- `Complex`
- `PComplex`
- `Date/Time`
- `Record`

Location

`Data` ⟹ `Constant`

Notes

All `Constant` objects, except `Text`, support the use of arbitrary formulas in the input box. (However, `Global` variables are not allowed.) The formula is immediately parsed and evaluated and the resulting Scalar number is used as the value for the `Constant`. `Constant`s do not support input variables like the `Formula` object does, therefore you cannot use input variable names in the formulas. The typed in formula must evaluate to a Scalar value of the proper type or of a type that can be converted to the `Constant` you are using.

The syntax supported is the same as in the Formula box, but with some exceptions. You cannot use array building syntax `([, ])` operators to construct an array, since they would return a non-Scalar and the return value must be Scalar. Also, you cannot use input name variables in `Constant` objects.

In general, the use of any of the dyadic operators, parentheses for nesting, function calls, and the predefined numeric constant `PI (3.1416 ... )` is allowed.

Examples

When you type `(2, @PI/2)` into a `PComplex` object, it is evaluated and returns `(2, @90)` (if `Trig Mode` is set to Degrees).

When you type `14 + exp10(3)` into a `Real` object, it is evaluated and returns `1014`.

When you type `(1, 2)` into a `Real` object, an error is returned because HP VEE does not allow the conversion from `Complex` to `Real`.

See Also

`Complex`, `Coord`, `Date/Time`, `Enum`, `Integer`, `PComplex`, `Real`, `Record`, `Text`, and `Toggle`.

Cont

A button that causes all suspended threads to continue running after the model has been paused.

Use

Use `Cont` to resume the model execution after the model has been stopped. The model pauses if the `Stop` button is pressed once, if a breakpoint is reached, or if `Step` is used.

`Cont` is primarily used to continue execution between breakpoints and is usually used in conjunction with the other debugging tools.

Location

On the right side of the title bar.

Example

Type in `2, 5 + exp10(2)` to generate the `Coord` constant (2,105).

Notes

`Cont` is not available if `Stop` was pressed twice.

If you edit a model that is paused (by pressing `Stop` once) or if you pressed `Stop` twice, you must press `Run` or `Start` to restart the model's execution.

See Also

`Breakpoints`, `Run`, `Start`, `Step`, and `Stop`.

Coord

An object that outputs a constant Coord Scalar or Array 1D.

Use

Use `Coord` to set a Coord constant or to get user input. To input an array, press tab to enter the next value.

Location

`Data` ⟹ `Constant` ⟹ `Coord`

Object Menu

- `Auto Execute` - If set, the object operates whenever the values in the field are edited.
- `Config` - Sets the initial number of values to be output with this object. A value of `0` sets the container to a scalar, otherwise the container is an array of the length given.
- `Initialize` - Used to set this object to a particular value at PreRun and/or Activate time.
  - `Initial Value` - A dialog box that specifies the value to be set. Default value is the zero value of that container type.
  - `Initialize At PreRun` - Whether to set the `Initial Value` at PreRun time. Default is off.
  - `Initialize At Activate` - Whether to set the `Initial Value` at Activate time. Default is off.
- `Number Formats` - Specifies a different display format.

Notes

`Initialize` is most often used for initializing values inside a `UserObject`.

The other method for setting initial values is the `Default Value` control pin. The `Default Value` pin allows you to programmatically change the current value.

Note that the `Initial Value` field is always a scalar, even if `Coord` is configured to be an array. The `Default Value` input pin, however, requires its input container to match the shape of the `Coord`.

`Coord`s can contain more than two fields. For example, a `Coord` with three fields is (x, y, z).

The mathematic operations on `Coord`s operate on the dependent variable only, that is, they operate on the `y` value of the (x, y) pair.

Example

The `Coord` Array 1D can be used to define bounding envelopes on waveforms and as input to `Comparators` for waveforms. Use an Array 1D of `Coords` to define the limiting points on the bounding waveform. The limiting points are the endpoints of line segments that define the limiting waveform shape. Then send this array through the `Build Arb Waveform` box to create a limiting waveform. The new bounding waveform can be used in the `Comparator` box as one of the inputs to generate the points that fall outside the bounding waveform.

You can use the bounding waveform in the `clipUpper` or `clipLower` function to clip the waveform to the bounding limits.

See Also

`Complex`, `Constant`, `Date/Time`, `Enum`, `Integer`, `PComplex`, `Real`, `Text`, and `Toggle`.

Copy

Places a copy of the selected objects in the `Paste` buffer.

Use

Use `Copy` and then `Paste` to make multiple copies of a set of selected objects.

Location

`Edit` ⟹ `Copy`

Notes

`Copy` is only available after objects are selected.

If you want to copy a single object, use the `Clone` selection from its object menu.

Each `Copy` overwrites the previous content of the `Paste` buffer.

`Copy` is different from `Clone` in that `Clone` gives you a duplicate set of objects immediately and `Copy` only places the objects in the `Paste` buffer for later use.

See Also

`Clone`, `Cut`, `Object Menu`, `Paste`, and `Select Objects`.

Counter

An object that displays and outputs the number of times its data input pin has been activated.

Use

Use `Counter` to keep a running count. `Counter` keeps a running count of the number of times an input has been activated by a previous object's output. The `Counter` output is a Real scalar.

Location

`Device` ⟹ `Counter`

Object Menu

- `Clear` - Clears the contents of the `Counter`.
- `Clear at PreRun` - Clears the contents of the `Counter` at PreRun. Default is on (checked).
- `Clear at Activate` - Clears the contents of the `Counter` at Activate. Default is on (checked).

Notes

The data input for `Counter` does not require any particular type data and even counts an input with a nil value. It can also be cleared by activating the input `Clear` control pin.

The value of the `Counter`, a Real scalar, is available at the `Counter` output.

See Also

`Accumulator`.

Create UserObject

Puts selected objects into a UserObject.

Use

Use `Create UserObject` to group objects together logically and physically. Select the objects to be included and then select `Create UserObject`. The `UserObject` created contains the selected objects.

A `UserObject` operates just like any other object; no objects operate inside a `UserObject` until all data inputs to the `UserObject` are activated unless the optional XEQ pin is activated. All operations inside the `UserObject` must be complete before the data output is activated.

The right-most button on the title bar of the open view is a maximize button that increases the size of the `UserObject` to the size of the HP VEE work area.

If the objects in the `UserObject` were connected to other objects before they were in the `UserObject`, terminals are automatically created on the `UserObject` to maintain the connections. When objects inside the `UserObject` are connected to objects outside the `UserObject`, input and output pins are also automatically created.

Location

`Edit` ⟹ `Create UserObject`

Object Menu

You can select the object menu of the `UserObject` from the object menu button, but you cannot select it from within the work area of the `UserObject` open view. From within the work area, the right mouse button provides a pop-up `Edit` menu, just like the one available in the main work area. When the pointer is over an object within the `UserObject`, the right mouse button gives the object menu for that object. You can get `UserObject` object menu by placing the pointer over the borders of the `UserObject` and clicking the right mouse button.

- `Make UserFunction` - Converts the `UserObject` into a User Function. The `UserObject` will disappear from the screen and will be replaced with a `Call Function` object containing a call to the new User Function. Before you do this operation, enter a unique name into the title field. This name will become the User Function name, which will be the name by which `Call Function` or certain expressions can call the User Function. Should the name conflict with the name of an existing User Function, an error will be displayed. You will then need to enter a different name and repeat the operation.
- `Unpack` - Deletes the `UserObject`, but not the objects contained in it.
- `Secure` - Prevents the `UserObject` from being modified.
- `Show Panel on Exec` - When set, shows the panel view associated with the `UserObject` when the `UserObject` operates. This is only available after the `UserObject` panel view has been created (by way of `Add To Panel`).
- `Trig Mode` $\Longrightarrow$ - Specifies the trig mode used in the `UserObject` context (degrees, radians, or gradians).
- `Edit` $\Longrightarrow$ - A parent menu that leads to the `UserObject` context `Edit` menu. This menu contains the same choices as the main menu `Edit` menu. You can get the pop-up `Edit` menu by clicking the right mouse button on the `UserObject` work area.

  The following choices are context sensitive to the `UserObject`:

  - `Clean Up Lines` - Routes the lines in the `UserObject` around objects.
  - `Move Objects` - Moves several objects at once.
  - `Create UserObject` - Creates a `UserObject` of the currently selected objects.
  - `Add To Panel` - Creates a panel view (for the `UserObject`) containing the selected objects.

  The following choices are available when editing a User Function:

  - `Make UserObject` - The opposite operation from `Make UserFunction`. Turns the User Function back into a `UserObject`
  - Delete - Deletes the User Function from the HP VEE model.

Notes

If no objects are selected, `Create UserObject` is not available. To create an empty `UserObject`, select `UserObject` from the `Device` menu. You can place objects inside the `UserObject` after it has been created by moving them inside the `UserObject` boundaries.

It is convenient to save `UserObjects` of common functions (using `Save Object`) to create a library of functions to be used again.

When you `Step` through a `UserObject` the entire context runs when `UserObject` is an icon, has `Show on Exec` checked, or the `UserObject` panel view is displayed.

See Also

`Add to Panel`, `Secure`, `Select Objects`, `Step`, `Trig Mode`, `UserFunction` and `UserObject`.

Cut

Deletes the selected objects and places them in the Paste buffer.

Use

Use Cut to delete a set of selected objects or to remove the objects from one place on the work area and put multiple copies of them in another place on the work area (using Paste).

Location

Edit ⟹ Cut

Notes

Cut is only available after objects are selected.

Cut is not available when a model is running.

To delete a single object, use the Cut feature from its object menu.

Each Cut overwrites the previous content of the Paste buffer.

If you accidentally Cut objects, they can be recovered by using Paste.

To make multiple copies without deleting objects, use Clone or Copy.

See Also

Clone, Cut (Object Menu), Copy, Paste, and Select Objects.

Cut (Object Menu)

Deletes the object and places it in the Paste buffer.

Use

Use Cut to remove this object from your work area.

Location

On the object menu ⟹ Cut

Notes

Cut is available from all objects.

Each Cut overwrites the previous content of the Paste buffer.

Cut (Object Menu) is not available when a model is running.

If you accidentally Cut an object, it can be recovered by using Paste.

If you want to delete multiple objects, use the Cut selection from the Edit menu.

Short Cuts

To delete the object under a pointer, press CTRL D.

See Also

Clone, Clone (Object Menu), Copy, Cut, Paste, and Select Objects.

Date/Time

An object that outputs a constant date and time value. `Date/Time` takes an input date and time and converts it to the number of seconds between the input and the beginning of the Epoch (defined as beginning at 0000 hours UTC January 1, 0001 AD).

Use

Use `Date/Time` to display the time and date at the moment an object or data is created. To input an array, press tab to enter the next value.

`Date/Time` can be used to calculate elapsed time between a past and current event by comparing the two mathematically.

To change a value such as the day, date, or time, type the value. For example, typing `Fri` changes `Thu 28/Mar/1991 15:43:33` to `Fri 29/Mar/1991 15:43:33`. If you don't specify a value, the current day, date, or time is displayed. If you change just one of these the other time elements will stay the same.

Location

`Data` ⟹ `Constant` ⟹ `Date/Time`

Example

You can use `Date/Time` to determine the length of time that has passed between the moment a specific piece of experimental data was produced and another moment in time, either past or present. Use a `Date/Time` box to enter the new, arbitrary date then take the difference between the original, experimental data and the new arbitrary date.

To determine the length of time it takes for all or part of your model to execute, use `Timer` or `Time Stamp`.

Object Menu

- `Auto Execute` - If set, the object operates whenever the values in the field are edited.
- `Config` - Sets the initial number of values to be output with this object.
- `Initialize` - Used to set this object to a particular value at PreRun and/or Activate time.
  - `Initial Value` - A dialog box that specifies the value to be set. Default value is the zero value of that container type.
  - `Initialize At PreRun` - Whether to set the `Initial Value` at PreRun time. Default is off.
  - `Initialize At Activate` - Whether to set the `Initial Value` at Activate time. Default is off.

Notes

`Initialize` is most often used for initializing values inside a `UserObject`.

The other method for setting initial values is the `Default Value` control pin. The `Default Value` pin allows you to programmatically change the current value.

Note that the `Initial Value` field is always a scalar, even if `Date/Time` is configured to be an array. The `Default Value` input pin, however, requires its input container to match the shape of the `Date/Time`.

Notes

The default value control input should be a valid Date/Time string as given by the parse rules for date/time constants. These are the same rules as those for `Direct I/O`.

The `Date/Time` constant is different from the `Time Stamp` which gives the number of seconds between the beginning of the Epoch and the current date only.

You can display `Date/Time` output in `AlphaNumeric` or `Logging AlphaNumeric` by using the Real number format: `Time Stamp`.

See Also

Time Stamp and Timer.

Delay

An object that waits a specified period of time before activating its `Done` data output terminal.

Use

Use `Delay` to delay execution of a thread segment for a specified number of seconds.

Location

`Flow` ⟹ `Delay`

Example

Use `Delay` as a time out in a dialog box created in a `UserObject`.

Open View Parameters

Enter the number of seconds delay in the edit field. If you add a data input pin, the value supplied to that pin determines the delay period in seconds.

Notes

The delay value is specified in seconds and has a resolution dependent on the system clock and the nature of the model in which the `Delay` is operating. You can set the value for the `Delay` through a data input.

The delay interval is measured from the time when the sequence input pin is activated. After the specified interval, the `Done` data output terminal is activated. Note that `Delay` is "asleep" while it is waiting, thereby allowing other objects the opportunity to operate.

In most cases, either the `Done` data output terminal or the sequence out pin may be used to continue propagation after the `Delay` time expires. However, when the `Delay` object is part of a thread containing other objects in executing in parallel, the sequence out pin may not fire until a much longer time has elapsed. This is because sequence out pins are fired only when there are no other objects which may execute. To ensure that the delayed thread will

continue at the end of the `Delay` period, use the `Done` data output terminal, which always fires at the end of the `Delay` time period.

See Also

`OK`, `On Cycle`, `Device Event`, and `Interface Event`.

Delete (Object Menu)

Removes this object from the panel view.

Use

Use `Delete` to remove the object from the panel view. Note that the object remains on the detail view.

Location

On the object menu ⟹ `Delete`

Notes

`Delete` is only available from objects on the panel view.

Short Cuts

You can quickly delete a panel object by placing the cursor over the object and then pressing CTRL D.

See Also

`Cut`, `Object Menu`, and `Show Title`.

Delete Bitmap (Object Menu)

Deletes the bitmap displayed on this icon.

Use

Use `Delete Bitmap` to remove the bitmap displayed on this icon. If `Show Label` is checked, the icon only displays its name. If `Show Label` is not checked, the icon appears as a blank object.

Location

On the object menu $\Longrightarrow$ `Layout` $\Longrightarrow$ `Delete Bitmap`

Notes

`Delete Bitmap` is only available from the icon of an object.

See Also

`Layout`, `Object Menu`, `Select Bitmap`, and `Show Label`.

Delete Input (Object Menu)

Deletes an input pin from the object.

Use

Use `Delete Input` to delete any input pin (data, control, or XEQ) that was added to the object.

After selecting `Delete Input`, select the pin to be deleted from the dialog box displayed. The pin and any connections to it are deleted. If two pins have identical names HP VEE deletes the pin based on the order that the pins appear in the list.

Location

On the object menu ⟹ `Terminals` ⟹ `Delete Input`

Notes

`Delete Input` is available only if you have pins that may be deleted. Some pins cannot be deleted.

Short Cuts

You can quickly delete a data input terminal by placing the cursor over the input terminal display area and then pressing CTRL D.

See Also

`Add Control Input`, `Add Data Input`, `Add Data Output`, `Add XEQ Input`, `Delete Output`, `Object Menu`, `Select Data Output`, and `Terminals`.

Delete Library

An object that deletes (unloads) a library of User Functions, Compiled Function definitions, or Remote Function definitions from a running model.

Use

Use `Delete Library` to delete a library from your HP VEE model. The action taken depends on the type of library:

- `User Function` - `Delete Library` deletes all of the library User Functions from the HP VEE model. Those User Functions can then no longer be executed.
- `Compiled Function` - `Delete Library` detaches a shared library from the HP VEE process. However, the file containing the shared library is actually detached only when the last library pointing to that file is deleted. That is, if you have several different libraries pointing to the same shared library file (with either the same or a different definition file), the shared library file is detached from the HP VEE process only when there are no more libraries using the attached code. The operating system will not allow you to move or delete a shared library file once it is attached to a process. You must use the `Delete Lib` option in the `Import Library` object to detach the code if you want to create a new shared library file.
- `Remote Function` - `Delete Library` shuts down the remote HP VEE process and removes the Remote Function definitions from the local HP VEE process. `Delete Library` actually shuts down the remote process when the last library that references the remote process is deleted on the local host.

Location

`Device` ⟹ `Function` ⟹ `Delete Library`

Open View Parameters

- `Library Name` - Enter the name of the library that you want to delete.

Notes

`Delete Library` is generally used for advanced operations where you want to dynamically load, and then delete, library objects from a running HP VEE process.

See Also

`Call Function`, `Edit UserFunction`, `Import Library`, `User Function`, and `UserObject`.

Delete Line

Removes the line between two pins.

Use

Use `Delete Line` to eliminate wrong connections or to modify connections in your program. Select `Delete Line`, then click on or near the line to be deleted.

To exit this mode without deleting a line, click on an empty space on the work area, away from any lines.

Location

`Edit` ⟹ `Delete Line`

Notes

To view the endpoints of a line (without deleting it), select `Delete Line` and *hold down* the mouse button when the pointer is over the line. Move the pointer away from the line before releasing the button.

`Delete Line` is not available when the model is running.

Short Cuts

While holding down CTRL Shift place the pointer over line(s) and click the left mouse button to delete lines.

You can delete a series of lines this way, one at a time.

See Also

`Cut` and `Line Probe`.

Delete Output (Object Menu)

Deletes the output pin from the object.

Use

Use `Delete Output` to delete a data or error output pin that was added to the object.

After selecting `Delete Output`, select the pin to be deleted from the dialog box displayed. The pin and any connections to it are deleted. If two pins have identical names HP VEE deletes the pin based on the order that the pins appear in the list.

Location

On each object menu ⟹ `Terminals` ⟹ `Delete Output`

Notes

`Delete Output` is available only if the object has pins that may be deleted. Some pins cannot be deleted.

Short Cuts

You can quickly delete an output terminal by placing the cursor over the output terminal view and then pressing CTRL D.

See Also

`Delete Input`, `Object Menu`, and `Terminals`.

DeMultiplexer

An object that directs the input value to a selected output pin.

Use

Use `DeMultiplexer` to direct the input data to a particular output pin. The output that is activated depends on the value of the address input.

Location

`Device` ⟹ `DeMultiplexer`

Notes

`DeMultiplexer` has two inputs, one for data output and one for the address value (which determines the output to be propagated). Only one output is propagated each time the object operates. If the value of the address input is not within the range of the number of outputs [0⟹(N-1)], an error is returned. Additional outputs can be added to the object. Outputs can be deleted, but then the following outputs (if any) are renumbered in order.

See Also

`If/Then/Else` and `JCT`.

Detail

A button that toggles the view displayed from the panel view to the detail view.

Use

When pressed, `Detail` shows the detail view of the model or `UserObject` you created. `Detail` is also available on `UserObjects`.

Location

The upper left side of the title bar.

Notes

`Detail` is only visible after you've created a panel view.

After you've secured a panel view, the `Detail` button is not visible.

See Also

`Add to Panel`, `Panel`, and `Secure`.

Device Event

An object that captures events generated by HP-IB or VXI devices. This feature is available in HP VEE-Test Only.

Use

The `Device Event` object can be configured to detect various device events. Depending upon configuration, the object may wait for the event, giving up execution to other parallel threads, or it may simply return a boolean (`0` or `1`) or other indicator of the state of the device.

If the `Device Event` object has been configured to wait for the event, then upon the event the object will execute. Any thread hosted by this object will have priority over any other parallel threads, and will execute to completion. If the `Device Event` object has been configured to simply return an indicator of device state, the thread containing it will have normal priority, and will execute in parallel with any other threads.

Location

`I/O` ⟹ `Advanced I/O` ⟹ `Device Event`

Open View Parameters

- `Device` - The currently selected device is displayed in the `Device` field. Click on this field to display a list of the currently configured HP-IB and VXI devices.
- `Event` - Allows selection of specific events. Click on this field to display the choices:
  - `Spoll` - Returns the status byte of HP-IB devices and message-based VXI devices. If an HP-IB device has been selected, the only possible selection is `Spoll`. The returned status byte is used with the `Mask` value.
  - `SRQ` - For VXI, an `SRQ` is generated from a message-based device when the device sends either a VME interrupt or a signal register write that is a request-true event. When this asynchronous event occurs, the device's status byte is returned, thus causing the device to generate a request-false event. The returned status byte is used with the `Mask` value.

- Signal Interrupt - This asynchronous event is generated from either message-based or register-based VXI devices. The event occurs when the device sends either a VME interrupt or a signal register write that is a device-defined event or an undefined event. The returned 32-bit value is the value placed in the signal write register, or the value placed on the VXI Bus when the device does an interrupt acknowledge (IACK). This value is used with the Mask value.

- Mask - You can enter a value to be used as a bit mask (default = 0). The Mask value is used in a logical AND operation with the value returned by the device when the specified event occurs.

- Action - Determines the execution behavior of the Device Event object upon detection of the specified event, in conjunction with the specified bit mask.

  - No Wait - Execute immediately, outputting the current state of the configured event, as specified by a 32-bit integer, to the status pin. This value is either the boolean value FALSE (0) or a device-dependent bit pattern. If the event choice is Spoll, the low byte of the returned integer is the status byte of the device. For the two asynchronous VXI events, a FALSE (0) is returned if the event has not occurred. If the event has occurred, the device-dependent bit pattern is returned.

  - Any Set - Wait for the event to occur and use the returned value in a logical AND operation with the Mask value. If any bit is set in the resulting bit pattern, then execute by placing the original, unmasked value on the status output pin.

  - All Clear - Wait for the event to occur and use the returned value in a logical AND operation with the Mask value. If no bit is set, then execute by placing the original, unmasked value on the status output pin.

Notes

The execution behavior of the Device Event object can either be asynchronous or synchronous as determined by the choice of the Action parameter. No Wait specifies a synchronous, polling behavior. The object executes immediately and any attached thread will have the same priority as all other threads currently executing. Choosing Any Set or All Clear causes the Device Event object to wait until both the event has occurred, *and* the logical AND of the Mask value

and the value returned by the device satisfies the "any bit set" or "no bit set" criteria. When both of these conditions have been satisfied, the object executes. Any thread attached will have a higher priority, and will execute to completion, blocking all other concurrently executing threads.

Strictly speaking, the `Spoll` event is not a true asynchronous event as are the VXI `SRQ` and `Signal Interrupt`. By using a `Mask` value and the appropriate action, `Any Set` or `All Clear`, the status byte can be evaluated continuously with the `Mask` value until the correct bits are modified by the device. The object will wait until this occurs, allowing concurrent threads to continue execution. For the `Spoll` event, choosing the `No Wait Action` is identical to specifying the `All Clear Action` with a mask value of zero—the `Device Event` object executes immediately.

For the true asynchronous events `SRQ` and `Signal Interrupt`, the device waits until the event occurs, allowing concurrent threads to continue executing. Once the interrupt occurs, the resulting status byte, signal write register contents, or `IACK` value will be logically ANDed with the mask value. The object executes only if the resulting bit pattern satisfies the `Any Set` or `All Clear` criteria. If the `All Clear` action is specified with a mask value of zero, the `Device Event` object will execute when the interrupt occurs.

See Also

`Interface Event`, `Interface Operations`.

Direct I/O

An object that writes data to and reads data from an instrument using transaction statements.

Use

Use `Direct I/O` to output and input a variety of encodings and formats to instruments.

Location

`I/O` ⟹ `Instrument` ⟹ `Direct I/O`

Object Menu

- `Show Config` - Displays the instrument configuration and allows you to change the instrument parameters.
- `Add Trans` - Adds a transaction to the end (bottom) of the list.
- `Insert Trans` - Inserts a transaction before (above) the currently highlighted transaction.
- `Cut Trans` - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- `Copy Trans` - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- `Paste Trans` - Pastes a transaction, previously "cut" or "copied" to the "cut-and-paste" buffer, in the position before the currently highlighted transaction.

Open View Parameters

`Direct I/O` Action:

- READ: Reads data from an instrument using the specified encoding and format.
- WRITE: Writes data to an instrument using the specified encoding and format.

- EXECUTE: For an HP-IB instrument, sends either a "Selected Device Clear", "Go To Local," "Remote," or "Group Execute Trigger" command. For a serial instrument, sends "Break" or "Reset." For a GPIO instrument, sends "Reset." For a VXI message-based instrument, sends either a "Selected Device Clear", "Go To Local," "Remote," or "Trigger" command.
- WAIT: Waits the specific number of seconds before executing the next transaction. For an HP-IB instrument or a VXI message-based instrument, wait until the bitwise AND of the serial poll response with the mask is either zero (ALL CLEAR) or non-zero (ANY SET).

For GPIO instruments only, the READ action allows IOSTATUS and the WRITE action allows IOCTL.

For VXI instruments only, the READ and WRITE action allows REGISTER: and MEMORY:. The WAIT action also supports the ability to wait until the value in a memory or register offset AND a mask value satisfy the ANYSET, ALL CLEAR, or EQUAL.

Short Cuts

You can quickly add a data terminal by placing the cursor over the input or output terminal display area and then pressing CTRL A. Each press of CTRL A adds an additional data terminal.

You can quickly delete a data terminal by placing the cursor over the terminal view area and then pressing CTRL D.

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`Advanced I/O`, `Component Driver`, `Instrument`, and `State Driver`.

Using HP VEE, chapter 5.

Do

An object that creates a branch point to control the flow of execution of a thread.

Use

Use `Do` to create a branch point; the object connected to the data output pin is activated before the object connected to the sequence output pin. Note that this forces the order in which objects operate.

Location

`Flow` ⟹ `Do`

See Also

`Break` and `Start`.

Edit UserFunction

A menu selection that allows you to edit a locally-created User Function within a model.

Use

Use `Edit UserFunction` to edit a User Function created locally within a model by executing `Make UserFunction` from the object menu of a `UserObject`. If you have saved a series of User Functions in a library file, you will have to open the library file to edit those User Functions.

Location

`Edit` ⟹ `Edit UserFunction`

Notes

When you select `Edit UserFunction`, a dialog box appears listing the available local User Functions to edit. Once you select a User Function, a dialog box displays a workspace showing the functionality of the original UserObject from which the User Function was created. You can edit this workspace just as you would edit any UserObject. When you have finished, press `Done` to finish the edit session. When you save the file, the edited User Function will be saved with it.

See Also

`Call Function`, `Delete Library`, `Import Library`, `User Function`, and `UserObject`.

Enum

An object that outputs an enumerated value from a user-defined list.

Use

Use **Enum** to set an enumerated constant or to get user input that is chosen from a list of acceptable choices. To input an array, press tab to enter the next value. The first item in the list is assigned ordinal position 0; the nth item in the list is assigned ordinal position n-1.

Location

`Data` ⟹ `Enum`

Object Menu

- `Auto Execute` - If set, the object operates whenever the values in the field are edited.
- `Edit Enum Values` - Allows you to add valid selections to the enumerated list of choices or edit existing choices.
- `Format` ⟹- Specifies the appearance of the **Enum** object and the method of selecting a choice. Default is `List`.
  - `List` - The enumerated choices are presented in a list box when the button field is pressed. The selection is made when the choice is clicked on.
  - `Cyclic` - The enumerated choices cycle as the the button field is clicked on. This choice is best used if there are two choices such as On/Off or Stop/Go or in a dialog box where the choice is confirmed by another button.
  - `Buttons` - The enumerated choices are all displayed. The selection is made when the radio button next to the choice is pressed.
- `Initialize` - Used to set this object to a particular value at PreRun and/or Activate time.
  - `Initial Value` - A dialog box that specifies the value to be set. Default value is the zero value of that container type.

- `Initialize At PreRun` - Whether to set the `Initial Value` at PreRun time. Default is off.
- `Initialize At Activate` - Whether to set the `Initial Value` at Activate time. Default is off.

Notes

`Initialize` is most often used for initializing values inside `UserObject`.

The other method for setting an initial value is the `Default Value` control pin. The `Default Value` pin allows you to programmatically change the current value.

`Enum` containers are automatically converted into `Text` for many objects, including `Collector`, `Concatenator`, `Sliding Collector`, and most `Math` objects.

You must add values to each enumerated field to create a list of valid choices.

Example

To use `Enum` as a prompt on a user panel, change the name of the `Enum` object to a prompt such as "Choose the type of test:". The user selects an input that depends on the format of the `Enum` object.

See Also

`Complex`, `Constant`, `Coord`, `Date/Time`, `Integer`, `Number Formats`, `PComplex`, `Real`, `Text`, and `Toggle`.

Escape

This object has been renamed to `Raise Error`.

Execute Program

An object that spawns a child process, either directly or through a command shell.

Use

Use `Execute Program` to communicate with programs external to HP VEE. Communication to the external program is handled through the operating system mechanism of stdin and stdout.

Use `Execute Program` to run any executable user written program or operating system commands.

In general, you need to add or modify transactions to accomplish useful results. To add a transaction, select `Add Trans` in the object menu. To edit a transaction, double-click on the transaction and complete the resulting dialog box.

Location

`I/O` ⟹ `Execute Program`

Object Menu

- `Config` - Allows you to view and edit the formatting configuration for data transmitted to and from the spawned process.
- `Add Trans` - Adds a transaction to the end of the list.
- `Insert Trans` - Inserts a transaction before the currently highlighted transaction.
- `Cut Trans` - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- `Copy Trans` - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- `Paste Trans` - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Open View Parameters

- `Shell` - The operating system shell to communicate with. If the `Shell` field is set to `none`, the first token in the `Pgm with params` field is assumed to be the name of an executable file. Each token after the first is assumed to be a command-line parameter. The executable is spawned directly as a child process of HP VEE. All other things being equal, `Execute Program` operates fastest when `Shell` is set to `none`.

  If the `Shell` field specifies a shell, HP VEE spawns a process corresponding to the specified shell. The string contained in the `Pgm with params` field is passed to the specified shell for interpretation. Generally, the shell spawns additional processes.

- `Wait for child exit` - If `Wait for child exit` is set to `Yes`, HP VEE performs the following tasks:

  1. Spawns a child process.

  2. Executes all transactions specified in the `Execute Program` object.

  3. Closes all pipes to the child process (thus sending an EOF to the child).

  4. Waits until the child process terminates before activating the sequence output pin of the `Execute Program` object.

  If `Wait for child exit` is set to `No`, HP VEE performs the following tasks:

  1. Checks to see if a child process corresponding to the `Execute Program` object is active. If one is not already active, HP VEE spawns one.

  2. Executes all transactions specified in the `Execute Program` object.

  3. Activates the sequence output pin of the `Execute Program` object. The child process remains active and the corresponding pipes still exist.

  All other things being equal, `Execute Program` operates fastest when `Wait for child exit` is set to `No`.

- `Pgm with params` - The program you want to run.

  Here are examples of what you typically type into the `Pgm with params` field:

  To run a custom C program (`Shell field set to none`):

  ```
  MyProg -optiona -optionb
  ```

Execute Program

To run a shell command (`Shell` set to `ksh`):

```
ls -t *.dat
```

To run a program using one of the shell dependent features, you could set `Shell` to `sh` and enter the command into the `Shell command:` line:

```
MyProg *.dat
```

This forces the shell to pass as command line parameters all the files that end in ".dat".

A maximum of 256 parameters may be passed.

Notes

If you use shell-dependent features in the `Pgm with params` field, you must set a shell in the `Shell` field to achieve the desired result. Common shell-dependent features are:

- Standard input/output redirection (`<` and `>`).
- File name expansion using wildcards (`*`, `?`, and `[a-z]`).
- Pipes (`|`).

When sending input to a shell command such as `sort` or `wc`, you must include an `EXECUTE CLOSE WRITE PIPE` transaction after the data has been written to the command. Closing the write pipe will let the command know that it has received all of the data to process.

When reading an arbitrary amount of data back from a shell command such as `ls` or `grep`, use a `READ ... ARRAY 1D TO END: (*)` transaction. It will continue to read data until the pipe reading data from the shell is closed.

Example

Here is a simple C program to read a number from stdin, add one to it, and send it back to HP VEE by way of stdout. The stdout buffering must be turned off. This is done in one of two ways. The first way is with the "setbuf(stdout,NULL)" statement, which turns buffering off. The second technique is to leave the stdout buffering on, then use the "fflush(stdout)", command to flush the output buffer back to HP VEE.

```
#include <stdio.h>

main ()

{

     int c;
     double val;
     setbuf(stdout,NULL); /* turn stdout buffering off */
     while (((c=scanf("%lf",&val)) !=EOF) & c > 0) { /* read a value */
       fprintf(stdout,"%g\n",val+1;  /* add one to the input value */
       fflush(stdout);  /* force output back to main pgm */
     }
     exit(0);
}
```

Short Cuts

You can quickly add a data terminal by placing the cursor over the input or output terminal display area and then pressing CTRL A. Each press of CTRL A adds an additional data terminal.

You can quickly delete a data terminal by placing the cursor over the terminal view area and then pressing CTRL D.

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`HP BASIC/UX`, `From StdIn`, `To/From Named Pipes`, and `To StdOut`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

Exit

Leaves HP VEE.

Use

Use `Exit` to end this work session with HP VEE.

Location

`File` ⟹ `Exit`

Notes

If your model changes have not been saved when you select `Exit`, you are prompted to save them.

You are not allowed to `Exit` when a model is running. Use the `Stop` object to exit a running model.

Short Cuts

Press CTRL E to exit.

See Also

`Raise Error`, `Exit Thread`, `Exit UserObject`, `New`, `Open`, and `Stop`.

Exit Thread

An object that stops the execution of a thread.

Use

Use `Exit Thread` to terminate the execution of an independent thread. `Exit Thread` only stops the execution of the thread on which it resides.

Location

`Flow` ⟹ `Exit Thread`

See Also

`Break`, `Raise Error`, `Exit UserObject`, `Start`, and `Stop`.

Exit UserObject

An object that stops the execution of a `UserObject`.

Use

Use `Exit UserObject` to stop the execution of all threads in the `UserObject` and start execution in the next object after the `UserObject` in the model. Any data sent to the `UserObject`s output terminals before the `Exit UserObject` was encountered, is passed to the appropriate succeeding object(s) by activating these outputs.

Use `Exit UserObject` to make an early exit from a `UserObject`.

Location

`Flow` ⟹ `Exit UserObject`

See Also

`Create UserObject`, `Raise Error`, `Exit Thread`, `Start`, `Stop`, and `UserObject`.

For Count

An object that activates its data output pin a specified number of times.

Use

Use `For Count` to execute a subthread a specified number of iterations. The output of a `For Count` can be used as a value for succeeding objects in the subthread.

Location

`Flow` ⟹ `Repeat` ⟹ `For Count`

Open View Parameters

Enter the number of iterations in the entry field or as an input. The value must be an Real.

Notes

The `For Count` output value is a `Real` scalar ranging from `0` to `n-1` where `n` is the specified count value. All access to unmapped array objects is zero-based, therefore `For Count` is ideally suited for traversing the dimensions of such objects when required. The `For Count` value is the dimension size.

If the specified `For Count` value is zero or negative, the output is not activated, but the sequence output pin is.

Execution of the subthread hosted by the `For Count` output continues until one of the following occurs:

- All objects that can, have operated. The subthread is deactivated, the iteration counter is incremented, and, if the count value is less than the specified count, the subthread is reactivated by refiring the `For Count` output with the new iteration value.
- A `Break` object operated. The subthread is deactivated and the sequence output pin is activated. Note that the value that remains on the `For Count` output is the same value present when the `Break` was encountered.

- A `Next` object operated. The subthread is deactivated and the iteration counter is incremented. If the count value is less than the specified `Count`, the subthread is reactivated by refiring the `Count` output with the new iteration value, and execution of the subthread proceeds as usual. If the iteration counter has reached the `Count` value, the sequence output pin is activated.

When the subthread hosted by the `For Count` object finishes an iteration, all data containers sent during the previous iteration are invalidated before the next iteration. This prevents "old" data from a previous iteration from being reused in the current iteration. However, if flow branching is present within the iterative subthread, some objects may not execute on every iteration. Thus, data containers sent by those objects may be invalidated before other objects can execute on the data. To obtain the desired propagation in this case, use the `Sample & Hold` object. Refer to "Iteration with Flow Branching" in chapter 4 of *Using HP VEE* for more information.

See Also

`Break`, `For Log Range`,`For Range`, `Get Values`, `Next`, `On Cycle`, `Sample & Hold`, `Set Values`, and `Until Break`.

For Log Range

An object that activates its data output pin a specified number of times.

Use

Use `For Log Range` to execute a subthread for a specified number of iterations. Use `For Log Range` to specify a beginning, end, and increment, and generate output values that are evenly distributed along the log 10 scale.

Location

`Flow` ⟹ `Repeat` ⟹ `For Log Range`

Open View Parameters

- `From` - The beginning value of the iteration loop.
- `Thru` - The ending value of the iteration loop. When the value of the iterator is greater or equal to the `Thru` value, the loop is completed.
- `/Dec` - Determines the spacing of values from the iterator. The 0th value is `From`. The nth value is `From*exp10(n/(/Dec))`.

  `From`, `Thru`, and `/Dec` may be added as inputs.

Notes

The `For Log Range` output value is a Real scalar ranging from `From` to `Thru`.

Execution of the subthread hosted by the `For Log Range` output continues until one of the following occurs:

- All objects that can, have operated. The subthread is deactivated, the iteration counter is incremented by the `/Dec` value, and, if the count value is less than the `Thru` value, the subthread reactivates the `For Log Range` output with the new iteration value.
- A `Break` object operated. The subthread is deactivated and the sequence output pin is activated. Note that the value that remains on the `For Log Range` output is the same value present when the `Break` was encountered.

For Log Range

- A `Next` object operated. The subthread is deactivated and the iteration counter is incremented by the `/Dec` value.. If the count value is less than the `Thru` value, the subthread reactivated the `Data` output with the new iteration value, and execution of the subthread proceeds as usual. If the iteration counter has reached or exceeded the `Thru` value, the sequence output pin is activated. If the counter is exceeded, then the value that remains on the DATA output is an out-of-range value.

When you add any of the optional data inputs, they do not necessarily line up with the names of the fields to which they supply a value. Make sure you know which values you are inputting and the pins associated with them. (Use `Show Terminals` if necessary.)

If the `Thru` value is less than `From` value, `/Dec` must be negative or the loop does not execute.

If `/Dec = 0` the loop iterates infinitely or until a `Break` object is encountered in the subthread.

If `From` equals `Thru`, the loop executes once.

When the subthread hosted by the `For Log Range` object finishes an iteration, all data containers sent during the previous iteration are invalidated before the next iteration. This prevents "old" data from a previous iteration from being reused in the current iteration. However, if flow branching is present within the iterative subthread, some objects may not execute on every iteration. Thus, data containers sent by those objects may be invalidated before other objects can execute on the data. To obtain the desired propagation in this case, use the `Sample & Hold` object. Refer to "Iteration with Flow Branching" in chapter 4 of *Using HP VEE* for more information.

See Also

`Break`, `For Count`, `For Range`, `Next`, `On Cycle`, `Sample & Hold`, `Show Terminals`, and `Until Break`.

Formula

An object that performs a user-defined mathematical operation specified by a mathematical formula.

Use

Use `Formula` to execute any desired mathematical formula using the syntax defined in chapter 3. The default `Formula` object has one input (`A`) and an example formula (`2*A+3`). However, the example can be replaced with any desired formula, and any number of inputs can be added as required by the user-defined formula. (If the formula requires no inputs, you may delete all inputs.)

The input(s) can be of any data type. For a relational formula (e.g. `A <= B`), the resulting output is a scalar Int32 with the value `0` ("false") or `1` ("true"). For other mathematical formulas, the resulting output is of a data type consistent with the inputs and with the formula itself. Refer to the section "Mathematically Processing Data" in chapter 3 for a discussion of the rules that determine the data type and shape of the result of a formula.

Location

`Math` ⟹ `Formula`

Open View Parameters

- *Formula field* (unlabeled) - Default is `2*A+3`. You can enter any mathematical formula consisting of the operators and functions described in chapter 3. When you enter a formula, its syntax is automatically checked and any errors are reported. The *Formula field* can be added as a control input.

Notes

All of the supported dyadic operators and functions are described in chapter 3. Any of these operators and functions, including calls to User Functions, compiled functions, and remote functions can be used in a `Formula` object to construct a mathematical formula using the described syntax.

The other `Math` and `AdvMath` objects are really just simple examples of `Formula` objects, but only the `Formula` object allows you to edit the mathematical relationship in the open view.

Although you could perform a mathematical calculation using several individual `Math` and `AdvMath` objects, a single `Formula` object can perform the same calculation with higher performance. This is because a fixed amount of overhead is associated with each graphical object in the model, so a model with fewer objects will run faster.

See Also

Chapter 3, "Formula (Math and AdvMath) Reference", `User Function`

For Range

An object that activates the data output a specified number of times.

Use

Use `For Range` to execute a subthread a specified number of iterations specified by a beginning value, ending value, and increment. Use `For Range` to generate values that are evenly distributed between the `From` and `Thru` values.

Location

`Flow` ⟹ `Repeat` ⟹ `For Range`

Open View Parameters

- `From` - The beginning value of the iteration loop.
- `Thru` - The ending value of the iteration loop. When the value of the iterator is greater than or equal to the `Thru` value, the loop is completed.
- `Step` - The increment value between the current value of the iterator and its next value.

All open view parameters are available from data inputs.

Notes

The `For Range` output value is a `Real` scalar ranging from the `From` to the `Thru` value.

Execution of the subthread hosted by the `For Range` output continues until one of the following occurs:

- All objects that can, have operated. The subthread is deactivated, the iteration counter is incremented by the step value, and, if the count value is less than the `Thru` value, the subthread is reactivated by refiring the `For Range` output with the new iteration value.
- A `Break` object operated. The subthread is deactivated and the sequence output pin is activated. Note that the value that remains on the `For Range` output is the same value present when the `Break` was encountered.

- A `Next` object operated. The subthread is deactivated and the iteration counter is incremented by the `Step` value. If the count value is less than the specified `Thru`, the subthread is reactivated by refiring the `Data` output with the new iteration value, and execution of the subthread proceeds as usual. If the iteration counter has reached the `Thru` value, the sequence output pin is activated.

When you add any of the optional data inputs, they do not necessarily line up with the names of the fields to which they supply a value. Make sure you know which values you are inputting and the pins associated with them. (Use `Show Terminals` if necessary.)

If the `Thru` value is less than the `From` value, `Step` must be negative, otherwise the output is never activated.

If `Step = 0`, the loop iterates infinitely or until a `Break` object is encountered on the subthread.

If `From` equals `Thru`, the loop executes once.

When the subthread hosted by the `For Range` object finishes an iteration, all data containers sent during the previous iteration are invalidated before the next iteration. This prevents "old" data from a previous iteration from being reused in the current iteration. However, if flow branching is present within the iterative subthread, some objects may not execute on every iteration. Thus, data containers sent by those objects may be invalidated before other objects can execute on the data. To obtain the desired propagation in this case, use the `Sample & Hold` object. Refer to "Iteration with Flow Branching" in chapter 4 of *Using HP VEE-Engine and HP VEE-Test* for more information.

See Also

`Break`, `For Count`, `For Log Range`, `Next`, `On Cycle`, `Sample & Hold`, `Show Terminals`, and `Until Break`.

From

A menu item.

Use

Use `From` to access the following objects which are sources for I/O operations:

- `File`
- `DataSet`
- `String`
- `StdIn`

Location

`I/O` $\Longrightarrow$ `From` $\Longrightarrow$

See Also

`To`.

From DataSet

An object that allows the user to conditionally retrieve records from a data set.

Use

Use `From DataSet` to find and retrieve from a data set either one record or all records that meet the constraints of a logical expression formula. If the record retrieved meets the constraints of the query expression, the record is output on the `Rec` output pin.

The default expression in the `Formula` field is `1`, (true). Thus, either the first record found (`One`), or all records in the data set (`All`) will be output.

If you want to test values in record fields, you must use the form `Rec.A` for field `A`, `Rec.B` for field `B`, and so forth. For example, the expression:

```
Rec.A<10
```

tests each record to see if field `A` is less than ten. If `Get records: One` is specified, the first record in the dataset with field `A` less than 10 will be output on `Rec`. If `Get records: All` is specified, all records with field `A` less than 10 will be output. Note that the entire record will be output.

The following control pins may be added:

- `File Name` - This control pin allows the user to change the name of the file from which the records will be read.
- `Rewind` - This control pin "rewinds" the `From DataSet` file so that it can be re-read from the beginning of the file.
- `Formula` - This control pin allows the user to change the formula which is used to determine which records read from the file will be output by the device.

Location

`I/O` ⟹ `From DataSet`

Open View Parameters

- `From DataSet` - Click on the name field to display a dialog box, then select the name of the file that contains the DataSet.
- `Get Records` - Select `One` or `All`:
  - If `One` is selected, the first record found in the data set that meets the constraints of the selection formula will be output, and `From DataSet` will stop executing. When `One` is selected, the output shape is always a scalar.
  - If `All` is selected, all records found in the data set that meet the constraints of the selection formula will be output. The output shape will always be an array. `From DataSet` will stop executing when it reaches the end of the file.
- `Formula` - Enter a mathematical expression to test the records in the data set. Either the first `One`, or `All` records that satisfy the expression will be output, depending on the `Get Records` selection. The default expression is `1` (true).

Notes

You may use any valid mathematical expression in the conditional formula. You may add additional inputs for use in the expression, just as for a `Formula` object. If you want to test values in record fields, express them as `Rec.A`, `Rec.B`, and so forth. The expression can be fairly complex, for example:

```
Rec.A > Rec.B AND Rec.C < 2.3
```

You must observe the normal stipulations for determining True or False for conditional expressions:

- For any relational test (for example, equality between two operands), if one operand is an array, the other must be either a scalar or an array of the same size and shape. If the first operand is equal to the second, the result is True; otherwise it is False.
- If both operands are of type Coord, they must have all of their independent variables and dependent variables match exactly for the result to be True. If independent variables do not match, an error is returned. Complex,

PComplex, and Spectrum operands must have both parts match for the operation to return True. Enums are converted to Text for comparison.

- Arrays must have all the respective values of both operands equal for the operation to return True.

You may wish to bring in a DataSet (an array of records) from a file, edit it, and then save it back to the file. This can be accomplished with a `From DataSet`, a `Record Constant` using the `Default Value` input pin, and a `To DataSet`. See `Record Constant` for more information.

You may add an `EOF` output pin to this object. When the object executes and there are no records meeting the query expression constraints, or you are already at the end of the DataSet, this pin will fire. Otherwise, an `End of file with no Data Found` error will result when this condition occurs.

If you have multiple `From DataSet` objects reading from the same file, they will share the file pointer. That is, all of the objects read sequentially through the file, no matter how many objects there are reading the file. See `From File` for more detail.

See Also

`Build Record`, `Record Constant`, `To DataSet`, and `UnBuild Record`.

From File

An object that reads data from a file using I/O transactions.

Use

Use `From File` object to read a wide variety of file encodings and formats. `From File` is especially useful for importing data files generated by other software packages.

When several `From File` objects refer to the same disc file, a single read pointer is maintained for all such objects even if they are in different threads or different contexts. This means that data read by one `From File` object is not reread by another `From File` object using the same file unless a `REWIND` operation has been done on the file. The file is opened and the file pointer is reset to the beginning of the file (implied `REWIND`) when the first `From File` object for that file operates after PreRun. All files are closed when the model stops.

`From File` Actions:

- `READ` - Reads data from a file using the specified encoding and format.
- `EXECUTE` - `REWIND` repositions the file's read pointer to the beginning of the file without erasing the contents of the file. `CLOSE` closes an open file.
- `WAIT` - Waits the specified number of seconds before executing the next transaction.

`From File` Encodings:

- `TEXT` - Reads text ASCII files written by HP VEE or other software packages.
- `BINARY` - Reads all data types from a machine-specific binary format.
- `BINBLOCK` - Reads all HP VEE data types from binary files with IEEE 488.2 definite length block headers.
- `CONTAINER` - Reads all data types from a machine-specific text format.

Location

I/O ⟹ From ⟹ File

Open View Parameters

Open View provides a field for the data file name, followed by a list of available transactions. The data file name can be added as a control input.

Open View shows the list of file transactions to be executed.

Object Menu

- Add Trans - Adds a transaction to the end (bottom) of the list.
- Insert Trans - Inserts a transaction before (above) the currently highlighted transaction.
- Cut Trans - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- Copy Trans - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- Paste Trans - Pastes a transaction, previously "cut" or "copied" to the "cut-and-paste" buffer, in the position before the currently highlighted transaction.

Notes

An EOF (End of File) output terminal can be added with Add Data Output. EOF is a special output terminal that propagates with a nil container when a transaction attempts to read data past the End of File. When an End of File condition is detected and the EOF output is activated, none of the other data outputs are activated, even if they contain newly read data.

If EOF is detected and there is no EOF output pin, an error is returned.

To read all of the data in the file, use a READ ... ARRAY TO END: transaction.

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`To File` and `To/From Named Pipe`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

From StdIn

An object that reads data from the operating system standard input using transactions.

Use

Use `StdIn` object to read a wide variety of file encodings and formats from the standard input of HP VEE.

All `From StdIn` objects use the same read pointer, even if they are in different threads or different contexts. This means that data read by one `From StdIn` object is not re-read by another `From StdIn` object.

To place data in HP VEE's standard in pipe, you can either type lines in the shell window where the HP VEE process was started or start the HP VEE process using a pipe command to redirect standard input. For example,

```
cat someFileName | veetest
```

sends the contents of file `someFileName` into HP VEE's standard input.

To read all of the data from standard input until standard input is closed use a `READ ... ARRAY TO END:` transaction.

`From StdIn` Actions:

- `READ` - Reads data using the specified encoding and format.
- `WAIT` - Waits the specified number of seconds before executing the next transaction.

`From StdIn` Encodings:

- `TEXT` - Reads all data types from an ASCII-data stream.
- `BINARY` - Reads all data types from an machine-specific binary format.
- `BINBLOCK` - Reads all HP VEE data types from binary data files with IEEE 488.2 definite length block headers.
- `CONTAINER` - Reads all data types from an HP VEE specific text format.

To help prevent a `READ` transaction from hanging until data is available on `stdin`, use a `READ IOSTATUS DATA READY` transaction in a separate `From StdIn`

object. This transaction returns a `1` if there is at least one byte to read, and a `0` if there are no bytes to read.

Location

`I/O` ⟹ `From` ⟹ `StdIn`

Open View Parameters

The open view shows the list of transactions to be executed.

Object Menu

- `Add Trans` - Adds a transaction to the end of the list.
- `Insert Trans` - Inserts a transaction before the currently highlighted transaction.
- `Cut Trans` - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- `Copy Trans` - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- `Paste Trans` - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`From File`, `To StdOut`, and `To/From Named Pipe`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

From String

An object that reads data from a string by using transactions.

Use

Use `From String` to read a wide variety of encodings and formats from textual data in a string.

If a one-dimension array of Text is passed to the `AString` input, the `From String` transactions will act in the same way as if those lines of text were being read from a file with the `From File` object. That is, an array of Text will be treated as a single stream of text with new-line characters between each string element.

From String Actions:

- READ - Reads data using the specified encoding and format.
- WAIT - Waits the specified number of seconds before executing the next transaction.

From String Encodings:

- TEXT - Reads all data types from the AString input.

Location

`I/O` ⟹ `From` ⟹ `String`

Open View Parameters

The open view shows the list of transactions to be executed.

Object Menu

- `Add Trans` - Adds a transaction to the end of the list.
- `Insert Trans` - Inserts a transaction before the currently highlighted transaction.
- `Cut Trans` - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.

- `Copy Trans` - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- `Paste Trans` - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Notes

`From String` is a useful debug tool to explore how READ TEXT transactions operate. Connect a `Text Constant` object to the `From String` input terminal labeled `AString`, and connect a `Logging AlphaNumeric` display to the `From String` output terminal to immediately view the results.

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`From`, `From File`, `From StdIn`, `To/From Named Pipe`, and `To String`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

Function Generator

An object that outputs a Waveform.

Use

Use `Function Generator` to generate a Waveform of your own specifications.

Location

`Device` ⟹ `Virtual Source` ⟹ `Function Generator`

Object Menu

- `Error on Aliasing` - Evaluates whether `Num Points` divided by `Time Span` (same frequency) is less than twice the specified frequency. If it is, an error is returned. The purpose of this evaluation is to determine whether the points being generated provide an accurate representation of the function being generated. An unchecked checkbox generates a literal presentation of points. Default is on (checked).

Open View Parameters

All of these parameters may be set from the open view or added as data inputs.

- `Function` - The shape of the waveform (`Sine, Square, Triangle`, and so forth). Default is `Cosine`.
- `Frequency` - The frequency of the waveform in Hertz. This parameter is ignored if the `Function` is set to `DcOnly`. Default is 1000 Hertz.
- `Amplitude` - The absolute value of the maximum and minimum values in linear units before any `dcOffset` is added. Default is 1.
- `dcOffset` - The offset of the waveform (the amount it is shifted up or down). Default is 0.
- `Phase Units` - Degrees (`Deg`), radians (`Rad`), or gradians (`Grad`). `Phase Units` overrides the current `Trig Mode` for the phase value. Default is `Deg`.

- `Phase` - The shift of the waveform by a portion of a period. The waveform is shifted toward negative time for positive values of phase. `Phase` is ignored when `Function` is set to `DcOnly`. Default is 0.
- `Time Span` - The duration of the waveform (in seconds). Default is 20m seconds and is set in `Waveform Defaults`.
- `Num Points` - The number of points in the waveform. The time between points in the waveform is `Time Span/Num Points`. `Num Points` must be positive. Default is 256 and is set in `Waveform Defaults`.

Notes

There are the same number of points as sampling intervals. Each point is at the beginning of the sampling interval.

See Also

`Build Arb Waveform`, `Comparator`, `Noise Generator`, `Pulse Generator`, and `Waveform Defaults`.

Gate

An object that passes the input data to the data output pin when the sequence in pin is activated.

Use

Use `Gate` to hold the container until another action occurs (and activates the sequence input pin).

Location

`Flow` ⟹ `Gate`

Notes

If the sequence input pin is not connected, there is no "holdoff" effect. Therefore, `Gate` passes the data from data input pin to data output pin whenever the data input pin is activated.

The data that flows into `Gate` is not stored; the container on the data input pin is the container output when the sequence input pin is activated.

See Also

`DeMultiplexer` and `JCT`.

Get Field

An object that allows the user to unbuild a record and recover one of its fields or sub-fields.

Use

Use `Get Field` to extract a field or sub-field of a record. The `Get Field` object is simply a `Formula` object that is initialized with an input terminal `Rec` and the expression `Rec.A`. Use the "dot" syntax to specify the field that you want to extract from the record input. For example, `Rec.A` extracts the `A` field from the `Rec` record input.

Location

`Data` ⟹ `Access Record` ⟹ `Get Field`

Open View Parameters

- *Formula field* (unlabeled) - The default is `Rec.A`. You can enter any mathematical formula allowed in the `Formula` object, but normally you would want to use the `A.B` dot syntax to extract a field from a record. You can add a `Formula` control input.

Notes

The `Get Field` object is actually just a `Formula` object that has been initialized with a `Rec` input and the expression `Rec.A`. You can use any expression in `Get Field` that you can use in a `Formula` object, but to extract a field you will need to use the "dot" syntax like `Rec.A` or `A.B.C`.

The `A.B` dot syntax in `Get Field` is fully supported in all `Formula` objects. These include `Formula`, `If/Then/Else`, `Get Values`, and all transaction devices (`To File`, `To String`, etc.) See "Using Records in Expressions" in chapter 3 of this manual for further information about this syntax.

Note the distinction between the `Get Field`, `UnBuild Record`, and `SubRecord` objects.

- The `Get Field` object works like a `Formula` object, in that it uses dot syntax (`A.B`) to unbuild a record. Note that `Get Field` allows you to unbuild a record of records in one step by using an expression such as `A.B.C`. This process would require two `UnBuild Record` objects.
- The `UnBuild Record` object has outputs for the `Name List` and `Type List` of the input record fields. The other (optional) outputs `A`, `B`, etc. of the `UnBuild Record` object return the same results as would multiple `Get Field` objects.
- The `SubRecord` object differs from `UnBuild Record` and `Get Field` in that its output is always a record. The `SubRecord` device allows you to either *include* or *exclude* a list of fields from a record to form a subrecord.

See Also

`Build Record`, `Formula`, `Record Constant`, `Set Field`, `SubRecord`, and `UnBuild Record`.

Get Global

An object that gets the value of a global variable that has already been created in the execution of the current model.

Use

Use the `Get Global` object to get a global variable (by name), for use as data in an HP VEE model. The global variable must have previously been created by a `Set Global` object within the model.

Global variables created (with `Set Global`) in one context of a model can be used as data in another context of that model. For example, a `Set Global` in the root context of a model could create a global variable, which could then be used as data by including a `Get Global` in a `UserObject`. This is especially useful when the model contains several nested layers of `UserObjects`.

Global variables that are set with `Set Global` may also be used by name in the expression fields of the following objects: `Formula`, `If/Then/Else`, `Get Values`, `Get Field`, and all objects using expressions in transactions, including `To File`, `From File`, `Direct I/O`, `From Stdin`, `To/From Named Pipes`, and `Sequencer`. Refer to "Using Global Variables in Expressions" in chapter 3 for further information.

Location

`Data` ⟹ `Globals` ⟹ `Get Global`

Open View Parameters

The open view displays a field for the name of the global variable. The name is not case sensitive (either lower-case or upper-case letters may be used). Thus `globalA` is the same variable as `GLOBALa`. The name field may be added as a control input.

Notes

To avoid unexpected results, your model must ensure that a global variable is set with `Set Global` *before* a `Get Global` object (or an object that includes the global variable name in an expression) executes. Generally, the best way to ensure this is to connect the sequence output pin of the `Set Global` object to the sequence input pin of the `Get Global` object, or other object that uses the global variable. However, there are cases when the sequence input pin need not be connected. For further information about this, refer to "Using Global Variables" in chapter 3 of *Using HP VEE*.

All global variables are deleted at the beginning of every `Run`, `Start`, or auto-execution. Global variables are always deleted by either `File` ⟹ `New` or `File` ⟹ `Open`. Global variable values are not saved with the model.

Global variables are truly global since they are not defined at a `UserObject` level. A global variable that is defined in one context of a model can be used in any other context within the model. For example, you can define a global variable with a `Set Global` object in the root context of the model, and then include a `Get Global` to get that global variable in a `UserObject`. However, you will need to avoid name conflicts throughout the model:

- If two or more `Set Global` objects attempt to set the same global variable (with the same name), the current value will be overwritten as each `Set Global` executes. This may result in unexpected behavior.
- If a there is a local input variable with the same name as a global variable, the local variable will take precedence.

For further information, refer to "Using Global Variables in UserObjects" in chapter 6 of *Using HP VEE*, and to "Using Global Variables in Expressions" in chapter 3 of this manual.

See Also

`Set Global`, and `View Globals`.

Get Mappings

An object that outputs mapping information about a container.

Use

Use `Get Mappings` to get information about mapped arrays. `Get Mappings` outputs the type of mapping (log or linear) and the beginning and ending values of each mapping for each dimension.

Location

`Data` ⟹ `Access Array` ⟹ `Get Mappings`

Open View Parameters

`NumDims` - The number of dimensions in the input array. An Int32 value with a minimum value of `1` and a maximum value of `10`.

Notes

If the input array is not mapped, `Get Mapping` outputs Linear (0,n-1), where "`n`" is the number of points in that dimension of the input array.

`Get Mapping` does not allow a Coord as input, because the Coord data type has explicit mappings.

`Num Dims` must be greater than or equal to `1` and match the number of dimensions in the input array.

If the `Num Dims` value is changed, the view automatically creates or deletes output terminals so that the number of outputs, three for each dimension, matches the value of the `Num Dims` value. For each dimension, `Get Mappings` has three data output terminals.

Short Cuts

To view the mappings of a mapped array, use `Line Probe` on a line or open the terminal to view the container data.

See Also

`Line Probe`, `Set Mappings`, `Show Terminals`, `UnBuild`, and `UnBuild Data`.

Get Values

An object that extracts elements from an array.

Use

Use `Get Values` to access particular elements of an array. You can extract single or multiple elements, columns, or rows from one or more arrays.

Use `Get Values` to get information about a set of data such as: the data type, the number of dimensions, the dimension sizes, and the total size of all the array elements of the input array.

Location

`Data` ⟹ `Access Array` ⟹ `Get Values`

Open View Parameters

Formula Box - Specifies the elements of the array that you are extracting. The specific symbols associated with arrays are:

- Comma (,) - Separates element dimensions.
- Colon (:) - Indicates a set of elements inclusively.
- Asterisk (*) - A wild card to indicate the entire column or row.

Example

To extract the 2nd column from a two dimensional array [3,3] elements, keeping in mind that array indices are zero-based:

```
-       -                          -   -
|1  2  3 |                         | 2 |
|4  5  6 |   Array [*,1] outputs:  | 5 |
|7  8  9 |                         | 8 |
-       -                          -   -
```

Notes

`Get Values` accepts expressions in the entry box.

All arrays are zero-based.

There must be exactly one specification for each array dimension.

For more information on the array syntax, see the `Formula Reference` chapter.

See Also

`Data`, `Set Values`, and `UnBuild Data`.

`Array` in the "Formula Reference" chapter.

Globals

A menu item.

Use

Use `Globals` to access the following objects which set and get the values of global variables.

- `Set Global`
- `Get Global`

Location

`Data` ⟹ `Globals` ⟹

Notes

A global variable that is defined in one context of a model can be used in any other context within the model. For example, you can define a global variable with a `Set Global` object in the root context of the model, and then include a `Get Global` to get that global variable in a `UserObject`. You need to avoid name conflicts throughout the model:

- If two or more `Set Global` objects attempt to set the same global variable (with the same name), the current value will be overwritten as each `Set Global` executes. This may result in unexpected behavior.
- If a there is a local input variable with the same name as a global variable, the local variable will take precedence.

For further information, refer to "Using Global Variables in UserObjects" in chapter 6 of *Using HP VEE*, and to "Using Global Variables in Expressions" in chapter 3 of this manual.

See Also

`Get Global`, `Set Global`, and `View Globals`.

Glossary

Displays definitions of common HP VEE terms.

Use

Use `Glossary` to clarify HP VEE terminology.

Location

`Help` ⟹ `Glossary`

Notes

`Glossary` presents the information from the glossary in the *HP VEE Reference Manual*.

See Also

`Help`.

Help (Object Menu)

Gives help on this object.

Use

Use `Help` to view the reference page for this object. To exit help, press the `Done` button.

Location

On each object menu $\Longrightarrow$ `Help`

Notes

To get help on multiple objects, terminology, or procedural information, use the features under the `Help` menu.

`Help` for `Math` and `AdvMath` objects give the entire list of functions to choose from. `Help` for `State Driver` or `Component Driver` contains help about the driver itself.

See Also

`Glossary`, `How To`, `Object Menu`, and `On Features`.

How To

Displays information summaries to help you build models.

Use

Use `How To` to get summary information to help you build, debug, and document models.

Location

`Help` ⟹ `How To`

Notes

`How To` presents information on model building summaries.

See Also

`Glossary`, `Help`, `On Features`, `On Help`, `On Instruments`, and `Short Cuts`.

HP BASIC/UX

A menu item available in HP VEE-Test running on the HP 9000 Series 300 and 400 only.

Use

Use `HP BASIC/UX` to access the following objects that control HP BASIC/UX programs:

- `Init HP BASIC/UX`
- `To/From HP BASIC/UX`

Location

`I/O` ⟹ `HP BASIC/UX` ⟹

See Also

`Init HP BASIC/UX` and `To/From HP BASIC/UX`.

HP-UX Escape

This object has been renamed to `Execute Program`.

If A == B

An object that branches output if the data input values are equal.

Use

Use `If A == B` to test for equality and branch based on that test.

Location

`Flow` ⟹ `Conditional` ⟹ `If A == B`

Notes

`If A == B` is a preconfigured `If/Then/Else` object that tests the equality of the inputs. You cannot modify the condition or change the data inputs. You can use `If/Then/Else` to change the condition or data inputs.

If the inputs are equal, the `Then` output is propagated with an Int32 scalar `1`, else the `Else` output is propagated with an Int32 scalar `0`.

See Also

`Conditional`, `If/Then/Else`, `If A != B`, `If A < B`, `If A > B`, `If A <= B`, and `If A >= B`.

`Relational` in the "Formula Reference" chapter.

If A >= B

An object that branches output if one input is greater than or equal to the other.

Use

Use `If A >= B` to test for one input being greater than or equal to the other, and branch based on that test.

Location

`Flow` ⟹ `Conditional` ⟹ `If A >= B`

Notes

`If A >= B` is a preconfigured `If/Then/Else` object. You cannot modify the condition or change the data inputs. You can use `If/Then/Else` to change the condition or data inputs.

If the value of A is greater than or equal to the value of B, the `Then` output is propagated with an Int32 scalar `1`, else the `Else` output is propagated with an Int32 scalar `0`.

See Also

`Conditionals`, `If/Then/Else`, `If A == B`, `If A != B`, `If A < B`, `If A > B`, and `If A <= B`.

`Relational` in the "Formula Reference" chapter.

If A > B

An object that branches output if one input is greater than the other.

Use

Use `If A > B` to test for one input being greater than the other and branch based on that test.

Location

`Flow` ⟹ `Conditional` ⟹ `If A > B`

Notes

`If A > B` is a preconfigured `If/Then/Else` object. You cannot change the condition or modify the data inputs. You can use `If/Then/Else` to change the condition or data inputs.

If the value of A is greater than the value of B, the `Then` output is propagated with an Int32 scalar `1`, else the `Else` output is propagated with an Int32 scalar `0`.

See Also

`Conditional`, `If/Then/Else`, `If A == B`, `If A != B`, `If A < B`, `If A <= B`, and `If A >= B`.

`Relational` in the "Formula Reference" chapter.

If A <= B

An object that branches output if one input is less than or equal to the other.

Use

Use `If A <= B` to test for one input being less than or equal to the other, and branch based on that test.

Location

`Flow` ⟹ `Conditional` ⟹ `If A <= B`

Notes

`If A <= B` is a preconfigured `If/Then/Else` object which tests for the inequality of the inputs. You cannot modify the condition or change the data inputs. You can use `If/Then/Else` to change the condition or data inputs.

If the value of `A` is less than or equal to the value of `B`, the `Then` output is propagated with an Int32 scalar `1`, else the `Else` output is propagated with an Int32 scalar `0`.

See Also

`Conditionals`, `If/Then/Else`, `If A == B`, `If A != B`, `If A < B`, `If A > B`, and `If A >= B`.

`Relational` in the "Formula Reference" chapter.

If A < B

An object that branches output if one input is less than the other.

Use

Use `If A < B` to test for one input being less than the other, and branch based on that test.

Location

`Flow` ⟹ `Conditional` ⟹ `If A < B`

Notes

`If A < B` is a preconfigured `If/Then/Else` object. You cannot modify the condition or change the data inputs. You can use `If/Then/Else` to change the condition or data inputs.

If the value of `A` is less than the value of `B`, the `Then` output is propagated with an Int32 scalar `1`, else the `Else` output is propagated with an Int32 scalar `0`.

See Also

`Conditional`, `If/Then/Else`, `If A == B`, `If A != B`, `If A > B`, `If A <= B`, and `If A >= B`.

`Relational` in the "Formula Reference" chapter.

If A != B

An object that branches output if the inputs are not equal.

Use

Use `If A != B` to test for inequality and branch based on that test.

Location

`Flow` ⟹ `Conditional` ⟹ `If A != B`

Notes

`If A != B` is a preconfigured `If/Then/Else` object which tests for the inequality of the inputs. You cannot change the condition or modify the data inputs. You can use `If/Then/Else` to change the condition or data inputs.

If the value of `A` is not equal to the value of `B`, the `Then` output is propagated with an Int32 scalar `1`, else the `Else` output is propagated with an Int32 scalar `0`.

See Also

`Conditional`, `If/Then/Else`, `If A == B`, `If A < B`, `If A > B`, `If A <= B`, and `If A >= B`.

`Relational` in the "Formula Reference" chapter.

If/Then/Else

An object that branches execution flow based on the values of its input(s). Formerly called `If/Then`.

Use

Use `If/Then/Else` to make decisions about which subthread branch to execute. You may enter any legal math expression in the formula boxes on the open view.

Only the output pin associated with a true condition is activated. The container output is the value of the expression.

`If/Then/Else` evaluates the expressions in the `If/Else If` clauses (top to bottom) until one evaluates to a non-zero value. This non-zero container is copied to that expression's corresponding output and that one output is propagated. Only one output is propagated. If none of the expressions evaluate to a non-zero value, then the `Else` output is propagated with the zero value of the last evaluated expression.

If any of the expressions evaluate to a value that is an array, the array values are checked to see if they are either all zero or all non-zero. If the values are mixed, an error is returned since it cannot be determined whether the array is `true` or `false`. To produce a scalar, use `expr == 0` or `expr!=0` since the equality relationals always return a scalar.

Location

`Flow` ⟹ `If/Then/Else`

Open View Parameters

Enter an expression in the edit field.

Object Menu

- `Add Else/If` - Adds a condition that is tested if the previous condition is not met and an output pin that is activated if the condition is true.
- `Delete Else/If` - Deletes a condition. After selecting `Delete Else/If`, select a condition to delete from a dialog box that is displayed.

  `Delete Else/If` is not available when only one condition is displayed. This feature is only available if you have at least two `Else/If` conditions.

Notes

You may use any legal mathematical statements in the conditional statement. You may add additional inputs for use in the expressions, just as in the `Formula` box.

The objects under `Conditional` are preconfigured `If/Then/Else` objects that test for a specific condition.

To get a TRUE|FALSE (1|0) output, use the `Relational` objects. They are available under the `Math` menu.

Also note the difference between menu items under `Relational` and under `Conditional`. `Relationals` are formulas with output `0` or `1`. `Conditionals` are `If/Then/Else` and have two outputs, of which one activates.

See Also

`Conditional`, and `Triadic Operator`.

`Relational` in the "Formula Reference" chapter.

Import Library

An object that loads a library of User Functions, Compiled Function definitions, or Remote Function definitions into a running HP VEE model.

Use

Use `Import Library` to load a library of one of the following kinds of functions into your HP VEE model at run time:

- `User Function` - `Import Library` loads all of the User Functions from a specified file. (User Functions are created by selecting the `Make User Function` selection on the object menu of a `UserObject`. You create a library by creating several User Functions and saving them to a file.)
- `Compiled Function` - `Import Library` attaches an *shared library* to the HP VEE process, and parses the `Definition File` declarations. (Compiled Functions are programs written in a programming language such as C, which are dynamically linked to the HP VEE process. Refer to *Using HP VEE* for further information.)
- `Remote Function` - `Import Library` starts an HP VEE process on a remote host and loads the `Remote File` into the HP VEE process. (Remote Functions are actually User Functions loaded by the remote HP VEE process, but callable on the local host. Refer to *Using HP VEE* for further information.)

Once the library of User Functions, Compiled Functions, or Remote Functions is loaded, the functions are executed (called) by name using the `Call Function` object.

Location

`Device` ⟹ `Function` ⟹ `Import Library`

Open View Parameters

- `Library Type` - A dialog box allows you to choose `User Function`, `Compiled Function`, or `Remote Function`.
- `Library Name` - Enter the name of the library that you want to import.
- `File Name` - (User Function or Compiled Function only.) Select the file that contains the library to import.
- `Definition File` - (Compiled Function only.) Select the definition file for the Compiled Function.
- `Host Name` - (Remote Function only.) Enter the name of the remote host where the Remote Function will operate.
- `Remote File Name` - (Remote Function only.) Enter the file name (complete path) of the library (e.g. `/users/MyUserName/MyDir/MyFile`).
- `Remote Timeout` - (Remote Function only.) Set the timeout in seconds for the Remote Function.

Object Menu

- `Load Lib` - Immediately loads the specified library into HP VEE. If a library with the same name was previously loaded, that library is deleted and overwritten with the new library.
- `Delete Lib` - Immediately deletes the specified library from HP VEE. (This provides the same functionality as the `Delete Library` object.)

Notes

The `Import Library` object is generally used for advanced operations where you have developed sets of User Functions, Compiled Functions, or Remote Functions into libraries. `Import Library` allows you to load such libraries dynamically at run time. For User Functions you can save considerable space by creating a library and loading it with `Import Library`—you don't have to save copies of the User Functions in each individual model. Also, when you develop a library of standard User Functions, you can keep the source code for those functions in a single place. Refer to *Using HP VEE* for a

detailed discussion of using User Functions, Compiled Functions, and Remote Functions.

User Functions loaded at run time by `Import Library` operate exactly like any locally-created User Function within the model. You can execute any User Function with the `Call Function` object. However, only locally created User Functions can be edited within the model. If you want to edit any of the external User Functions, you must open the library file that contains the User Functions and use the `Edit User Functions` selection on the `Edit` menu. Once you have edited the User Function, save the file back to the disk.

See Also

`Call Function`, `Delete Library`, `Edit UserFunction`, `User Function`, and `UserObject`.

Init HP BASIC/UX

An object that spawns an HP BASIC/UX process and runs a specified program. `Init HP BASIC/UX` is available in HP VEE-Test on HP 9000 Series 300 and 400 only.

Use

Use `Init HP BASIC/UX` to begin communications with HP BASIC programs.

Enter the complete path and file name and any options for the HP BASIC program you wish to run in the `Program` field. The program may be in either STOREd or SAVEd format. You can use relative paths; these paths are relative from the present working directory where HP VEE started.

`Init HP BASIC/UX` does not provide any data path to or from the HP BASIC process; use `To/From HP BASIC/UX` for that purpose.

Location

`I/O` ⟹ `HP BASIC/UX` ⟹ `Init HP BASIC/UX`

Notes

You can use more than one `Init HP BASIC/UX` object in a model and you can use more than one in a single thread.

There is no direct way to terminate an HP BASIC/UX process from an HP VEE model other than deleting the `Init` object. There are two possible ways to terminate the `HP BASIC/UX` processes:

- Your HP BASIC program runs a QUIT statement when it receives a certain data value from HP VEE.
- An `Execute Program` object kills the HP BASIC/UX process using a shell command, such as `rmbkill`.

When you exit HP VEE, any `HP BASIC/UX` processes still attached are killed.

See Also

`To/From HP BASIC/UX`, `To/From Named Pipe`, and `Execute Program`.

Instrument

Selects an I/O object to control any configured instrument or adds a new instrument to the list of configured instruments. This feature is available in HP VEE-Test only.

Use

Click on `I/O` ⟹ `Instrument` and examine the list of configured instruments in the Select an I/O Device dialog box.

If the instrument you want is not properly configured, go to `Configure I/O` and configure it. If the instrument you want is in the list and is properly configured, follow these instructions:

1. Click once on the desired instrument to highlight it.

2. Click one of the buttons at the bottom of the dialog box to select the type of object you wish to use to control the instrument. If you are not sure which type of object to use to control the selected instrument, try `State Driver`.

If the instrument you want is not in the list or is not properly configured:

1. Click on `Add` to add a new instrument.

2. Complete the resulting dialog boxes. Refer to the `Configure I/O` entry for details about how to complete these dialog boxes.

Location

`I/O` ⟹ `Instrument`

Notes

All instruments should be configured before they can be opened by way of the `Instruments` menu selection. The best way to configure instruments is to use the `Configure I/O` menu selection.

Note that an I/O object may be operated with or without actual instruments connected to the computer. If you wish to control a live instrument, you must set a correct, non-zero address and enable Live Mode. The address and Live Mode setting are controlled by way of the `I/O` ⟹ `Configure I/O` menu

selection. If the address is zero or if Live Mode is off, the instrument object operates but does not attempt to communicate with a physical instrument.

State drivers can be used interactively or within a model. To set the value of an individual component, click on the field containing the value of the component and complete the resulting dialog box. To make a measurement and display the result, click on the corresponding numeric readout or XY display inside the State Driver open view.

It is possible to have more than one object controlling a single instrument. It is also possible to have multiple copies of the same driver, each controlling a different physical instrument. In either case, it is the configured name that determines which object controls which instrument.

Your system administrator must properly configure your computer before it is possible to communicate between HP VEE and any hardware interface. If you believe that you have properly followed all HP VEE procedures properly and you still cannot achieve any level of communication with an instrument, the problem may be with your computer configuration. Ask your system administrator to read this explanation and verify proper configuration of your system's interface drivers. (These interface drivers are different from the instrument driver files included with HP VEE).

See Also

`Advanced I/O`, `Bus I/O Monitor`, and `Configure I/O`.

Using HP VEE, chapter 5.

Integer

An object that outputs a constant integer Scalar or Array 1D. To input an array, press tab to enter the next value.

Use

Use `Integer` to set an integer constant or to get user input.

Location

`Data` ⟹ `Constant` ⟹ `Integer`

Example

To use `Integer` as a prompt on a panel view, change the name of the `Integer` object to a prompt such as `Enter the number of seconds to delay:`. The user fills in the requested information in the entry field.

Type in the formula `1024*1024` into the `Integer` and it will calculate the result.

Object Menu

- `Auto Execute` - If set, the object operates whenever the values in the field are edited.
- `Config` - Sets the initial number of values to be output with this object. A value of `0` sets the container to a scalar, otherwise the container is an array of the length given.
- `Initialize` - Used to set this object to a particular value at PreRun and/or Activate time.
  - `Initial Value` - A dialog box that specifies the value to be set. Default value is zero.
  - `Initialize At PreRun` - Whether to set the `Initial Value` at PreRun time. Default is off.
  - `Initialize At Activate` - Whether to set the `Initial Value` at Activate time. Default is off.

- `Number Formats` - Specifies a different display format.

Notes

`Initialize` is most often used for initializing values inside a `UserObject`.

The other method for setting initial values is the `Default Value` control pin available on most data constants. The `Default Value` pin allows you to programmatically change the current value.

Note that the `Initial Value` field is always a scalar, even if `Integer` is configured to be an array. The `Default Value` input pin, however, requires its input container to exactly match the size and shape of `Integer`.

See Also

`Alloc Integer`, `Complex`, `Constant`, `Coord`, `Date/Time`, `Enum`, `Integer`, `Number Formats`, `PComplex`, `Real`, `Text`, and `Toggle`.

Integer Slider

An object that outputs the Integer value of the slider.

Use

Use `Integer Slider` to input Int32 values. `Integer Slider` is particularly useful on a panel view.

Location

`Data` ⟹ `Integer Slider`

Open View Parameters

The open view displays fields for slider value, min, max, and slider control.

Object Menu

- `Auto Execute` - If set, the object operates whenever the values in the field are edited.
- `Detents` - Sets the distance between values. Any Real `Detents` are truncated to Int32.
- `Initialize` - Used to set this object to a particular value at PreRun and/or Activate time.
  - `Initial Value` - A dialog box that specifies the value to be set. Default value is the zero value of that container type.
  - `Initialize At PreRun` - Whether to set the `Initial Value` at PreRun time. Default is off.
  - `Initialize At Activate` - Whether to set the `Initial Value` at Activate time. Default is off.
- `Number Formats` - Specifies a different display format.
- `Layout` - Specifies either horizontal or vertical slider format.

Notes

`Initialize` is most often used for initializing values inside a `UserObject`.

The other method for setting initial values is the `Default Value` control pin. The `Default Value` pin allows you to programmatically change the current value.

See Also

`Complex`, `Enum`, `Integer`, `PComplex`, `Real`, `Real Slider`, `Text`, and `Toggle`.

Interface Event

An object that captures asynchronous interface events for VXI and HP-IB. This feature is available in HP VEE-Test only.

Use

The `Interface Event` object can be configured to detect various interface events. Depending on the configuration, the object may wait for the event, giving up execution to other parallel threads, or it may simply return a boolean (`0` or `1`) or other indicator of the state of the interface.

If the `Interface Event` object has been configured to wait for the event, then upon the event the object will execute. Any thread hosted by this object will have priority over any other parallel threads, and will execute to completion. If the `Interface Event` object has been configured to simply return an indicator of interface state, the thread containing it will have normal priority, and will execute in parallel with any other threads.

Location

`I/O` ⟹ `Advanced I/O` ⟹ `Interface Event`

Open View Parameters

- `Interface` - The currently selected interface is displayed in this field. Click on this field to display a list of the currently configured interfaces. For an interface to appear in this list, at least one device for that interface must be configured using `Config I/O`. Multiple interfaces of the same type are distinguished by the select code (e.g.: "hpib7" and "hpib8").
- `Action` - Determines the execution behavior of the `Interface Event` object upon detection of the event.
  - `Wait` - Wait for the event to occur. The object will execute when the event occurs. Any thread hosted by this object will have priority over any other parallel threads, and will execute to completion when the event occurs, but is "blocked" until the event occurs. When the event occurs, a boolean TRUE (`1`) or an interface-event-specific bit pattern is output.

- `No Wait` - Execute immediately, placing the current state of the configured event, as specified by a 32-bit integer, on the `event` output pin. This value is the boolean value FALSE (`0`) if the event has not occurred. If the event has occurred, a TRUE (`1`) or an interface-event-dependent bit pattern is returned. In the `No Wait` configuration, any thread hosted by the `Interface Event` object will have normal execution priority, and will execute in parallel with any other threads.

- `Event` - Allows selection of specific interface events. Click on this field to display the choices. For each choice below, both the `Wait` and the `No Wait` actions are described. Note that in each case, the thread propagation priorities for `Wait` and `No Wait` are as described above.

  - `SRQ` - This is the only choice for the HP-IB interface:

    —If the `Action` choice is `Wait`, when an HP-IB device "pulls" on the service request line the `Interface Event` object will execute, placing a boolean value TRUE (`1`) on the `event` output pin.

    —If the `Action` choice is `No Wait`, a boolean value FALSE (`0`) is output if the event has not occurred. If the event has occurred, a boolean TRUE (`1`) will be output on the `event` pin.

  - `Sys Reset` - The `Interface Event` object will execute when a system reset occurs on the VXI interface:

    —If the `Action` choice is `Wait`, a boolean TRUE (`1`) will be output on the `event` pin when the system reset occurs.

    —If the `Action` choice is `No Wait`, a FALSE (`0`) is output if the system reset has not occurred.

  - `Sys Active` - When the VXI Resource Manager has finished its VXI initialization and management, it sends a Begin Normal Operation (BNO) command to the VXI interface. If `Sys Active` is configured, the `Interface Event` object will execute when this occurs:

    —If the `Action` choice is `Wait`, a boolean TRUE (`1`) will be output on the `event` pin when the BNO command occurs.

    —If the `Action` choice is `No Wait`, a FALSE (`0`) is output if the event has not occurred.

- `Sys Deactive` - When normal VXI operation is aborted an Abort Normal Operation (ANO) or End Normal Operation (ENO) command is sent to the interface. If `Sys Deactive` is configured, the `Interface Event` object will execute when this occurs:

  —If the `Action` choice is `Wait`, a boolean TRUE (`1`) will be output on the `event` pin when the ANO or ENO command occurs.

  —If the `Action` choice is `No Wait`, FALSE (`0`) is output if the event has not occurred.

- `TTL Trig` - When `TTL Trig` is configured, the `Interface Event` object "monitors" the VXI TTL trigger lines:

  —If the `Action` choice is `Wait`, the `Interface Event` object executes when a VXI device "pulls" on one or more of the eight TTL trigger lines in the VXI backplane, placing a 32-bit integer on the `event` output pin. Only the low byte of this integer is significant. It is a bit pattern showing which of the eight TTL trigger lines are active (bit 0 corresponds to trigger line 0, bit 7, to trigger line 7). Each bit set (`1`) indicates that the corresponding line has been triggered. For example, `00010001` indicates that TTL lines 0 and 4 have been triggered.

  —If the `Action` choice is `No Wait`, the boolean FALSE (`0`) is output if no triggers have occurred.

- `ECL Trig` - When `ECL Trig` is configured, the `Interface Event` object "monitors" the VXI ECL trigger lines:

  —If the `Action` choice is `Wait`, the `Interface Event` object executes when a VXI device "pulls" on one or more of the four ECL trigger lines in the VXI backplane, placing a 32-bit integer on the `event` output pin. The low byte of this integer is a bit pattern showing which of the four ECL trigger lines are active. (Bit 0 corresponds to trigger line 0, bit 3, to trigger line 3.) Each bit set (`1`) indicates that the corresponding line has been triggered. For example, `00001001` indicates that ECL lines 0 and 3 have been triggered.

  —If the `Action` choice is `No Wait`, the boolean FALSE (`0`) is output if no triggers have occurred. Note that some VXI card cages may choose to implement only two ECL trigger lines.

- `EXT Trig` - When `EXT Trig` is configured, the `Interface Event` object "monitors" the VXI EXT trigger lines:

  —If the `Action` choice is `Wait`, the `Interface Event` object will execute when an external device "fires" on an embedded controller's "Trig In" input. Up to four external trigger inputs can occur. A 32-bit integer is output on the `event` pin when the `Interface Event` object executes. The low byte of this integer is a bit pattern showing which of the four EXT trigger lines are active. (Bit 0 corresponds to external trigger 0, bit 3, to external trigger 3.) Each bit set (`1`) indicates that the corresponding trigger has occurred. For example, `00001001` indicates that EXT trigger lines 0 and 3 have been triggered.

  —If the `Action` choice is `No Wait`, the boolean FALSE (`0`) is output if no triggers have occurred. Some controller implementations may have no external trigger inputs. The HP 75000 Series C Model V382 embedded controller has one trigger input, which corresponds to the least significant bit of the returned 32-bit integer.

- `Unknown Interrupt` - If `Unknown Interrupt` is configured, the `Interface Event` object checks for any VME interrupts from a VME or VXI device not in the VXI controller's servant area:

  —If the `Action` choice is `Wait`, such an interrupt will cause the `Interface Event` object to execute. The 32-bit value returned is the value placed on the VXI Bus when the device does an interrupt acknowledge (IACK).

  —If the `Action` choice is `No Wait`, the boolean FALSE (`0`) is output if no interrupt has occurred.

- `Unknown Signal` - If `Unknown Signal` is configured, the `Interface Event` object checks for any signal register write from a VXI device not in the VXI controller's servant area:

  —If the `Action` choice is `Wait`, such a signal will cause the `Interface Event` object to execute. The 32-bit value returned is the contents of the signal write register.

  —If the `Action` choice is `No Wait`, the boolean FALSE (`0`) is output if no signal has occurred.

Notes

The execution behavior of the `Interface Event` object can either be asynchronous or synchronous, as determined by the choice of the `Action` parameter. `NO WAIT` specifies a synchronous, polling behavior. The object executes immediately, and any attached thread will have the same priority as all other threads currently executing. Choosing `WAIT` causes the `Interface Event` object to wait until the event has occurred. Any thread hosted by the object will have a higher priority, and will execute to completion, blocking all other concurrently executing threads.

See Also

`Device Event`, and `Interface Operations`.

Interface Operations

An object that uses transactions to send low-level interface commands and data by way of HP-IB and VXI. This feature is available in HP VEE-Test only.

`Interface Operations` should be used for interface-specific tasks, or for special cases where communication with devices via `Direct I/O` or an `Instrument Driver` does not afford the necessary level of control.

Use

Use `Interface Operations` for low-level operations such as:

- Sending non-standard HP-IB addressing sequences.
- Sending an HP-IB group trigger message, or pulling on VXI trigger lines.
- Sending arbitrary bytes via HP-IB, with or without ATN true.
- Sending clear and reset commands to either the VXI or HP-IB interfaces.
- Sending IEEE 488.1 defined multi-line messages.

To use `Interface Operations` you must add one or more transactions. To do this, click on `Add Trans` in the object menu. Transactions are listed as lines of text in the open view of the object. To edit a transaction, double click on the transaction and complete the resulting dialog box.

Location

`I/O` ⟹ `Advanced I/O` ⟹ `Interface Operations`

Object Menu

- `Config` - Displays the bus configuration and allows you to change to another interface or to specify a timeout. Multiple interfaces of the same type are distinguished by their select codes (e.g., "hpib7" and "hpib8").
- `Add Trans` - Adds a transaction to the end (bottom) of the list.
- `Insert Trans` - Inserts a transaction before (above) the currently highlighted transaction.

- `Cut Trans` - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- `Copy Trans` - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- `Paste Trans` - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`Advanced I/O`, `Bus I/O Monitor`, and `Configure I/O`.

Chapter 12, "Using Transaction I/O," in *Using HP VEE*.

JCT

An object that outputs the value from the most recent data input (one of multiple inputs) that is activated.

Use

Use JCT like a wired OR to connect the data output pins of two or more objects to one data input pin of another object. JCT is often used in feedback loops for data initialization. JCT passively transmits data.

Location

Flow ⟹ Junction

Notes

JCT is useful for setting up initial conditions.

Inputs to JCT are asynchronous data inputs. This means that JCT operates any time *any* of its data inputs is activated. When more than one pin on the JCT is activated before the JCT can operate, the incoming data is queued up in the order in which it was received. The JCT then activates its output repeatedly much like an iterator until the data queue is empty. If the input data pin is activated by a feedback loop, multiple activations are ignored.

One of the common uses of JCT is in a feedback loop where an initial value is first supplied through one of the inputs, and feedback values are then supplied through one or more of the other data inputs.

See Also

DeMultiplexer.

Layout (Object Menu)

A menu item.

Use

Use `Layout` to access the following features which affect the appearance of the icon:

- `Show Label` - Toggles the display of the object name on the icon.
- `Select Bitmap` - Selects the bitmap displayed on the icon.
- `Delete Bitmap` - Deletes the bitmap displayed on the icon.

Location

On each object menu ⟹ `Layout` ⟹

Notes

`Layout` is only available from the icon view of an object. However, it is not available for the `OK`, `Start`, and `Toggle` icons.

See Also

`Delete Bitmap`, `Object Menu`. `Select Bitmap`, `Show Label`, and `Show Title`.

Line Probe

Displays the container information that is transmitted on a line between two objects.

Use

Use `Line Probe` as a debugging tool to examine the container transmitted on a line. When you select `Line Probe`, the cursor changes to crosshairs. Click on the line you wish to examine. A dialog box displays the current container on the line.

If the container on the line is not nil, the dialog box displays the current type, shape, and size of the containers. Data values of up to Array 2D are shown. If any of the dimensions of the container are mapped, the mapping values are shown for each dimension.

Control inputs often have no data on them. Sequence output pins never have data on them. If an object has not yet operated, the lines connected to its outputs may show nil or old data.

Location

`Edit` ⟹ `Line Probe`

Notes

Check the endpoints of a line by selecting `Line Probe` and dragging the pointer to the line. When you release the mouse button over the highlighted line, current attributes of the line are displayed.

Also, to view the endpoints of a line without showing its data, select `Line Probe` and *hold down* the mouse button when the pointer is over the line. Move the pointer away from the line before releasing the button.

You can check the data inside terminals of an object by double-clicking on the terminals in the open view.

Short Cuts

Press (Shift). Place pointer over line and click left mouse button. You can look at the data on series of lines this way, one at a time.

See Also

`Show Data Flow`, `Show Exec Flow`, and `Show Terminals`.

Logging AlphaNumeric

An object that displays a Scalar or Array 1D of alphanumeric data. `Logging AlphaNumeric` allows data to be displayed without overwriting the display.

Use

Use `Logging AlphaNumeric` to display consecutive input (either Scalar or Array 1D) as a history of previous values.

Location

`Display` ⟹ `Logging AlphaNumeric`

Object Menu

- `Config` - Specifies the number of lines that the display buffer holds. If more than that number of lines is input to `Logging AlphaNumeric` the last values are kept and the previous are overwritten. Default is 256.
- `Clear` - Clears all the data that was displayed on `Logging AlphaNumeric`. `Clear` may be added as a control input.
- `Clear at PreRun` - Clears the contents of the `Logging AlphaNumeric` at PreRun. Default is on (checked).
- `Clear at Activate` - Clears the contents of the `Logging AlphaNumeric` at Activate. Default is on (checked).
- `Number Formats` - Specifies a display format different than the global format.

Notes

A row of asterisks "***" is displayed if the current width of the `Logging AlphaNumeric` object is too small to display the default precision of the numeric type. Resize the object and rerun the model to display the data.

See Also

`AlphaNumeric` and `Number Format`.

Magnitude Spectrum

An object that graphically displays the magnitude of a frequency-domain spectrum.

Use

Use `Magnitude Spectrum` to display Spectrums or Waveforms in the frequency domain. Waveforms are automatically converted to spectrums by way of a Fast Fourier Transform. The X axis is in the sampling units of the input Spectrum (typically Hertz).

Location

`Display` ⟹ `Spectrum (Frequency)` ⟹ `Magnitude Spectrum`

Open View Parameters

- `Auto Scale` - Automatically scales the display to show the entire trace.
- `Mag` - The name of the Y axis.
- `Trace1` - The name of the first trace.
- `Freq` - The name of the X axis.

Object Menu

- `Auto Scale` ⟹ - Automatically scales the display to show the entire trace.
  - `Auto Scale` - Automatically scales both axes.
  - `Auto Scale X` - Automatically scales the X axis.
  - `Auto Scale Y` - Automatically scales the Y axis.

  These parameters may be added as control inputs.
- `Clear Control` ⟹ - Parameters that specify when to clear the display.
  - `Clear` - Clears the displayed trace(s). This parameter may be added as a control input.

- `Clear At PreRun` - Clears the displayed trace(s) when the model or thread is PreRun.
- `Clear At Activate` - Clears the displayed trace(s) when the User Object is activated.
- `Next Curve` - Resets the pen to display the next curve in a family of curves (data from the next time the display operates) *without* clearing the previous curve. `Next Curve` must be selected (or activated) before each new curve in the family. This parameter may be added as a control input.

■ `Zoom` ⟹ - Scales the display.

- `In` - Magnifies the display to contain only the rectangular region that you selected with the pointer. You select the region after selecting this feature by dragging on the graph area. This action outlines the information with a "rubber band" box.
- `Out 2|5|10|20|50|100 x` - Expands the scales of the display by a factor in both the X and Y directions about the center.

■ `Markers` ⟹ - Allows you to find the exact value of a data point on the displayed curve. If the data is plotted on a log scale, the values shown on the marker are the linear data points.

To move markers to a different trace, click on the button to the left of the marker values near the bottom of the display. The button cycles through the different line types and colors of the different traces.

- `Off` - No markers are shown.
- `One On` - One marker is available.
- `Two On` - Two markers are available.
- `Delta On` - Two markers are available and the x and y differences between them are displayed.
- `Interpolate` - When checked, you can place markers in between the displayed data points. The marker values displayed are calculated by linear interpolation. Default is off.
- `Center` - If markers are available, brings them to the center of the visible part of the trace. This is useful if you've scrolled the display and markers are not visible.

- `Grid Type` ⟹ - Sets the type of grid marks. The value of the major x division is shown below the maximum x value (to the right of the x scale name). The value of the major y division is shown below the y scale name.
  - `No Grid` - No grid lines are shown.
  - `Tic Marks` - Shows tic marks at the major and minor divisions on all four sides of the graph.
  - `Axis` - Shows tic marks at the major and minor divisions on the X and Y axes of the graph. If the actual axes are scrolled off the graph area, axis lines are drawn on the edge closest to the axes.
  - `Lines` - Shows lines at the major divisions and tic marks at the minor divisions. The X and Y axes are shown as thick lines.
- `Panel Layout` ⟹ - Sets the appearance of the open view.
  - `Graph Only` - The open view shows only the graph area and the marker information (if it exists). No buttons, scales, scale names, or traces names are shown. This layout redraws quickest and provides the largest display area.
  - `Scales` - The open view shows the graph area, the scales, the scale names, and trace names. These fields are not recessed and may not be edited. This layout is useful when recessed fields might be distracting, such as printing graphs, or when the fields should be protected from editing on a User Panel.
  - `Scales & Sliders` - The open view shows all information about the graph. It includes the most information and allows you to modify the most elements interactively. It is the default.
- `Traces & Scales` - A control panel that allows you to specify values such as the names, colors, line and point characteristics, minimum values, and maximum values for the traces and/or scales.

  Traces:
  - `Name:` - The name of the trace that is displayed to the left of the graph area and the name of the corresponding input terminal.
  - `Scale:` - If you have multiple Y scales, selects which Y scale is to be used for this trace.

- `Color:` - The color of the trace. Each added trace has a different color than the existing traces. Default is `Pen 4` (yellow).
- `Lines:` - The format of the line connecting data points. Default is a continuous line.
- `Points:` - The symbol that marks each data point. To show unconnected data points, select the single dot `Lines` format and the desired `Points` symbol. Default is a dot.

You can add a `Traces` control input pin. The control input data must be a record with the following fields: 1) A `TraceNum` field with an Integer value (1 is the top trace), *and* 2) one or more of the following fields: `Name`, `Pen`, `LineType`, `PointType`. (The `Pen`, `LineType`, and `PointType` values are integers from 0 to n, where 0 draws nothing.) Refer to "Records and DataSets" in *Using HP VEE* for further information.

Scales:

- `Show Scale:` - If you have multiple Y scales, a selection (using a check box) to specify if the end points and an axis of each additional right scale are displayed to the right of the graph area.
- `Scale Name:` - The names of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Maximum:` - The maximum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Minimum:` - The minimum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Mapping:` - The way the x and y data is mapped to the display. The mapping may be linear or log. To make a log-log plot, change both `X` and `Y` axes to `Log`. Default is `Linear`.
- `Log Cycles:` - The maximum number of decades shown (counting down from the maximum x and y values) when `AutoScale` is activated. `Log Cycles` is only used when the `Mapping` is `Log`. `Log Cycles` is useful when a trace contains a large dynamic range. Default is 4.
- `Scale Colors` - The color of any background grid or tic marks. Default is `Gray`.

You can add a `Scales` control input pin. The control input data must be a record with the following fields: 1) A Text field `Scale` with a value `X`, `Y` (or `Y1`), `Y2`, or `Y3`, *and* 2) one or more of the following fields: `Name`, `Min`, `Max`, and `Mapping`. (The `Mapping` text value may be `Linear` or `Log`). Refer to "Records and DataSets" in *Using HP VEE* for further information.

- `Add Right Scale` - Adds up to two additional scales to permit traces to have different scale ranges. After adding a right scale, use `Traces & Scales` to assign a trace to the scale.
- `Plot` - Presents the `Plotter Configuration` control panel. When `OK` is pressed, a copy of the device's entire display is plotted on the selected plotter. This parameter may be added as a control input. If the current `Plotter Configuration` is in `Plot to File` mode, you may specify the destination file or directory name as string data on the `Plot` control input. If no control input value is given, the file or directory name specified in `Plotter Configuration` will be used. See `Plotter Config` for more information.

Notes

Inputs must be of type `Spectrum`, `Waveform`, or `Coord` (Scalar or Array 1D).

Add traces with the `Terminals` ⟹ `Add Data Input` object menu selection. Up to twelve traces are allowed.

Input data of type Coord is plotted by simply using its x and y values without first being converted to type Spectrum.

A `Title` control input may be added, which sets the title bar name to the specified text value. This allows programmatic control over the title shown when the display is printed or plotted.

See Also

`Magnitude vs Phase`, `Phase Spectrum`, `Plotter Config`, `Spectrum (Freq)`, and `Waveform (Time)`.

Magnitude vs Phase

An object that displays a polar plot of Magnitude against Phase of a complex Spectrum.

Use

Use `Magnitude vs Phase` to display Spectrums or Waveforms Waveforms are automatically converted to spectrums by way of a Fast Fourier Transform (fft). The `Radius` of each data point is the spectrum's magnitude; the `Angle` is the Spectrum's phase in the current trig units.

You can change the display between `Polar` and `Smith` modes by way of `Grid Type` on the Object Menu.

Marker values are displayed in polar format (r:radius, a:angle) for the Polar "grid type" and complex impedance or admittance format for `Smith` and `Inverse Smith` "Grid Type".

Location

`Display` ⟹ `Spectrum (Frequency)` ⟹ `Magnitude vs Phase (Polar)`

`Display` ⟹ `Spectrum (Frequency)` ⟹ `Magnitude vs Phase (Smith)`

Open View Parameters

- `Auto Scale` - Automatically scales the display to show the entire trace. `Auto Scale` resets the polar reference location to `Center`.
- `Mag` - The name of the radius scale.
- `Trace1` - The name of the first trace.
- `Polar Reference Location` - Specifies the part of the polar plot displayed. This is quicker than scrolling to the area. Changing the reference point in the entry field scrolls the part of the graph specified (such as Center) to the location specified (such as (0, @0)).
- `Span:` - The vertical dimension of the display. When the origin is centered on the display, the `Span` is twice the radius. This parameter controls the scaling of the display. Default is `2`.

- `Ref Radius:` - The radius value of the bold graticule circle.

Object Menu

- `Auto Scale` ⟹ - Automatically scales the display to show the entire trace.
  - `Auto Scale` - Automatically scales both axes and resets the polar reference location to `Center: (0,@0)`.
  - `Auto Scale Y` - Automatically scales the Y axis.

  This parameter may be added as a control input.
- `Clear Control` ⟹ - Parameters that specify when to clear the display.
  - `Clear` - Clears the displayed trace(s). This parameter may be added as a control input.
  - `Clear At PreRun` - Clears the displayed trace(s) when the model or thread is PreRun.
  - `Clear At Activate` - Clears the displayed trace(s) when the User Object is activated.
  - `Next Curve` - Resets the pen to display the next curve in a family of curves (data from the next time the display operates) *without* clearing the previous curve. `Next Curve` must be selected (or activated) before each new curve in the family. This parameter may be added as a control input.
- `Zoom` ⟹ - Scales the display.
  - `In` - Magnifies the display to contain only the rectangular region that you selected with the pointer. You select the region after selecting this feature by dragging on the graph area. This action outlines the information with a "rubber band" box.
  - `Out 2|5|10|20|50|100 x` - Expands the scales of the display by a factor in both the X and Y directions about the center.
- `Markers` ⟹ - Allows you to find the exact value of a data point on the displayed curve. If the data is plotted on a log scale, the values shown on the marker is the linear data points.

To move markers to a different trace, click on the button to the left of the marker values near the bottom of the display. The button cycles through the different line types and colors of the different traces.

- `Off` - No markers are shown.
- `One On` - One marker is available.
- `Two On` - Two markers are available.
- `Delta On` - Two markers are available and the x and y differences between them are displayed.
- `Interpolate` - When checked, you can place markers in between the displayed data points. The marker values displayed are calculated by linear interpolation. Default is off.
- `Center` - If markers are available, brings them to the center of the visible part of the trace. This is useful if you've scrolled the display and markers are not visible.

■ `Grid Type` ⟹ - Sets the type of grid marks. The value of the major x division is shown below the maximum x value (to the right of the x scale name). The value of the major y division is shown below the y scale name.

- `No Grid` - No grid lines are shown.
- `Tic Marks` - Shows tic marks at the major and minor divisions on all four sides of the graph.
- `Axis` - Shows tic marks at the major and minor divisions on the X and Y axes of the graph. If the actual axes are scrolled off the graph area, axis lines are drawn on the edge closest to the axes.
- `Lines` - Shows circles at the major divisions and tic marks at the minor divisions. The bold (thick) circle is the "Reference Radius"; its radius is shown in the right corner of the display.
- `Smith Chart` - Displays data in a Smith Chart (also called an "impedance chart"). Marker values are given as real and imaginary impedances. The bold circle is the "reference radius" and its linear radius value is shown in the lower right corner of the display. The Smith Chart Grid and markers only represent properly scaled values if the input data has been normalized. To normalize the data, divide all magnitudes by the characteristic impedance.

- Inv Smith Chart - Displays data in an Inverse Smith Chart (also called an "admittance chart"). An Inverse Smith Chart is used in the same way as a Smith chart except grid and marker values are admittance values rather than impedance values.

- Panel Layout ⟹ - Sets the appearance of the open view.
  - Graph Only - The open view shows only the graph area and the marker information (if it exists). No buttons, scales, scale names, or traces names are shown. This layout redraws quickest and provides the largest display area.
  - Scales - The open view shows the graph area, the scales, the scale names, and trace names. These fields are not recessed and may not be edited. This layout is useful when recessed fields might be distracting, such as printing graphs, or when the fields should be protected from editing on a User Panel.
  - Scales & Sliders - The open view shows all information about the graph. It includes the most information and allows you to modify the most elements interactively. It is the default.

- Traces & Scales - A control panel that allows you to specify values such as the names, colors, line and point characteristics, minimum values, and maximum values for the traces and/or scales.

  Traces:
  - Name: - The name of the trace that is displayed to the left of the graph area and the name of the corresponding input terminal.
  - Scale: - If you have multiple Y scales, selects which Y scale is to be used for this trace.
  - Color: - The color of the trace. Each added trace has a different color than the existing traces. Default is Pen 4 (yellow).
  - Lines: - The format of the line connecting data points. Default is a continuous line.
  - Points: - The symbol that marks each data point. To show unconnected data points, select the single dot Lines format and the desired Points symbol. Default is a dot.

You can add a `Traces` control input pin. The control input data must be a record with the following fields: 1) A `TraceNum` field with an Integer value (1 is the top trace), *and* 2) one or more of the following fields: `Name`, `Pen`, `LineType`, `PointType`. (The `Pen`, `LineType`, and `PointType` values are integers from 0 to n, where 0 draws nothing.) Refer to "Records and DataSets" in *Using HP VEE* for further information.

Scales:

- `Show Scale:` - If you have multiple Y scales, a selection (using a check box) to specify if the end points and an axis of each additional right scale are displayed to the right of the graph area.
- `Scale Name:` - The names of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Vert Span` - The vertical dimension of the display area for each scale. This parameter may be set here or on the `Scales & Sliders` layout.
- `Scale Colors` - The color of any background grid or tic marks. Default is `Gray`.
- `Polar Reference Location` - The location of the given reference point for the `R MAIN` scale. This parameter may be set here or on the `Scales` or `Scales & Sliders` layout.

You can add a `Scales` control input pin. The control input data must be a record with one or more of the following fields: `Name`, `Span` (or `Name1`, `Span1`), `Name2`, `Span2`, `Name3`, `Span3`, and `RefPt` (the value of `RefPt` must be PComplex). Refer to "Records and DataSets" in *Using HP VEE* for further information.

- `Add Right Scale` - Adds up to two additional scales to permit traces to have different scale ranges. After adding a right scale, use `Traces & Scales` to assign a trace to the scale.
- `Plot` - Presents the `Plotter Configuration` control panel. When `OK` is pressed, a copy of the device's entire display is plotted on the selected plotter. This parameter may be added as a control input. If the current `Plotter Configuration` is in `Plot to File` mode, you may specify the destination file or directory name as string data on the `Plot` control input. If no control input value is given, the file or directory name specified in

`Plotter Configuration` will be used. See `Plotter Config` for more information.

Notes

Inputs must be of the type Spectrum, Waveform, or Coord (Scalar or Array 1D).

Add traces with the `Terminals` ⟹ `Add Data Input`. Up to twelve traces are allowed.

Input data of type Coord is plotted by simply using its x and y values as rectangular coordinates without first being converted to type Spectrum.

A `Title` control input may be added, which sets the title bar name to the specified text value. This allows programmatic control over the title shown when the display is printed or plotted.

See Also

`Magnitude Spectrum`, `Phase Spectrum`, `Spectrum (Freq)`, and `Plotter Config`.

Merge

Adds the contents of a model or set of saved objects into the work area while keeping the existing contents of the work area.

Use

Use `Merge` to combine a previously saved model or set of objects into an existing model in the HP VEE work area. `Merge` is useful for bringing in previously created models or sets of objects from a library of functions that are used often.

Location

`File` ⟹ `Merge`

Notes

Any `Preferences` (for example, the `Trig Mode`) that are saved with a model will *not* be loaded or used when you `Merge` that model into the HP VEE workspace. The currently active `Preferences` for the workspace will remain unchanged.

`Merge` presents a dialog box that allows you to select the name of the file you wish to open. After you select the file, the pointer changes to a pair of glasses while HP VEE is reading in the file.

You can use `Merge` to load a file that was saved with `Save` or `Save As`.

If the model saved has a panel view, the panel is not imported into the model. The panel views of `UserObject` are input.

When you open `Merge` for the first time, it points to `/usr/lib/veeengine/lib/` (or `/usr/lib/veetest/lib` if you have HP VEE-Test). You can choose to point somewhere else and then choose `Save Preferences`. The new default directory is saved so next time you open `Merge`, it opens in the new default directory.

See Also

`Open`, `Save`, `Save Objects`, and `Save Preferences`.

Merge Library

Loads a library of User Functions into the work area.

Use

Use `Merge Library` to merge User Functions previously saved in a library file of functions, into the current work area.

`Merge Library` loads all of the User Functions from a specified file. (User Functions are created by selecting the `Make User Function` selection on the object menu of a `UserObject`. You create a library by creating several User Functions and saving them to a file.)

Once the library of User Functions is loaded, the functions are executed (called) by name using the `Call Function` object.

Location

`File` ⟹ `Merge Library`

Notes

`Merge Library` is used for advanced operations where you have developed sets of User Functions into libraries. When you develop a library of standard User Functions, you can keep the source code for those functions in a single place. Refer to *Using HP VEE* for a detailed discussion of using User Functions.

User Functions loaded by `Merge Library` operate exactly like any locally-created User Function within the model. You can execute any User Function with the `Call Function` object. You can edit any of the external User Functions, loaded by `Merge Library`, as they are now merged into the HP VEE work area.

Any `Preferences` (for example, the `Trig Mode`) that are saved with a function will *not* be loaded or used. The currently active `Preferences` for the workspace will remain unchanged.

`Merge Library` presents a dialog box that allows you to select the name of the file you wish to open. After you select the file, the pointer changes to a pair of glasses while HP VEE is reading in the file.

When you open Merge Library for the first time, it points to /usr/lib/veeengine/lib/ (or /usr/lib/veetest/lib if you have HP VEE-Test). You can choose to point somewhere else and then choose Save Preferences. The new default directory is saved so next time you open Merge Library, it opens in the new default directory.

See Also

Call Function, Delete Library, Edit UserFunction, Import Library, Open, Save, Save Objects, Save Preferences, User Function, and UserObject.

Merge Record

An object that allows the user to merge two or more records of the same shape into a single record.

Use

Use `Merge Record` to merge two or more records of the same shape into a single record. Any number of records can be merged together as long as they are the same shape—either all scalars or all arrays with the same number of elements.

Location

`Data` ⟹ `Access Record` ⟹ `Merge Record`

Notes

Any number of input pins (`A`, `B`, and so forth) may be added, but each must be of the `Record` data type. The merged record is output on the `Record` output pin.

The records to be merged must not have any duplicate field names. The field names are taken from the input records (the input terminal names are ignored).

See Also

`Build Record`, `From DataSet`, `Record Constant`, `Set Field`, `To DataSet`, and `UnBuild Record`.

Meter

An object that graphically displays a Scalar numeric value.

Use

Use `Meter` to display a value on an analog scale and digital numeric field.

Location

`Display` ⟹ `Meter`

The value of the input is displayed at the bottom of the meter.

Object Menu

- `Clear At PreRun` - Clears the displayed value when the model or thread is PreRun.
- `Clear At Activate` - Clears the displayed value when the `UserObject` is activated.
- `Sub-Range Config` - Sets different colors (green, yellow, and red) for different ranges of the display. The color ranges overlay each other with red on the bottom and green on top. This allows red sub-ranges at each end of the scale.
- `Number Formats` - Specifies a different display format than the global format.

Notes

The upper and lower limits of the meter are set by clicking on the present value and typing in a new value.

These values may also be added as control inputs.

The input data must be Scalar and able to be converted to Real (Complex, PComplex, Spectrum, and Coord must be first "unbuilt".)

See Also

`AlphaNumeric`, `Logging AlphaNumeric`, and `Number Format`.

Move Objects

Moves selected objects.

Use

Use `Move Objects` to change the arrangement of a set of selected objects in your model.

Location

`Edit` ⟹ `Move Objects`

Notes

`Move Objects` retains all connections between objects. If `Automatic Line Routing` is set, lines between selected objects and unselected objects will be re-routed.

If you want to move a single object, use the `Move` feature on its object menu.

See Also

`Move (Object Menu)` and `Select Objects`.

Move (Object Menu)

Moves this object.

Use

Use `Move` to reposition the object on the work area by dragging it.

Location

On each object menu ⟹ `Move`

Notes

`Move` retains the connections with other objects. If `Automatic Line Routing` is set, it may take a few moments to redraw lines.

To move multiple objects, use the `Move Objects` selection from the Edit menu.

Short Cuts

Instead of selecting `Move`, you can move an object by positioning the pointer over the object and dragging it (using the left mouse button). The pointer must not be over a action area of the object (like a button or a entry field).

See Also

`Move Objects`, and `Object Menu`.

New

Clears the work area.

Use

Use `New` to give yourself a new work area, empty of objects.

Location

`File` ⟹ `New`

Notes

If your model has not been saved, `New` prompts you to save your changes.

`New` does not clear the `Paste` buffer. `New` reads in the default `.veerc` file for default `Preferences` settings.

`New` maintains the directory paths used for `Save` and `Open` instruments.

Short Cuts

Press Clear display to erase the model.

See Also

`Preferences`.

Next

An object that causes a `Repeat` ⟹ object to immediately start the next iteration.

Use

Use `Next` to skip to the next iteration of the current loop without executing the remaining objects on the iteration subthread.

If `Next` is not on an iteration subthread, it stops all execution on the subthread.

Location

`Flow` ⟹ `Repeat` ⟹ `Next`

Notes

`Next` is often used with an `If/Then` object to conditionally control iteration.

See Also

`Break`, `For Count`, `For Log Range`, `For Range`, `If/Then`, `On Cycle`, and `Until Break`.

Noise Generator

An object that generates a waveform of noise data.

Use

Use `Noise Generator` to generate a waveform of simulated white noise.

Location

`Device` ⟹ `Virtual Source` ⟹ `Noise Generator`

Open View Parameters

- `Amplitude` - The absolute value of the maximum and minimum values. Default is 1. No value is greater than Amplitude. No value is less than -Amplitude.
- `Time Span:` - The duration of the waveform (in seconds) Default is 20m seconds and is set in `Waveform Defaults`.
- `Num Points:` - The number of points in the waveform. The time between points in the waveform is `Time Span/N`. `Num Points` must be positive. Default is 256 and is set in `Waveform Defaults`.

All of these parameters may be set from the open view or added as inputs.

Notes

The random noise generated is distributed evenly across the frequency spectrum.

See Also

`Build Arb Waveform`, `Build Waveform`, `Comparator`, `Function Generator`, and `Pulse Generator`.

Note Pad

An object that displays a block of text.

Use

Use `Note Pad` to document your model, label the panel view, or write notes.

Location

`Display` ⟹ `Note Pad`

Object Menu

`Enable Editing` - When checked, allows you to edit the text in the panel. When not checked, no editing is allowed; this is useful when a `Note Pad` in on a panel view and you don't want other users to modify your note.

Open View Parameters

To enter information in the `Note Pad`, click on the panel, then type the note.

Notes

Each `Note Pad` is a stand-alone object and is not connected to any other objects.

After editing or resizing, scroll bars are added or removed from the `Note Pad` as required.

To edit the text area, all the usual HP VEE edit functions work including the following keys: Clear line, Insert line, Delete line, Insert char, Delete char, Prev, Next, and the cursor keys.

See Also

`Show Description`.

Number Formats

Specifies the default display format for numbers.

Use

Use `Number Format` to globally set the numeric format for real and integer numbers. Real number formats are `Fixed`, `Scientific`, `Engineering` and `Standard`. You can specify the precision of the real format. The Integer number formats are `Decimal`, `Octal`, `Hexadecimal`, and `Binary`. Numbers displayed in the last three formats are denoted by leading `#Q, #H, and #B`.

The current values of `Number Formats` is saved with each model. The defaults are read in from `.veerc` when HP VEE is started or `New` is selected.

Save Preferences will save the current `Number Format` to the `.veerc` file as the future default.

Location

`File` ⟹ `Preferences` ⟹ `Number Formats`

Notes

`Real Slider`, `Integer Slider`, `Alphanumeric`, `Logging Alphanumeric`, `VU Meter`, `Integer`, `Real`, `Coord`, `Complex`, and `PComplex` may be set to numeric display formats different from the global by using the `Number Formats` feature on the object menu. `AlphaNumeric` and `Logging AlphaNumeric` additionally allow a `Time Stamp` format that displays time and data information in several combinations.

See Also

`Alphanumeric`, `Complex`, `Coord`, `Integer`, `Integer Slider`, `Logging Alphanumeric`, `PComplex`, `Real`, `Real Slider`, `Save Preferences`, and `VU Meter`.

Object Menu

The menu associated with every object that contains selections that affect that object.

Use

Use the object to set status, edit, set controls, or change the terminals of an object.

Use `Object Menu` to access the following object actions:

- `Move`
- `Size`
- `Clone`
- `Help`
- `Show Description`
- `Breakpoint`
- `Show Title` (only available from the open view)
- `Terminals` ⟹
  - `Show Terminals` (only available from the open view)
  - `Add Data Input`
  - `Add Control Input`
  - `Add XEQ Input`
  - `Delete Input`
  - `Add Data Output`
  - `Add Error Output`
  - `Delete Output`
- `Layout` ⟹ (only available from the icon)
  - `Show Label`
  - `Select Bitmap`
  - `Delete Bitmap`
- `Cut`

Use `Object Menu` to access the following object actions on the panel view:

- `Move`
- `Size`
- `Show Title`
- `Delete`

Location

To access an object's `Object Menu`, position the pointer over the object and click the right mouse button. Note that to access the object menu of the open view of a `UserObject` you must position the pointer over the title bar or terminals, otherwise, the `Edit Menu` is presented.

You can also access the `Object Menu` from the open view by positioning the pointer over the object menu button (the square with a bar in the middle to the left of the title bar) and pressing the left mouse button.

Notes

Most object menus have additional menu features specific to that object. Refer to information on the object for details about those object menu choices.

The `Layout` menu is not available for the `Start`, `Toggle`, and `OK` objects.

OK

An object that pauses the execution flow of a subthread until it is pressed.

Use

Use OK to wait for a user response.

OK allows the addition of XEQ input. If the XEQ input is activated, OK operates even if the button has not been pressed by the user.

Location

Flow ⟹ Confirm (OK)

Notes

OK often is used when creating a dialog box or with the Enum constant.

The button part of the OK doesn't pop out until the object operates.

Since the icon of this object represents its greatest utility, the object menu accessed from the icon is larger and more useful than the object menu accessed from its open view.

You can use either the Go data output pin or the sequence output pin to continue the thread.

The name of the button (default = OK) can be changed by changing the text in the open view.

See Also

Gate, JCT, Start and Stop.

On Cycle

An object that begins execution of a subthread at a specified interval.

Use

Use `On Cycle` to execute a subthread that must be repeated periodically. The output of `On Cycle` is the current date and time. This date and time can be displayed on any object that supports a `Time Stamp` number format.

In HP VEE-Test, use `On Cycle` to trigger instruments at specific intervals.

Location

`Flow` ⟹ `Repeat` ⟹ `On Cycle`

Open View Parameters

Enter the number of seconds per iteration period in the entry field or as an input. If this value is `0` or `-1`, `On Cycle` operates continuously (the same as `Until Break`).

Notes

Execution of the subthread hosted by the `On Cycle` output continues until one of the following occurs:

- All objects that can, have operated. The subthread is deactivated, the iteration timer is reset, and the subthread is reactivated after the number of seconds have elapsed.
- A `Break` object operates. The subthread is deactivated and the sequence output pin is activated. Note that the value that remains on the `On Cycle` output is the same value present when the `Break` was encountered.
- A `Next` object operates. The subthread is deactivated and the iteration timer is reset.

`On Cycle` starts operation at the specified time period. If the thread takes longer to complete than the period of the `On Cycle` object, then the next iteration starts at the next period synchronization point. For example, if the

period in the `On Cycle` is 1 second and the thread it hosts takes 1.5 seconds to complete, the next iteration starts .5 seconds after the thread completes.

While waiting for the next time period to begin, `On Cycle` "sleeps" allowing other objects on this or other threads to operate.

When the subthread hosted by the `On Cycle` object finishes an iteration, all data containers sent during the previous iteration are invalidated before the next iteration. This prevents "old" data from a previous iteration from being reused in the current iteration. However, if flow branching is present within the iterative subthread, some objects may not execute on every iteration. Thus, data containers sent by those objects may be invalidated before other objects can execute on the data. To obtain the desired propagation in this case, use the `Sample & Hold` object. Refer to "Iteration with Flow Branching" in chapter 4 of *Using HP VEE* for more information.

See Also

`AlphaNumeric`, `Break`, `For Count`, `For Log Range`, `For Range`, `Next`, `Sample & Hold`, and `Until Break`.

On Features

Displays help on all menu choices (objects, actions, settings, cascading menus, and title bar buttons).

Use

Use `On Features` to get Reference help on the use, location, and parameters of each menu item. `On Features` also presents examples, notes, and related topics (See Also) for menu items.

Location

`Help` ⟹ `On Features`

Notes

Items in cascading menus are listed by the name on the object (which is usually the name of the menu feature).

See Also

`Help (Object Menu)`.

On Help

Displays information about the use of the `Help` menu items.

Use

Use `On Help` to become familiar with the types of information under the `Help` menu and navigation around the help screens.

Location

`Help` ⟹ `On Help`

See Also

`Glossary`, `How To`, `On Features`, `On Instruments`, `On Version`, and `Short Cuts`.

On Instruments

Displays information about instrument drivers. `On Instruments` is available in HP VEE-Test only.

Use

Use `On Instruments` to get help on the use and parameters of the instrument drivers supplied with HP VEE.

Location

`Help` ⟹ `On Instruments`

Notes

The instrument help files are designed to be used with several Hewlett-Packard software products and may refer to products other than HP VEE.
For information about those products, contact your Hewlett-Packard representative.

See Also

`Help` and `Help (Object Menu)`.

On Version

Displays the HP VEE version number.

Use

Use `On Version` to verify the version of HP VEE that you're running. You may want to do this to check your installation or when you're communicating with Hewlett-Packard Company.

`On Version` also displays the file name for the model you `Open`ed.

Location

`Help` ⟹ `On Version`

See Also

`Help`.

Open

Loads a model file into the work area, replacing the existing model.

Use

Use `Open` to bring a previously saved model into HP VEE for running or editing. Move to the directory where the file is located, click on the file name, and press OK. The mode is displayed on your screen.

Location

`File` ⟹ `Open`

Notes

`Open` presents a dialog box that allows you to select the name of the model file you wish to open. When HP VEE is loading in the file, the pointer icon changes to a pair of glasses.

You can `Open` a file that was saved with `Save Objects`, `Save` or `Save As`.

Short Cuts

CTRL O opens files.

Spacebar completes filenames.

See Also

`Merge`, `Save`, and `Save Objects`.

Panel

A button that toggles the view displayed from the detail view to the panel view.

Use

When pressed, `Panel` shows the panel view you created as an interface to your model on either the main HP VEE window or a `UserObject`.

Location

The upper left side of the title bar on the main HP VEE window and on User Objects when the associated panel is present.

Notes

`Panel` is only visible when you've created a panel view.

After you've secured a panel view, the `Panel` button is not visible.

See Also

`Add to Panel`, `Detail`, and `Secure`.

Paste

Places a copy of the `Paste` buffer on the work area.

Use

After using `Cut`, `Copy`, or `Clone`, use `Paste` to retrieve a copy of a set of objects.

`Paste` allows you to recover objects that have been accidentally `Cut`.

Location

`Edit` ⟹ `Paste`

Notes

The `Paste` buffer only holds one set of objects. The `Paste` buffer is replaced each time you `Clone`, `Cut` or `Copy`.

The `Paste` buffer is not cleared when you use `Open`, `Merge`, or `New`.

See Also

`Clone`, `Clone (Object Menu)`, `Copy`, `Cut`, and `Cut (Object Menu)`.

PComplex

An object that outputs a constant PComplex number. To input an array, press tab to enter the next value.

Use

Use PComplex to set a PComplex constant or to get user input.

Location

Data ⟹ Constant ⟹ PComplex

Example

To use PComplex as a prompt on a panel view, change the name of the PComplex object to a prompt such as Enter the AC voltage:. The user fills in the requested information in the entry field.

Type in (2,@PI/2) and it will calculate the PComplex Constant formula.

Object Menu

- Auto Execute - If set, the object operates whenever the values in the field are edited.
- Config - Sets the initial number of values to be output with this object. with this object. A value of 0 sets the container to a scalar, otherwise the container is an array of the length given.
- Initialize - Used to set this object to a particular value at PreRun and/or Activate time.
  - Initial Value - A dialog box that specifies the value to be set. Default value is (0,@0) (value of that container type).
  - Initialize At PreRun - Whether to set the Initial Value at PreRun time. Default is off.
  - Initialize At Activate - Whether to set the Initial Value at Activate time. Default is off.

- `Number Formats` - Sets the numeric format for Real and Integers.

Notes

`Initialize` is most often used for initializing values inside a `UserObject`.

The other method for setting initial values is the `Default Value` control pin. The `Default Value` pin allows you to programmatically change the current value.

Note that the `Initial Value` field is always a scalar, even if the `PComplex` is configured to be an array. The `Default Value` input pin, however, requires its input container to match the shape of the `PComplex`. `Trig Mode` specifies phase value units.

You can enter `Magnitude` and `Phase` values (separated by commas) and HP VEE formats automatically.

See Also

`Alloc PComplex`, `Complex`, `Constant`, `Coord`, `Date/Time`, `Enum`, `Integer`, `Number Formats`, `Real`, `Text`, `Trig Mode`, and `Toggle`.

Phase Spectrum

An object that graphically displays the phase of a frequency-domain spectrum.

Use

Use `Phase Spectrum` to display phase against frequency of Spectrums or Waveforms (in the frequency domain). Waveforms are automatically converted to spectrums by way of a Fast Fourier Transform (fft). The X axis is in the sampling units of the input spectrum (typically Hertz). The Y axis units are the current `Trig Mode` setting.

Location

`Display` ⟹ `Spectrum (Frequency)` ⟹ `Phase Spectrum`

Open View Parameters

- `Auto Scale` - Automatically scales the display to show the entire trace.
- `Phase` - The name of the Y axis.
- `Trace1` - The name of the first trace.
- `Freq` - The name of the X axis.

Object Menu

- `Auto Scale` ⟹ - Automatically scales the display to show the entire trace.
  - `Auto Scale` - Automatically scales both axes.
  - `Auto Scale X` - Automatically scales the X axis.
  - `Auto Scale Y` - Automatically scales the Y axis.

  These parameters may be added as control inputs.
- `Clear Control` ⟹ - Parameters that specify when to clear the display.
  - `Clear` - Clears the displayed trace(s). This parameter may be added as a control input.

- `Clear At PreRun` - Clears the displayed trace(s) when the model or thread is PreRun.
- `Clear At Activate` - Clears the displayed trace(s) when the User Object is activated.
- `Next Curve` - Resets the pen to display the next curve in a family of curves (data from the next time the display operates) *without* clearing the previous curve. `Next Curve` must be selected (or activated) before each new curve in the family. This parameter may be added as a control input.

■ `Zoom` ⟹ - Scales the display.

- `In` - Magnifies the display to contain only the rectangular region that you selected with the pointer. You select the region after selecting this feature by dragging on the graph area. This action outlines the information with a "rubber band" box.
- `Out 2|5|10|20|50|100 x` - Expands the scales of the display by a factor in both the X and Y directions about the center.

■ `Markers` ⟹ - Allows you to find the exact value of a data point on the displayed curve. If the data is plotted on a log scale, the values shown on the marker are the linear data points.

To move markers to a different trace, click on the button to the left of the marker values near the bottom of the display. The button cycles through the different line types and colors of the different traces.

- `Off` - No markers are shown.
- `One On` - One marker is available.
- `Two On` - Two markers are available.
- `Delta On` - Two markers are available and the x and y differences between them are displayed.
- `Interpolate` - When checked, you can place markers in between the displayed data points. The marker values displayed are calculated by linear interpolation. Default is off.
- `Center` - If markers are available, brings them to the center of the visible part of the trace. This is useful if you've scrolled the display and markers are not visible.

- `Grid Type` ⟹ - Sets the type of grid marks. The value of the major x division is shown below the maximum x value (to the right of the x scale name). The value of the major y division is shown below the y scale name.
  - `No Grid` - No grid lines are shown.
  - `Tic Marks` - Shows tic marks at the major and minor divisions on all four sides of the graph.
  - `Axis` - Shows tic marks at the major and minor divisions on the X and Y axes of the graph. If the actual axes are scrolled off the graph area, axis lines are drawn on the edge closest to the axes.
  - `Lines` - Shows lines at the major divisions and tic marks at the minor divisions. The X and Y axes are shown as thick lines.
- `Panel Layout` ⟹ - Sets the appearance of the open view.
  - `Graph Only` - The open view shows only the graph area and the marker information (if it exists). No buttons, scales, scale names, or traces names are shown. This layout redraws quickest and provides the largest display area.
  - `Scales` - The open view shows the graph area, the scales, the scale names, and trace names. These fields are not recessed and may not be edited. This layout is useful when recessed fields might be distracting, such as printing graphs, or when the fields should be protected from editing on a User Panel.
  - `Scales & Sliders` - The open view shows all information about the graph. It includes the most information and allows you to modify the most elements interactively. It is the default.
- `Traces & Scales` - A control panel that allows you to specify values such as the names, colors, line and point characteristics, minimum values, and maximum values for the traces and/or scales.

  Traces:
  - `Name:` - The name of the trace that is displayed to the left of the graph area and the name of the corresponding input terminal.
  - `Scale:` - If you have multiple Y scales, selects which Y scale is to be used for this trace.

Phase Spectrum

- `Color:` - The color of the trace. Each added trace has a different color than the existing traces. Default is `Pen 4` (yellow).
- `Lines:` - The format of the line connecting data points. Default is a continuous line.
- `Points:` - The symbol that marks each data point. To show unconnected data points, select the single dot `Lines` format and the desired `Points` symbol. Default is a dot.

You can add a `Traces` control input pin. The control input data must be a record with the following fields: 1) A `TraceNum` field with an Integer value (1 is the top trace), *and* 2) one or more of the following fields: `Name`, `Pen`, `LineType`, `PointType`. (The `Pen`, `LineType`, and `PointType` values are integers from 0 to n, where 0 draws nothing.) Refer to "Records and DataSets" in *Using HP VEE* for further information.

Scales:

- `Show Scale:` - If you have multiple Y scales, a selection (using a check box) to specify if the end points and an axis of each additional right scale is displayed to the right of the graph area.
- `Scale Name:` - The names of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Maximum:` - The maximum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Minimum:` - The minimum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Mapping:` - The way the x and y data is mapped to the display. The mapping may be linear or log. To make a log-log plot, change both `X` and `Y` axes to `Log`. Default is `Linear`.
- `Log Cycles:` - The maximum number of decades shown (counting down from the maximum x and y values) when `AutoScale` is activated. `Log Cycles` is only used when the `Mapping` is `Log`. `Log Cycles` is useful when a trace contains a large dynamic range. Default is 4.
- `Scale Colors` - The color of any background grid or tic marks. Default is `Gray`.

You can add a `Scales` control input pin. The control input data must be a record with the following fields: 1) A Text field `Scale` with a value `X`, `Y` (or `Y1`), `Y2`, or `Y3`, *and* 2) one or more of the following fields: `Name`, `Min`, `Max`, and `Mapping`. (The `Mapping` text value may be `Linear` or `Log`). Refer to "Records and DataSets" in *Using HP VEE* for further information.

- `Add Right Scale` - Adds up to two additional scales to permit traces to have different scale ranges. After adding a right scale, use `Traces & Scales` to assign a trace to the scale.
- `Plot` - Presents the `Plotter Configuration` control panel. When `OK` is pressed, a copy of the device's entire display is plotted on the selected plotter. This parameter may be added as a control input. If the current `Plotter Configuration` is in `Plot to File` mode, you may specify the destination file or directory name as string data on the `Plot` control input. If no control input value is given, the file or directory name specified in `Plotter Configuration` will be used. See `Plotter Config` for more information.

Notes

Inputs must be of the type Spectrum, Waveform, or Coord (Scalar or Array 1D).

Add traces with the `Terminals` ⟹ `Add Data Input`. Up to twelve traces are allowed.

Input data of type Coord is plotted by simply using its x and y values without first being converted to type Spectrum.

A `Title` control input may be added, which sets the title bar name to the specified text value. This allows programmatic control over the title shown when the display is printed or plotted.

See Also

`Magnitude Spectrum`, `Magnitude vs Phase`, `Spectrum (Freq)`, and `Plotter Config`.

Plotter Config

Changes the plotter configuration choices.

Use

Use `Plotter Config` to change how or where HP VEE plots two-dimensional graphical displays.

Location

`File` ⟹ `Preferences` ⟹ `Plotter Config`

Dialog Information

- `Plotter Device` - Allows you to select from the set of plotters/printers configured on your system. You may also use the default printer.
- `Plot to File` - Allows you to specify the file or directory on the file system where the plot will be written, rather than being sent directly to a plotter. If you specify a file, each plot will erase the old contents of the file (if any) and write the plotter HPGL commands to the file. If you specify an existing directory, the first plot made during any session of HP VEE will generate a file named "plot1" in that directory. Additional plots to a directory during the same HP VEE session will be named "plot2", "plot3", etc. Any of the resulting plot files can be sent to the plotter using a standard `lp` command such as:

  ```
  lp -d <plotter> /tmp/veeplot/plot3
  ```

- `Plotter Type` - Identifies the plotter to be used as compatible with either the HP-GL or the HP-GL/2 command set. Use HP-GL/2 for plotters that can also be used as printers, such as the LaserJet III or the PaintJet XL with the HP-GL/2 Cartridge.
- `Number of Pens` - Identifies the number of physical pens in the plotter. If 6 pens are specified, such as for an HP 7475 plotter, and a display trace attempts to use pen 9, the pen numbers will "wrap" and pen 3 will be used.
- `Label Using` - This option is only available when one or more stroke fonts (e.g., Kanji for Japanese) have been specified in app-defaults. Refer to

"Using Two-Byte Character Sets" in appendix A of *Using HP VEE* for further information. When present, this option gives you two choices:

- `Stroke Fonts` - With this selection, text and numeric labels are plotted using the specified stroke font. Since most plotters do not directly support Kanji, use this option for Kanji support.
- `Plotter ROM` - With this selection, text and numeric labels are output as HPGL and are to be converted to strokes by the plotter. This won't work for fonts not directly supported by the plotter. However, this option may be desirable if the HPGL commands are to be used by another software package.

- `Paper Size` - Allows you to select the size of the paper in the plotter. Choosing the size `Special` will allow you to edit the `P1` and `P2` fields to use a non-standard paper size, or to plot on only a portion of the page. The plotted image will always be scaled to the maximum size that can fit within the area defined by `P1` and `P2`. However, the aspect ratio of the original display object is maintained, so the plotted image may fill the defined area in only one direction.
  - `P1 (left, bottom)` - Specifies the offset from the left side and bottom of the plotter's hard clip limits to the bottom left corner of the plotting area to be used.
  - `P2 (right, top)` - Specifies the offset from the right side and top of the plotter's hard clip limits to the top right corner of the plotting area to be used.
  - `Size Units` - Allows you to select whether the `P1` and `P2` units will be displayed and entered in either `Inches` or `Millimeters`.

Notes

HP VEE holds only one configuration for plotting. It can be set with the `Plotter Config` menu item or the `Plot` menu item on a graphical display object. Use the `Preferences` ⟹ `Save Preferences` menu pick to save the current plotter configuration.

To send plots to a local or networked plotter, your system administrator must first add the plotter as a spooled device on your system.

Plotter Config

In addition to standard HP-GL plotters such as the HP 7475, the HP ColorPro (HP 7440), or the HP 7550, some printers can be used as plotters, such as the PaintJet XL, and the LaserJet III. The HP ColorPro plotter requires the Graphics Enhancement Cartridge to plot Polar or Smith Chart graticules. The PaintJet XL requires the HP-GL/2 Cartridge to make any plots. In order to make plots on the LaserJet III, at least two megabytes of optional memory expansion is required, and the `Page Protection` configuration option should be enabled. Plots of many vectors, especially with Polar or Smith chart graticules, may require even more optional memory in the LaserJet III. If a plot is to be output to a printer, the `Plotter Type` must be set to `HP-GL/2`, which causes the proper HP-GL/2 setup sequence to be included with the plot information.

To generate a plot, either interactively choose the `Plot` menu entry from the display's object menu, or programmatically use the optional `Plot` input control pin on the display. If the current `Plotter Configuration` is in the `Plot to File` mode, you may specify the destination file or directory name as string data on the `Plot` control input. If no control input value is given, the file or directory name specified in `Plotter Configuration` will be used.

The entire view of the display object will be plotted, and scaled to fill the defined plotting area, while retaining the aspect ratio of the original display object. By re-sizing the display object, you can control the aspect ratio of the plotted image. By making the display object larger, you can reduce the relative size of the text and numeric labels around the plot.

See Also

`XY Trace`, `Complex Plane`, `Magnitude Spectrum`, `Polar Plot`, `Strip Chart`, `Waveform (Time)`, `X vs Y Plot`, and `Save Preferences`.

Polar Plot

An object that displays a graphical plot in polar coordinates.

Use

Use `Polar Plot` to display a value on a polar scale when separate polar information is available for radius and angle data.

When more than one trace is to be plotted, each execution of the `Polar Plot` object uses the single angle input data with each trace's Radius input data, therefore, all traces share the single angle input.

Location

`Display` ⟹ `Polar Plot`

Open View Parameters

- `Auto Scale` - Automatically scales the display to show the entire trace. `Auto Scale` resets the polar reference location to `Center`.
- `R name` - The name of the radius scale.
- `RData1` - The name of the first trace.
- Polar Reference Location - Specifies the part of the polar plot displayed. This is quicker than scrolling to the area. Changing the reference point in the entry field scrolls the point of the display specified (such as Center) to the coordinate specified (such as (0, @0)).
- `Span:` - The vertical dimension of the display. When the origin is centered on the display, the `Span` is twice the radius. This parameter controls the scaling of the display. Default is 2.
- `Ref Radius:` - The radius value of the bold graticule circle.

Object Menu

- `Auto Scale` ⟹ - Automatically scales the display to show the entire trace.
  - `Auto Scale` - Automatically scales both axes and resets the polar reference location to "Center: (0,@0)".

  This parameter may be added as a control input.
- `Clear Control` ⟹ - Parameters that specify when to clear the display.
  - `Clear` - Clears the displayed trace(s). This parameter may be added as a control input.
  - `Clear At PreRun` - Clears the displayed trace(s) when the model or thread is PreRun.
  - `Clear At Activate` - Clears the displayed trace(s) when the User Object is activated.
  - `Next Curve` - Resets the pen to display the next curve in a family of curves (data from the next time the display operates) *without* clearing the previous curve. `Next Curve` must be selected (or activated) before each new curve in the family. This parameter may be added as a control input.
- `Zoom` ⟹ - Scales the display.
  - `In` - Magnifies the display to contain only the rectangular region that you selected with the pointer. You select the region after selecting this feature by dragging on the graph area. This action outlines the information with a "rubber band" box.
  - `Out 2|5|10|20|50|100 x` - Expands the scales of the display by a factor in both the X and Y directions about the center.
- `Markers` ⟹ - Allows you to find the exact value of a data point on the displayed curve. If the data is plotted on a log scale, the values shown on the marker are the linear data points.

  To move markers to a different trace, click on the button to the left of the marker values near the bottom of the display. The button cycles through the different line types and colors of the different traces.
  - `Off` - No markers are shown.
  - `One On` - One marker is available.

- `Two On` - Two markers are available.
- `Delta On` - Two markers are available and the x and y differences between them are displayed.
- `Interpolate` - When checked, you can place markers in between the displayed data points. The marker values displayed are calculated by linear interpolation. Default is off.
- `Center` - If markers are available, brings them to the center of the visible part of the trace. This is useful if you've scrolled the display and markers are not visible.

■ `Grid Type` ⟹ - Sets the type of grid marks. The value of the major x division is shown below the maximum x value (to the right of the x scale name). The value of the major y division is shown below the y scale name.

- `No Grid` - No grid lines are shown.
- `Tic Marks` - Shows tic marks at the major and minor divisions on all four sides of the graph.
- `Axis` - Shows tic marks at the major and minor divisions on the X and Y axes of the graph. If the actual axes are scrolled off the graph area, axis lines are drawn on the edge closest to the axes.
- `Lines` - Shows circles at the major divisions and tic marks at the minor divisions. The bold (thick) circle is the "Reference Radius"; its radius is shown in the lower right corner of the display.
- `Smith Chart` - Displays data in a Smith Chart (also called an "impedance chart"). Marker values are given as real and imaginary impedances. The bold circle is the "reference radius" and its linear radius value is shown in the lower right corner of the display. The Smith Chart grid and markers only represent properly scaled values if the input data has been normalized. To normalize the data, divide all magnitudes by the characteristic impedance.
- `Inv Smith Chart` - Displays data in an Inverse Smith Chart (also called an "admittance chart"). An Inverse Smith Chart is used in the same way as a Smith Chart except grid and marker values are admittance values rather than impedance values.

■ `Panel Layout` ⟹ - Sets the appearance of the open view.

- `Graph Only` - The open view shows only the graph area and the marker information (if it exists). No buttons, scales, scale names, or traces names are shown. This layout redraws quickest and provides the largest display area.
- `Scales` - The open view shows the graph area, the scales, the scale names, and trace names. These fields are not recessed and may not be edited. This layout is useful when recessed fields might be distracting, such as printing graphs, or when the fields should be protected from editing on a User Panel.
- `Scales & Sliders` - The open view shows all information about the graph. It includes the most information and allows you to modify the most elements interactively. It is the default.

- `Traces & Scales` - A control panel that allows you to specify values such as the names, colors, line and point characteristics, minimum values, and maximum values for the traces and/or scales.

  Traces:

  - `Name:` - The name of the trace that is displayed to the left of the graph area and the name of the corresponding input terminal.
  - `Scale:` - If you have multiple Y scales, selects which Y scale is to be used for this trace.
  - `Color:` - The color of the trace. Each added trace has a different color than the existing traces. Default is `Pen 4` (yellow).
  - `Lines:` - The format of the line connecting data points. Default is a continuous line.
  - `Points:` - The symbol that marks each data point. To show unconnected data points, select the single dot `Lines` format and the desired `Points` symbol. Default is a dot.

  You can add a `Traces` control input pin. The control input data must be a record with the following fields: 1) A `TraceNum` field with an Integer value (1 is the top trace), *and* 2) one or more of the following fields: `Name`, `Pen`, `LineType`, `PointType`. (The `Pen`, `LineType`, and `PointType` values are integers from 0 to n, where 0 draws nothing.) Refer to "Records and DataSets" in *Using HP VEE* for further information.

Scales:

- `Show Scale:` - If you have multiple Radius scales, a selection (using a check box) to specify if the span and an axis of each additional right scale will be displayed to the right of the graph area.
- `Scale Name:` - The names of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Vert Span` - The vertical dimension of the display area for each scale. This parameter may be set here or on the `Scales & Sliders` layout.
- `Scale Colors` - The color of any background grid or tic marks. Default is `Gray`.
- `Polar Reference Location` The location of the given reference point for the `R MAIN` scale. This parameter may be set here or on the `Scales` or `Scales & Sliders` layout.

You can add a `Scales` control input pin. The control input data must be a record with one or more of the following fields: `Name`, `Span` (or `Name1`, `Span1`), `Name2`, `Span2`, `Name3`, `Span3`, and `RefPt` (the value of `RefPt` must be PComplex). Refer to "Records and DataSets" in *Using HP VEE* for further information.

- `Add Right Scale` - Adds up to two additional scales to permit traces to have different scale ranges. After adding a right scale, use `Traces & Scales` to assign a trace to the scale.
- `Plot` - Presents the `Plotter Configuration` control panel. When `OK` is pressed, a copy of the device's entire display is plotted on the selected plotter. This parameter may be added as a control input. If the current `Plotter Configuration` is in `Plot to File` mode, you may specify the destination file or directory name as string data on the `Plot` control input. If no control input value is given, the file or directory name specified in `Plotter Configuration` will be used. See `Plotter Config` for more information.

Notes

Inputs must be Scalar or Array 1Ds that can be converted to the type Real.

You can add traces as data inputs. Up to twelve traces are allowed. Angle units are in the current `Trig Mode`.

All inputs must be the same size and shape. Mapping information on the inputs data is ignored.

A `Title` control input may be added, which sets the title bar name to the specified text value. This allows programmatic control over the title shown when the display is printed or plotted.

See Also

`Complex Plane`, `Magnitude Spectrum`, `Polar Plot`, `Strip Chart`, `Trig Mode`, `Waveform (Time)`, and `Plotter Config`.

Preferences

A menu item.

Use

Use `Preferences` to access the following work area options.

- `Trig Mode`
  - `Degrees`
  - `Radians`
  - `Gradians`
- `Number Formats`
- `Waveform Defaults`
- `Auto Line Routing`
- `Printer Config`
- `Plotter Config`
- `Save Preferences`

Location

`File` ⟹ `Preferences` ⟹

Notes

Default `Preferences` are stored in `.veerc` in your $HOME directory. If they are changed within HP VEE, the changes are valid only for the current work session unless you have selected `Save Preferences`.

See Also

`Auto Line Routing`, `Number Formats`, `Printer Config`, `Plotter Config`, `Save Preferences`, `Trig Mode`, and `Waveform Defaults`.

Print All

Outputs the entire detail view of the model to a printer or a print file.

Use

Use `Print All` to get hardcopy of your entire model to document it. `Print All` displays a dialog box that allows you to specify what to output, followed by the printer configuration dialog box.

`Print All` produces pages that can be assembled to show models. The output includes a total view of the top-level model, and a page for each object in the model, which shows all views of the object and its `Description` data. Each page of the output contains information describing the image's place in the model. (If you are using a Postscript printer, the entire view of the model will be scaled, if necessary, to fit on one page.)

All aspects of the printing process may be controlled independently.

Location

`File` ⟹ `Print All`

Dialog Information

The `Print All` dialog box displays four fields:

- `Print Complete Network` - If checked (the default), HP VEE prints the detail view of the entire model (even if the entire detail view is not visible). If the printout will not fit on one page, multiple pages are generated. If `Include All UserObjects` is also checked, complete detail views for each `UserObject` in the model are also printed. If `Include All UserFunctions` is also checked, complete detail views for each local `UserFunction` in the model are also printed. (If you are using a Postscript printer, the entire view of the model will be scaled, if necessary, to fit on one page.)
- `Print All Objects` - If checked, HP VEE prints a page containing the icon and open view and a description of each object. If the object is a `UserObject` or a `UserFunction`, it's panel view will also be printed. Multiple objects will be placed on a page if they will fit. If `Include All UserFunctions` is also checked, the detail view for each local `UserFunction`

in the model is also printed. This option may cause a large number of pages to be output (especially if `Include All UserObjects` is also checked). The default is not checked.

- `Include All UserObjects` - If checked (the default), the objects within each `UserObject` are printed. If not checked, `UserObjects` are treated just like other objects in the model.
- `Include All User Functions` - If checked (the default), the entire detail view and the object information of each User Function (its background functionality) will be included in the printout.

When you press `OK`, the printer configuration dialog box appears:

- `Graphics Printer` - Allows you to select from the set of printers configured on your system for graphics. You may also use the default printer. Click the mouse on `Default` for a selection of available printers. If you click on `Graphics Printer`, the field toggles to `Graphics Directory` (below).
- `Graphics Directory` - Allows you to specify a destination directory for the files that output is printed to. The output files are named "page1", "page2", and so forth.
- `Printer Type` - Identifies the particular type of printer being used for graphics printing. When `Graphics Directory` is selected, this field becomes `File Type` (below).
- `File Type` - Selects the type of output file that is to be written.
- `Magnification` - Allows you to select enlargement or reduction of the graphic being printed. A value of `1` generates the same physical size print on paper as is viewed on the screen, if the printer allows it. A `.5` reduces the graphic to half the screen size. If you are printing to a Postscript printer, a `Magnification` of `1` automatically scales the picture to fit on the page. (The picture is scaled first to the page size, then the `Magnification` factor is applied.)
- `Darkness` - Allows you to select the intensity of print you want, either light, medium, or dark.
- `Landscape` - Allows you to select the landscape (horizontal) presentation of graphics.

Notes

`Print All` takes a few moments to capture the image of your model. While the image is being captured, the pointer changes to an hourglass. You can't use HP VEE until the capture is completed (the pointer changes back to a crosshair).

If you print to a file in xwd format, HP VEE prints to multiple files. To print the files, first convert them to PCL using `pctrans`. Then use the HP-UX command

```
lp -oraw filename
```

to print the PCL files to an HP LaserJet printer (or other PCL printer).

If you print to a file in Postscript format, HP VEE prints to a single file. To print the file, use the HP-UX "`lp`" command

```
lp -d<PostscriptPrinter> filename
```

to print to a Postscript printer.

HP VEE only saves one configuration for printing. It can be set with `Printer Config`, `Print All`, `Print Objects`, or `Print Screen`.

See Also

`Print Screen`, `Print Objects`, and `Printer Config`.

Print Objects

Outputs a page for each of the selected objects, containing all views and `Description` data, to a printer or a print file.

Use

Use `Print Objects` to get hardcopy of selected objects. `Print Objects` displays a dialog box that specifies information about the printer configuration.

`Print Objects` will print all views (icon, open, detail, or panel), along with `Description` data, for each object.

If no objects are currently selected, the `Print Objects` menu item will be grayed out.

Location

`File` ⟹ `Print Objects`

Dialog Information

`Print Objects` displays the `Printer Config` dialog box. The following fields are displayed:

- `Graphics Printer` - Allows you to select from the set of printers configured on your system for graphics. You may also use the default printer. Click the mouse on `Default` for a selection of available printers. If you click on `Graphics Printer`, the field toggles to `Graphics Directory` (below).
- `Graphics Directory` - Allows you to specify a destination directory for the files that output is printed to. The output files are named "page1", "page2", and so forth.
- `Printer Type` - Identifies the particular type of printer being used for graphics printing. When `Graphics Directory` is selected, this field becomes `File Type` (below).
- `File Type` - Selects the type of output file that is to be written.
- `Magnification` - Allows you to select enlargement or reduction of the graphic being printed. A value of `1` generates the same physical size print on

paper as is viewed on the screen, if the printer allows it. A `.5` reduces the graphic to half the screen size. If you are printing to a Postscript printer, a `Magnification` of `1` automatically scales the picture to fit on the page. (The picture is scaled first to the page size, then the `Magnification` factor is applied.)

- `Darkness` - Allows you to select the intensity of print you want, either light, medium, or dark.
- `Landscape` - Allows you to select the landscape (horizontal) presentation of graphics.

Notes

`Print Objects` may take a few moments to capture the image of the objects, particularly if you select a large number of objects. While the image is being captured, the pointer changes to an hourglass. You can't use HP VEE until the capture is completed (the pointer changes back to a crosshair).

HP VEE only saves one configuration for printing. It can be set with `Printer Config`, `Print All`, `Print Objects`, or `Print Screen`.

If HP VEE prints to a file in xwd format, you can print the file by first converting it to PCL using `pctrans`. Then use the HP-UX command

```
lp -oraw filename
```

to print the PCL file to an HP LaserJet printer (or other PCL printer).

If HP VEE prints to a file in Postscript format, you can print it to a Postscript printer using the HP-UX command

```
lp -d<PostscriptPrinter> filename
```

where `<PostscriptPrinter>` is the device name for your Postscript printer.

See Also

`Print All`, `Print Screen`, and `Printer Config`.

Print Screen

Outputs the contents of the HP VEE window to a printer or a print file.

Use

Use `Print Screen` to get hardcopy of the portion of your model that is visible on your screen. `Print Screen` displays a dialog box that specifies information about the printer configuration.

Location

`File` ⟹ `Print Screen`

Dialog Information

`Print Screen` displays the `Printer Config` dialog box. The following fields are displayed:

- `Graphics Printer` - Allows you to select from the set of printers configured on your system for graphics. You may also use the default printer. Click the mouse on `Default` for a selection of available printers. If you click on `Graphics Printer`, the field toggles to `Graphics Directory` (below).
- `Graphics Directory` - Allows you to specify a destination directory for the files that output is printed to. The output files are named "page1", "page2", and so forth.
- `Printer Type` - Identifies the particular type of printer being used for graphics printing. When `Graphics Directory` is selected, this field becomes `File Type` (below).
- `File Type` - Selects the type of output file that is to be written.
- `Magnification` - Allows you to select enlargement or reduction of the graphic being printed. A value of `1` generates the same physical size print on paper as is viewed on the screen, if the printer allows it. A `.5` reduces the graphic to half the screen size. If you are printing to a Postscript printer, a `Magnification` of `1` automatically scales the picture to fit on the page. (The picture is scaled first to the page size, then the `Magnification` factor is applied.)

- `Darkness` - Allows you to select the intensity of print you want, either light, medium, or dark.
- `Landscape` - Allows you to select the landscape (horizontal) presentation of graphics.

Notes

`Print Screen` takes a few moments to capture the image in the window. While the image is being captured, the pointer changes to an hourglass. You can't use HP VEE until the capture is completed (the pointer changes back to a crosshair).

The size of the HP VEE display window determines the size of the print area. To get a large output, maximize the HP VEE window (but too large an image gets clipped). Larger images take longer to scan and print.

HP VEE only saves one configuration for printing. It can be set with `Printer Config`, `Print All`, `Print Objects`, or `Print Screen`.

If HP VEE prints to a file in xwd format, you can print the file by first converting it to PCL using `pctrans`. Then use the HP-UX command

```
lp -oraw filename
```

to print the PCL file to an HP LaserJet printer (or other PCL printer).

If HP VEE prints to a file in Postscript format, you can print it to a Postscript printer using the HP-UX command

```
lp -d<PostscriptPrinter> filename
```

where `<PostscriptPrinter>` is the device name for your Postscript printer.

Short Cuts

Just press Shift-Print to cause an immediate `Print Screen`. The current options specified in `Printer Config` will be used. This is useful for printing open dialog boxes.

See Also

`Print All`, `Print Objects`, and `Printer Config`.

Print Screen (Object)

An object that prints a copy of the visible portion of the HP VEE window on the system printer.

Use

Use `Print Screen` to print a copy of the visible work area while the model is running.

Location

`I/O` ⟹ `Print Screen`

Notes

The printer used by Print Screen (the HP VEE system printer) is configured by `File` ⟹ `Preferences` ⟹ `Printer Config`. `Print Screen` directs output to the configured graphics printer or directory.

A running model is paused when `Print Screen` operates. It resumes when `Print Screen` is finished.

See Also

`Print All`, `Print Screen`, `Print Objects` and `Printer Config`.

Printer Config

Changes the printer configuration choices.

Use

Use `Printer Config` to change how or where you get your HP VEE printouts.

Location

`File` ⟹ `Preferences` ⟹ `Printer Config`

Dialog Information

- `Graphics Printer` - Allows you to select from the set of printers configured on your system for graphics. You may also use the default printer. Click the mouse on `Default` for a selection of available printers. If you click on `Graphics Printer`, the field toggles to `Graphics Directory` (below).
- `Graphics Directory` - Allows you to specify a destination directory for the files that output is printed to. The output files are named "page1", "page2", and so forth.
- `Printer Type` - Identifies the particular type of printer being used for graphics printing. When `Graphics Directory` is selected, this field becomes `File Type` (below).
- `File Type` - Selects the type of output file that is to be written.
- `Magnification` - Allows you to select enlargement or reduction of the graphic being printed. A value of `1` generates the same physical size print on paper as is viewed on the screen, if the printer allows it. A `.5` reduces the graphic to half the screen size. If you are printing to a Postscript printer, a `Magnification` of `1` automatically scales the picture to fit on the page. (The picture is scaled first to the page size, then the `Magnification` factor is applied.)
- `Darkness` - Allows you to select the intensity of print you want, either light, medium, or dark.

- `Landscape` - Allows you to select the landscape (horizontal) presentation of graphics.

The following are used only for text printing using the `To Printer` object.

- `Text Printer` - Allows you to select from the set of printers configured on your system for text (`To Printer` objects). You may also use the default printer.
- `Wrap Column` - The number of characters that can be entered on a single line before the column wraps to the next line. Default is 80 characters.
- `Header Title` - Enter your header title here. You can also select `No Header` by clicking on the `Header Title` bar. If a header is specified, it is printed at the top of each page with the date and page number.

Notes

HP VEE only saves one configuration for printing. It can be set with `Printer Config`, `Print All`, `Print Objects`, or `Print Screen`.

To save the printer configuration for the next work session, be sure to execute `File` ⟹ `Preferences` ⟹ `Save Preferences` before you exit HP VEE.

See Also

`Print All`, `Print Screen`, `Print Objects`, and `To Printer`.

Pulse Generator

An object that generates and outputs a pulse waveform.

Use

Use `Pulse Generator` to generate a pulse waveform using your own specifications.

Location

`Device` ⟹ `Virtual Source` ⟹ `Pulse Generator`

Object Menu

- `Error on Aliasing` - Evaluates whether the period of any high or low pulse state is less than twice the sampling period (Time Span/Num Points). If it is, an error is returned. The purpose of this evaluation is to determine whether the points being generated provide an accurate representation of the pulse being generated. An unchecked checkbox generates a literal presentation of points. Default is on (checked).

Open View Parameters

- `Frequency` - The rate in Hertz at which pulses are generated.
- `Pulse Width` - The width, in time, of the pulses. The time is measured from the mid-point of the low-to-high transition to the mid-point of the high-to-low transition. Pulse width must be a positive Real number. Default is 5m seconds.
- `Pulse Delay` - The time from the start of the waveform to the beginning of the first transition. `Pulse Delay` must be a positive Real number. Default is 0.
- `Thresholds` - Determine the thresholds used by `Rise Time` and `Fall Time`.
  Low Threshold = (first percentage) * (high - low)
  High Threshold = (second percentage) * (high - low)
  - 0%-100%
  - 10%-90%

- 20%-80%

- **Rise Time** - The time, in seconds, to make the transition from the low threshold to the high threshold. **Rise Time** must be a positive Real number.
- **Fall Time** - The time, in seconds, to make the transition from the high threshold to the low threshold. **Fall Time** must be a positive Real number.
- **High** - The value of the waveform when the pulse is active. **High** must be a Real number. Default is 1.
- **Low** - The value of the waveform when the pulse is not active. **Low** must be a Real number. If the value of **Low** is greater then the value of **High** the pulse generated is inverted. Default is 0.
- **Burst Mode** - Enables or disables multiple pulses being generated at the **Frequency**. When **Burst Rate** is **OFF**, only one pulse is generated at the **Frequency**. When **Burst Mode** is **On**, a **Burst Count** number of pulses is generated at the **Frequency**. Default is **OFF**.
- **Burst Count** - The number of pulses generated at the **Rep Rate**. **Burst Count** must be a positive integer. Default is 2.
- **Burst Rep Rate** - The rate at which the **Burst Count** pulses are generated. **Burst Rep Rate** must be a Real number greater than 1/**pulse width**.
- **Time Span:** - The duration of the waveform (in seconds). Default is 20m seconds set in **Waveform Defaults**.
- **Num Points:** - The number of points in the waveform. The time between points in the waveform is **Time Span/numPoints**. **Num Points** must be positive. Default is **256** set in **Waveform Defaults**.

All of these parameters may be set from the open view or added as control inputs.

See Also

Build Arb WF, **Build Waveform**, **Comparator**, **Function Generator**, **Noise Generator**, and **Waveform Defaults**.

Raise Error

Generates a user-defined error condition with associated error number (escape code) and text message. (This object was formerly called `Escape`.)

Use

Use `Raise Error` to stop execution of the current `UserObject` and generate an error that may be trapped by the error pin of an enclosing `UserObject` or the main work area. If the error is not trapped (by an error pin on a `UserObject`) and the escape code is non-zero, `Raise Error` stops the model and generates an error dialog box.

Location

`Flow` ⟹ `Raise Error`

Open View Parameters

`Code` - An error number; it must be an Int32. HP VEE uses error numbers between 300 and 1,000; it is best to use error numbers exclusive of this range. You can enter the `Code` in the entry box or add it as a data input.

`Message` - The text message to output. You can enter the `Message` in the entry box or add it as a data input.

Notes

If HP VEE is invoked with the `-r` command line option, an untrapped error generated by `Raise Error` causes HP VEE to terminate. The `Code` and `Message` are written to `stderr` and 255 is returned to the operating system as the exit code.

To generate a specific exit code, use the `Stop` object.

See Also

`Add Error Output`, `Create UserObject`, `Exit UserObject`, `Exit Thread`, `Start`, and `Stop`.

Random Number

An object that outputs a random number.

Use

Use `Random Number` to get a uniform random distribution.

Location

`Device` ⟹ `Random Number`

Open View Parameters

- `Range from` - The lower limit of the random number distribution (inclusive). `Range From` may be added as an input. Default is 0.
- `Range to` - The upper limit of the random number distribution (exclusive). `Range To` may be added as an input. Default is 1.

See Also

`Random Seed`.

`randomize(x)` and `random(l,h)` in the "Formula Reference" chapter.

Random Seed

An object that accepts a new seed for the random number generator.

Use

Use `Random Seed` to get a set of repeatable random numbers from `Random Number`.

Location

`Device` ⟹ `Random Seed`

Notes

If you have more than one `Random Seed` operating in your model, the `Random Seed` resets the seed value for the random number generator each time the `Random Seed` object operates.

The same `Random Seed` value is used for the `Random Number` object and the `random(l,h)` and `randomize(x)` functions in the `Formula Reference` chapter.

See Also

`Random Number`.

`randomize(x)` and `random(l,h)` in the "Formula Reference" chapter.

Real

An object that outputs a constant Real number. To input an array, press tab to enter the next value.

Use

Use `Real` to set a real constant or to get user input.

Location

`Data` ⟹ `Constant` ⟹ `Real`

Example

To use `Real` as a prompt on a panel view, change the name of the `Real` object to a prompt such as `Enter the offset:`. The user fills in the requested information in the entry field.

Type in `100M*PI` to have the Real `Constant` evaluate this formula and display the Real number answer.

Object Menu

- `Auto Execute` - If set, the object operates whenever the values in the field are edited.
- `Config` - Sets the initial number of values to be output with this object. A value of `0` sets the container to a scalar, otherwise the container is an array of the length given.
- `Initialize` - Used to set this object to a particular value at PreRun and/or Activate time.
  - `Initial Value` - A dialog box that specifies the value to be set. Default value is the zero value of that container type.
  - `Initialize At PreRun` - Whether to set the `Initial Value` at PreRun time. Default is off.
  - `Initialize At Activate` - Whether to set the `Initial Value` at Activate time. Default is off.

- `Number Formats` - Sets the numeric format.

Notes

`Initialize` is most often used for initializing values inside a `UserObject`.

The other method for setting initial values is the `Default Value` control pin. The `Default Value` pin allows you to programmatically change the current value.

Note that the `Initial Value` field is always a scalar, even if `Real` is configured to be an array. The `Default Value` input pin, however, requires its input container to match the shape of `Real`.

See Also

`Alloc Real`, `Complex`, `Constant`, `Coord`, `Date/Time`, `Enum`, `Integer`, `PComplex`, `Text`, and `Toggle`.

Real Slider

An object that outputs the Real value of the slider.

Use

Use `Real Slider` to input values. `Real Slider` is particularly useful on a panel.

Location

`Data` ⟹ `Real Slider`

Open View Parameters

The open view displays a field for min, max, slider value, and slider control.

Object Menu

- `Auto Execute` - If set, the object operates whenever the values in the field are edited.
- `Detents` - Sets the distance between values.
- `Initialize` - Used to set this object to a particular value at PreRun and/or Activate time.
  - `Initial Value` - A dialog box that specifies the value to be set. Default value is the zero value of that container type.
  - `Initialize At PreRun` - Whether to set the `Initial Value` at PreRun time. Default is off.
  - `Initialize At Activate` - Whether to set the `Initial Value` at Activate time. Default is off.
- `Number Formats` - Sets the numeric format.
- `Layout` - Specifies either horizontal or vertical slider format.

Notes

`Initialize` is most often used for initializing values inside a `UserObject`.

The other method for setting initial values is the `Default Value` control pin. The `Default Value` pin allows you to programmatically change the current value.

See Also

`Complex`, `Enum`, `Integer`, `Integer Slider`, `PComplex`, `Real`, `Text`, and `Toggle`.

Record Constant

An object that outputs a constant of the Record data type.

Use

Use the `Record Constant` object to define and build a Record constant, or to interactively edit an existing record or array of records.

The following control input pins may be added:

- `Default Value`—This control pin allows you to set the entire record to an initial "value," which is actually an entire record. That is, all fields in the record are set to the values from the record connected as the "default value."
- `Reset`—This control pin "zeros" out all the data in the record.

Location

`Data` ⟹ `Constant` ⟹ `Record`

Open View Parameters

In the open view there is a `Field name` button and a `Value` button (or type-in field) for every field in the record. (By default, the `Field name` buttons `A` and `B` correspond to the `Value` fields "`Text field`" and "`1.25`", respectively.) If the record is an Array 1D (that is, `Array Elements` is non-zero), there is an additional field in the open view for the index into the array (`0`, `1`, etc.) To step through the array values, `First`, `Prev`, `Next`, and `Last` buttons are provided.

- `Field name`—A button that, when pressed, pops up a dialog box that allows the user to re-name the record field, and change its type and shape. An error will occur if you attempt to rename a field to an existing field name. Field names are not case sensitive (lowercase and uppercase letters are equivalent). When changing the data type, HP VEE will attempt to coerce the data into the new data type, subject to the existing rules about data type coercion. Refer to the section "Converting Data Types on Input Terminals" in chapter 3 of *Using HP VEE*.

- `Value`—Allows you to edit the values in the field. If the field is a scalar container of a data type other than Enum, `Value` is a type-in field. That is, you can click on the field and type in the desired value. If the field is an Enum or Array 1D container, the `Value` field is a button, which brings up a dialog box to allow you to select the values. If the field is an array of two or more dimensions, the values may not be edited.

Object Menu

- `Config`—Sets the record size through a dialog box. A value of `0` for `Array Elements` sets the container to a scalar. Otherwise the container is an array of the length given. You can fix the size of the record (so the user can't change it) by checking `Size fixed?`, and you can fix the format ("schema") of the record by checking `Schema fixed?`. Note that only Scalar and Array 1D records are allowed.
- `Number Formats`—Sets the number format of a numeric field. A different number format can be applied to each numeric record field. A dialog box prompts you to select a numeric field, and a second dialog box allows you to set the number format for that field.
- `Add Record`—Adds a record at the end of the record array. The new record is a copy of the last record in the array. (This menu selection is not active for record scalars.)
- `Insert Record`—Inserts a record at the position in the record array that you are currently viewing. The new record is a copy of the record that you were viewing. (This menu selection is not active for record scalars.)
- `Delete Record`—Deletes the record that you are currently viewing from the array. (This menu selection is not active for record scalars. Also, for record arrays, the selection is active only if there is more than one record in the array.)
- `Add Field`—Adds a new field to the record at the end of the field list. A dialog box will allow you to define the name, type, and shape of the new field.
- `Insert Field`—Inserts a new field to the record at the position above the current edit position in the array. A dialog box will allow you to define the name, type, and shape of the new field.

- `Delete Field`—Allows you to delete record fields. A dialog box will list the available fields to delete. This selection is available only if the record has more than one field.

Notes

Two control pins may be added to the `Record Constant` object: `Reset` and `Default Value`.

The `Reset` pin "zeros out" all fields in the record. Numeric fields become zero and Text fields become null strings.

The `Default Value` pin is useful for editing large record arrays (for example, a record array "pulled in" with the `From DataSet` object). Just send the record array into the `Default Value` pin to set the initial value for the `Record Constant`. You can then view and interactively edit the entire record. The data values may be changed, and also the type, shape, and mappings of the fields may be changed (unless the field is itself a record). Once you have edited the record, it can be output to a new dataset or file (using `To DataSet` or `To File`). The `Record Constant` object does not allow you to edit a field that is itself a record—it will only allow you to view the field and its data.

Note that the `Record Constant` object does not allow you to create a record of records. That is, the `Record Constant` does not allow you to add a field that is a record. However, all other HP VEE record devices (such as `Build Record` and `Set Field`) do allow this. Recursion is allowed up to the limit of available user memory. That is, a record may have a field that is a record, which has a record field within it, and so forth. The shape of a record container may only be either a scalar or a one-dimensional array.

See Also

`Build Record`, `From DataSet`, `Merge Record`, `Set Field`, `To DataSet`, and `UnBuild Record`.

Repeat

A menu item.

Use

Use `Repeat` to access the following iterators and iteration-control objects:

- `For Count`
- `For Range`
- `For Log Range`
- `Until Break`
- `On Cycle`
- `Next`
- `Break`

Location

`Flow` ⟹ `Repeat` ⟹

Notes

You can nest iterators.

Execution of the subthread hosted by the iterator continues until one of the following occurs:

- All objects that can, have operated. The subthread iterations have been completed.
- A `Break` object operated. The subthread is deactivated and the sequence output pin is activated.
- A `Next` object operated. The subthread is deactivated and the iteration counter is incremented.

When the iteration subthread has completed, the sequence output pin is activated.

See Also

Break, For Count, For Range, For Log Range, Next, On Cycle, and Until Break.

Run

A button that causes all threads to run.

Use

Use `Run` to begin the model execution. If `Start` objects are used on threads in the main work area, execution begins with those objects.

When you press `Run`, HP VEE PreRuns all threads to clear data and check for feedback.

Location

On the right side of the title bar.

Notes

If you have several threads in your model and want to run only one of them, connect a `Start` object to that thread. Press the `Start` object on the thread you want to run.

See Also

`Cont`, `Start`, `Step`, and `Stop`.

Sample & Hold

An object that stores the most recently received data container. Activating the `XEQ` pin outputs the stored container.

Use

Use `Sample & Hold` to capture the last data present on the input pin when `XEQ` is activated.

Location

`Flow` ⟹ `Sample & Hold`

Example

Use `Sample & Hold` to preserve data generated by objects in subsequent thread segments of either `If/Then/Else` or `Conditional` flow objects that are also in an iteration sub-thread.

For an example and further information, refer to "Iteration with Flow Branching" in chapter 4 of *Using HP VEE-Engine and HP VEE-Test*.

Notes

The `Sample & Hold` is generally not needed unless *both* iteration and flow branching are present in a thread. Objects will normally retain the most recently received data container on their input pin(s) by default.

Only the most recent data container arriving at the `InData` input will be saved by the `Sample & Hold`. As new data arrives, it over-writes the existing data.

If the `Sequence In` pin is connected, both the `InData` input and the `Sequence In` pin must be activated before a data container will be accepted to replace the existing stored data.

The `Sample & Hold` operates when the `XEQ` pin is activated. Propagation will proceed as far as possible before the `Sequence Out` pin is activated.

The contents of the `Sample & Hold` are cleared by Prerun and Activate. If `XEQ` is activated with no stored data, a nil (empty) container is generated.

Because XEQ is required on the Sample & Hold, you cannot delete it or add another XEQ pin.

Multiple XEQ activations each re-send the most recent stored data if no new data has arrived.

See Also

Repeat, If/Then/Else, Conditionals, User Objects.

Save

Copies the model in the work area to a previously specified filename.

Use

Use `Save` periodically to write your model to a file.

Location

`File` ⟹ `Save`

Notes

`Save` writes your model to the file name specified in `Save As` or `Open`. If a model was created from a `New` work area and no `Save As` was done, `Save` prompts you for a file name. As the model is being written, the pointer changes to a pencil.

`Save` does not ask for confirmation when overwriting a file.

Short Cuts

Press CTRL S to save a model.

See Also

`Merge`, `Open`, `Save As`, and `Save Objects`.

Save As

Copies the model in the work area to a specific file.

Use

Use `Save As` to specify a file name for your model and write the model to that file.

Location

`File` ⟹ `Save As`

Notes

If you attempt to overwrite an existing file, you are asked to confirm that action. As the model is being written, the pointer changes to a pencil.

The file name specified in `Save As` becomes the name used in subsequent `Save` operations.

The current file is saved as *filename*.bak; the new saved version is just *filename*. Note that if the filename already includes an extension (such as `.vee`) that extension is replaced with `.bak`.

Short Cuts

Press CTRL W to `Save As`.

See Also

`Merge`, `Open`, `Save`, and `Save Objects`.

Save Objects

Copies the selected objects to a specific file.

Use

Use `Save Objects` to specify a file name for a set of objects and write the objects to that file. `Save Objects` is generally used to save a group of objects into a library of common functions that is included in other models.

Location

`File` ⟹ `Save Objects`

Notes

Use `Save Objects` to copy a set of objects from your model that you use in other models. `Save Objects` allows you to build up a library of your own functions. As the objects are being written, the pointer changes to a pencil.

If you attempt to overwrite an existing file, you are asked to confirm that action.

See Also

`Merge`, `Open`, `Save`, and `Select Objects`.

Save Preferences

Saves default preferences.

Use

Use `Save Preference` to save all the default preferences under the `Preferences` menu, plus the last directory referenced by a `Merge` or `Save Objects` command in a file called `.veerc` in your `$HOME` directory.

Starting up HP VEE or clearing the work area by selecting `New` causes `.veerc` to be read, thus setting all preferences and the `Merge/Save Objects` default directory to the values stored with the last `Save Preferences` command.

Location

`File` ⟹ `Preferences` ⟹ `Save Preferences`

Notes

In addition to the defaults in `.veerc`, the current values for `Trig Mode`, `Number Formats`, and `Waveform Defaults` are saved with each model and restored when the saved model is `Open`ed.

See Also

`Preferences`.

Secure

Locks the panel view of a model.

Use

Use `Secure` to lock the main panel view or a `UserObject` after it is completed and before users have access to it. Access to the detail view is no longer available.

When you use `Secure`, you are prompted to save the unsecured model. After the unsecured model is saved, save the secured model under another name. There is no `UnSecure`, so be sure to save your model or `UserObject` before securing it.

Location

`UserObject (Object Menu)` $\Longrightarrow$ `Secure`

`UserObject` $\Longrightarrow$ `Secure`

Notes

A secured model loads and runs faster than an unsecured model.

You can `Secure` an individual `UserObject` panel view, too.

See Also

`UserObject`.

Select Bitmap (Object Menu)

Selects a bitmap to be displayed on the icon.

Use

Use `Select Bitmap` to choose a bitmap from your file system to be displayed on this icon.

Location

On the object menu ⟹ `Layout` ⟹ `Select Bitmap`

Notes

`Select Bitmap` is only available from the icon of an object.

HP VEE will examine the file to determine the file format. The currently supported formats include X11 Bitmap, X11 Window Dump (xwd), and GIF.

If you have HP VEE-Test, the bitmaps supplied with HP VEE are stored in the `/usr/lib/veetest/bitmaps` directory; if you have HP VEE-Engine, bitmaps are stored in `/usr/lib/veeengine/bitmaps`.

`Start`, `Toggle`, and `OK` do not have this menu feature.

See Also

`Delete Bitmap`, `Layout`, `Object Menu`, and `Show Label`.

Select Objects

Selects a set of objects to be acted on.

Use

Before you use the object features in the `Edit` menu, you must select a set of objects to be acted on. Choose `Select Objects` and then click on the objects to put in the set. When an object is selected, it has a shadow highlight. Click on an empty area of work area to end `Select Objects` mode. Then choose an editing function such as `Cut` or `Copy`.

Location

`Edit` ⟹ `Select Objects`

Notes

`Select` only selects objects, it does not deselect them. If you select an object by mistake, end select mode by clicking on the work area, then deselect all objects by clicking on the work area a second time.

Any objects previously selected are deselected when `Select Object` is used.

Short Cuts

Hold down the CTRL key while clicking on the left mouse button objects. This shortcut allows you to toggle the selection of objects. This method does *not* deselect any previously selected objects.

Clicking the left button on an object selects it, however, it also deselects objects already selected.

See Also

`Add To Panel`, `Clone`, `Copy`, `Create UserObject`, `Cut`, `Move Objects`, and `Paste`.

Sequencer

An object that executes a series of sequence transactions, each of which may call a `UserFunction`, `Compiled Function`, or `Remote Function`.

Use

Use the `Sequencer` to control the order of calling of a series of UserFunctions, Compiled Functions, Remote Functions or any other HP VEE function by specifying a series of sequence transactions. Typically, the `Sequencer` is used to perform a series of tests.

The `Sequencer` contains a list of sequence transactions. Each of these transactions evaluates an HP VEE expression, which may contain a call to a UserFunction, Compiled Function, or Remote Function. After evaluating the HP VEE expression, a transaction compares the value returned by that expression against a test specification. Depending on whether the test passes or fails, the transaction then evaluates different expressions and selects the next transaction to be executed. Transactions may optionally log their results to the `Log` output pin, or to a UserFunction, Compiled Function, or Remote Function specified in the `Logging Config` dialog box.

The `Return` output pin of the `Sequencer` can be used for returning a result when the sequence has completed. By default the `Return` output pin has a value of zero, but a transaction with a `Next Operation` of `Then Return` will set the `Return` output pin to the specified value.

The `Log` output pin of the sequencer contains a record that contains one logging record for each transaction that has logging enabled. See "Logging Config" below for more information.

Location

`Device` ⟹ `Sequencer`

Open View Parameters

The open view shows the list of transactions to be executed.

Dialog Information

You can change the parameters for an individual transaction. Just double-click on a transaction to display the `Sequence Transaction` dialog box for that transaction. The available parameters are as follows:

- *Sequence Mode:*
  - `TEST` - Executes the specified expression and tests the result against a specification. (If the expression calls a `UserFunction`, a compiled function or `Remote Function`, the result on the top-most output pin of the corresponding `UserObject` is used.)
  - `EXEC` - Executes the specified expression without performing a pass/fail test against a specification. *Logging is never performed on* `EXEC` *transactions.*
- *Transaction Name* - Unique name of this transaction. Must be a valid HP VEE variable name (must begin with a letter, but may include letters and numbers). This name can be referenced by expressions in this or other transactions, and will contain the logging record for the most recent execution of this transaction.
- *Enable Condition Type:*
  - `ENABLED` - Sequencer unconditionally executes this transaction.
  - `DISABLED` - Sequencer never executes this transaction.
  - `ENABLED IF:` - Sequencer executes this transaction only if the given expression is non-zero.
  - `DISABLED IF:` - Sequencer does not execute this transaction if the given expression is non-zero.
- `SPEC NOMINAL:` - Nominal (expected) value of the test result. This value is not used in calculating `RANGE` or `LIMIT` specification tests, but may be included in the logging record.

- *Spec Type* - All specification tests are done in the same manner as the `Comparator` object. Thus, the test result may be a waveform, and the test limit(s) may be another waveform or an array of type `Coord`. The spec test will automatically perform the necessary interpolation to determine if each data point is in or out of specification. Also, all tests use an "almost equal" algorithm that checks equality or inequality to at least six significant digits. See the `Comparator` object description for more information. Options include:
  - `RANGE:` - Specifies a lower and upper limit and the comparison operators that apply.
  - `LIMIT:` - Specifies a single limit and the comparison operator that applies. May be used to test for an upper or lower limit, or for an exact match. The limit value will be logged in the `HighLimit` record field.
  - `TOLERANCE:` - Specifies a plus and minus tolerance from the `Nominal` value. A tolerance of zero is permitted. `HighLimit` and `LowLimit` logging record fields contain the specification range after the tolerance has been applied to the `Nominal` value.
  - `%TOLERANCE:` - Specifies a plus and minus percent tolerance of and from the `Nominal` value. A tolerance of zero is permitted. `HighLimit` and `LowLimit` logging record fields contain the specification range after the tolerance has been applied to the `Nominal` value.
- `FUNCTION` - An HP VEE expression that specifies the test. The expression may just be a call to a `UserFunction`, `Compiled Function`, or `Remote Function`. Or it may be a larger expression of input pin names, `UserFunctions`, `Compiled Functions`, `Remote Functions`, and test records from previous transactions in this `Sequencer`. If this is a test transaction, the single result from this field will be tested against the specification. The result of a `UserFunction` or `Remote Function` call is the value of the top-most output pin of the corresponding `UserObject`, or nil if no output pins are present.
- *Logging Mode:*
  - `LOGGING ENABLED` - Add a field for this transaction to the `Log` record output pin (if present). If a "`Log Each Transaction To:`" procedure has been specified in `Logging Config` in the object menu, that logging procedure will be called immediately after this transaction is completed.

- LOGGING DISABLED - A log record will not be generated for this transaction, and the "Log Each Transaction To:" procedure will not be called. (See "Logging Config" under "Object Menu" below for options and more information.)

- *Pass Operation:*
  - IF PASS - If the spec test passes (test value within limits), the specified "then" action will be performed.
  - IF PASS CALL: - If the spec test passes (test value within limits), the specified HP VEE expression—which may include a call to a UserFunction, Compiled Function, or Remote Function—will be evaluated and the specified "then" action will be performed.
- *Fail Operation:*
  - IF FAIL - If the spec test fails (test value outside of limits), the specified "then" action will be performed.
  - IF FAIL CALL: - If the spec test fails (test value outside of limits), the specified HP VEE expression—which may include a call to a UserFunction, Compiled Function, or Remote Function—will be evaluated and the specified "then" action will be performed.
- *Next Operation:* (Applies to EXEC transactions, or to either "Pass" or "Fail" operations in TEST transactions.)
  - THEN CONTINUE - Goes to the next transaction in the Sequencer list. (This is the default operation.)
  - THEN RETURN - Quits executing transactions in this Sequencer and places the value of the specified expression on the Return output pin of the Sequencer. The Sequencer object will then fire its output pins and execution flow within the model will continue normally.
  - THEN GOTO - Goes to the transaction in this Sequencer with the specified transaction name.
  - THEN REPEAT - Executes this transaction again, repeating up to the specified number of times. If the Pass/Fail condition still exists after the maximum number of repeats, continues to the next transaction.

- THEN ERROR - Stops execution of the Sequencer by generating an error condition with the given error number. An error can be trapped with an Error output pin on this Sequencer, or on any enclosing UserObject.
- THEN EVALUATE - Evaluates the given expression—which may call a UserFunction, Compiled Function, or Remote Function—and uses the string result to determine the next operation. Valid string results from the expression are:

```
CONTINUE
RETURN <expr>
GOTO <name>
REPEAT <expr>
ERROR <expr>
```

  Where <expr> is any valid HP VEE expression and <name> is the name of a transaction in this Sequencer. This operation can be useful when a UserFunction, possibly with user interaction, needs to determine what action the Sequencer should perform next.

- DESCRIPTION - "Comment" field for this transaction. Text entered into this field will be shown at the end of the transaction's abbreviated text in the Sequencer open view.

Object Menu

- Step Trans - Causes the currently highlighted transaction to be executed, using the last data sent to the input pins. The IF PASS or IF FAIL expressions are executed as usual, and the next transaction specified by these rules will be highlighted. (Note that Ctrl X is a shortcut for Step Trans when the cursor is over the Sequencer.)
- Logging Config - Selects which fields will be generated in the logging record for each transaction. (The Log output pin contains a record of records — one logging record for each transaction with logging enabled. In each transaction logging record the selected fields, specified below, may be present.) The names (without spaces) are the field names of the logging record. The choices are:
  - Name - A unique string that identifies this transaction (not the procedure name).

- Result - The value returned from the evaluation of the test procedure.
- Nominal - The nominal test value.
- High Limit - The upper limit used in the specification test. Actual limit values, not offsets are given for Tolerance and %Tolerance specifications.
- Low Limit - Lower limit used in specification test. Actual limit values, not offsets, are given for Tolerance and %Tolerance specifications.
- Pass - An Int32 with value "1" for "Pass", "0" for "Fail".
- Time Stamp - A Real value containing day, date, and time of completion of the test. (Refer to the Time Stamp object for more information.)
- Description - A Text value containing the description "comment" field in the Transaction.

Under Logging Config you can also choose the type of logging:

- Log to Output Pin Only - A logging record will be present in the record on the Log output pin for each transaction that has Logging Enabled. If a transaction is executed more than once during one execution of the Sequencer, only the last logging record for that transaction will be present in the record on the Log output pin. If a transaction has not been executed during an execution of the Sequencer, the logging record for that transaction will contain "0" (Real) values for each field except the Name and Description fields, which contain null strings.
- Log Each Transaction To: - Specify a UserFunction, Compiled Function, or Remote Function expression that will be called after each transaction that has Logging Enabled. To pass the logging record for the current transaction to the UserFunction, Compiled Function, or Remote Function, use the reserved variable name "ThisTest". The Log output pin will continue to operate in the same manner as for the "Log to Output Pin Only" mode.

- Add Trans - Adds a transaction to the end of the list.
- Insert Trans - Inserts a transaction before the currently highlighted transaction.
- Cut Trans - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.

- `Copy Trans` - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- `Paste Trans` - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Notes

You can add an `Exec Trans` control input pin. The `Exec Trans` control input executes only the transaction(s) with the specified name(s) in the order given by a `Text` scalar or `1D` array on the control input. The highlight bar will be moved to the transaction currently being executed. The `If Pass` or `If Fail` expressions will be evaluated as usual, but the result of the `Then` field will have no effect on the next transaction to be executed.

All field expressions (such as `ENABLED IF`, `FUNCTION`, or spec limits) may contain input pin names, UserFunction calls, Compiled Function calls, Remote Function calls, HP VEE math functions, or record names of previously run transactions. In test transactions with `LOGGING ENABLED`, the reserved variable, `thisTest` will contain a record of the logging fields that have been calculated. For example, to use the test value from transaction `test4`, which had logging enabled, use the name `test4.result` in any expression. Or to use the current test name in any expression, use `thisTest.name`. Or, an `Enabled If` expression might be `random()<0.25` to cause the test to be executed only 25 percent of the time.

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

Ctrl X is a shortcut for `Step Trans` when the cursor is over the `Sequencer`.

See Also

Call Function, Comparator, Raise Error, Record, Time Stamp, Unbuild Record, and UserObject.

Set Breakpoints

Creates execution breakpoints at the selected objects.

Use

Use `Set Breakpoint` as a debugging tool to stop the execution of the model before each selected object. To continue execution, press the `Cont` button in the upper right corner of the HP VEE window; to operate the object with the breakpoint only, press `Step`.

After the breakpoint is set, the object is highlighted with an outline border (black, by default).

Location

`Edit` ⟹ `Breakpoints` ⟹ `Set Breakpoints`

Notes

To set the breakpoint of a single object, use the `Breakpoint` selection on the object menu.

When an object has a breakpoint set, it is checked on the object menu next to the breakpoint menu feature.

If no objects are selected, `Set Breakpoints` is not available.

`Breakpoints` are saved with the model. If you do not want the `Breakpoint` to reappear when the model is opened, use `Clear Breakpoints` before saving.

See Also

`Activate Breakpoints`, `Breakpoint (Object Menu)`, `Clear All Breakpoints`, `Clear Breakpoint`, `Cont`, `Select Objects`, and `Step`.

Set Field

An object that allows you to modify a field of an existing record.

Use

Use `Set Field` to modify the value of a field of an existing record. The `Set Field` object is actually an assignment object that assigns the value specified by the *right-hand expression* to the field specified by the *left-hand expression*. The value in the right hand expression can evaluate to a scalar, an array, another record, or a field in a record.

`Set Field` is a short-cut for un-building a record, changing one field, and re-building the data into a record. Note that the output terminal name must match the name of the input record to be modified. Also, the left and right-hand expression results must match in type and shape. You cannot use `Set Field` to change the "schema" (type and shape) of the input record.

Location

`Data` ⟹ `Access Record` ⟹ `Set Field`

Open View Parameters

- *Left-hand expression*—The left-hand expression allows a subset of `Formula` mathematical syntax, which allows you to specify a field of a record. The `A.B` dot syntax, as well as a subset of the `A[2]` sub-array syntax, is supported to specify a particular field of the record to be changed. See "Notes" below for further details of the allowed syntax.
- *Right-hand expression*—The right-hand expression allows any mathematical expression that is allowed in a `Formula` object. This expression has access to all input containers, global variables, and functions. The right-hand expression is evaluated and the resulting value is stored in the field of the record specified by the left-hand expression.

Notes

The name of the record output pin must match the name of the input record to be modified, and the left-hand expression must begin with that name. For example, if the record pins are named `Rec`, the left-hand expression must be something like `Rec.A` or `Rec[1]`. The left-hand expression must be a single expression that specifies which field of the input container is to be modified.

There may be confusion over the names of inputs, outputs, and record fields. For example, if the name of the record input pin and the record output pin is `Rec` (they must match), the expression `Rec.A` specifies the `A` field in the input record. On the other hand, the expression `A*2` refers to the `A` data input of the `Set Field` object. Thus, `Rec.B=A*2` sets the `B` field of the output record equal to twice the value of the `A` data input. On the other hand, `Rec.B=Rec.A` sets the `B` field of the output record equal to the value of the `A` field of the input record.

The complete input record container is output on the record output pin, but the specified field is modified. For example, suppose the input `Rec` is a record with two fields `x` and `y`, where `x` is a Text scalar with value "hello" and `y` is a Real scalar with the value 1.2. If the left-hand expression specifies `Rec.y`, and the right-hand expression evaluates to 5.6, the output container will be a record with two fields `x` and `y`, where `x` is a Text scalar with value "hello" and `y` is a Real scalar with the value 5.6.

The field specified by the left-hand expression and the result of the right-hand expression must match in type and shape ("schema"). That is, the operation of the `Set Field` object cannot change the schema of the input record—only the data in the specified field can be changed. For example, suppose input `Rec` is a record with two fields `x` (a Text scalar) and `y` (a Real scalar). If the left-hand expression specifies `Rec.y`, the right-hand expression must result in either a Real scalar, or something that can promote to a Real scalar. Otherwise, an error will result.

Record arrays require special attention. A record array is an array consisting of individual record elements, each consisting of fields. You can use `Set Field` to change one field of one element in a record array. For example, if you create a record 1D array and you want to change the `f` field of the third record element of the array, specify `Rec[2].f` (for a zero-based array) in the left-hand expression. In this case, the result of the right-hand expression is placed in the

f field of the third element of the record 1D array. None of the other fields of Rec[2] are changed, nor are any fields of the other record elements of the array (for example, Rec[0].f and Rec[1].f).

On the other hand, you can use Set Field to modify a particular field in every record element of a record array. For example, suppose you specify Rec.f in the left-hand expression where Rec is a record 1D array with n elements. In this case, the result of the right-hand expression is copied into every Rec[x].f field, where x is from 0 through n-1. In other words, the f field is changed in every record element of the array.

The left-hand expression must be a single expression specifying which part of the input record is to be modified. For example, Rec+2 and sine(Rec) are not valid as left-hand expressions. Also global variables are not allowed in the left-hand expression.

The syntax allowed for the left-hand expression is limited:

- The A.B dot sub-record syntax may be used any number of times for record of record containers (there is no limit on recursion). Suppose that record A has a field b, which is itself a record. The record field b has a field c, which is a record containing field d. To specify that lowest level field d, you can use A.b.c.d as the left-hand expression.
- The A[x] sub-array syntax is more limited, and may be used only once, before the last dot. For example, A.b.c[1].d is valid as a left-hand expression, but A[2].b.c.d and A[2].b[2].c are not allowed. Also, A[1] and A.b.c[1:2] are not allowed. However, the A[x] sub-array syntax, supported before the last dot, does support the "colon and asterisk" syntax. For example A[1:2].b and a.b.c[2:*].d are valid left-hand expressions.

Note the difference between A[1].b and A.b[1]. The first is a record 1D with a field b. The second is a scalar record containing a field b, which is a 1D array.

See Also

Build Record, Formula, Get Field, Record Constant, SubRecord, and UnBuild Record.

Set Global

An object that sets the data value of a global variable.

Use

Use the `Set Global` object to set the data container (type, shape, and values) of a global variable (by name). This global variable can then be used by name in other parts of the HP VEE model.

Global variables created (with `Set Global`) in one context of a model can be used as data in another context of that model. For example, a `Set Global` in the root context of a model could create a global variable, which could then be used as data by including a `Get Global` in a `UserObject`. This is especially useful when the model contains several nested layers of `UserObjects`.

Global variables that are set with `Set Global` may also be used by name in the expression fields of the following devices: `Formula`, `If/Then/Else`, `Get Values`, `Get Field`, `Set Field`, `From DataSet` and all devices using expressions in transactions, including `To File`, `From File`, `Direct I/O`, `From Stdin`, `To/From Named Pipes`, and `Sequencer`. Refer to "Using Global Variables in Expressions" in chapter 3 for further information.

Location

`Data` ⟹ `Globals` ⟹ `Set Global`

Open View Parameters

The open view displays a field for the name of the global variable. The name is not case sensitive (either lower-case or upper-case letters may be used). Thus `globalA` is the same variable as `GLOBALa`. The name field may be added as a control input.

Notes

To avoid unexpected results, your model must ensure that a global variable is set with `Set Global` *before* a `Get Global` object (or an object that includes the global variable name in an expression) executes. Generally, the best way to ensure this is to connect the sequence output pin of the `Set Global` object to the sequence input pin of the `Get Global` object, or other object that uses the global variable. However, there are cases when the sequence input pin need not be connected. For further information about this, refer to "Using Global Variables" in chapter 3 of *Using HP VEE*.

All global variables are deleted at the beginning of every `Run`, `Start`, or auto-execution. Global variables are always deleted by either `File` ⟹ `New` or `File` ⟹ `Open`. Global variable values are not saved with the model.

Global variables are truly global since they are not defined at a `UserObject` level. A global variable that is defined in one context of a model can be used in any other context within the model. For example, you can define a global variable with a `Set Global` object in the root context of the model, and then include a `Get Global` to get that global variable in a `UserObject`. However, you will need to avoid name conflicts throughout the model:

- If two or more `Set Global` objects attempt to set the same global variable (with the same name), the current value will be overwritten as each `Set Global` executes. This may result in unexpected behavior.
- If there is a local input variable with the same name as a global variable, the local variable will take precedence.

For further information, refer to "Using Global Variables in UserObjects" in chapter 6 of *Using HP VEE*, and to "Using Global Variables in Expressions" in chapter 3 of this manual.

See Also

`Get Global`, and `View Globals`.

Set Mappings

An object that assigns mapping to an array.

Use

Use `Set Mappings` to map one or more dimensions of an array linearly or logarithmically over a range of real values.

Location

`Data` ⟹ `Access Array` ⟹ `Set Mappings`

Open View Parameters

- `Num Dimensions` - The number of dimensions in the input array. `Num Dimensions` must be an integer. Minimum is 1. Maximum is 10. Default is 1.

  Changing the value of `Num Dimensions` affects `Mapping`, `From`, and `To` by corresponding adding or deleting open view fields.

- `Mapping` - `Linear` or `Log` for each dimension. Default is `Linear`.
- `From` - The lower limit of the mapping range (inclusive) for each dimension.
- `To` - The upper limit of the mapping range (inclusive) for each dimension.

`From` and `To` may be added as data inputs.

Notes

If a Waveform or Spectrum is input, its elements are remapped to your specifications. Log mapping is not allowed on a Waveform.

There are the same number of points as sampling intervals. Each point is at the beginning of the sampling interval.

`Set Mappings` does not allow an array of Coord as input because the Coord array is already explicitly mapped.

The mappings on arrays are used in the `XY` displays for Autoscale and various math functions.

The `From` and `To` values must be different.

The value of the `Num Dimensions` field must match the number of dimensions of the input array.

See Also

`Alloc Array`, `Collector`, `Get Mappings`, `Line Probe`, and `Sliding Collector`.

Using HP VEE, chapter 3.

Set Values

An object that allows you to change an element of an array container.

Use

Use `Set Values` to modify elements of an array of any type and size. The new array is generated only when the XEQ pin is activated.

You can connect several containers to the inputs on the `Set Values` object: the array to be changed, the scalar data to be added, and the indices into the array where in the array to put the scalar data.

Input new values for the array and the index of the element you wish to replace. When all changes are done, activate the `XEQ` pin.

Location

`Data` ⟹ `Access Array` ⟹ `Set Values`

Open View Parameters

`Num Dimensions` - The number of dimensions in the input array. `Num Dimensions` must be an integer (not an expression). Minimum is 1. Maximum is 10. Default is 1.

Notes

All arrays are zero-based; all indices for arrays are zero based.

To initially create an array use the `Allocate Array` ⟹ objects.

The open view of the `Set Values` object has a field on it for the number of dimensions of the array coming in. This allows the object to add or delete input pins to give you the correct number of inputs for the indices needed. For example, if you change the number of dimensions field to `2`, the object automatically adds an input for the second index into the array.

The `Set Values` object begins with its internal buffer cleared. When the first three data pins are activated, it starts to operate. It copies the array container on the first input pin into its internal buffer, making sure the number of

dimensions match the field on the open view. It then takes the second data input and makes sure it can be converted to the same type as the input array. Then, HP VEE takes the value(s) on the index input(s) and makes sure they are within range of the array sizes. Finally, HP VEE places the data value in the array at the appropriate indices. When the XEQ pin is activated, the `Set Values` box sends the modified array out the output pin. The `Set Values` object clears its internal buffer.

The `Set Values` object always clears its internal buffer at PreRun and at Activate time.

See Also

`Alloc Array`, `Build Data`, `Collector`, `Get Values`, `ramp(numElem, from, thru)`, `Set Mappings`, and `Sliding Collector`.

Shift Register

An object that outputs the previous values of the inputs.

Use

Use `Shift Register` to access the previous values of the input.

Location

`Device` ⟹ `Shift Register`

Object Menu

- `Clear` - Clears the contents of the `Shift Register`.
- `Clear at PreRun` - Clears the contents of the `Shift Register` at PreRun. Default is on (checked).
- `Clear at Activate` - Clears the contents of the `Shift Register` at Activate. Default is on (checked).

Notes

Add outputs to `Shift Register` to access previous values.

The `Shift Register` starts with all its outputs set to nil. Each time the object is activated, the data from the input is copied to the "current" output terminal. Data that was in the current output is moved to the `1 Prev` output. The data from previous executions is shifted down the output terminals to the last output terminal.

Additional outputs may be added so the user may access the data from the "n" previous executions of a thread or model. Turn `Clear at PreRun` and `Clear at Activate` off from the object menu to retain data over successive model executions.

See Also

`Logging AlphaNumeric` and `Sliding Collector`.

Short Cuts

Displays information about keyboard accelerators.

Use

Use `Short Cuts` to get a list of easier, faster ways to accomplish common tasks.

Location

`Help` ⟹ `Short Cuts`

See Also

`Help`.

Show Config (Object Menu)

Displays a dialog box that shows the I/O configuration of an instrument driver panel or a direct I/O object.

Use

From the object menu of an instrument driver panel or a direct I/O object, select `Show Config` to view the configuration specified with `Configure I/O`.

To change any configuration parameter use `I/O` ⟹ `Configure I/O`. Note that any change will affect all other objects using that instrument with Direct I/O. To change the device in a `Direct I/O` object, edit the name field to choose a new device. Note that the transactions in the `Direct I/O` object may not be compatible with the new device.

Location

On an instrument driver panel or a direct I/O object:

From the object menu ⟹ `Show Config`

See Also

`Configure I/O`.

Show Data Flow

Shows the route that data takes through the model.

Use

Use `Show Data Flow` as a debugging tool to show the flow of data between input and output pins. A small square marker is used to trace the flow of data propagation along the lines. Even nil data is shown this way

When `Show Data Flow` is set, the menu selection has a checkmark on its left.

Location

`Edit` ⟹ `Show Data Flow`

Notes

`Show Data Flow` is often used with `Show Exec Flow`. These debugging tools slow the execution of your model.

See Also

`Breakpoints`, `Line Probe`, `Show Exec Flow`, and `Step`.

Show Description

Displays an editable dialog box that displays user-supplied information. Accessible from the object menu for information about an object, or from the `File` menu for information about the entire model.

Use

Use `Show Description` to document an object, and explain how or why it's used. Or, from the file menu, use `Show Description` to document the entire file.

To write a description, click in the text area and type the description. Click on `OK` when you're finished.

Location

On each object menu ⟹ `Show Description`

or

`File` ⟹ `Show Description`

Notes

To edit the text area, all the usual HP VEE edit functions work including the following keys: Clear line, Insert line, Delete line, Insert char, Delete char, Prev, Next, and the cursor keys.

When printing using `Print All` with `Print All Objects` selected or printing using `Print Objects`, the description for the object will also be printed.

See Also

`Note Pad` and `Object Menu`.

Show Exec Flow

Highlights the object that is currently executing.

Use

Use `Show Exec Flow` as a debugging tool to show the currently operating object. A highlighted border (default color yellow) appears around the object and remains until the operation is complete. The currently executing transaction will also be highlighted in objects that include a list of transactions, such as `Direct I/O`, `Sequencer`, `To File`, `From File`, etc. When `Show Execution Flow` is on, the menu selection has a checkmark on its left.

Location

`Edit` ⟹ `Show Exec Flow`

Notes

`Show Exec Flow` is often used with `Show Data Flow`. These debugging tools slow the execution of your model.

See Also

`Breakpoints`, `Line Probe`, `Show Data Flow`, and `Step`.

Show Label (Object Menu)

Toggles the display of the object name on the icon.

Use

When checked, `Show Label` displays the name of the object on the icon. When `Show Label` is not checked, the object name is not displayed on the icon. This feature is used if you only want a bitmap to be displayed on the icon.

Location

On the object menu ⟹ `Layout` ⟹ `Show Label`

Notes

`Show Label` is only available from the icon of an object.

See Also

`Delete Bitmap`, `Layout`, `Object Menu`, and `Select Bitmap`.

Show Terminals (Object Menu)

Toggles the display of the input and output terminals on an open view.

Use

When checked, `Show Terminals` displays the input and output terminals on the open view. The displayed terminals show the terminal name and type. To access complete terminal information, double-click on the terminal. When `Show Terminals` is not checked, the input and output terminals are not shown on the open view.

Location

On the object menu ⟹ `Terminals` ⟹ `Show Terminals`

Notes

`Show Terminals` is only available from the open view of an object.

To get or change information about the terminal, double-click on the displayed terminal. You can change any value that is in an entry field or a button.

See Also

`Line Probe`, `Object Menu` and `Terminals`.

Show Title (Object Menu)

Toggles the display of the title bar on the open view of the object.

Use

When checked, `Show Title` displays the object's title bar. When `Show Title` is not checked, the title bar is not displayed. Remember that you can access the Object Menu by placing the cursor over the object and then pressing the right mouse button.

Location

On the object menu ⟹ `Show Title`

See Also

`Layout`, and `Show Label`

Size (Object Menu)

Changes the size of this object.

Use

Use `Size` to change the size of the icon or the open view. After you select `Size` the pointer changes to a ⌟ (a lower right corner bracket). Drag the bracket to the new size of the object; an outline rectangle shows you the size of the object. Another way to resize is to position the bracket pointer to where you want the lower right corner of the object and click.

Location

On the object menu ⟹ `Size`

Notes

`Size` retains the connections with other objects. If `Automatic Line Drawing` is set, it may take a few moments to redraw lines.

Short Cuts

You may resize an object before you place it on the work area by positioning the outline rectangle at the desired location, pressing the left mouse button, then dragging the bracket pointer to the position you want for the lower right corner of the object. Release the mouse button and you have the size and placement you need.

See Also

`Object Menu`.

Sliding Collector

An object that outputs an Array 1D.

Use

Use `Sliding Collector` to create an Array 1D from scalar input data.

Location

`Data` ⟹ `Sliding Collector`

Object Menu

- `Clear` - Clears the contents of the `Sliding Collector`.
- `Clear at PreRun` - Clears the contents of the `Sliding Collector` at PreRun. Default is on (checked).
- `Clear at Activate` - Clears the contents of the `Sliding Collector` at Activate. Default is on (checked).

Open View Parameters

- `Array Size` - The number of elements in the output array. `Array Size` can be a data input. Default is 10.
- `Trigger Every` - The number of times the input pin must be activated before an array is output. Default is 10.

Both parameters may be set from the open view or from data input pins. Parameters must be integers.

`Enum` inputs are converted to type `Text` before the data is placed into the array.

Notes

`Sliding Collector` builds an Array 1D by automatically sequencing through the array index. Generally, `Array Size` and `Trigger Every` are the same size.

The output array is always an Array 1D of length `Array Size`. The type of the output array is determined by the first input data after a `Clear`.

If `Trigger Every` is smaller than `Array Size`, `Sliding Collector` generates intermediate outputs. For example, if `Array Size` is `4` and `Trigger Every` is `2`, the `Sliding Collector` outputs an array four long after two data inputs. The first two elements of the output are the data elements input; the last two elements of the output are `0`.

If `Trigger Every` is larger than `Array Size`, some of the input elements are discarded. For example, if `Array Size` is 2 and `Trigger Every` is 4, the output is an array two long with the last two pieces of data in it. The first two pieces of data input are discarded.

See Also

`Allocate Array`, `Collector`, `Concatenator`, `Get Values`, and `Set Values`.

Spectrum (Freq)

A menu item that contains frequency-domain displays.

Use

Use `Spectrum (Freq)` to access the following frequency-domain displays:

- `Magnitude Spectrum`
- `Phase Spectrum`
- `Magnitude vs Phase (Polar)`
- `Magnitude vs Phase (Smith)`

Location

`Display` ⟹ `Spectrum (Freq)` ⟹

Notes

Inputs must be Waveform, Spectrum, or an array of Coords.

If a Waveform is input to a Spectrum display, it is automatically transformed into a Spectrum by way of a Fast Fourier Transform (fft).

See Also

`Magnitude Spectrum`, `Magnitude vs Phase`, `Phase Spectrum`, and `Waveform`.

SPOLL

This menu item has been replaced with a serial poll option under `Device Event`.

Start

An object that, when its button is pressed, starts the execution of a thread.

Use

The `Start` object is normally used to define where to initiate execution in a thread that includes feedback.

The `Start` object, when its button is pressed, begins the execution of the thread to which it is connected, but has no effect on other threads. (Pressing `Run` starts the execution of all threads in the model, regardless of whether they have `Start` objects.)

Location

`Flow` ⟹ `Start`

Notes

The `Start` object is not required, except for threads that include feedback. In a feedback thread, the `Start` button is necessary to specify where execution is to begin.

If a model includes more than one independent thread, you can use `Start` to initiate a single thread without running the other threads.

When you press `Run` or `Start`, propagation begins with `Start` object(s). If more than one `Start` object is on a thread, propagation begins at each `Start` and proceeds in parallel as far as allowed by the propagation rules.

If you press `Start` inside a `UserObject`, the `UserObject` runs but its data output pins *do not* activate; therefore, propagation does not continue.

See Also

`Raise Error`, `Exit Thread`, `Exit UserObject`, and `Stop`.

State Driver

Selects an I/O object to control an instrument using an instrument driver where all the panels are available.

Use

Click on `I/O` ⟹ `Instrument` and examine the list of configured instruments in the `Select an I/O Device` dialog box.

If the instrument you want is in the list and is properly configured:

l. Click once on the desired instrument to highlight it. (If the instrument is not configured with an ID Filename, the `State Driver` and `Component Driver` buttons are flat (grayed)).

2. Click the button at the bottom of the dialog box labeled `State Driver`.

If the instrument you want is not in the list or is not properly configured, use `Configure I/O` to change the existing instrument, then:

1. Click on `Add` to add a new instrument.

2. Complete the resulting dialog boxes. Refer to the `Configure I/O` entry for details about how to complete these dialog boxes.

All instruments must be configured before they can be accessed by way of the `Instruments` menu selection. The best way to configure instruments is to use the `Configure I/O` menu selection.

Instrument objects may be operated with or without live instruments connected to the computer. If you wish to control a live instrument, you must set a correct, non-zero address and enable Live Mode. The address and Live Mode setting are controlled by the `I/O` ⟹ `Configure I/O` menu selection. If the address is zero or if Live Mode is off, the instrument object operates but does not attempt to communicate with a physical instrument.

State drivers can be used interactively or within a model. To set the value of an individual component, click on the field containing the value of the component and complete the resulting dialog box. To make a measurement and display the result, click on the corresponding numeric readout or XY display inside the State Driver open view.

State Driver

It is possible to have more than one object controlling a single instrument, that is, two `State Driver` objects for "fgen" in the same model. These objects performs state recalls when operated.

It is possible to add inputs and outputs for the components inside the driver, similar to `Component Drivers`. Inputs perform the set actions of the component, outputs perform the get actions of the component.

Your system administrator must properly configure your computer before it is possible to communicate between HP VEE and any hardware interface. If you believe that you have properly followed all HP VEE procedures properly and you still cannot achieve any level of communication with an instrument, the problem may be with your computer configuration. Ask your system administrator to read this explanation and verify proper configuration of your system's interface drivers. (These interface drivers are different from the instrument driver files included with HP VEE.)

Location

`I/O` ⟹ `Instrument` ⟹ `State Driver`

Object Menu

- `Add Terminals` ⟹ `Select Input Component` - Use to add fields visible on the panel as inputs to a `State Driver`.
- `Add Terminals` ⟹ `Select Output Component` - Use to add fields visible on the panel as outputs to a `State Driver`.
- `Show Config` - Displays the configuration of the instrument. `State Driver` can only be edited using `Configure I/O`.

See Also

`Component Driver`, `Direct I/O`, and `Instrument`.

Using HP VEE, chapter 5.

Step

A button that causes all threads to operate one primitive object at a time.

Use

Use `Step` to debug your model. `Step` operates one primitive object every time the `Step` button is pressed. An arrow points to the object currently operating.

Location

On the right side of the title bar.

Notes

All objects except `UserObjects` are primitive objects. A `UserObject` is *not* treated as a primitive object when it is displayed as a detail view and `Show On Exec` is off.

`Step` is not available from the panel view.

`PreRun` occurs the first time you click on `Step`.

See Also

`Cont`, `Run`, `Show Data Flow`, `Show Exec Flow`, and `Start`, and `Stop`.

Stop

A button that causes all threads to stop running.

Use

Use `Stop` to stop model execution. After `Stop` is pressed, you may press `Cont` to continue execution.

Location

On the right side of the title bar.

Notes

Pressing `Stop` once puts the model in the "paused" state. Both `Step` and `Cont` are active and can be used to proceed.

If `Stop` is pressed a second time, the model goes to the "stopped" state which clears any highlighting and closes any open files and I/O channels.

Short Cuts

Press CTRL-C to pause a running model or cancel an edit.

See Also

`Cont`, `Run`, `Show Exec`, `Step`, and `Stop (Object)`.

Stop (Object)

An object that stops the execution of a model.

Use

Use `Stop` to pause the execution of all threads in the model and return an error code. `Stop` is usually used after testing for a certain condition.

Location

`Flow` ⟹ `Stop`

Open View Parameters

`Exit Code` - A number that indicates the status of the model. You can type the `Exit Code` in the recessed rectangle or add it as a data input. The `Exit Code` is an 8-bit integer that is output to the operating system; if the output is greater than 255, it is output modulo 256 (256 is output as 0).

If you run HP VEE with the `-r` option, `Stop` exits HP VEE.

See Also

`Exit Thread`, `Exit UserObject`, `Raise Error`, `Start`, and `Stop`.

Strip Chart

An object that displays continuously-generated data.

Use

Use `Strip Chart` to display the recent history of data that is continuously generated. For each `y` input value, the `x` value is incremented by the specified `Step` size. When new data runs off the right side of the display, the display automatically scrolls to show you the latest data.

`Strip Chart` saves all data sent to it until it's cleared.

Unlike other graphical displays, array data will be appended to the end of the `Strip Chart` trace, rather than first clearing the trace.

Location

`Display` ⟹ `Strip Chart`

Object Menu

- `Auto Scale` ⟹ - Automatically scales the display to show the entire trace.
  - `Auto Scale` - Automatically scales both axes.
  - `Auto Scale X` - Automatically scales the X axis.
  - `Auto Scale Y` - Automatically scales the Y axis.

  These parameters may be added as control inputs.
- `Clear Control` ⟹ - Parameters that specify when to clear the display.
  - `Clear` - Clears the displayed trace(s). This parameter may be added as a control input.
  - `Clear At PreRun` - Clears the displayed trace(s) when the model or thread is PreRun.
  - `Clear At Activate` - Clears the displayed trace(s) when the User Object is activated.

- `Next Curve` - Resets the pen to display the next curve in a family of curves (data from the next time the display operates) *without* clearing the previous curve. `Next Curve` must be selected (or activated) before each new curve in the family. This parameter may be added as a control input.

- `Zoom` ⟹ - Scales the display.
  - `In` - Magnifies the display to contain only the rectangular region that you selected with the pointer. You select the region after selecting this feature by dragging on the graph area. This action outlines the information with a "rubber band" box.
  - `Out 2|5|10|20|50|100 x` - Expands the scales of the display by a factor in both the X and Y directions about the center.
- `Markers` ⟹ - Allows you to find the exact value of a data point on the displayed curve. If the data is plotted on a log scale, the values shown on the marker are the linear data points.

  To move markers to a different trace, click on the button to the left of the marker values near the bottom of the display. The button cycles through the different line types and colors of the different traces.
  - `Off` - No markers are shown.
  - `One On` - One marker is available.
  - `Two On` - Two markers are available.
  - `Delta On` - Two markers are available and the x and y differences between them are displayed.
  - `Interpolate` - When checked, you can place markers in between the displayed data points. The marker values displayed are calculated by linear interpolation. Default is off.
  - `Center` - If markers are available, brings them to the center of the visible part of the trace. This is useful if you've scrolled the display and markers are not visible.
- `Grid Type` ⟹ - Sets the type of grid marks. The value of the major x division is shown below the maximum x value (to the right of the x scale name). The value of the major y division is shown below the y scale name.
  - `No Grid` - No grid lines are shown.

- Tic Marks - Shows tic marks at the major and minor divisions on all four sides of the graph.
- Axis - Shows tic marks at the major and minor divisions on the X and Y axes of the graph. If the actual axes are scrolled off the graph area, axis lines are drawn on the edge closest to the axes.
- Lines - Shows lines at the major divisions and tic marks at the minor divisions. The X and Y axes are shown as thick lines.

■ Panel Layout ⟹ - Sets the open view's appearance.

- Graph Only - The open view shows only the graph area and the marker information (if it exists). No buttons, scales, scale names, or traces names are shown. This layout redraws quickest and provides the largest display area.
- Scales - The open view shows the graph area, the scales, the scale names, and trace names. These fields are not recessed and may not be edited. This layout is useful when recessed fields might be distracting, such as printing graphs, or when the fields should be protected from editing on a User Panel.
- Scales & Sliders - The open view shows all information about the graph. It includes the most information and allows you to modify the most elements interactively. It is the default.

■ Traces & Scales - A control panel that allows you to specify values such as the names, colors, line and point characteristics, minimum values, and maximum values for the traces and/or scales.

Traces:

- Name: - The name of the trace that is displayed to the left of the graph area and the name of the corresponding input terminal.
- Scale: - If you have multiple Y scales, selects which Y scale is to be used for this trace.
- Color: - The color of the trace. Each added trace has a different color than the existing traces. Default is Pen 4 (yellow).
- Lines: - The format of the line connecting data points. Default is a continuous line.

- `Points:` - The symbol that marks each data point. To show unconnected data points, select the single dot `Lines` format and the desired `Points` symbol. Default is a dot.

You can add a `Traces` control input pin. The control input data must be a record with the following fields: 1) A `TraceNum` field with an Integer value (1 is the top trace), *and* 2) one or more of the following fields: `Name`, `Pen`, `LineType`, `PointType`. (The `Pen`, `LineType`, and `PointType` values are integers from 0 to n, where 0 draws nothing.) Refer to "Records and DataSets" in *Using HP VEE* for further information.

Scales:

- `Show Scale:` - If you have multiple Y scales, a selection (using a check box) to specify if the end points and an axis of each additional right scale is displayed to the right of the graph area.
- `Scale Name:` - The names of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Maximum:` - The maximum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Minimum:` - The minimum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Mapping:` - The way the x and y data is mapped to the display. The mapping may be linear or log, but the x mapping must be linear. Default is `Linear`.
- `Log Cycles:` - The maximum number of decades shown (counting down from the maximum x and y values) when `AutoScale` is activated. `Log Cycles` is only used when the `Mapping` is `Log`. `Log Cycles` is useful when a trace contains a large dynamic range in the data. Default is 4.
- `Scale Colors` - The color of any background grid or tic marks. Default is `Gray`.

You can add a `Scales` control input pin. The control input data must be a record with the following fields: 1) A Text field `Scale` with a value `X`, `Y` (or `Y1`), `Y2`, or `Y3`, *and* 2) one or more of the following fields: `Name`, `Min`, `Max`, and `Mapping`. (The `Mapping` text value may be `Linear` or `Log`). Refer to "Records and DataSets" in *Using HP VEE* for further information.

- `Add Right Scale` - Adds up to two additional scales to permit traces to have different scale ranges. After adding a right scale, use `Traces & Scales` to assign a trace to the scale.
- `Plot` - Presents the `Plotter Configuration` control panel. When `OK` is pressed, a copy of the device's entire display is plotted on the selected plotter. This parameter may be added as a control input. If the current `Plotter Configuration` is in `Plot to File` mode, you may specify the destination file or directory name as string data on the `Plot` control input. If no control input value is given, the file or directory name specified in `Plotter Configuration` will be used. See `Plotter Config` for more information.

Notes

Inputs must be Scalar or Array 1D that can be converted to type Real. `Complex`, `PComplex`, and `Spectrum` values must be "unbuilt" to Real Values before being input to a `Strip Chart`.

Since the `Strip Chart` retains all the input data until it is cleared, you may need to occasionally clear the `Strip Chart` in a continuous loop to prevent the model from running out of memory.

To get the best update speed on the `Strip Chart`, set the grid type to `No Grid`. Use a `Panel Layout` of `Graph Only` or `Scales` and set the `X` scale maximum value so display scrolling is infrequent.

Add traces with the `Terminals` ⟹ `Add Data Input` object menu section. Up to twelve traces are allowed.

A `Title` control input may be added, which sets the title bar name to the specified text value. This allows programmatic control over the title shown when the display is printed or plotted.

See Also

`Complex Plane`, `Magnitude Spectrum`, `Polar Plot`, `Waveform (Time)`, `XY Trace`, `X vs Y Plot`, and `Plotter Config`.

SubRecord

An object that allows the user to cut out a number of fields from a record.

Use

Use `SubRecord` to output a subset of the original record, with specified fields "cut" from the record, on the `Record` output pin. The record to be edited is input on the `Record` input pin. The list of fields to be "included" or "excluded" in the output record is input on the `Fields` input pin.

- If `Include fields` is selected in the open view, the output record will consist *only* of those fields in the `Fields` input.
- If `Exclude fields` is selected in the open view, the output record will consist of all fields in the original record *excluding* those listed in the `Fields` input.

Location

`Data` ⟹ `Access Record` ⟹ `SubRecord`

Open View Parameters

- `Include fields`—(The default mode.) The output record is constructed from the input record with only those fields in the list of field names on the `Fields` data input pin. This selection toggles with `Exclude fields`.
- `Exclude fields`—The output record is constructed from the input record with only those fields that are *not* in the list of field names on the `Fields` data input pin.

Notes

The `Fields` input pin can be a text scalar consisting of the names of the record fields to include/exclude in the output record.

If you attempt to include/exclude a field that is not in the input record, an error will occur. If you exclude all the fields of the record, a nil output signal is sent.

SubRecord

Note the distinction between the `UnBuild Record`, `SubRecord`, and `Get Field` objects.

- The `UnBuild Record` object has outputs for the `Name List` and `Type List` of the input record fields. The other (optional) outputs `A`, `B`, etc. of the `UnBuild Record` object return the same results as would multiple `Get Field` objects.
- The `Get Field` object works like a `Formula` object, in that it uses dot syntax (`A.B`) to unbuild a record. Note that `Get Field` allows you to unbuild a record of records in one step by using an expression such as `A.B.C`. This process would require two `UnBuild Record` objects.
- The `SubRecord` object differs from `UnBuild Record` and `Get Field` in that its output is always a record. The `SubRecord` device allows you to either *include* or *exclude* a list of fields from a record to form a subrecord.

See Also

`Build Record`, `From DataSet`, `Get Field`, `Record Constant`, `Set Field`, `To DataSet`, and `UnBuild Record`.

Terminals

A menu item.

Use

Use `Terminals` to access the following features which show, add, and delete terminals:

- `Show Terminals` (only available from the open view, except for `Do`, `Start`, and `Toggle`)
- `Add Data Input`
- `Add Control Input`
- `Add XEQ Input`
- `Delete Input`
- `Add Data Output`
- `Add Error Output`
- `Delete Output`

Location

On each object menu ⟹ `Terminals` ⟹

Short Cuts

You can quickly add a data terminal by placing the cursor over the input or output terminal display area and then pressing CTRL A. Each press of CTRL A adds an additional data terminal.

You can quickly delete a data terminal by placing the cursor over the terminal view area and then pressing CTRL D.

See Also

`Show Terminals` and `Line Probe`.

Text

An object that outputs a constant string Scalar or Array 1D.

Use

Use `Text` to set a string constant or to request user input. To input an array, press tab to enter the next value.

Location

`Data` ⟹ `Constant` ⟹ `Text`

Example

To use `Text` as a prompt on a panel view, change the name of the `Text` object to a prompt such as `What is your name?` The user fills in the requested information in the entry field.

Object Menu

- `Auto Execute` - If set, the object operates whenever the values in the field are edited.
- `Config` - Sets the initial number of values to be output with this object.
- `Initialize` - Used to set this object to a particular value at PreRun and/or Activate time.
  - `Initial Value` - A dialog box that specifies the value to be set. Default value is the zero value of that container type.
  - `Initialize At PreRun` - Whether to set the `Initial Value` at PreRun time. Default is off.
  - `Initialize At Activate` - Whether to set the `Initial Value` at Activate time. Default is off.

Notes

`Initialize` is most often used for initializing values inside a `UserObject`.

There is no interpretation of escaped characters in this constant. Your entry is replicated exactly on the output pin. The other method for setting an initial value is the `Default Value` control pin. The `Default Value` pin allows you to programmatically change the current value.

Note that the `Initial Value` field is always a scalar, even if `Text` is configured to be an array. The `Default Value` input pin, however, requires its input container to exactly match the size and shape of `Text`.

See Also

`Complex`, `Constant`, `Coord`, `Date/Time`, `Enum`, `Integer`, `PComplex`, `Real`, and `Toggle`.

Timer

An object that outputs the difference in seconds between the activation times of the top and bottom data input pins.

Use

Use `Timer` to measure how much time passes between two events, such as objects operating. This happens when the first and second inputs are activated. If the second input is activated before the first, the container on the second input is ignored and the `Timer` object does not execute.

Location

`Device` ⟹ `Timer`

Notes

Time is displayed in seconds.

See Also

`Date/Time`, `Do`, and `Time Stamp`.

To

A menu item.

Use

Use `To` for accessing the following objects which are destinations for I/O operations:

- `To File`
- `To DataSet`
- `To Printer`
- `To String`
- `To StdOut`
- `To StdErr`

Location

`I/O` ⟹ `To` ⟹

See Also

`From`.

To DataSet

An object that allows the user to collect records into a file.

Use

Use `To DataSet` to collect records that have the same schema definition into a file for later retrieval. The following control pins can be added:

- `File Name`—This control pin allows the user to change the name of the file that the records will be written to.
- `Rewind`—This control pin "rewinds" the file and makes it available for rewriting with an entirely new record type. The effect is the same as sending the first record to the file with the `Clear File At PreRun` selection checked.

Location

`I/O` ⟹ `To` ⟹ `DataSet`

Open View Parameters

- `To DataSet:`—Click on the name field to display a dialog box, then select the name of the file that contains the DataSet.
- `Clear File At PreRun`—If this box is checked, the entire DataSet is cleared and rewritten when `To DataSet` executes (when it receives the first record). Because the DataSet is completely erased, records with a new schema definition can be sent—the first record establishes the new schema definition.

Notes

The records going into the `To DataSet` file must have the same schema (data shape and type). The first record written to the file will define the schema that all subsequent records must have.

Every time the `To DataSet` object executes with its one input (only one input is allowed), the new record is appended to the file. If the new record is an array, every record element of it is appended. The schema (number of fields, and each field name, type, and shape) must match for every record, but the shape of the record (scalar/array) does not matter. The `To DataSet` object will

continue appending to the file until either a `Rewind` control input is activated, or `Clear File At PreRun` is checked True and PreRun occurs. The next time the `To DataSet` object executes, it begins writing to the file at its beginning, again starting with the schema of the first record. This schema may be different than what was in the file.

See Also

`Build Record`, `From DataSet`, `Record Constant`, and `UnBuild Record`.

To File

An object that writes data to a file using transaction statements.

Use

Use `To File` to output a wide variety of encodings and formats to allow flexible output to files. A single pointer is maintained for each file; therefore, data output by different `To File` objects is written to the file in the order that the objects operated.

`To File` Actions:

- `WRITE` - Writes data to a file using the specified encoding and format.
- `EXECUTE` - Repositions the file's write pointer to the beginning of the file with (CLEAR) or without (REWIND) erasing the contents of the file. `EXECUTE REWIND` can only be used when `Clear File at PreRun & Open` is checked. Note that the file is automatically closed when the HP VEE model stops. However, you may use the `EXECUTE CLOSE` transaction to close the file during model execution.
- `WAIT` - Waits the specified number of seconds before executing the next transaction.

`To File` Encodings;

- `TEXT` - Writes all data types to human-readable files that are easily edited or ported to other software applications. HP VEE numeric data is automatically converted to text as it is written.
- `BYTE` - Converts numeric data to binary integer and writes the least significant byte.
- `CASE` - Maps an enumerated value or an integer to a string and writes the string. For example, you could use CASE to accept error numbers and write error messages.
- `BINARY` - Writes all data types in an machine-specific binary format.
- `BINBLOCK` - Writes all HP VEE data types in binary files with IEEE 488.2 definite length block headers.
- `CONTAINER` - Writes all data types in an HP VEE specific text format.

Location

I/O ⟹ To ⟹ File

Open View Parameters

- Clear at PreRun & Open - Erases file contents and sets file pointer to the beginning of the file. The REWIND transaction command can only be operated when Clear File At PreRun & Open is checked. When Clear File At PreRun & Open is not checked, data is appended to the end of the file.

The open view shows the list of transactions to be executed.

Object Menu

- Config - Allows you to view and edit the configuration that determines the end-of-line character and formatting for arrays.
- Add Trans - Adds a transaction to the end of the list.
- Insert Trans - Inserts a transaction before (above) the currently highlighted transaction.
- Cut Trans - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- Copy Trans - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- Paste Trans - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To File

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`From File` and `To/From Named Pipe`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

To/From HP BASIC/UX

An object that communicates with an HP BASIC/UX process using transactions by way of specified named pipes. This object is available in HP VEE-Test running on HP 9000 Series 300 or 400 only.

Use

Use `To/From HP BASIC/UX` to communicate with an HP BASIC/UX program. Be certain that you understand how to use named pipes in the HP-UX environment.

Type in the names of the pipes you wish to use in the `Read Pipe` and `Write Pipe` fields. Be certain that they match the names of the pipes used by your HP BASIC/UX program and that the read and write names are not inadvertently swapped. Use different pipes for the `To/From HP BASIC/UX` objects in different threads.

A `To/From HP BASIC/UX` object is generally preceded by an `Init HP BASIC/UX` object.

Location

`I/O` ⟹ `HP BASIC/UX` ⟹ `To/From HP BASIC/UX`

Open View Parameters

The open view shows the list of transactions to be executed.

The `Write Pipe` and `Read Pipe` fields can be added as control inputs.

Object Menu

- `Config` - Allows you to view and edit end-of-line sequences and formatting for arrays.
- `Add Trans` - Adds a transaction to the end (bottom) of the list.
- `Insert Trans` - Inserts a transaction before (above) the currently highlighted transaction.

- `Cut Trans` - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- `Copy Trans` - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- `Paste Trans` - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Notes

Because of the behavior of named pipes, it is easiest to structure your HP VEE transactions and corresponding HP BASIC/UX OUTPUTs and ENTERs to transmit known or easily parsed data blocks. For example, if you are transmitting strings, determine the maximum length block you wish to transmit and pad shorter strings with blanks. This avoids the problems of trying to read more data from a pipe than is available and of leaving unwanted data in a pipe.

To help prevent a `READ` transaction from hanging until data is available, use a `READ IOSTATUS DATA READY` transaction in separate `To/From HP BASIC/UX` object. This transaction returns a `1` if there is at least one byte to read, and a `0` if there are no bytes to read.

To read all the data available on the read pipe until the read pipe is closed, use a `READ ... ARRAY TO END` transaction.

If you are running discless, be certain you `Read` and `Write` to uniquely named pipes.

Example

Here are typical `To/From HP BASIC/UX` settings and the corresponding HP BASIC/UX program:

To/From HP BASIC/UX Object

```
Write Pipe : /tmp/to_rmb
Read Pipe  : /tmp/from_rmb
```

(These default pipes are created for you the first time the object operates.)

HP BASIC/UX Program

```
100  ASSIGN @From_vee TO "/tmp/to_rmb"
110  ASSIGN @To_vee TO "/tmp/from_rmb"
120  ! Your code here
130  ENTER  @From_vee;Vee_data
140  OUTPUT @To_vee;Rmb_data
150  END
```

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`Init HP BASIC/UX` and `To/From Named Pipe`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

To/From Named Pipe

An object that creates and uses named pipes for I/O. `To/From Named Pipe` is available only in HP VEE-Test.

Use

Named pipes are tools for programmers who wish to implement interprocess communication.

You must add one or more transactions to `To/From Named Pipe` to read or write data. To do this, click on `Add Trans` in the object menu.

Location

`I/O` ⟹ `To/From Named Pipe`

Open View Parameters

The open view shows the list of transactions to be executed.

The `Write Pipe` and `Read Pipe` fields can be added as control inputs.

Object Menu

- `Config` - Allows you to view and edit the configuration that determines the end-of-line character and formatting for arrays.
- `Add Trans` - Adds a transaction to the end of the list.
- `Insert Trans` - Inserts a transaction before the currently highlighted transaction.
- `Cut Trans` - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- `Copy Trans` - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- `Paste Trans` - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Notes

All To/From Named Pipe objects contain the same default names for read and write pipes. Be certain that you know which pipe you really want to read or write.

If the pipes do not exist before To/From Named Pipe operates, then they are created. However, there are some overhead costs; if the pipes exist beforehand, the model runs quicker. These pipes are created for you automatically if they do not already exist:

- /tmp/read_pipe
- /tmp/write_pipe

To create additional pipes, use the operating system command mknod.

Named pipes are opened when the first Read or Write transaction to that pipe operates after PreRun. All named pipes are closed at PostRun. The EXECUTE CLOSE READ PIPE and EXECUTE CLOSE WRITE PIPE transactions allow the named pipes to be closed at any time.

Because of the behavior of named pipes, it is easiest to structure your ToFrom Named Pipe transactions to transmit known or easily parsed data blocks. For example, if you are transmitting strings, determine the maximum length block you wish to transmit and pad shorter strings with blanks. This avoids the problems of trying to read more data from a pipe than is available and of leaving unwanted data in a pipe.

To help prevent a READ transaction from hanging until data is available, use a READ IOSTATUS DATA READY transaction in separate To/From HP BASIC/UX object. This transaction returns a 1 if there is at least one byte to read, and a 0 if there are no bytes to read.

To read all the data available on the read pipe until the read pipe is closed, use a READ ... ARRAY TO END transaction.

If you are running diskless, be certain you Read and Write to uniquely named pipes. Otherwise several workstations on the same diskless cluster may attempt to read/write to the same named pipe, which will cause contention problems.

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`From`, `From File`, `FromStdIn`, `To File`, `ToStdIn`, and `To`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

To Printer

An object that sends data to a printer.

Use

Use `To Printer` to output data to a printer using a wide variety of encodings and formats.

`To Printer` Actions:

- `WRITE` - Prints data using the specified encoding and format.
- `WAIT` - Waits the specified number of seconds before executing the next transaction.

`To Printer` Encodings:

In almost all cases, you should use `TEXT` or `CASE` encoding to print data. Other encodings should be used by advanced users only.

- `TEXT` - Writes all data types in human-readable form. HP VEE numeric data is automatically converted to text as it is written.
- `BYTE` - Converts numeric data to binary integer and writes the least significant byte.
- `CASE` - Maps an enumerated value or an integer to a string and writes the string. For example, you could use `CASE` to accept error numbers and write error messages.
- `BINARY` - Writes all data types in an machine-specific binary format.
- `BINBLOCK` - Writes all HP VEE data types in binary files with IEEE 488.2 definite length block headers.
- `CONTAINER` - Writes all data types in a machine-specific text format.

Location

I/O ⟹ To ⟹ Printer

Open View Parameters

The open view shows the list of transactions to be executed.

Object Menu

- Config - Allows you to view and edit, end-of-line sequences and array print formatting.
- Add Trans - Adds a transaction to the end of the list.
- Insert Trans - Inserts a transaction before the currently highlighted transaction.
- Cut Trans - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- Copy Trans - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- Paste Trans - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Notes

The printer which receives To Printer data is configured by way of File ⟹ Preferences ⟹ Printer Config using the Text Printer portion of the Printer Configuration dialog box.

When the first To Printer object in a model operates, it starts a print job. The print job is shared by all To Printer objects in the model, therefore their output appears in sequence on the same printout.

The print job remains open until PostRun, or until an EXECUTE CLOSE transaction is executed or the End Job control input on any To Printer is activated. PostRun occurs when all the threads in a model complete execution or when you press Stop twice. You access the End Job control input pin by way of the Add Control Input selection in the object menu.

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`Printer Config`, `To`, `To File`, `To/From Named Pipe`, and `To StdOut`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

To StdErr

An object that uses transactions to transmit data to the operating system standard error output.

Use

Use `To StdErr` to output a wide variety of file encodings and formats to the standard error output of HP VEE.

`To StdErr` Actions:

All `ToStdErr` objects use the same write pointer, even if they are in different threads or different contexts. Data written by different `ToStdErr` objects appears on stderr in the order the objects operated.

- `WRITE` - Writes data to standard error using the specified encoding and format.
- `WAIT` - Waits the specified number of seconds before executing the next transaction.

`To StdErr` Encodings:

- `TEXT` - Writes all data types in a human-readable form. HP VEE numeric data is automatically converted to text as it is written.
- `BYTE` - Converts numeric data to binary integer and writes the least significant byte.
- `CASE` - Maps an enumerated value or an integer to a string and writes the string. For example, you could use CASE to accept error numbers and write error messages.
- `BINARY` - Writes all data types in an machine-specific binary format.
- `BINBLOCK` - Writes all HP VEE data types in binary form with IEEE 488.2 definite length block headers.
- `CONTAINER` - Writes all data types in a machine-specific text font.

Location

I/O ⟹ To ⟹ StdErr

Open View Parameters

The open view shows the list of transactions to be executed.

Object Menu

- Config - Allows you to view and edit the configuration that determines the end-of-line character and formatting for arrays.
- Add Trans - Adds a transaction to the end of the list.
- Insert Trans - Inserts a transaction before the currently highlighted transaction.
- Cut Trans - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- Copy Trans - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- Paste Trans - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Notes

In general, standard error for HP VEE is the terminal window in which you ran veetest or veeengine.

You can redirect standard error by starting the HP VEE process using a command that redirects standard error, such as

```
veetest 2> someFileName     For sh and ksh
veetest >& someFileName     For csh
```

to send the stderr data to the file "someFileName".

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`From File`, `From StdIn`, and `To/From Named Pipe`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

To StdOut

An object that uses transactions to transmit data to the operating system standard output.

Use

Use `To StdOut` to output a wide variety of file encodings and formats to standard output of HP VEE.

`To StdOut` objects use the same write pointer, even if they are in different threads or different contexts. Data written by different `To StdOut` objects appears on stdout in the order the objects operated.

`To StdOut` Actions:

- `WRITE` - Writes data to standard output using the specified encoding and format.
- `WAIT` - Waits the specified number of seconds before executing the next transaction.

`To StdOut` Encodings:

- `TEXT` - Writes all data types in a human-readable form. HP VEE numeric data is automatically converted to text as it is written.
- `BYTE` - Converts numeric data to binary integer and writes the least significant byte.
- `CASE` - Maps an enumerated value or an integer to a string and writes the string. For example, you could use CASE to accept error numbers and write error messages.
- `BINARY` - Writes all data types in an machine-specific binary format.
- `BINBLOCK` - Writes all HP VEE data types in binary files with IEEE 488.2 definite length block headers.
- `CONTAINER` - Writes all data types in a machine-specific text format.

Location

`I/O ⟹ To ⟹ StdOut`

Open View Parameters

The open view shows the list of transactions to be executed.

Object Menu

- `Config` - Allows you to view and edit the configuration that determines the end-of-line character and formatting for arrays.
- `Add Trans` - Adds a transaction to the end of the list.
- `Insert Trans` - Inserts a transaction before the currently highlighted transaction.
- `Cut Trans` - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- `Copy Trans` - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- `Paste Trans` - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Notes

In general, standard output for HP VEE is the terminal window in which you ran `veetest` or `veeengine`.

You can redirect standard output by starting the HP VEE process using a command that redirects standard output, such as

```
veetest > someFileName
```

to send the stdout data to the file "someFileName".

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`From File`, `From StdIn`, `To/From Named Pipe`, and `To StdErr`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

To String

An object that uses transactions to write formatted text to a string.

Use

Use `To String` to created formatted text using transactions.

If only one string is generated by the `To String` transactions, the output container is a string scalar. When more than one string is generated, the output container is an Array 1D.

`WRITE` transactions ending with "EOL ON" will terminate the current output string, causing the next transaction in the `To String` list to write data to the next array element in the output container.

`Write` Transactions ending with "EOL OFF" will not terminate the output string, causing the characters generated by the next `Write` transaction in the `To String` list to be appended to the end of the current output string. The last transaction in the `To String` list will always terminate the output string, regardless of that transaction's "EOL" mode.

`To String` Actions:

- `WRITE` - Writes data using the specified encoding and format.
- `WAIT` - Waits the specified number of seconds before executing the next transaction.

`To String` Encodings:

In almost all cases, you should use `TEXT` encoding to write data.

- `TEXT` - Writes all data types in human-readable form. HP VEE numeric data is automatically converted to text as it is written.
- `BYTE` - Converts numeric data to binary form and writes the ASCII text equivalent that maps to the least significant byte. For example, the following transaction writes the character `A` (ASCII 65 decimal):

  ```
  WRITE BYTE 65
  ```

- `CASE` - Maps an enumerated value or an integer to a string and writes the string. For example, you could use CASE to accept error numbers and write error messages.

Location

`I/O` ⟹ `To` ⟹ `String`

Open View Parameters

The open view shows the list of transactions to be executed.

Object Menu

- `Config` - Allows you to view and edit formatting for arrays.
- `Add Trans` - Adds a transaction to the end of the list.
- `Insert Trans` - Inserts a transaction before the currently highlighted transaction.
- `Cut Trans` - Cuts (deletes) the currently highlighted transaction, but saves it in the transaction "cut-and-paste" buffer.
- `Copy Trans` - Copies the currently highlighted transaction to the transaction "cut-and-paste" buffer.
- `Paste Trans` - Pastes a transaction, previously "cut" or "copied" to the buffer, in the position before the currently highlighted transaction.

Example

`To String` is a useful debug tool to explore how `TEXT` transactions operate. Connect a `Logging AlphaNumeric` display to the `To String` output terminal to immediately view the results.

Short Cuts

To quickly insert a transaction, place the cursor on a transaction. Press CTRL O to insert a transaction over the transaction where you placed the cursor.

To quickly delete ("kill") a transaction, place the cursor on that transaction and press CTRL K.

To paste a transaction from the "kill" buffer press CTRL Y.

To quickly move to the next or previous transaction, press CTRL N or CTRL P respectively.

See Also

`To`, `To File`, `To/From Named Pipe`, `To StdOut`, and `To StdErr`.

"Using Transaction I/O" in *Using HP VEE*, chapter 12.

Toggle

An object that outputs a `1` if pressed (or checked) and a `0` if left up (or unchecked).

Use

Use `Toggle` to set a one or a zero output. `Toggle` is commonly used with an `If/Then` object that contains an `A==1` condition.

Location

`Data` ⟹ `Toggle`

Object Menu

- `Auto Execute` - If set, the object operates whenever the values in the field are edited.
- `Format` ⟹ - Specifies the appearance of the object and the method of selecting a choice.
  - `Check Box` - The choice is made by clicking on a recessed box. A check mark denotes the selection.
  - `Button` - The choice is made by pressing a button.
- `Initialize` - Used to set this object to a particular value at PreRun and/or Activate time.
  - `Initial Value` - A dialog box that specifies the value to be set. Default value is the first string value in the list.
  - `Initialize At PreRun` - Whether to set the `Initial Value` at PreRun time. Default is off.
  - `Initialize At Activate` - Whether to set the `Initial Value` at Activate time. Default is off.

Notes

`Initialize` is most often used for initializing values inside a `UserObject`.

The other method for setting initial values `Default Value` control pin available on most data constants. The `Default Value` pin allows you to programmatically change the current value.

Since the icon of this object represents its greatest utility, the object menu accessed from the icon is larger and more useful than the object menu accessed from its open view.

The name of the button or check box (default = `Toggle`) can be changed by changing the text in the open view.

See Also

`Complex`, `Constant`, `Coord`, `Date/Time`, `Enum`, `If/Then/Else`, `Integer`, `PComplex`, `Real`, and `Text`.

Trig Mode

Specifies the units used for trigonometric calculations.

Use

Use `Trig Mode` to specify the trigonometric units used in this context to `Degrees`, `Radians`, or `Gradians`.

The current value of `Trig Mode` is saved with each model. The default value is read from `.veerc` in your $HOME when HP VEE is started or `New` is selected.

Location

`File` ⟹ `Preferences` ⟹ `Trig Mode`
or
`UserObject` (Object Menu) ⟹ `Trig Mode`

Notes

`Trig Mode` is context sensitive.

Be careful when using `Merge` as the `Trig Mode` preferences for objects at the root context will not be read from the merge file. The merged objects will use the currently active `Preferences` setting for `Trig Mode`. This may cause unexpected results.

See Also

`Preferences` and `UserObject`.

UnBuild Complex

An object that outputs the real and imaginary components of a Complex number as Real numbers.

Use

Use `UnBuild Complex` to extract the real and imaginary components from a rectangular Complex number.

Location

`Data` ⟹ `UnBuild Data` ⟹ `Complex`

Notes

You can also get the components of a Complex number by using the `re(x)` and `imag(x)` functions or objects.

Even if inputs are mapped, the outputs are not.

See Also

`Build Complex`, `UnBuild Data`, and `UnBuild PComplex`.

"Formula Reference" chapter.

UnBuild Coord

An object that outputs the values of a Coord as Real numbers.

Use

Use `UnBuild Coord` to extract the `x, y, z ...` values of a Coord data type. Additional outputs can be added for the `z` and other coordinate dimensions. The number of outputs must match the number of coordinate dimensions of the input data.

Location

`Data` ⟹ `UnBuild Data` ⟹ `Coord`

See Also

`Build Coord` and `UnBuild Data`.

UnBuild Data

A menu item.

Use

Use `UnBuild` to access the following data type conversion objects:

- `Coord`
- `Complex`
- `PComplex`
- `Waveform`
- `Spectrum`
- `Record`

Location

`Data` ⟹ `UnBuild Data`

See Also

`Access Array`, `Build`, `UnBuild Complex`, `UnBuild Coord`, `UnBuild PComplex`, `UnBuild Record`, `UnBuild Spectrum`, and `UnBuild Waveform`.

UnBuild PComplex

An object that extracts the magnitude and phase components of a PComplex number and outputs them as Real numbers.

Use

Use `UnBuild PComplex` to extract the magnitude and phase components from a PComplex number.

Location

`Data` ⟹ `UnBuild Data` ⟹ `PComplex`

Notes

You can also get the components of a `PComplex` number by using the `mag(x)` and `phase(x)` functions or objects.

Even if inputs are mapped, the output are not.

See Also

`Build PComplex`, `Trig Mode`, `UnBuild Complex`, and `UnBuild Data`.

UnBuild Record

An object that allows the user to unbuild a record and recover its components.

Use

Use `UnBuild Record` to extract the data components from a record (`A`, `B`, and so forth), and to obtain a `Name List` and `Type List` for those components.

- The `Name List` output pin outputs a Text array of all the names of the input record field names.
- The `Type List` output pin outputs a Text array of all the data types of the input record.
- The `A`, `B`, etc. output pins output the individual data components unbuilt from the record.

Location

`Data` ⟹ `UnBuild Data` ⟹ `Record`

Notes

You may add an arbitrary number of output pins to this object. Each output pin name must correspond to a name of a record field. Field names are not case sensitive (lowercase and uppercase letters are equivalent).

Note the distinction between the `UnBuild Record`, `SubRecord`, and `Get Field` objects.

- The `UnBuild Record` object has outputs for the `Name List` and `Type List` of the input record fields. The other (optional) outputs `A`, `B`, etc. of the `UnBuild Record` object return the same results as would multiple `Get Field` objects.
- The `Get Field` object works like a `Formula` object, in that it uses dot syntax (`A.B`) to unbuild a record. Note that `Get Field` allows you to unbuild a record of records in one step by using an expression such as `A.B.C`. This process would require two `UnBuild Record` objects.

- The `SubRecord` object differs from `UnBuild Record` and `Get Field` in that its output is always a record. The `SubRecord` device allows you to either *include* or *exclude* a list of fields from a record to form a subrecord.

See Also

`Build Record`, `From DataSet`, `Get Field`, `Record Constant`, `Set Field`, `SubRecord`, and `To DataSet`.

UnBuild Spectrum

An object that outputs the PComplex values and frequency range of a Spectrum.

Use

Use `UnBuild Spectrum` to extract an Array 1D of PComplex values and interval information from a Spectrum.

Location

`Data` ⟹ `UnBuild Data` ⟹ `Spectrum`

Open View Parameters

`Start/Stop` | `Center/Span` - Changes the values in the outputs that describe the frequency range of the `Spectrum`.

See Also

`Build Spectrum`, `UnBuild Data`, and `UnBuild Waveform`.

UnBuild Waveform

An object that outputs the amplitude values and time span of a Waveform.

Use

Use `UnBuild Waveform` to extract an Array 1D of the amplitude values and a real value for the time span from a Waveform.

Location

`Data` ⟹ `UnBuild Data` ⟹ `Waveform`

Notes

The Real array output will be unmapped.

See Also

`Build Waveform`, `UnBuild Data`, and `UnBuild Spectrum`.

Until Break

An object that repeats execution of a subthread until a `Break` is encountered.

Use

Use `Until Break` to start a set of operations that is repeated until a `Break` is encountered.

Location

`Flow` ⟹ `Repeat` ⟹ `Until Break`

Notes

Execution of the subthread hosted by the `Until Break` output continues until one of the following occurs:

- All objects that can, have operated; the subthread is deactivated. The subthread is reactivated by refiring the `Until Break` output until a `Break` object is encountered.
- A `Break` object operates . The subthread is deactivated and the sequence output pin is activated.
- A `Next` object operates. The subthread is deactivated then reactivated by refiring the `Until Break` output until a `Break` object is encountered.

`Until Break` outputs an output execution signal that has no value (nil).

When the subthread hosted by the `Until Break` object finishes an iteration, all data containers sent during the previous iteration are invalidated before the next iteration. This prevents "old" data from a previous iteration from being reused in the current iteration. However, if flow branching is present within the iterative subthread, some objects may not execute on every iteration. Thus, data containers sent by those objects may be invalidated before other objects can execute on the data. To obtain the desired propagation in this case, use the `Sample & Hold` object. Refer to "Iteration with Flow Branching" in chapter 4 of *Using HP VEE* for more information.

See Also

Break, For Count, For Log Range, For Range, Next, On Cycle, and Sample & Hold.

User Function

A user-defined function created from a `UserObject`.

Use

You can create a User Function by creating a `UserObject` and then executing `Make UserFunction` (refer to `UserObject` for details). You can use `Edit User Function` to edit the User Function once it has been created. User Functions exist in the "background" and can be called with `Call Function` or from certain expressions.

An advantage of making a User Function out of a `UserObject` is that the same User Function can be called multiple times within the model, but it exists as source code only in one place.

The following choices are also available when editing a User Function:

- `Make UserObject` - The opposite operation from `Make UserFunction`. Turns the User Function back into a `UserObject`
- `Delete` - Deletes the User Function from the HP VEE model.

Notes

You can create a library of User Functions by creating several `UserObjects`, turning them into User Functions, and then saving them all to a file. This library then can be imported into a model using `Import Library`, and deleted with `Delete Library`.

Not only can you call a User Function with `Call Function`, but you can call it from any expression whose evaluation is delayed until run time. These include expressions in `Formula`, `If/Then/Else`, `Get Values`, `Get Field`, `Set Field`, or `From DataSet` devices, or expressions in `Sequencer` or `I/O` transactions. However, the syntax of function calls in an expression allows only a single return value. Thus, a call to a User Function from such an expression will return only the value on the first output. If there are additional outputs (excluding the sequence output) their values will be dropped. If there are no outputs, the returned value will be undefined.

See Also

Call Function, Delete Library, Edit UserFunction, Formula, If/Then/Else, Import Library, Sequencer, and UserObject.

UserObject

An object that may contain other objects.

Use

Use `UserObject` to logically and physically group objects together. A `UserObject` constitutes a separate "context" from the root context of a model. You can use multiple `UserObjects`, or nested `UserObjects` within a model.

To create the desired `UserObject`, you can move other objects into the work area of the open view of the `UserObject`, or you can select objects and use `Create UserObject` from the `Edit` menu.

The `Make UserFunction` item in the object menu allows you to make the `UserObject` into a User Function, which can then be called using the `Call Function` object, or from certain expressions.

Operation:

A `UserObject` operates just like any other object—no objects inside of a `UserObject` operate until all data inputs to the `UserObject` are activated. All operations inside of the `UserObject` must be completed before the data outputs are activated.

The right-most button on the title bar of the open view is a maximize button—it increases the size of the `UserObject` to the full size of the HP VEE work area.

When objects inside the `UserObject` are connected to objects outside the `UserObject`, input and output pins are automatically created.

Creating a User Function:

Once you have created a `UserObject`, you can use `Make UserFunction` (see "Object Menu," below) to create a User Function with the same functionality. The `UserObject` will disappear from the screen and will be replaced with a `Call Function` object containing a call to the new User Function. The User Function will be added to the list of available User Functions, which exists in the "background" within the model. You can call the User Function using the `Call Function` object, or from certain expressions. You can edit the User Function by using `Edit UserFunction` in the `Edit` menu.

Location

`Device` ⟹ `UserObject`

Object Menu

You can select the object menu of the `UserObject` from the object menu button, but you cannot select it from within the work area of the `UserObject` open view. From within the work area, the right mouse button provides a pop-up `Edit` menu, just like the one available in the main work area. When the pointer is over an object within the `UserObject`, the right mouse button gives the object menu for that object. You can also get `UserObject` object menu by placing the pointer over the borders of the `UserObject` and clicking the right mouse button.

- `Make UserFunction` - Converts the `UserObject` into a User Function. The `UserObject` will disappear from the work area and it is replaced with a `Call Function` object containing a call to the new User Function. Before you do this operation, enter a unique name into the title field. This name will become the User Function name, which will be the name by which `Call Function` or certain expressions can call the User Function. Should the name conflict with the name of an existing User Function, an error will be displayed. You will then need to enter a different name and repeat the operation.
- `Unpack` - Deletes the `UserObject`, but not the objects contained in it.
- `Secure` - Prevents the `UserObject` from being modified.
- `Show Panel on Exec` - When set, shows the panel view associated with the `UserObject` when the `UserObject` operates. This is only available after the `UserObject` panel view has been created (by way of `Add To Panel`).
- `Trig Mode` ⟹ - Specifies the trig mode used in the `UserObject` context (degrees, radians, or gradians).
- `Edit` ⟹ - A parent menu that leads to the `UserObject` context `Edit` menu. This menu contains the same choices as the main menu `Edit` menu. You can get the pop-up `Edit` menu by clicking the right mouse button on the `UserObject` work area.

  The following choices are context sensitive to the `UserObject`:

- `Clean Up Lines` - Routes the lines in the `UserObject` around objects.
- `Move Objects` - Moves several objects at once.
- `Create UserObject` - Creates a `UserObject` of the currently selected objects.
- `Add To Panel` - Creates a panel view (for the `UserObject`) containing the selected objects.

Notes

To create a `UserObject` of the currently selected objects, use `Create UserObject`.

It is convenient to save `UserObjects` of common functions (using `Save Objects`) to create a library of functions to be used again.

You can create a library of User Functions by creating several `UserObjects`, turning them into User Functions, and then saving them all to a file. This library then can be imported into a model using `Import Library`, and deleted with `Delete Library`.

`UserObject` allows the addition of `XEQ` inputs.

If the XEQ input is activated, `UserObject` operates even if all its data input elements are not satisfied.

When a breakpoint is set on an object in a `UserObject` that is displayed as an icon view and `Show Panel on Exec` is off, `Step` ignores the breakpoint.

See Also

`Add To Panel`, `Call Function`, `Create UserObject`, `Delete Library`, `Direct I/O`, `Edit UserFunction`, `Formula`, `If/Then/Else`, `Import Library`, `Secure`, `Sequencer`, and `User Function`.

View Globals

A menu item that allows you to view the list of global variables that are currently defined.

Use

Use `View Globals` to look at the type, shape, and data values of any currently defined global variables. (This menu item is "grayed" if no globals exist.)

Location

`Edit` ⟹ `View Globals`

Notes

Before you may view any global variables, one or more must have been created by running a model with one or more `Set Global` objects.

All global variables are deleted at the beginning of every `Run`, `Start`, or auto-execution. Global variables are always deleted by either `File` ⟹ `New` or `File` ⟹ `Open`. Global variable values are not saved with the model.

See Also

`Get Global`, and `Set Global`.

Virtual Source

A menu item.

Use

Use `Virtual Source` to access the following virtual signal generators.

- `Function Generator`
- `Pulse Generator`
- `Noise Generator`

Location

`Device` ⟹ `Virtual Source` ⟹

Notes

`Virtual Source` objects output Waveforms.

See Also

`Build Arb Waveform`, `Function Generator`, `Instruments`, `Noise Generator`, and `Pulse Generator`.

VU Meter

This object has been renamed to `Meter`.

Wait for SRQ

This menu item has been replaced with an SRQ option under `Device Event`.

Waveform (Time)

An object that displays time-domain information (waveforms) on a two-dimensional graphical display.

Use

Use `Waveform (Time)` to display Waveforms or Spectrums in the time domain. Spectrums are automatically converted to waveforms by way of an Inverse Fast Fourier Transform (ifft). The X axis is in the sampling units of the input waveform (typically, seconds).

Location

`Data` ⟹ `Build Data` ⟹ `Waveform`

Open View Parameters

- `Auto Scale` - Automatically scales the display to show the entire trace.
- `Mag` - The name of the Y axis.
- `Trace1` - The name of the first trace.
- `Time` - The name of the X axis.

Object Menu

- `Auto Scale` ⟹ - Automatically scales the display to show the entire trace.
  - `Auto Scale` - Automatically scales both axes.
  - `Auto Scale X` - Automatically scales the X axis.
  - `Auto Scale Y` - Automatically scales the Y axis.

  These parameters may be added as control inputs.
- `Clear Control` ⟹ - Parameters that specify when to clear the display.
  - `Clear` - Clears the displayed trace(s). This parameter may be added as a control input.

- `Clear At PreRun` - Clears the displayed trace(s) when the model or thread is PreRun.
- `Clear At Activate` - Clears the displayed trace(s) when the User Object is activated.
- `Next Curve` - Resets the pen to display the next curve in a family of curves (data from the next time the display operates) *without* clearing the previous curve. `Next Curve` must be selected (or activated) before each new curve in the family. This parameter may be added as a control input.

■ `Zoom` ⟹ - Scales the display.

- `In` - Magnifies the display to contain only the rectangular region that you selected with the pointer. You select the region after selecting this feature by dragging on the graph area. This action outlines the information with a "rubber band" box.
- `Out 2|5|10|20|50|100 x` - Expands the scales of the display by a factor in both the X and Y directions about the center.

■ `Markers` ⟹ - Allows you to find the exact value of a data point on the displayed curve. If the data is plotted on a log scale, the values shown on the marker are the linear data points.

To move markers to a different trace, click on the button to the left of the marker values near the bottom of the display. The button cycles through the different line types and colors of the different traces.

- `Off` - No markers are shown.
- `One On` - One marker is available.
- `Two On` - Two markers are available.
- `Delta On` - Two markers are available and the x and y differences between them are displayed.
- `Interpolate` - When checked, you can place markers in between the displayed data points. The marker values displayed are calculated by linear interpolation. Default is off.
- `Center` - If markers are available, brings them to the center of the visible part of the trace. This is useful if you've scrolled the display and markers are not visible.

- **Grid Type ⟹** - Sets the type of grid marks. The value of the major x division is shown below the maximum x value (to the right of the x scale name). The value of the major y division is shown below the y scale name.
  - **No Grid** - No grid lines are shown.
  - **Tic Marks** - Shows tic marks at the major and minor divisions on all four sides of the graph.
  - **Axis** - Shows tic marks at the major and minor divisions on the X and Y axes of the graph. If the actual axes are scrolled off the graph area, axis lines are drawn on the edge closest to the axes.
  - **Lines** - Shows lines at the major divisions and tic marks at the minor divisions. The X and Y axes are shown as thick lines.
- **Panel Layout ⟹** - Sets the appearance of the open view.
  - **Graph Only** - The open view shows only the graph area and the marker information (if it exists). No buttons, scales, scale names, or traces names are shown. This layout redraws quickest and provides the largest display area.
  - **Scales** - The open view shows the graph area, the scales, the scale names, and trace names. These fields are not recessed and may not be edited. This layout is useful when recessed fields might be distracting, such as printing graphs, or when the fields should be protected from editing on a User Panel.
  - **Scales & Sliders** - The open view shows all information about the graph. It includes the most information and allows you to modify the most elements interactively. It is the default.
- **Traces & Scales** - A control panel that allows you to specify values such as the names, colors, line and point characteristics, minimum values, and maximum values for the traces and/or scales.

  Traces:
  - **Name:** - The name of the trace that is displayed to the left of the graph area and the name of the corresponding input terminal.
  - **Scale:** - If you have multiple Y scales, selects which Y scale is to be used for this trace.

- `Color:` - The color of the trace. Each added trace has a different color than the existing traces. Default is `Pen 4` (yellow).
- `Lines:` - The format of the line connecting data points. Default is a continuous line.
- `Points:` - The symbol that marks each data point. To show unconnected data points, select the single dot `Lines` format and the desired `Points` symbol. Default is a dot.

You can add a `Traces` control input pin. However, the control input data must be a record with the following fields: 1) A `TraceNum` field with an Integer value (1 is the top trace), *and* 2) one or more of the following fields: `Name`, `Pen`, `LineType`, `PointType`. (The `Pen`, `LineType`, and `PointType` values are integers from 0 to n, where 0 draws nothing.) Refer to "Records and DataSets" in *Using HP VEE* for further information.

Scales:

- `Show Scale:` - If you have multiple Y scales, a selection (using a check box) to specify if the end points and an axis of each additional right scale is displayed to the right of the graph area.
- `Scale Name:` - The names of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Maximum:` - The maximum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Minimum:` - The minimum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Mapping:` - The way the x and y data is mapped to the display. The mapping may be linear or log. To make a log-log plot, change both `X` and `Y` axes to `Log`. Default is `Linear`.
- `Log Cycles:` - The maximum number of decades shown (counting down from the maximum x and y values) when `AutoScale` is activated. `Log Cycles` is only used when the `Mapping` is `Log`. `Log Cycles` is useful when a trace contains a large dynamic range. Default is 4.
- `Scale Colors` - The color of any background grid or tic marks. Default is `Gray`.

You can add a Scales control input pin. However, the control input data must be a record with the following fields: 1) A Text field Scale with a value X, Y (or Y1), Y2, or Y3, *and* 2) one or more of the following fields: Name, Min, Max, and Mapping. (The Mapping text value may be Linear or Log). Refer to "Records and DataSets" in *Using HP VEE* for further information.

- Add Right Scale - Adds up to two additional scales to permit traces to have different scale ranges. After adding a right scale, use Traces & Scales to assign a trace to the scale.
- Plot - Presents the Plotter Configuration control panel. When OK is pressed, a copy of the device's entire display is plotted on the selected plotter. This parameter may be added as a control input. If the current Plotter Configuration is in Plot to File mode, you may specify the destination file or directory name as string data on the Plot control input. If no control input value is given, the file or directory name specified in Plotter Configuration will be used. See Plotter Config for more information.

Notes

Inputs must be Scalar or Array 1D.

You can add traces as data inputs. Up to twelve traces are allowed.

Input data of type Coord is plotted by simply using its x and y values without first being converted to type Waveform.

A Title control input may be added, which sets the title bar name to the specified text value. This allows programmatic control over the title shown when the display is printed or plotted.

See Also

Complex Plane, Magnitude Spectrum, Polar Plot, Strip Chart, X vs Y Plot, XY Trace, and Plotter Config.

Waveform Defaults

Changes the default Waveform sampling parameters.

Use

Use `Waveform Defaults` to change the default sampling parameters in newly created `Function Generator`, `Pulse Generator`, `Noise Generator`, `Build Waveform`, and `Build Arb Waveform` objects.

The current values for `Waveform Defaults` are saved with each model. The defaults are read in from the `.veerc` file when HP VEE is started or `New` is selected.

Location

`File` ⟹ `Preferences` ⟹ `Waveform Defaults`

Dialog Information

- `Time Span` - Allows the specifying of the time interval covered by the entire Waveform in time units.
- `Num Points` - Allows the specifying of the number of data points in the Waveform.

See Also

`Build Arb Waveform`, `Build Waveform`, `Function Generator`, `Noise Generator`, `Preferences`, and `Pulse Generator`.

X vs Y Plot

An object that displays a Cartesian plot.

Use

Use `X vs Y` plot to graphically display values when separate data information is available for `X` and `Y` data. When you plot more than one trace, each execution of the `X vs Y Plot` object uses the single `X` input data with each trace's `Y` input data, therefore all traces share the same `X` input data. If you want different `X` data values, you must build coordinates or mapped arrays for each trace.

All inputs must be the same size and shape. Mapping information on the input's data is ignored.

Location

`Display` ⟹ `X vs Y Plot`

Open View Parameters

- `Auto Scale` - Automatically scales the display to show the entire trace.
- `Y name` - The name of the Y axis.
- `YData1` - The name of the first trace.
- `X name` - The name of the X axis.

Object Menu

- `Auto Scale` ⟹ - Automatically scales the display to show the entire trace.
  - `Auto Scale` - Automatically scales both axes.
  - `Auto Scale X` - Automatically scales the X axis.
  - `Auto Scale Y` - Automatically scales the Y axis.

  These parameters may be added as control inputs.
- `Clear Control` ⟹ - Parameters that specify when to clear the display.

- `Clear` - Clears the displayed trace(s). This parameter may be added as a control input.
- `Clear At PreRun` - Clears the displayed trace(s) when the model or thread is PreRun.
- `Clear At Activate` - Clears the displayed trace(s) when the User Object is activated.
- `Next Curve` - Resets the pen to display the next curve in a family of curves (data from the next time the display operates) *without* clearing the previous curve. `Next Curve` must be selected (or activated) before each new curve in the family. This parameter may be added as a control input.

■ `Zoom` ⟹ - Scales the display.

- `In` - Magnifies the display to contain only the rectangular region that you selected with the pointer. You select the region after selecting this feature by dragging on the graph area. This action outlines the information with a "rubber band" box.
- `Out 2|5|10|20|50|100 x` - Expands the scales of the display by a factor in both the X and Y directions about the center.

■ `Markers` ⟹ - Allows you to find the exact value of a data point on the displayed curve. If the data is plotted on a log scale, the values shown on the marker are the linear data points.

To move markers to a different trace, click on the button to the left of the marker values near the bottom of the display. The button cycles through the different line types and colors of the different traces.

- `Off` - No markers are shown.
- `One On` - One marker is available.
- `Two On` - Two markers are available.
- `Delta On` - Two markers are available and the x and y differences between them are displayed.
- `Interpolate` - When checked, you can place markers in between the displayed data points. The marker values displayed are calculated by linear interpolation. Default is off.

- Center - If markers are available, brings them to the center of the visible part of the trace. This is useful if you've scrolled the display and markers are not visible.

- **Grid Type ⟹** - Sets the type of grid marks. The value of the major x division is shown below the maximum x value (to the right of the x scale name). The value of the major y division is shown below the y scale name.
  - No Grid - No grid lines are shown.
  - Tic Marks - Shows tic marks at the major and minor divisions on all four sides of the graph.
  - Axis - Shows tic marks at the major and minor divisions on the X and Y axes of the graph. If the actual axes are scrolled off the graph area, axis lines are drawn on the edge closest to the axes.
  - Lines - Shows lines at the major divisions and tic marks at the minor divisions. The X and Y axes are shown as thick lines.
- **Panel Layout ⟹** - Sets the appearance of the open view.
  - Graph Only - The open view shows only the graph area and the marker information (if it exists). No buttons, scales, scale names, or traces names are shown. This layout redraws quickest and provides the largest display area.
  - Scales - The open view shows the graph area, the scales, the scale names, and trace names. These fields are not recessed and may not be edited. This layout is useful when recessed fields might be distracting, such as printing graphs, or when the fields should be protected from editing on a User Panel.
  - Scales & Sliders - The open view shows all information about the graph. It includes the most information and allows you to modify the most elements interactively. It is the default.
- **Traces & Scales** - A control panel that allows you to specify values such as the names, colors, line and point characteristics, minimum values, and maximum values for the traces and/or scales.

  Traces:

- `Name:` - The name of the trace that is displayed to the left of the graph area and the name of the corresponding input terminal.
- `Scale:` - If you have multiple Y scales, selects which Y scale is to be used for this trace.
- `Color:` - The color of the trace. Each added trace has a different color than the existing traces. Default is `Pen 4` (yellow).
- `Lines:` - The format of the line connecting data points. Default is a continuous line.
- `Points:` - The symbol that marks each data point. To show unconnected data points, select the single dot `Lines` format and the desired `Points` symbol. Default is a dot.

You can add a `Traces` control input pin. However, the control input data must be a record with the following fields: 1) A `TraceNum` field with an Integer value (1 is the top trace), *and* 2) one or more of the following fields: `Name`, `Pen`, `LineType`, `PointType`. (The `Pen`, `LineType`, and `PointType` values are integers from 0 to n, where 0 draws nothing.) Refer to "Records and DataSets" in *Using HP VEE* for further information.

Scales:

- `Show Scale:` - If you have multiple Y scales, a selection (using a check box) to specify if the end points and an axis of each additional right scale is displayed to the right of the graph area.
- `Scale Name:` - The names of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Maximum:` - The maximum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Minimum:` - The minimum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Mapping:` - The way the x and y data is mapped to the display. The mapping may be linear or log. To make a log-log plot, change both `X` and `Y` axes to `Log`. Default is `Linear`.
- `Log Cycles:` - The maximum number of decades shown (counting down from the maximum x and y values) when `AutoScale` is activated. `Log`

Cycles is only used when the Mapping is Log. Log Cycles is useful when a trace contains a large dynamic range in the data. Default is 4.

- Scale Colors - The color of any background grid or tic marks. Default is Gray.

You can add a Scales control input pin. However, the control input data must be a record with the following fields: 1) A Text field Scale with a value X, Y (or Y1), Y2, or Y3, *and* 2) one or more of the following fields: Name, Min, Max, and Mapping. (The Mapping text value may be Linear or Log). Refer to "Records and DataSets" in *Using HP VEE* for further information.

- Add Right Scale - Adds up to two additional scales to permit traces to have different scale ranges. After adding a right scale, use Traces & Scales to assign a trace to the scale.
- Plot - Presents the Plotter Configuration control panel. When OK is pressed, a copy of the device's entire display is plotted on the selected plotter. This parameter may be added as a control input. If the current Plotter Configuration is in Plot to File mode, you may specify the destination file or directory name as string data on the Plot control input. If no control input value is given, the file or directory name specified in Plotter Configuration will be used. See Plotter Config for more information.

Notes

Inputs must be Scalar or Array 1D.

Add Y data inputs with the Terminals ⟹ Add Data Input object menu selection. Up to twelve traces are allowed.

A Title control input may be added, which sets the title bar name to the specified text value. This allows programmatic control over the title shown when the display is printed or plotted.

See Also

Complex Plane, Magnitude Spectrum, Polar Plot, Strip Chart, Waveform (Time), XY Trace, and Plotter Config.

XY Trace

An object that displays a two-dimensional Cartesian plot.

Use

Use `XY Trace` to display mapped arrays or a set of values when `y` data is generated with evenly-spaced `x` values. An `XY Trace` is useful for a quick look at data when you don't need (or have) scaled `x` data values.

The `x` value that is automatically generated depends on the data type of the trace data. For example, if trace is a Real value, the `x` values is 0, 1, 2, and so forth. If the trace is a Waveform, the `x` values are time values.

Location

`Display` ⟹ `XY Trace`

Open View Parameters

- `Auto Scale` - Automatically scales the display to show the entire trace.
- `Y name` - The name of the Y axis.
- `Trace1` - The name of the first trace.
- `X name` - The name of the X axis.

Object Menu

- `Auto Scale` ⟹ - Automatically scales the display to show the entire trace.
  - `Auto Scale` - Automatically scales both axes.
  - `Auto Scale X` - Automatically scales the X axis.
  - `Auto Scale Y` - Automatically scales the Y axis.

  These parameters may be added as control inputs.
- `Clear Control` ⟹ - Parameters that specify when to clear the display.
  - `Clear` - Clears the displayed trace(s). This parameter may be added as a control input.

- Clear At PreRun - Clears the displayed trace(s) when the model or thread is PreRun.
- Clear At Activate - Clears the displayed trace(s) when the User Object is activated.
- Next Curve - Resets the pen to display the next curve in a family of curves (data from the next time the display operates) *without* clearing the previous curve. Next Curve must be selected (or activated) before each new curve in the family. This parameter may be added as a control input.

■ Zoom ⟹ - Scales the display.

- In - Magnifies the display to contain only the rectangular region that you selected with the pointer. You select the region after selecting this feature by dragging on the graph area. This action outlines the information with a "rubber band" box.
- Out 2|5|10|20|50|100 x - Expands the scales of the display by a factor in both the X and Y directions about the center.

■ Markers ⟹ - Allows you to find the exact value of a data point on the displayed curve. If the data is plotted on a log scale, the values shown on the marker are the linear data points.

To move markers to a different trace, click on the button to the left of the marker values near the bottom of the display. The button cycles through the different line types and colors of the different traces.

- Off - No markers are shown.
- One On - One marker is available.
- Two On - Two markers are available.
- Delta On - Two markers are available and the x and y differences between them are displayed.
- Interpolate - When checked, you can place markers in between the displayed data points. The marker values displayed are calculated by linear interpolation. Default is off.
- Center - If markers are available, brings them to the center of the visible part of the trace. This is useful if you've scrolled the display and markers are not visible.

- `Grid Type` ⟹ - Sets the type of grid marks. The value of the major x division is shown below the maximum x value (to the right of the x scale name). The value of the major y division is shown below the y scale name.
  - `No Grid` - No grid lines are shown.
  - `Tic Marks` - Shows tic marks at the major and minor divisions on all four sides of the graph.
  - `Axis` - Shows tic marks at the major and minor divisions on the X and Y axes of the graph. If the actual axes are scrolled off the graph area, axis lines are drawn on the edge closest to the axes.
  - `Lines` - Shows lines at the major divisions and tic marks at the minor divisions. The X and Y axes are shown as thick lines.
- `Panel Layout` ⟹ - Sets the appearance of the open view.
  - `Graph Only` - The open view shows only the graph area and the marker information (if it exists). No buttons, scales, scale names, or traces names are shown. This layout redraws quickest and provides the largest display area.
  - `Scales` - The open view shows the graph area, the scales, the scale names, and trace names. These fields are not recessed and may not be edited. This layout is useful when recessed fields might be distracting, such as printing graphs, or when the fields should be protected from editing on a User Panel.
  - `Scales & Sliders` - The open view shows all information about the graph. It includes the most information and allows you to modify the most elements interactively. It is the default.
- `Traces & Scales` - A control panel that allows you to specify values such as the names, colors, line and point characteristics, minimum values, and maximum values for the traces and/or scales.

  Traces:
  - `Name:` - The name of the trace that is displayed to the left of the graph area and the name of the corresponding input terminal.
  - `Scale:` - If you have multiple Y scales, selects which Y scale is to be used for this trace.

- `Color:` - The color of the trace. Each added trace has a different color than the existing traces. Default is `Pen 4` (yellow).
- `Lines:` - The format of the line connecting data points. Default is a continuous line.
- `Points:` - The symbol that marks each data point. To show unconnected data points, select the single dot `Lines` format and the desired `Points` symbol. Default is a dot.

You can add a `Traces` control input pin. However, the control input data must be a record with the following fields: 1) A `TraceNum` field with an Integer value (1 is the top trace), *and* 2) one or more of the following fields: `Name`, `Pen`, `LineType`, `PointType`. (The `Pen`, `LineType`, and `PointType` values are integers from 0 to n, where 0 draws nothing.) Refer to "Records and DataSets" in *Using HP VEE* for further information.

Scales:

- `Show Scale:` - If you have multiple Y scales, a selection (using a check box) to specify if the end points and an axis of each additional right scale is displayed to the right of the graph area.
- `Scale Name:` - The names of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Maximum:` - The maximum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Minimum:` - The minimum values of the scales. They may be set here or on the `Scales & Sliders` layout.
- `Mapping:` - The way the x and y data is mapped to the display. The mapping may be linear or log. To make a log-log plot, change both `X` and `Y` axes to `Log`. Default is `Linear`.
- `Log Cycles:` - The maximum number of decades shown (counting down from the maximum x and y values) when `AutoScale` is activated. `Log Cycles` is only used when the `Mapping` is `Log`. `Log Cycles` is useful when a trace contains a large dynamic range in the data. Default is 4.
- `Scale Colors` - The color of any background grid or tic marks. Default is `Gray`.

You can add a Scales control input pin. However, the control input data must be a record with the following fields: 1) A Text field Scale with a value X, Y (or Y1), Y2, or Y3, *and* 2) one or more of the following fields: Name, Min, Max, and Mapping. (The Mapping text value may be Linear or Log). Refer to "Records and DataSets" in *Using HP VEE* for further information.

- Add Right Scale - Adds up to two additional scales to permit traces to have different scale ranges. After adding a right scale, use Traces & Scales to assign a trace to the scale.
- Plot - Presents the Plotter Configuration control panel. When OK is pressed, a copy of the device's entire display is plotted on the selected plotter. This parameter may be added as a control input. If the current Plotter Configuration is in Plot to File mode, you may specify the destination file or directory name as string data on the Plot control input. If no control input value is given, the file or directory name specified in Plotter Configuration will be used. See Plotter Config for more information.

Notes

Inputs must be Scalar or Array 1D that can be converted to type Real. You must "unbuild" Complex, PComplex, and Spectrum data before inputting to an XY Trace.

You can add traces as data inputs. Up to twelve traces are allowed.

Input data of type Coord is plotted by using its x and y values without first being convert to type Real.

A Title control input may be added, which sets the title bar name to the specified text value. This allows programmatic control over the title shown when the display is printed or plotted.

See Also

Complex Plane, Magnitude Spectrum, Polar Plot, Strip Chart, Waveform (Time), X vs Y Plot, and Plotter Config.

3

Formula (Math and AdvMath) Reference

This chapter contains detailed reference information about all formula-based functions in HP VEE, which includes all the features in the `Math` and `AdvMath` menus.

The first section of this chapter covers all the general concepts underlying the `Math` and `AdvMath` features. Then the rest of the chapter contains the detailed reference information about each feature. The detailed reference information is ordered alphabetically by feature name to help you find information about each feature quickly and easily.

Mathematically Processing Data

To process data, you operate on it with functions from the `Math` and `AdvMath` menus or combine the functions to create mathematical expressions.

Note

You can also process data before running a model by using numeric entry fields such as those in `Constant` objects. Numeric entry fields on some objects support the use of arbitrary formulas. The formula is immediately evaluated; the resulting Scalar is used as the value for the field. You cannot use input variable names in the formula. You also cannot use global variables in `Constant`s. Also, the typed-in formula must evaluate to a Scalar value of the proper type or of a type that can be converted to that which the object expects. In general, you can use any of the dyadic operators, parentheses for nesting, function calls, and the predefined numeric constant `PI` (`3.1416 ...`) in numeric entry fields.

The `Math` and `AdvMath` menus contain a set of mathematical functions to process your data in numerous ways. All the features that are listed under the `Math` and `AdvMath` menus (except `Regression`) can be used in any object that allows expressions. The objects that allow expressions are:

- `Math` ⟹ `Formula`
- `Data` ⟹ `Access Array` ⟹ `Get Values`
- `Data` ⟹ `Access Array` ⟹ `Get Field`
- `Data` ⟹ `Access Array` ⟹ `Set Field`
- `Device` ⟹ `Sequencer`
- `Flow` ⟹ `If/Then/Else`
- `I/O` objects that use transactions

Expressions may contain the names of data input terminals, data output terminals (`I/O` transactions only), and any mathematical expression from the `Math` menu and `AdvMath` menu. Data input terminal names are used as variables. HP VEE is not case sensitive about names of input variables within expressions for USASCII keyboards. For non-USASCII keyboards, HP VEE is

case insensitive for 7-bit ASCII characters only. Expressions are evaluated at run-time.

General Concepts

Functions that are input an array operand perform the function on each element of the array, unless stated otherwise. For example, `sqrt` of a scalar returns a scalar; `sqrt(4)` returns `2`. But `sqrt` of an array returns an array of the same size; `sqrt( [1 4 9 64])` returns the array `[1 2 3 8]`.

All numbers in an expression field are considered Real values, unless you use parentheses to specify Complex or PComplex values. Therefore, `2` is considered to be a Real number, not an Int32. `(1, @2)` is a PComplex number, while `(1, 2)` is a rectangular Complex number.

Note

HP VEE interprets any value contained within parentheses as a Complex or PComplex value. If you need to use a Coord value in an expression, use the `coord(x, y)` function. The `coord` function takes 2 or more parameters. `coord(1, 2)` returns a Scalar Coord container with two fields.

All functions that operate on Coord data operate only on the dependent (last) field of each Coord. For example, `abs(coord(-1, -2, -3))` returns the Coord `(-1, -2, 3)`.

An Enum container is always converted to Text before every math operation except the function `ordinal(x)`. Enum arrays are not supported. If you try to create an Enum array, a Text array is created instead.

For information on specific data type definitions, please refer to the section titled "Understanding Containers" in the "Understanding Models" chapter of *Using HP VEE*.

Using Strings in Expressions

Strings within expressions must be surrounded by double quotes.

You may use the following escape sequences within strings:

Escape Character	Meaning
\n	Newline
\t	Horizontal Tab
\v	Vertical Tab
\b	Backspace
\r	Carriage Return
\f	Form Feed
\"	Double Quote
\'	Single Quote
\\	Backslash
ddd	Character Value. *d* is an octal digit.

Using Arrays in Expressions

Arrays in expressions can be used just like scalars, just refer to them by the terminal name. Array constants can be entered directly into an expression (such as `[1 2 3]`). Arrays used in functions, like `sin(x)`, have the `sin` function applied on every element of the array.

Please note, however, that negative constants in array constants are evaluated as expressions. For example, `[5 4 -3 2]` is evaluated as `[5 1 2]`. Therefore, you must specify `[5 4 (-3) 2]` instead.

Note

Array indices are 0-based. The indices start with zero and continue to n-1, where n is the number of elements in that particular dimension.

You can use expressions to access portions of an array. Once you've specified the sub-array in the expression, you can output the sub-array, or use it in further expression calculations.

You can access only contiguous sub-arrays from each array. To access sub-arrays, you *must* specify a parameter for each dimension in the array.

Use the following characters to specify array parameters:

- The comma, `","`, separates array dimensions. Each sub-array operation *must* have exactly one specification for each array dimension.
- The colon, `":"`, specifies a range of elements from one of the array dimensions.
- The asterisk, '*', is a wildcard to specify all elements from that particular array dimension.

Note

Waveform time spans, Spectrum frequency spans, and array mappings are adjusted according to the number of points in the sub-array. For example, if you have a 256 point Waveform (`WF`) and you ask for `WF[0:127]`, you'll get the first half of the Waveform and a time span that is half of the old one.

Examples. `A` is an Array 1D, 10 elements long.

- `A[1]` accesses the second element in `A` and outputs a Scalar.
- `A[0:5]` returns a one-dimensional sub-array that contains the first 6 elements of `A`.
- `A[1:1]` returns a one-dimensional sub-array that contains one element, which is the second element of `A`. Note the difference between this and the first example, `A[1]`.
- `A[2:*]` returns a one-dimensional sub-array that contains the third through the tenth elements of `A`.
- `A` or `A[*]` returns the entire array `A`.
- `A[1,2]` returns an error because it specifies parameters for a two-dimensional array.

`B` is a 5x5 matrix (an Array 2D).

- `B[*]` returns an error because it specifies only one parameter, and `B` is a two-dimensional array.
- `B[1,2]` returns a Scalar value from the second row, third element.
- `B[1,*]` returns all of row one as an Array 1D.
- `B[1,1:*]` returns all of row one, except for the first element, as an Array 1D.
- `B[4,1:4]` returns an Array 1D that contains four elements: the second through fifth values from row 4.
- `B[5,5]` returns an error because arrays are zero-based. The array can only be accessed through `B[4,4]`.
- `B[1 1]` returns an error because a comma must separate the dimension specifiers.

Building Arrays in Expressions

You can build an array from elements of other arrays or sub-arrays. Each element in the expression must specify the same number of dimensions and contain the same number of values in each dimension.

Examples. `A` is an Array 1D with ten elements. `B` is a 5x5 matrix.

- `[1 2 3]` returns a three element Real Array 1D that contains the values `1`, `2`, and `3`.
- `[A[0] A[5:7] A[9]]` causes an error because both Scalar and Array 1D elements are specified.
- `[A[0:4] B[0,*]]` returns a ten element Array 2D that contains the first five elements from `A` as the first row and the first row from `B` as the second row.
- `[A[0] A[1] B[2,3] A[5]]` returns a four element Array 1D that contains the first and second element of `A`, the element from the third row and fourth column of `B`, and the sixth element of `A`.

Using Global Variables in Expressions

You can create and set global variables by using the `Set Global` object, and you can access global variables by using the `Get Global` object. Refer to "Set Global" and "Get Global" in chapter 2 for further information.

In addition, you can access a global variable by including its name in a mathematical expression. You can include a global variable in a mathematical expression in a `Formula` object, or in any object with a delayed-evaluation expression field. These objects include `If/Then/Else`, `Get Values`, `Get Field`, `Set Field`, and all devices using expressions in transactions, including `To File`, `From File`, `From DataSet`, `Direct I/O`, `From Stdin`, `To/From Named Pipes`, and `Sequencer`.

To include a global variable in an expression, just use the global variable name as if it were an input variable. For example, suppose a model uses a `Set Global` device to define the global variable `numFiles`. Elsewhere in the model, a `Formula` object with input `A` may use the expression `numFiles+3*A`.

Note

Global variable names are case-insensitive. Either upper-case or lower-case letters may be used. Thus, `GLOBALA` is equivalent to `globalA`.)

To avoid errors or unexpected results, you should be aware of two limitations when you include global variables in an expression:

1. *Local input variables have higher precedence than global variables.* This means that, in case of duplicate variable names, the local variable is chosen over the global variable. For example, if the expression `Freq*10` is included in a `Formula` object that has a `Freq` input (a local variable), and there is also a global variable named `Freq`, the expression will be evaluated with the local variable `Freq`, not the global one. No error will be reported regarding this duplication.
2. *Depending on the flow of your model, an object that evaluates an expression containing a global variable may execute before the global variable is defined.* For example, suppose the global variable `globalA` is set with a `Set Global` object, and the expression `globalA*X^2` is included in a `Formula` object. Depending on the flow of your model, the `Formula` object may execute before the `Set Global` object executes. In this case, the `Formula` object

won't be able to evaluate the expression because `globalA` is undefined. An error message will appear.

It is important that you take steps to ensure correct propagation—that `Set Global` executes first. You can do this by connecting the sequence output pin of the `Set Global` object to the sequence input pin of the `Formula` object in this case, or of any other object that includes the global variable in an expression to be evaluated. If a `Get Global` object is used, its sequence input pin should also be connected to the sequence output pin of `Set Global`. For further information, refer to "Using Global Variables" in chapter 3 of *Using HP VEE*.

Global variables can be arrays. Just access a global variable array as if it were an input variable using array syntax, for example: `GlobAry[2]`. If a global variable is a `Record`, use the record access syntax, such as `globRecord.numFiles`.

Using Records in Expressions

You can use expressions to access a field or sub-field of a record. Use the `A.B` sub-field syntax to access the `B` field of a record `A`. If `A` is a record with a field `B`, which itself is a record which has a field `C`, you may use the `A.B` syntax recursively to access the `C` field. That is, use the expression `A.B.C`. If `A` does not have a `B` field, or `B` does not have a `C` field, an error will result.

There is no limit on the number of recursions of `A.b.c.d.e.f` that may be used in expressions. Note that field names are not case sensitive (lowercase and uppercase letters are equivalent). Field names may be duplicated in sub-Records, so you may use the expression `A.a.A`.

Records are very useful as global variables, so that one global variable may hold several different values. Note that a `Formula` object can be used in place of a `Get Global`. Thus, you can accomplish the `GlobRec.numFiles` access in one object, instead of using both a `Get Global` and a `Formula` object to unbuild the record.

The record and array syntax may be combined in expressions to access a field of a record array (for example `A[1].B`), or to access a portion of an array that is a field of a record (for example, `A.B[1]`). Note the difference between `A[1].b` and `A.b[1]` (both are supported):

- You would use the first for a record 1D with a field `b`. `A[1].b` accesses the field `b` of the second record element of the record array `A`.
- You would use the second for a scalar record with a field `b`, which is a 1D array. `A.b[1]` accesses the second element of the field `b` of the record `A`.

To change a field in a record, use the `Set Field` object. For example, suppose you have a record `R` with a field `A`, and you wish to change the value of `R.A` to be `sin(R.A)`. Just use `R.A` as the left-hand expression (specifying the field to change) and `sin(R.A)` as the right-hand expression (specifying the new value for the field) in a `Set Field` object. You can continue to use the record `R` (with the new value for field `A`) later in your HP VEE model.

Using Dyadic Operators

The set of dyadic operators have several additional conditions and guidelines. The dyadic operators are under the `Math` menu and are as follows:

- `+ - * / ⟹`
  - `+`
  - `-`
  - `*`
  - `/`
  - `^` (exponentiation)
  - `mod` (modulo - returns remainder of division)
  - `div` (integer division - no remainder)
- `Relational ⟹`
  - `==`
  - `!=`
  - `<`
  - `>`
  - `<=`
  - `>=`
- `Logical ⟹`
  - `AND`
  - `OR`
  - `XOR`
  - `NOT` (a monadic that follows the same guidelines as dyadics)

When using dyadic operators on arrays, the array size, array shape, and array mappings (if they exist) must match. For Coords, the values of the independent variable for each Coord must match.

Precedence of Dyadic Operators

This list is the order of precedence of the dyadic operators. They are listed from highest to lowest, with operators of the same precedence listed on the same level.

1. parentheses (and) used to group expressions
2. ^
3. unary minus -
4. * / MOD DIV
5. + -
6. == != < > <= >=
7. NOT
8. AND
9. OR XOR

Data Type Conversion

For the dyadic operators, the input values are promoted to the highest data type and then the operation is performed. The data type of the output is the highest input data type. For example, when the complex number (2, 3) is added to the String "Dog", "Dog"+(2,3), the result is the String "Dog(2, 3)".

Note

There is one exception to this rule. When you multiply a Text string by an Int32, the result is a repeated string. For example, "Hello"*3 returns HelloHelloHello. No data type promotion occurs in this case.

The data type order (from highest to lowest) is:

1. Record
2. Text (Enum is treated as Text)
3. Spectrum
4. PComplex
5. Complex
6. Coord (no conversion to any other numeric type possible)
7. Waveform
8. Real
9. Int32

Record Considerations. Records have the highest precedence of all data types, but other data types can be converted to the Record data type *only* by using special objects such as `Build Record`. Records will not automatically demote to other types, nor will other types automatically promote to the Record type.

The dyadic operators do support combining records and other data types, but they will always return a record in this case. A dyadic operation on a record and non-record will apply the operation with the non-record to every field of the record. For example, consider a record `R` with two fields `A`, a scalar Real value (2.0), and `B`, a scalar Complex value (3,30). The expression `R+2` will produce a record `R` with two fields `A`, a scalar Real with value 4, and `B`, a scalar Complex with value (5,30). If the operation cannot be performed on every field in the record, an error occurs.

Dyadic operations on a record and any other type will return a record with the same "schema," so the resulting record will have the same fields with the same names, types, and shapes. The dyadic operation may not change the type or shape of a field of a record. For example, consider a record `R` with two fields `A`, a scalar Real, and `B`, a scalar Complex. The expression `R+(2,3)` will cause an error. HP VEE will first try to add `(2,3)` to `R.A`, then do the same with `R.B`. The error occurs because the `R.A` field is a Real and the result of `R.A+(2,3)` would be a Complex. The Complex cannot be demoted to a Real to be stored back into `R.A`.

Dyadic operations on records using arrays treat the record as having higher precedence than the array. For example, `[1 2 3] + [3 4 5]` produces `[4 6 8]`, so the arrays are combined piece by piece. But records have higher precedence than arrays. This means that if `R` is a record with two fields `A` and `B`, the expression `R + [1 2]` will try to add the array `[1 2]` to each field of R. It will *not* add `1` to `R.A`, and `2` to `R.B`.

Things get even more complicated when you combine arrays with record arrays. For example, suppose `R` is a record 1D array, two long, with three fields `A`, `B`, and `C`. The expression `R + [1 2 3]`, or the expression `R + [1 2]` will add the entire array to each field `A`, `B`, and `C` for every element of R. Even though `R` is an array, the fact that it is a record is more important.

A dyadic operation on two records will combine them field by field, so the two records must have the same "schema." That is, each record must have the same number of fields, and each field must have the same name, type and shape, in the same order.

If you want to add `1` to field `A`, add `2` to field `B`, and so forth, there are two ways to do this. The first is to use multiple `Set Field` objects, one for each field, to change a field of an existing record. (See `Set Field` for more information.) The other way is to create a record of the same shape and "schema" as the original; put `1` in its `A` field, `2` in its `B` field, and so forth; and then add the two records.

Coord Considerations. The Coord data type has some special rules associated with it:

- Although arrays of Int32 and Real data types can be promoted to Coord, a Coord cannot be converted to any other numeric type.
- When unmapped arrays are converted to Coord, the independent Coord values (the first Coord fields) are created from the array indexes; the dependent Coord value (the last Coord field) contains the element value. For example, if array `A` is converted to a Coord and `A` contains `[1 5 7]`, it is converted to a Coord array with `[(0,1)(1,5)(2,7)]` in it.
- When mapped arrays are converted to Coord, the independent Coord parameter ranges from the low value of the mapping to the value `Xmin+(Xmax-Xmin/N)*(N-1)`.

Spectrum Considerations. If you choose to use dB scaling, you must keep track of it yourself. Although dB-scaled data displays correctly (except on the `Waveform (Time)` display), many math functions such as `fft(x)`, `ifft(x)`, and those involving PComplex numbers don't operate correctly on dB-scaled data. If you need to use these operations, convert the dB-scaled data to linear scaling before operating on it. HP VEE supplies library models for dB conversions in `/usr/lib/veeengine/lib/conversions/` or `/usr/lib/veetest/lib/conversions/`.

When you are using particular dB units, some math functions give meaningful results, but only within the confines of those units. For example, if you add `20` to a dBW-scaled Spectrum, `20` is added to the magnitude of each element (which has the same effect as converting the Spectrum to a linear scale, multiplying each element by `100`, and converting back to dBW.).

Data Shape Considerations

For dyadic operations where both operands (inputs) are arrays, the size and shape of the arrays must match. The result of the operation is an array with the same size and shape as the input arrays, except for the relational operators (`==`, `<`, and so on, which always return a Scalar.) If arrays have a different number of dimensions or are of different sizes, HP VEE returns an error. For example, `[1 2] + [1 2 3]` returns an error.

If you are operating on a scalar and an array, the scalar is treated as if it were a constant array of the same size and shape as the array operand.

For example, `2 + [1 2 3]` is treated as `[2 2 2] + [1 2 3]`. The result is `[3 4 5]`.

When an n-dimensional array is converted to a Coord, the Coord data shape is an Array 1D with $n+1$ fields in each Coord element.

Math Output Types

The following table summarizes the `Math` output types and mappings. Given the following parameter type (in the top row headings of the following table), the particular `Math` function returns the type and mapping listed.

Note

The Record data type is not included in this table. Most `Math` and `AdvMath` functions will not operate on records. If you attempt to do this you will generate an `Invalid operation for data type Record` error. The only functions supported for records are: `sort(x)`, `init(x, val)`, `concat(x,y)`, and `totSize(x)` (see the "AdvMath Output Types" table).

Table 3-1. Math Output Types

Math Function	Int32	Real	Coord	Wave-form	Complex, PComplex	Spec-trum	Enum, Text	Inputs Mapped	Output Mapped?
bit(x,n)	0/1	0/1	0/1	0/1	*	*	*	dc	same as x
bits(x)	*	*	*	*	*	*	Int32	dc	same
setBit(x,n)	same	Int32	Int32	Int32	*	*	*	dc	same as x
clearBit(x,n)	same	Int32	Int32	Int32	*	*	*	dc	same as x
bitAnd(x,y)	same	same(1)	same(1)	same(1)	*	*	*	==	same
bitOr(x,y)	same	same(1)	same(1)	same(1)	*	*	*	==	same
bitXor(x,y)	same	same(1)	same(1)	same(1)	*	*	*	==	same
bitCmpl(x)	same	same(1)	same(1)	same(1)	*	*	*	dc	same
bitShift(x,y)	same	same(1)	same(1)	same(1)	*	*	*	dc	same as x
abs(x)	same	same	same	same	Real	Real	*	dc	same
signof(x)	-1/0/1	-1/0/1	-1/0/1	-1/0/1	*	*	*	dc	same
ordinal(x)	same	same	same	same	*	*	Int32(2)	dc	same
round(x)	same	same	same	same	*	*	*	dc	same
floor(x)	same	same	same	same	*	*	*	dc	same
ceil(x)	same	same	same	same	*	*	*	dc	same
intPart(x)	same	same	same	same	*	*	*	dc	same
fracPart(x)	same	same	same	same	*	*	*	dc	same

Table 3-1. Math Output Types (continued)

Math Function	Int32	Real	Coord	Wave-form	Complex, PComplex	Spec-trum	Enum, Text	Inputs Mapped	Output Mapped?
j(x)	Cpx	Cpx	*	Cpx	*	*	*	dc	same
re(x)	same	same	same	same	Real	Real	*	dc	same
im(x)	(3)	(3)	(3)	(3)	Real	Real	*	dc	same
mag(x)	same	same	same	same	Real	Real	*	dc	same
phase(x)	(3)	(3)	(3)	(3)	Real	Real	*	dc	same
conj(x)	*	*	*	*	same	same	*	dc	same
strUp(str)	Text	Text	Text	Text	Text	Text	Text	dc	same
strDown(str)	Text	Text	Text	Text	Text	Text	Text	dc	same
strRev(str)	Text	Text	Text	Text	Text	Text	Text	dc	same
strTrim(str)	Text	Text	Text	Text	Text	Text	Text	dc	same
strLen(str)	Int32	Int32	Int32	Int32	Int32	Int32	Int32	dc	same
strFromThru (str,from,thru)	Text	Text	Text	Text	Text	Text	Text	dc	same
strFromLen (str,from,len)	Text	Text	Text	Text	Text	Text	Text	dc	same
strPosChar (str,char)	Int32	Int32	Int32	Int32	Int32	Int32	Int32	dc	same
strPosStr (str,str1,str2)	Int32	Int32	Int32	Int32	Int32	Int32	Int32	dc	same
ramp(n,f,t)	Real	Real	*	*	*	*	*	Scalar	no
logRamp(n,f,t)	Real	Real	*	*	*	*	*	Scalar	no
xramp(n,f,t)	Real	Real	*	*	*	*	*	Scalar	no
xlogRamp(n,f,t)	Real	Real	*	*	*	*	*	Scalar	no
sq(x)	same	same	same	same	same	same	*	dc	same
sqrt(x)	Real	same	same	same	same	same	*	dc	same
cubert(x)	Real	same	same	same	same	same	*	dc	same
recip(x)	Real	same	same	same	same	same	*	dc	same
log(x)	Real	same	same	same	same	same	*	dc	same
log10(x)	Real	same	same	same	same	same	*	dc	same
exp(x)	Real	same	same	same	same	same	*	dc	same
exp10(x)	Real	same	same	same	same	same	*	dc	same

Table 3-1. Math Output Types (continued)

Math Function	Int32	Real	Coord	Wave-form	Complex, PComplex	Spec-trum	Enum, Text	Inputs Mapped	Output Mapped?
poly(n,vec)	Real	Real	*	*	*	*	*	dc	same as n
sin(x)	Real	same	same	same	same	same	*	dc	same
cos(x)	Real	same	same	same	same	same	*	dc	same
tan(x)	Real	same	same	same	same	same	*	dc	same
cot(x)	Real	same	same	same	same	same	*	dc	same
asin(x)	Real	same	same	same	same	same	*	dc	same
acos(x)	Real	same	same	same	same	same	*	dc	same
atan(x)	Real	same	same	same	same	same	*	dc	same
acot(x)	Real	same	same	same	same	same	*	dc	same
atan2(y,x)	Real	same	same	same	same	same	*	==	same
sinh(x)	Real	same	same	same	same	same	*	dc	same
cosh(x)	Real	same	same	same	same	same	*	dc	same
tanh(x)	Real	same	same	same	same	same	*	dc	same
coth(x)	Real	same	same	same	same	same	*	dc	same
asinh(x)	Real	same	same	same	same	same	*	dc	same
acosh(x)	Real	same	same	same	same	same	*	dc	same
atanh(x)	Real	same	same	same	same	same	*	dc	same
acoth(x)	Real	same	same	same	same	same	*	dc	same
now() no parm	—	—	—	—	—	—	—	dc	Scalar
wday(aDate)	Real	same	same	same	*	*	*	dc	same
mday(aDate)	Real	same	same	same	*	*	*	dc	same
month(aDate)	Real	same	same	same	*	*	*	dc	same
year(aDate)	Real	same	same	same	*	*	*	dc	same
dmyToDate(d,m,y)	Real	same	same	same	*	*	*	==	same
hmsToSec(h,m,s)	Real	same	same	same	*	*	*	==	same
hmsToHour(h,m,s)	Real	same	same	same	*	*	*	==	same

Legend for the Math Output Types Table

same	Return same type.
*	Error, not implemented or doesn't make sense.
Real	Performs the action as Real and returns a Real.
—	Returns a new container. For details, see the reference information about that particular function later in this chapter.

Input Mappings Key for the Math Output Types Table

dc	Don't care, don't look at mappings.
==	Multi-parameter functions. Mappings, if they exist, must be equal.
Scalar	Input is Scalar, therefore unmapped.

Output Mappings Key for the Math Output Types Table

same	Same as `x` or `y` or equally mapped parameters.
same as x	Same as `x`, even if second parameter is mapped differently.
no	Output is not mapped, regardless of input mappings.
Scalar	Output is Scalar, therefore unmapped.

Notes Referenced in the Math Output Types Table

(1)	Performs action as an Int32 (error if overflow), returns same type.
(2)	Text will return an error, Enum types will return the ordinal value of the enum, an Int32.
(3)	Will return same type, but value(s) will always be zero.

AdvMath Output Types

The following table summarizes the `AdvMath` output types. Given the following parameter type (in the top row headings of the following table), the particular `AdvMath` function returns the type and mapping listed.

Note

The Record data type is not included in this table. Most `Math` and `AdvMath` functions will not operate on records. The only functions supported for records are: `sort(x)`, `init(x, val)`, `concat(x,y)`, and `totsize(x)`.

Table 3-2. AdvMath Output Types

AdvMath Function	Int32	Real	Coord	Wave-form	Complex, PComplex	Spec-trum	Enum, Text	Inputs Mapped	Output Mapped?
init(x,val)	same	same	same	same	same	same	Text	dc	same as val
totSize(x)	Int32	Int32	Int32	Int32	Int32	Int32	Int32	dc	same
rotate(x,n)	same	same	same	same	same	same	same	dc	same as x
concat(x,y)	same(4)	same(4)	same(4)	same(4)	same(4)	same(4)	same(4)	dc	special
sum(x)	same	same	Real	Real	same	Pcx	Text	dc	Scalar
prod(x)	same	same	Real	Real	same	Pcx	*	dc	Scalar
sort (x,dir,field)	same	same	same	same	same	same	same	dc	same
det(x)	Real	same	*	*	same	*	*	dc	Scalar
inverse(x)	Real	same	*	*	same	*	*	dc	same
transpose(x)	same	same	*	*	same	*	same	dc	transposed
identity(x)	same	same	*	*	same	*	*	dc	same
minor(x,r,c)	Real	same	*	*	same	*	*	dc	Scalar
cofactor (x,r,c)	Real	same	*	*	same	*	*	dc	Scalar
matMultiply (A,B)	Real	same	*	*	same	*	*	dc	no
matDivide (n,d)	Real	same	*	*	same	*	*	dc	no
integral(x)	Real	Real	same(4)	same	*	*	*	dc	same as x
deriv(x,o)	Real	Real	same(4)	same	*	*	*	dc	same as x
defIntegral (x,a,b)	Real	Real	Real(4)	Real	*	*	*	dc	Scalar
derivAt (x,o,pt)	Real	Real	same(4)	Real	*	*	*	dc	Scalar
polySmooth (x)	Real	same	same(4)	same	*	*	*	dc	same
meanSmooth (x,n)	Real	same	same(4)	same	*	*	*	dc	same as x

Table 3-2. AdvMath Output Types (continued)

AdvMath Function	Int32	Real	Coord	Wave-form	Complex, PComplex	Spec-trum	Enum, Text	Inputs Mapped	Output Mapped?
movingAvg (x,n)	Real	same	same	same	*	*	*	dc	same as x
clipUpper (x,a)	same	same	same	same	*	*	same	==	same
clipLower (x,a)	same	same	same	same	*	*	same	==	same
minIndex(x)	Int32	Int32	Int32	Int32	*	*	Int32	dc	Scalar
maxIndex(x)	Int32	Int32	Int32	Int32	*	*	Int32	dc	Scalar
minX(x)	Real	same	Real	Real	*	*	Real	dc	Scalar
maxX(x)	Real	same	Real	Real	*	*	Real	dc	Scalar
random(l,h)	Real	Real	*	*	*	*	*	Scalar	Scalar
randomize (x,low,high)	same	same	same	same	same	same	*	dc	same as x
randomSeed (s)	same	same	*	*	*	*	*	Scalar	Scalar
perm(n,r)	Real	same(5)	same(5)	same(5)	*	*	*	==	same
comb(n,r)	Real	same(5)	same(5)	same(5)	*	*	*	==	same
gamma(x)	Real	same	same	same	*	*	*	dc	same
beta(x,y)	Real	same	same	same	*	*	*	==	same
factorial(n)	Real	same(5)	same(5)	same(5)	*	*	*	dc	same
binomial (a,b)	Real	same(5)	same(5)	same(5)	*	*	*	==	same
erfc(x)	Real	same	same	same	*	*	*	dc	same
erf(x)	Real	same	same	same	*	*	*	dc	same
min(x)	same	same	same	Real	*	*	same	dc	Scalar
max(x)	same	same	same	Real	*	*	same	dc	Scalar
median(x)	Real	same	Real	Real	*	*	*	dc	Scalar
mode(x)	same	same	Real	Real	*	*	*	dc	Scalar
mean(x)	Real	same	Real	Real	*	*	*	dc	Scalar
sdev(x)	Real	same	Real	Real	*	*	*	dc	Scalar

Table 3-2. AdvMath Output Types (continued)

AdvMath Function	Int32	Real	Coord	Wave-form	Complex, PComplex	Spec-trum	Enum, Text	Inputs Mapped	Output Mapped?
vari(x)	Real	same	Real	Real	*	*	*	dc	Scalar
rms(x)	Real	same	Real	Real	*	*	*	dc	Scalar
magDist (x,f,t,s)	Real	same	Real	Real	Real(6)	Real(6)	*	dc	no
logMagDist (x,f,t,l)	Real	same	Real	Real	Real(6)	Real(6)	*	dc	no
j0(x)	Real	same	same	same	*	*	*	dc	same
j1(x)	Real	same	same	same	*	*	*	dc	same
jn(x,n)	Real	same	same	same	*	*	*	==	same
y0(x)	Real	same	same	same	*	*	*	dc	same
y1(x)	Real	same	same	same	*	*	*	dc	same
yn(x,n)	Real	same	same	same	*	*	*	==	same
Ai(x)	Real	same	same	same	*	*	*	dc	same
Bi(x)	Real	same	same	same	*	*	*	dc	same
i0(x)	Real	same	same	same	*	*	*	dc	same
i1(x)	Real	same	same	same	*	*	*	dc	same
k0(x)	Real	same	same	same	*	*	*	dc	same
k1(x)	Real	same	same	same	*	*	*	dc	same
fft(x)	Cpx	Cpx	Cpx(4)	Spectrum	*	*	*	dc	special
ifft(x)	*	*	*	*	Real	Waveform	*	dc	special
convolve (x,y)	Real	same	same	Waveform	*	same(7)	*	dc	special
xcorrelate (x,y)	Real	same	same	Waveform	*	same(7)	*	dc	special
bartlet(x)	Real	same	same	same	*	same(8)	*	dc	same
hamming(x)	Real	same	same	same	*	same(8)	*	dc	same
hanning(x)	Real	same	same	same	*	same(8)	*	dc	same
blackman(x)	Real	same	same	same	*	same(8)	*	dc	same
rect(x)	Real	same	same	same	*	same(8)	*	dc	same

Legend for the AdvMath Output Types Table

same	Return same type.
*	Error, not implemented or doesn't make sense.
Real	Performs the action as Real and returns a Real.

Input Mappings Key for the AdvMath Output Types Table

dc	Don't care, don't look at mappings.
==	Multi-parameter functions. Mappings, if they exist, must be equal.
Scalar	Input is Scalar, therefore unmapped.

Output Mappings Key for the AdvMath Output Types Table

same	Same as `x` or `y` or equally mapped parameters.
same as x	Same as `x`, even if second parameter is mapped differently.
no	Output is not mapped, regardless of input mappings.
Scalar	Output is Scalar, therefore unmapped.
special	Output container mappings are handled specially for each of the functions. See the reference for each function later in this chapter for information on how it is handled.
transposed	Mappings for the two dimensions are transposed.

Notes Referenced in the AdvMath Output Types Table

(4)	The dependent variable(s) of the Coord must be equidistant. That is, the x-interval between the points must be a constant.
(5)	The input parameters are converted to Int32 before the function is performed.
(6)	Uses the `mag(x)` magnitude of the Complex/PComplex/Spectrum, and then performs the function as Real.
(7)	Using an `ifft(x)`, converts the Spectrum to Waveform. Applies the convolution or autoCorrelation function, and then uses an `fft(x)` to convert back to Spectrum.
(8)	Using an `ifft(x)`, converts the Spectrum to a Waveform. Applies the weighting for the windowing function, and then uses an `fft(x)` to convert back to Spectrum.

abs(x)

An object that returns the absolute value of `x`.

Use

Use `abs(x)` to obtain the absolute value of the number in a container. `x` may be any shape and of the type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For types Int32, Real, Coord, and Waveform, the same type is returned. For types Complex, PComplex, and Spectrum, the absolute value is the magnitude of the complex number; therefore, a Real of the same shape is returned.

Location

`Math` ⟹ `Real Parts` ⟹ `abs(x)`

Example

`abs(-34)` returns `34`.

Notes

Mappings are retained in the result. The largest negative Int32 will cause an error.

See Also

`Complex Parts`, `mag(x)`, `Real Parts`, and `signof(x)`.

acos(x)

An object that returns the arccosine of `x`.

Use

Use `acos(x)` to generate the arccosine of the `x` data, with the result in the range of `0` to `PI`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The return value will be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Trig` ⟹ `acos(x)`

Example

`acos(1)` returns `0` with `Trig Mode` set to Degrees.

`acos((1.54308, @0))` returns `(0.9999, @1.87079)` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`acosh(x)`, `cos(x)`, `sin(x)`, `tan(x)`, and `Trig`.

`Trig Mode` in the "General Reference" chapter.

acosh(x)

An object that returns the hyperbolic arccosine of `x`.

Use

Use `acosh(x)` to generate the hyperbolic arccosine of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The return value will be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Hyper Trig` ⟹ `acosh(x)`

Example

`acosh(1)` returns `0` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`acos(x)`, `cosh(x)`, `Hyper Trig`, `sinh(x)`, and `tanh(x)`.

`Trig Mode` in the "General Reference" chapter.

acot(x)

An object that returns the arccotangent of `x`.

Use

Use `acot(x)` to generate the arccotangent of the `x` data, simply returning `atan(1/x)`. `acot(0)` will return `PI/2`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The return value will be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Trig` ⟹ `acot(x)`

Example

`acot(1)` returns `45` with `Trig Mode` set to Degrees.

`acot((1.31303, @PI/2))` returns `(1.00, @-1.57079)` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`acoth(x)`, `atan2(y,x)`, `cos(x)`, `sin(x)`, `tan(x)`, and `Trig`.

`Trig Mode` in the "General Reference" chapter.

acoth(x)

An object that returns the hyperbolic arccotangent of `x`.

Use

Use `acoth(x)` to generate the hyperbolic arccotangent of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The return value will be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Hyper Trig` ⟹ `acoth(x)`

Example

`acoth(1.5)` returns `0.8047` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`acot(x)`, `cosh(x)`, `coth(x)`, `Hyper Trig`, `sinh(x)`, and `tanh(x)`.

`Trig Mode` in the "General Reference" chapter.

+ (add)

An object that performs an arithmetic addition on two operands.

Use

Use + to add the values of two containers. The two containers may be of any type and of any shape. If one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a container of the highest type with the same shape as the operands.

If both operands are of type Coord, they must have their independent variable(s) match exactly or an error is returned. The addition is only performed on the dependent (last) variable.

Text performs addition as a concatenation. Enums convert to Text for the addition. Note that this addition on two strings is not the same as the `concat(x, y)` function. With the `concat` function, `concat("hello","there")` yields a one-dimensional array, two long `["hello" "there"]`.

Location

`Math` ⟹ `+ - * /` ⟹ `+`

Example

Array plus a scalar: `[1 2 3] + 3` returns `[4 5 6]`.

Two Complex scalars: `(2,4) + (1,3)` returns `(3,7)`.

Two arrays: `[1 2 3] + [4 5 6]` returns `[5 7 9]`.

Two Coord scalars: `coord(1,3) + coord(1,5)` returns `coord(1,8)`.

Two Coord scalars: `coord(1,3) + coord(2,5)` returns an error.

Two Text: `"hello" + "there"` returns `"hellothere"`.

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings. The return value will have the same mappings as the operands; otherwise an error is returned.

See Also

`concat(x,y)`, `/ (divide)`, `* (multiply)`, and `- (subtract)`.

+ - * /

A menu item.

Use

Use + - * / to access the following objects which perform arithmetic functions on two operands:

- +
- -
- *
- /
- ^
- mod
- div

Location

Math ⟹ + - * / ⟹

Notes

Dyadic operations which are given two different types convert the lower type to the higher of the two types. See the type conversion information and the hierarchy of types in the "Using Dyadic Operators" section at the beginning of this chapter.

In the dyadic operation example below, adding an array and a scalar performs the operation on each element of the array. The result is an array of the same size and shape as the array operand. For example, a scalar plus a linear array of four elements, `5 + [1 2 3 6]`, produces the linear array of four elements `[6 7 8 11]`. The same is true for operations like exponentiation. Thus, `2 ^ [3 4]` produces `[8 16]`.

Dyadic operations which are given two arrays require the operands to be conformant, that is, have the same size and shape. The result is an array with the same size and shape as the operands. The operation is done on an element by element basis. For example, adding two linear arrays of three elements each

+ - * /

results in a linear array of three elements long. `[1 2 3] + [4 5 6]` produces `[5 7 9]`. If the two arrays are not conformant, the result is an error.

Ai(x)

An object used to calculate the `Airy` fractional order Bessel function of `x`.

Use

Use `Ai(x)` to find the Fractional order Bessel function `Airy` of the `x`. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` $\Longrightarrow$ `Bessel` $\Longrightarrow$ `Ai(x)`

Example

`Ai(10)` returns `0.110475325528986`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`Bi(x)`, `j0(x)`, `j1(x)`, `jn(x,n)`, `y0(x)`, `y1(x)`, and `yn(x,n)`.

~= (almost equal to)

An object that performs an *almost equal to* operation on two operands.

Use

Use `~=` to determine whether the value(s) of one container is equal to the value(s) of another container for the first seven decimal digits. The two containers may be of any type and of any shape. Integer, Enum and Text types will be compared exactly. If one of the containers is an array (or a record), the other must be either a scalar or an array (or a record) of the same size and shape. The result is a scalar Int32 with the value `0` or `1`. If the first operand is almost equal to the second, the value of the result is `1`; otherwise the value is `0`.

The operation of `X ~= 0` (zero) is meaningless in this context since zero has an infinite number of digits of precision and no order of magnitude. This requires that any comparison to zero match exactly. Therefore, comparing any value `X` to `0` (zero) will always return a `0` (false) unless `X` exactly equals `0`.

Almost equals does not perform the same comparison as `Device` ⟹ `Comparator`. The Comparator should be used when comparing waveforms or arrays which contain zeros. The Comparator can derive a value for "virtual zero" from the reference data's magnitude.

If both operands are of type Coord, they must have all their dependent variables match for the first seven decimal digits for the the result to be `1`. If independent variables do not match, an error is returned. Complex, PComplex, and Spectrum containers must have both parts almost match for the operation to return `1`. Enums are converted to Text for the comparison.

Arrays must have all the respective values of both containers almost equal for the operation to return `1`.

Location

`Math` ⟹ `Relational` ⟹ `~=`

Example

Two scalars: `3.1234560 ~= 3.1234559` returns `1`.

Two scalars: `3.1234560 ~= 3.1234459` returns `0`.

A scalar and an array: `3.1234560 ~= [3.1234559 3.1234562 3.1234563]` returns `1`.

A scalar and an array: `3.1234560 ~= [3.1234459 3.1234562 3.1234569]` returns `0`.

Two Complex scalars: `(2.12345,3.12345) ~= (2.12344,3.12345)` returns `0`.

Two Complex scalars: `(2.1234567,3.12345) ~= (2.1234566,3.12345)` returns `1`.

Two Coord scalars: `coord(1,3.123456) ~= coord(1,3.123446)` returns `0`.

Two Coord scalars: `coord(1,3.123456) ~= coord(2,3.123456)` returns `Values for independent variables must match`.

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings; otherwise an error is returned.

The test for "almost equal" is slightly different for Complex and Polar Complex numbers. For two Real values (`a` and `b`) the test for "almost equal" is:

```
a - b <= a * 10E-7
```

However, for two Complex values (`a` and `b`) the test is:

```
re(a) - re(b) <= mag(a) * 10E-7
```

where:

`re(a)` is the real component of `a`

`re(b)` is the real component of `b`

`mag(a)` is the vector magnitude of `a`

~= (almost equal to)

This test is used for both Complex and Polar Complex numbers to avoid any ambiguity with regard the rectangular or polar treatment of "almost equal."

See Also

`AND`, `== (equal to)`, `> (greater than)`, `>= (greater than or equal to)`, `< (less than)`, `<= (less than or equal to)`, `NOT`, `!= (not equal to)`, `OR`, `Relational`, and `XOR`.

`Comparator`, `Conditional`, and `If/Then/Else` in the "General Reference" chapter.

AND

An object that performs a logical `AND` operation on two operands.

Use

Use `AND` to determine whether the value(s) of two containers are both logically true (non-zero). The two containers may be of any type and of any shape. If one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is an Int32 of the same shape as the operands, with value(s) `0` or `1`. If both operands are non-zero, the value of the `AND` operation is `1`; otherwise the value is `0`.

If both operands are of type Coord, they must have their independent variable(s) match exactly or an error is returned. Only the dependent (last) variable is considered for the `AND` operation.

For Complex, PComplex, and Spectrum containers, the value of the operand is true if either part is non-zero. Text is true if non-null. Enums are converted to Text for the operation.

Location

`Math` ⟹ `Logical` ⟹ `AND`

Example

A scalar and an array: `3 AND [3 3 3]` returns `[1 1 1]`.

A scalar and an array: `3 AND [-3 0 3]` returns `[1 0 1]`.

Two arrays: `[1 2 3] AND [0 1 (-1)]` returns `[0 1 1]`.

Two arrays: `(1,@90) AND (1,@85)` returns `1`.

Two Complex scalars: `(2,3) AND (0,1)` returns `1`.

Two Complex scalars: `(0,1) AND (1,0)` returns `1`.

Two Complex scalars: `(0,0) AND (0,0)` returns `0`.

Two Coord scalars: `coord(1,3) AND coord(1,5)` returns `1`.

AND

Two Coord scalars: `coord(1,3) AND coord(2,3)` returns an error.

A Text scalar and a scalar number: `"too" AND 2` returns `1`.

A Text scalar and a scalar number: `"" AND 0` returns `0` because one string is null and the other is not (`"0"`).

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings. The return value will have the same mappings as the operands; otherwise an error is returned.

Note that the `If/Then/Else` object requires the expression(s) inside it to evaluate to either a scalar or an array, which is either all zeros or all ones.

See Also

`NOT`, `OR`, `Relational`, and `XOR`.

`Conditional` and `If/Then/Else` in the "General Reference" chapter.

Array

A menu item.

Use

Use `Array` to access the following objects which perform miscellaneous functions on arrays.

- `init(x,val)`
- `totSize(x)`
- `rotate(x,numElem)`
- `concat(x,y)`
- `sum(x)`
- `prod(x)`
- `sort(x)`

Location

`AdvMath` ⟹ `Array` ⟹

asin(x)

An object that returns the arcsine of `x`.

Use

Use `asin(x)` to generate the arcsine of the `x` data with the result in the range of `-PI/2` to `+PI/2` for Real values. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The return value will be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Trig` ⟹ `asin(x)`

Example

`asin(1)` returns `90` with `Trig Mode` set to Degrees.

`asin((1.1752012, @90))` returns `(1, @90)` with `Trig Mode` set to Degrees.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`asinh(x)`, `cos(x)`, `sin(x)`, `tan(x)`, and `Trig`.

`Trig Mode` in the "General Reference" chapter.

asinh(x)

An object that returns the hyperbolic arcsine of `x`.

Use

Use `asinh(x)` to generate the hyperbolic arcsine of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The return value will be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Hyper Trig` ⟹ `asinh(x)`

Example

`asinh(0)` returns `0` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`asin(x)`, `cosh(x)`, `Hyper Trig`, `sinh(x)`, and `tan(x)`.

`Trig Mode` in the "General Reference" chapter.

atan(x)

An object that returns the arctangent of `x`.

Use

Use `atan(x)` to generate the arctangent of the `x` data, with the result in the range of `-PI/2` to `+PI/2` for Real values. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The return value will be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Trig` ⟹ `atan(x)`

Example

`atan(1)` returns `45` with `Trig Mode` set to Degrees.

`atan((1.5574, @0))` returns `(0.999, @0)` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`atanh(x)`, `atan2(y,x)`, `cos(x)`, `sin(x)`, `tan(x)`, and `Trig`.

`Trig Mode` in the "General Reference" chapter.

atan2(y,x)

An object that returns the arctangent of `y` divided by `x`.

Use

Use `atan2(y,x)` to generate the arctangent of `y` divided by `x`, with the proper sign, in the range of `-PI` to `+PI` for Real values. `x` and `y` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. If one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a container of the highest type, with the same shape as the operands. `x` and `y` are assumed to be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Trig` ⟹ `atan2(y,x)`

Example

`atan2(1, -1)` returns `135` with `Trig Mode` set to Degrees.

`atan2((1, 1), (5, 5))` returns `(0.19739, 0)` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`atan(x)`, `cos(x)`, `sin(x)`, `tan(x)`, and `Trig`.

`Trig Mode` in the "General Reference" chapter.

atanh(x)

An object that returns the hyperbolic arctangent of `x`.

Use

Use `atanh(x)` to generate the hyperbolic arctangent of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The return value will be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Hyper Trig` ⟹ `atanh(x)`

Example

`atanh(0.5)` returns `0.5493` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`atan(x)`, `cosh(x)`, `Hyper Trig`, `sinh(x)`, and `tanh(x)`.

`Trig Mode` in the "General Reference" chapter.

bartlet(x)

An object used to apply a Bartlet window to a time series of values.

Use

Use `bartlet(x)` to filter the values in `x` in the same manner as convolving `x` with the spectral transform of the Bartlet function. This has the effect of suppressing some of the noise due to the tails of the input sequence and the potential discontinuities they represent when sampling periodic signals.

The input `x` must be an Array 1D of type Int32, Real, Coord, or a Waveform, or a Spectrum. The same type is returned, except for Int32, which returns Real.

If `x` is a Spectrum, it is first converted to a Waveform using an `ifft(x)` before the window is applied. The result of the window is then converted back to type Spectrum using an `fft(x)`. A Spectrum is returned.

Location

`AdvMath` ⟹ `Signal Processing` ⟹ `bartlet(x)`

Example

`bartlet([1 1 1 1 1 1 1 1])` returns
`[0.125 0.375 0.625 0.875 0.875 0.625 0.375 0.125]`.

Notes

The `Bartlet` function in the time domain is represented as `2*(n+0.5)/N`, where `n` is the position (index) in the array, and `N` is the size of the array. The result will be an array of the same type as `x` and will have the same mappings as `x` (if any).

For a discussion of sidelobe levels and coherent gains, see: Ziemer, Tranter, and Fannin, *Signals and Systems*, Macmillan Publishing, New York, NY, 1983. ISBN #0-02-431650-4.

See Also

blackman(x), convolve(a,b), fft(x), hamming(x), hanning(x), ifft(x), and rect(x).

Bessel

A menu item.

Use

Use `Bessel` to calculate the bessel function of input data.

- `j0(x)`
- `j1(x)`
- `jn(x,n)`
- `y0(x)`
- `y1(x)`
- `yn(x,n)`
- `Ai(x)`
- `Bi(x)`

Location

`AdvMath` ⟹ `Bessel` ⟹

See Also

`Hyper Bessel`.

beta(x,y)

An object used to calculate the beta function of the input (x,y).

Use

Use beta(x,y) to calculate the beta function on x. The beta function is defined as the:

(gamma(x) * gamma(y)) / gamma(x+y)

The x and y values must be positive.

The x input may be of any shape and size and of the type Int32, Real, Coord, or Waveform. For x input of all types the same output type is returned, except for Int32 which returns a Real type. The second y parameter must be of the same type or be able to be converted to the same type as the x input value. If both of the inputs are arrays, they must be of exactly the same shape and size.

Location

AdvMath ⟹ Probability ⟹ beta(x,y)

Example

beta(3, 1) returns 0.333333333333333.

beta(8, 7) returns 4.16250416250416E-05.

beta(-1, 3) returns an error.

Notes

If both of the inputs are mapped then the mappings must be the same. The return value has the same mappings as the input if either input is mapped. If neither of the inputs is mapped, then the output is unmapped.

See Also

comb(n,r), factorial(x), gamma(x), and perm(n,r).

Bi(x)

An object used to calculate the `B-Airy` fractional order Bessel function of the second kind of `x`.

Use

Use `Bi(x)` to find the Fractional order Bessel function `B-Airy` of the second kind of `x`. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` $\Longrightarrow$ `Bessel` $\Longrightarrow$ `Bi(x)`

Example

`Bi(1)` returns `1.20742359495287`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`Ai(x)`, `j0(x)`, `j1(x)`, `jn(x,n)`, `y0(x)`, `y1(x)`, and `yn(x,n)`.

binomial(a,b)

An object used to calculate the number of combinations of `a` things taken `b` at a time.

Use

Use `binomial(a,b)` to calculate the number of combinations of `a` things taken `b` at a time using the formula:

```
binomial(a, b) = a!/((a-b)! * b!)
```

The `!` symbol means factorial.

The `a` input may be of any shape and size and of the type Int32, Real, Coord, or Waveform. For `a` input of all types the same output type is returned, except for Int32 which returns a Real type. The second `b` parameter must be of the same type or be able to be converted to the same type as the `a` input value. If both of the inputs are arrays, they must be of exactly the same shape and size.

The `binomial(a,b)` operation is only defined for integer operands so the input numbers, while of unique type, are converted to Int32 type before the calculation is done.

Location

`AdvMath` $\Longrightarrow$ `Probability` $\Longrightarrow$ `binomial(a, b)`

Example

`binomial(10,3)` will return as given by the formula:

```
10!/((10-3)!*3!) = 10!/(7!*3!) =(10*9*8)/(3*2*1) = 120.
```

`binomial(10.4,3.9)` will return as given by the formula:

```
10!/((10-3)!*3!) = 10!/(7!*3!) = (10*9*8)/(3*2*1) = 120.
```

The rule about converting Real to Int32 forces `10.4` to `10` and `3.9` to `3`.

Notes

`a` and `b` must both be positive and `a` must be greater than `b`.

This function is identical to `comb(n,r)`.

If both of the inputs are mapped then the mappings must be the same. The return value has the same mappings as the input if either input is mapped. If neither of the inputs is mapped, then the output is unmapped.

See Also

`beta(x,y)`, `comb(n,r)`, `factorial(n)`, `gamma(x)`, and `perm(n,r)`.

bit(x,n)

An object that returns the binary value in a particular bit position.

Use

Use `bit(x,n)` to return the bit value in the `n`th position of `x`. `x` can be any shape and of type Int32, Real, Coord, or Waveform. If `x` is not of type Int32, it is converted to Int32, retaining shape. The return type is the same as `x` with a value of zero or one. `n` must be a container that is, or can be converted to, Int32 and has a value between `0` and `31` inclusive. `n` must be either scalar or match the shape of `x`.

Location

`Math` $\Longrightarrow$ `Bitwise` $\Longrightarrow$ `bit(x,n)`

Example

`bit(9,0)` returns `1` because the binary value of `9` is `1001` and the `0` bit is set.

Notes

The return value has the same mappings as `x`. Mappings on `n` are ignored.

See Also

`bits(str)`, `Bitwise`, `clearBit(x,n)`, and `setBit(x,n)`.

bitAnd(x,y)

An object that returns the `Bitwise AND` of `x` and `y`.

Use

Use `bitAnd(x,y)` to perform a binary AND on two containers. `x` and `y` can be any shape. But if both are arrays, the shape and sizes must match. `x` and `y` may be of types Int32, Real, Coord, or Waveform. Both `x` and `y` are converted to Int32 before doing the `bitAnd`. The return value is of type Int32.

Location

`Math` ⟹ `Bitwise` ⟹ `bitAnd(x,y)`

Example

`bitAnd(5, [3 12])` returns `[1 4]`.

Notes

If any of the input parameters are mapped, then those mappings must be the same. The resultant container retains the mappings of the input container. If only one of the inputs is mapped, the resultant container has those mappings.

See Also

`AND`, `bit(x,n)`, `bitCmpl(x)`, `bitOr(x,y)`, `Bitwise`, `bitXor(x,y)`, `NOT`, `OR`, and `XOR`.

bitCmpl(x)

An object that returns the `Bitwise` `1`'s complement of `x`.

Use

Use `bitCmpl(x)` to perform a binary `1`'s complement on a container. `x` may be any shape and of the type Int32, Real, Coord, or Waveform. `x` is converted to Int32 before the operation and the result is converted back to `x`'s original type.

Location

`Math` ⟹ `Bitwise` ⟹ `bitCmpl(x)`

Example

`bitCmpl(7)` returns `-8`.

Notes

Mappings are retained in the result.

See Also

`AND`, `bit(x,n)`, `bitAnd(x,y)`, `bitOr(x,y)`, `Bitwise`, `bitXor(x,y)`, `NOT`, `OR`, and `XOR`.

bitOr(x,y)

An object that returns the `Bitwise OR` of `x` and `y`.

Use

Use `bitOr(x,y)` to perform a binary OR on two containers. `x` and `y` can be any shape. But if both are arrays, the shapes and sizes must match. `x` and `y` may be of types Int32, Real, Coord, or Waveform. Both `x` and `y` are converted to Int32 before the `bitOr`. The return value is of type Int32.

Location

`Math` ⟹ `Bitwise` ⟹ `bitOr(x,y)`

Example

`bitOr(5, [3 12])` returns `[7 13]`.

Notes

If any of the input parameters are mapped, then those mappings must be the same. The resultant container retains the mappings of the input container. If only one of the inputs is mapped, the resultant container has those mappings.

See Also

`AND`, `bit(x,n)`, `bitAnd(x,y)`, `bitCmpl(x)`, `Bitwise`, `bitXor(x,y)`, `NOT`, `OR`, and `XOR`.

bits(str)

An object that returns the binary value of a string of 0s and 1s.

Use

Use `bits(str)` to return an Int32 with the value of the digits in the Text `str`, converted to Int32 as base 2. `str` must be a Text or Enum container of any shape. (Enum is converted to Text.) `str` may only contain leading white space, then 0s and 1s to form a binary number.

Location

`Math` ⟹ `Bitwise` ⟹ `bits(str)`

Example

`bits("011")` returns 3.

Notes

The return value has the same mappings as the input container. The least significant bit is on the right, and the most significant bit is on the left.

See Also

`bit(x,n)`, `Bitwise`, `clearBit(x,n)`, and `setBit(x,n)`.

bitShift(x,y)

An object that returns `x` with the bits shifted left or right `n` places.

Use

Use `bitShift(x,y)` to shift the bits of container `x` by `y` places. `x` may be any shape and of the types Int32, Real, Coord, or Waveform. `x` is converted to Int32 before the operation and the result is converted back to `x`'s original type. `y` must be a container which is or can be converted to Int32. If `y` is negative, the bits are shifted right; otherwise the bits are shifted left. `y` must be either scalar or match the shape of `x`. If `y` is greater than `31` or less than `-31`, the return value will be zero.

Location

`Math` ⟹ `Bitwise` ⟹ `bitShift(x,y)`

Example

`bitShift([1 7 7], [3 2 (-2)])` returns `[8 28 1]`.

`bitShift(8,2)` returns `32`.

`bitShift(8,-2)` returns `2`.

Notes

Mappings are retained in the result.

See Also

`bit(x,n)`, `bitAnd(x,y)`, `bitOr(x,y)`, `Bitwise`, `bitXor(x,y)`, `clearBit(x,n)`, `rotate(x,numElem)`, and `setBit(x,n)`.

Bitwise

A menu item.

Use

Use `Bitwise` to access the following objects which set, clear, access, or shift bits in an Int32, or perform binary AND, OR, and XORs on Int32s.

- `bit(x,n)`
- `bits(str)`
- `setBit(x,n)`
- `clearBit(x,n)`
- `bitAnd(x,y)`
- `bitOr(x,y)`
- `bitXor(x,y)`
- `bitCmpl(x)`
- `bitShift(x,y)`

Location

`Math` $\Longrightarrow$ `Bitwise` $\Longrightarrow$

Notes

The objects on the `Bitwise` menu make sense only for the type Int32. But for ease of use, most of the objects accept and return most types. Note that internally most of these objects convert the arguments to Int32 before performing the function, then convert the result back to the original type. This means that the values of Reals, Coords, and Waveforms are truncated inside the objects.

Bits are numbered from zero.

See Also

`AND`, `NOT`, `OR`, `Relational`, and `XOR`.

bitXor(x,y)

An object that returns the `Bitwise XOR` of `x` and `y`.

Use

Use `bitXor(x,y)` to perform a binary XOR (exclusive or) on two containers. `x` and `y` can be any shape. But if both are arrays, the shapes and sizes must match. `x` and `y` may be of the type Int32, Real, Coord, or Waveform. Both `x` and `y` are converted to Int32 before the `bitXor()`. The return value is of type Int32.

Location

`Math` $\Longrightarrow$ `Bitwise` $\Longrightarrow$ `bitXor(x,y)`

Example

`bitXor(5, [3 12])` returns `[6 9]`.

Notes

If any of the input parameters are mapped, then those mappings must be the same. The resultant container retains the mappings of the input container. If only one of the inputs is mapped, the resultant container has those mappings.

See Also

`AND`, `bit(x,n)`, `bitAnd(x,y)`, `bitCmpl(x)`, `bitOr(x,y)`, `Bitwise`, `NOT`, `OR`, and `XOR`.

blackman(x)

An object used to apply a Blackman window to a time series of values.

Use

Use `blackman(x)` to filter the values in `x` in the same manner as convolving `x` with the spectral transform of the Blackman function. This has the effect of suppressing some of the noise due to the tails of the input sequence and the potential discontinuities they represent when sampling periodic signals.

The input `x` must be an Array 1D of type Int32, Real, Coord, or a Waveform, or a Spectrum. The same type is returned, except for Int32, which returns Real.

If `x` is a Spectrum, it is first converted to a Waveform using an `ifft(x)` before the window is applied. The result of the window is then converted back to type Spectrum using an `fft(x)`. A Spectrum is returned.

Location

`AdvMath` ⟹ `Signal Processing` ⟹ `blackman(x)`

Example

`blackman([1 1 1 1 1 1 1 1])` returns
`[0.014 0.172 0.555 0.938 0.938 0.555 0.172 0.014]`.

Notes

The `Blackman` function is represented in the time domain as:

```
0.42 - 0.5*cos(2*PI*(n+0.5)/N) + 0.08*cos(4*PI*(n+0.5)/N)
```

where `n` is the position (index) in the array, and `N` is the size of the array. The result will be an array of the same type as `x` and will have the same mappings as `x` (if any).

For a discussion of sidelobe levels and coherent gains, see: Ziemer, Tranter, and Fannin, *Signals and Systems*, Macmillan Publishing, New York, NY, 1983. ISBN #0-02-431650-4.

See Also

`bartlet(x)`, `convolve(a,b)`, `fft(x)`, `hamming(x)`, `hanning(x)`, `ifft(x)`, and `rect(x)`.

Calculus

A menu item.

Use

Use `Calculus` to take the integral or the derivative of a one-dimensional array of simple numeric values (Int, Real, Waveform, ...) or lists of two-dimensional (two field) coordinates.

- `integral(x)`
- `deriv(x,1)`
- `deriv(x,2)`
- `deriv(x,order)`
- `defIntegral(x,a,b)`
- `derivAt(x,1,pt)`
- `derivAt(x,2,pt)`
- `derivAt(x,order,pt)`

Location

`AdvMath` ⟹ `Calculus` ⟹

Notes

The integral and derivative functions are applicable to one-dimensional arrays of simple numeric values (Int, Real, Waveform, and so forth) and lists of two-dimensional (two field) coordinates which represent equally spaced ordered data. For unmapped arrays, it is assumed that the data is equally spaced and ordered, and a value of `1` is assumed for the interval `dx` between points. For mapped arrays, the operation is performed with the interval value `dx` equal to `(Xmax-Xmin)/N` and is thus automatically scaled appropriately. For a coordinate list, the independent (first field) values are first checked to be sure that they are ordered and equally spaced, then the operation is performed on the dependent (second field) values using the previously determined spacing as the value for `dx`.

If an unmapped array is used, but it is known otherwise that the interval between points is some value `dx`, then the correct value for the operation can

still be obtained. For the integral, multiply the result by the known `dx` value. For the derivative, divide the result by the known `dx` value.

ceil(x)

An object that returns the rounded-up value of `x` to the nearest integer of a container.

Use

`x` may be any shape and of the types Int32, Real, Coord, or Waveform. The `ceil(x)` function returns the smallest integer (as the same type) greater than or equal to `x`; that is, the value of `x` rounded to the next larger integer, the nearest integer towards positive infinity.

Location

`Math` ⟹ `Real Parts` ⟹ `ceil(x)`

Example

`ceil([23.0 23.1 23.9 23.5 (-23.5)])` returns `[23 24 24 24 -23]`.

Notes

Mappings are retained in the result.

See Also

`abs(x)`, `Complex Parts`, `floor(x)`, `fracPart(x)`, `intPart(x)`, `Real Parts`, and `round(x)`.

clearBit(x,n)

An object that returns `x` with the `n`th bit set to `0`.

Use

Use `clearBit(x,n)` to set a particular binary digit of a container `x` to `0`. `x` can be any shape and of types Int32, Real, Coord, or Waveform. If `x` is not of type Int32, it is converted to Int32, retaining shape. The return value is of type Int32. `n` must be a container which is, or can be converted to, Int32 and has a value between `0` and `31` inclusive. `n` must be either scalar or match the shape of `x`.

Location

`Math` ⟹ `Bitwise` ⟹ `clearBit(x,n)`

Example

`clearBit(7,1)` returns `5` because the binary value of `7` is `111`, and the first bit is cleared yielding `101` (decimal `5`). (Bit numbering starts with `0`.)

Notes

The mappings of the resultant container are the same as the `x` parameter.

See Also

`bit(str)`, `bits(x,n)`, `Bitwise`, and `setBit(x,n)`.

clipLower(x,a)

An object used to clip the input data `x` to a certain minimum value `a`.

Use

Use `clipLower(x,a)` to clip the data to a minimum value `a`, that is, any data that is less than `a` will be changed to `a`.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, Waveform, or Text. For `x` input of all types, the same output type is returned. The clip parameter `a` must be of the same type, or be able to be converted to the same type, as the `x` input value and must either be Scalar in shape or the same shape and size as `x`. If one of the inputs is Scalar and one of the inputs is non-Scalar, the Scalar input is treated as if it were an array of the same shape and size as the non-Scalar input.

Location

`AdvMath` ⟹ `Data Filtering` ⟹ `clipLower(x,a)`

Example

Using the `clipLower(x,.5)` function on data, where `x` is the array `[.1, .4, .6, .9]`, returns `[.5, .5, .6, .9]`.

Using `clipLower([.1, .4, .6, .9],[.2, .2, .7, .7])` returns `[.2, .4, .7, .9]`.

You can use `clipLower` to clip the values of a Waveform to an arbitrary lower limit. Use an array of Coords to define the limiting points on the waveform. The limiting points are the endpoints of line segments that define the limiting waveform shape. There might be just a few of these points. Send the Coord array through the `Build Arb Waveform` object to create a lower limit waveform with the same number of points as your actual waveform. Send that new waveform into a `clipLower` function as the `a` input. When the `clipLower(x,a)` is done, the input waveform `x` will have all values below the lower limit clipped to the minimum allowed by the `a` input.

Notes

If both of the inputs are mapped, then the mappings must be the same. The return value has the same mappings as the input if either input is mapped. If neither of the inputs is mapped, then the output is unmapped.

See Also

`clipUpper(x,a)`.

`Build Arb Waveform` and `Comparator` in the "General Reference" chapter.

clipUpper(x,a)

An object used to clip the input data `x` to a certain maximum value `a`.

Use

Use `clipUpper(x,a)` to clip the data to a maximum value `a`. That is, any data that is greater than `a` will be changed to `a`.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, Waveform, or Text. For `x` input of all types, the same output type is returned. The clip parameter `a` must be of the same type, or be able to be converted to the same type, as the `x` input value and must either be Scalar in shape or the same shape and size as `x`. If one of the inputs is Scalar and one of the inputs is non-Scalar, the Scalar input is treated as if it were an array of the same shape and size as the non-Scalar input.

Location

`AdvMath` ⟹ `Data Filtering` ⟹ `clipUpper(x,a)`

Example

Using the `clipUpper(x,.5)` function on data, where `x` is the array `[.1, .4, .6, .9]`, returns `[.1, .4, .5, .5]`.

Using `clipUpper([.1, .4, .6, .9],[.2, .2, .5, .5])` returns `[.1, .2, .5, .5]`.

You can use `clipUpper` to clip the values of a waveform to an arbitrary upper limit. Use an array of Coords to define the limiting points on the waveform. The limiting points are the endpoints of line segments that define the limiting waveform shape. Send the Coord array through the `Build Arb Waveform` object to create an upper limit waveform with the same amount of points as your actual waveform. Send that new waveform into a `clipUpper` function as the `a` input. When the `clipUpper(x,a)` is done, the input waveform `x` will have all values above the upper limit clipped to the maximum allowed by the `a` input.

Notes

If both of the inputs are mapped then the mappings must be the same. The return value has the same mappings as the input if either input is mapped. If neither of the inputs is mapped, then the output is unmapped.

See Also

`clipLower(x,a)`.

`Build Arb Waveform` and `Comparator` in the "General Reference" chapter.

cofactor(x,row,col)

An object used to calculate the cofactor of the input matrix `x`, at row `r` and column `c`.

Use

Use `cofactor(x,row,col)` to calculate the cofactor of the square matrix `x`. The cofactor of a matrix is defined as the determinant of the submatrix of the input matrix `x` obtained by deleting the `r`th row and `c`th column, times `(-1)` raised to the `(r+c)` power.

The `x` input must be a square matrix shape and of the type Int32, Real, Complex or PComplex. For `x` input of all types the same output type is returned and is Scalar in shape, except for Int32 which returns a Real. The input for the row and column to delete, `r` and `c`, must be an Int32 Scalar or be able to be converted to Int32 type.

A square matrix has the same number of rows as columns.

Location

`AdvMath` $\Longrightarrow$ `Matrix` $\Longrightarrow$ `cofactor(x,row,col)`

Example

`cofactor(a,1,2)`, where `a` is a matrix `[ [3 -1 2] [4 5 6] [7 1 2] ]`, returns `34`.

Notes

Mappings on the operand are ignored.

The `r` and `c` inputs are expected to be of `Int32` type or be able to be converted to that type. The rows and columns of the matrix are numbered from 0 to n-1. Therefore, be careful when specifying which row and column to use.

See Also

`det(x)` and `minor(x,row,col)`.

comb(n,r)

An object used to calculate the number of combinations of `n` things taken `r` at a time.

Use

Use `comb(n,r)` to calculate the number of combinations of `n` things taken `r` at a time using the formula:

```
comb(n, r) = n!/((n -r)! * r!)
```

The `!` symbol means factorial.

The `n` input may be of any shape and size and of the type Int32, Real, Coord, or Waveform. For `n` input of all types the same output type is returned, except for Int32 which returns a Real type. The second `r` parameter must be of the same type or be able to be converted to the same type as the `n` input value. If both of the inputs are arrays, they must be of exactly the same shape and size.

The `comb(n, r)` operation is only defined for integer operands so the input numbers, while of unique type, are converted to Int32 type before the calculation is done.

Location

`AdvMath` ⟹ `Probability` ⟹ `comb(n, r)`

Example

`comb(10,3)` will return `120` as given by the formula:

`10!/((10-3)!*3!) = 10!/(7!*3!) = (10*9*8)/(3*2*1) = 120.`

`comb(10.4,3.9)` will return `120` as given by the formula:

`10!/((10-3)!*3!) = 10!/(7!*3!) = (10*9*8)/(3*2*1) = 120.`

The rule about converting Real to Int32 forces `10.4` to `10` and `3.9` to `3`.

Notes

Both `n` and `r` must be positive and `n` must be greater than `r`.

If both of the inputs are mapped, then the mappings must be the same. The return value has the same mappings as the input if either input is mapped. If neither of the inputs is mapped, then the output is unmapped.

This function is identical to `binomial(a,b)`.

See Also

`beta(x,y)`, `binomial(a,b)`, `factorial(n)`, `gamma(x)`, and `perm(n,r)`.

Complex Parts

A menu item.

Use

Use `Complex Parts` to access the following objects which return different parts of Complex and PComplex containers.

- `j(x)`
- `re(x)`
- `im(x)`
- `mag(x)`
- `phase(x)`
- `conj(x)`

Location

`Math` ⟹ `Complex Parts` ⟹

Notes

The objects on the `Complex Parts` menu make sense only for the type Complex and PComplex. But for ease of use, most of the objects accept and return most types.

See Also

`Real Parts`.

`Build Complex`, `Build PComplex`, `Complex`, `UnBuild Complex`, and `UnBuild PComplex` in the "General Reference" chapter.

concat(x,y)

An object used to concatenate containers.

Use

Use `concat(x,y)` to concatenate two containers. The input parameters can be of any type and any shape. The `concat(x,y)` function will flatten arrays that are greater than one dimension into an Array 1D. The resulting 1D arrays are then concatenated together. For `x` and `y` input of all types, a 1D container of the highest type is returned.

Location

`AdvMath` $\Longrightarrow$ `Array` $\Longrightarrow$ `concat(x,y)`

Example

If `x` is a Real Scalar container with `3` in it, and `y` is a Text array container with `["This" "is" "a" "test"]` in it, then `concat(x,y)` will return a Text array container `["3" "This" "is" "a" "test"]`. In this case, the numeric `3` has been converted to the higher type Text string `"3"` before the `concat` operation.

Notes

If the `x` value is mapped, the `y` value may be either mapped with the same point spacing, or unmapped. The result will be mapped from `Xmin` to `Xmin+dx*N`, where `Xmin` is the minimum value of the `x` mapping, `dx` is the distance between two consecutive points in `x`, and `N` is the size of the resultant array.

Note the difference between the `concat(x,y)` and `sum(x)` functions on Text inputs. The `concat(x,y)` creates an Array 1D in all cases. The `sum(x)` function will simply link all the text strings together to form one large output string. That is, `concat(x,y)`, where `x` is the Scalar Text value `"a"` and `y` is the Scalar Text input `"b"`, yields the Array 1D `["a","b"]`. On the other hand, `sum(["a","b"])` will return the Scalar container with the text value of `"ab"` in it.

All nil inputs are automatically converted to Real scalars with the value `0`. Thus, where `x` is nil and `y` is an Int32 scalar value `5`, `concat(x,y)` results in the Array 1D two long of Real with values `[0 5]`. In contrast, the `Concatenator` object with two inputs with nil and an Int32 scalar value `5` will output an Int32 array one long with value `[5]`.

See Also

`sum(x)`.

`Concatenator` in the "General Reference" chapter.

conj(x)

An object that returns the complex conjugate of a Complex number `x`.

Use

Use `conj(x)` to generate the complex conjugate of a Complex number `x`. `x` can be any shape and of type Complex, PComplex, or Spectrum. The complex conjugate of a Complex number simply reverses the sign of the imaginary component. The complex conjugate of a PComplex number simply reverses the sign of the phase.

Location

`Math` ⟹ `Complex Parts` ⟹ `conj(x)`

Example

`conj( (2, 3) )` returns `(2, -3)`.

`conj( (8, @45) )` returns `(8, @-45)` with `Trig Mode` set to Degrees.

Notes

Mappings are retained in the result.

See Also

`im(x)`, `j(x)`, `mag(x)`, `phase(x)`, `re(x)`, and `Real Parts`.

`Build Complex`, `Trig Mode`, and `UnBuild Complex` in the "General Reference" chapter.

convolve(a,b)

An object used to convolve two arrays of data.

Use

Use `convolve(a,b)` to calculate the discrete aperiodic convolution of two 1D arrays `a` and `b`. The result will be an array of the same type and of size `(Na+Nb)-1`, where `Na` is the size of input array `a`, and `Nb` is the size of input array `b`.

The input values `a` and `b` must be an Array 1D of type Int32, Real, Coord, or Waveform. The return type is the same as the highest type of the inputs, except Int32, which returns Real. `a` and `b` do not have to be the same size. The resultant values are not normalized.

Location

`AdvMath` ⟹ `Signal Processing` ⟹ `convolve(a,b)`

Example

`convolve([1 2 3 4 5],[1 1 1])` returns `[1 3 6 9 12 9 5]`.

Notes

The inputs `a` and `b` must represent equally spaced data. In addition, the interval between any two values of `a` must be the same as that between any two values of `b`. For two unmapped arrays, it is assumed that the interval is always `1`. For mapped arrays, the interval is `(Xmax-Xmin)/N`, where `Xmax` and `Xmin` are the mappings, and `N` is the size of the array. When one input is mapped and the other is not, the unmapped input is assumed to be sampled at the same frequency as the mapped input. The resultant values will never be normalized to any scale factor.

See Also

`fft(x)`and `xcorrelate(a,b)`.

cos(x)

An object that returns the cosine of `x`.

Use

Use `cos(x)` to generate the cosine of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. `x` is assumed to be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Trig` ⟹ `cos(x)`

Example

`cos([0 180])` returns `[1 -1]` with `Trig Mode` set to Degrees.

`cos((1, PI/4))` returns `(0.715684, -0.73096)` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`acos(x)`, `cosh(x)`, `sin(x)`, `tan(x)`, and `Trig`.

`Trig Mode` in the "General Reference" chapter.

cosh(x)

An object that returns the hyperbolic cosine of `x`.

Use

Use `cosh(x)` to generate the hyperbolic cosine of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. `x` is assumed to be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Hyper Trig` ⟹ `cosh(x)`

Example

`cosh(0)` returns `1` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`acosh(x)`, `cos(x)`, `Hyper Trig`, `sinh(x)`, and `tanh(x)`.

`Trig Mode` in the "General Reference" chapter.

cot(x)

An object that returns the cotangent of `x`.

Use

Use `cot(x)` to generate the cotangent of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. `x` is assumed to be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Trig` ⟹ `cot(x)`

Example

`cot([45 90])` returns `[1 61E-18]` with `Trig Mode` set to Degrees. Note that the second number, which should be zero, is smaller than the system precision.

`cot((3, @PI/2))` returns `(1.0049698, @-1.57079)` with `Trig Mode` set to Radians.

`cot((3, @90))` returns `(1.0049698, @-90)` with `Trig Mode` set to Degrees.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`acot(x)`, `atan2(y,x)`, `cos(x)`, `coth(x)`, `sin(x)`, `tan(x)`, and `Trig`.

`Trig Mode` in the "General Reference" chapter.

coth(x)

An object that returns the hyperbolic cotangent of `x`.

Use

Use `coth(x)` to generate the hyperbolic cotangent of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. `x` is assumed to be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Hyper Trig` ⟹ `coth(x)`

Example

`coth(0.4)` returns `2.631` with `Trig Mode` set to Radians, essentially, positive infinity.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`acoth(x)`, `cot(x)`, `Hyper Trig`, `sinh(x)`, and `tanh(x)`.

`Trig Mode` in the "General Reference" chapter.

cubert(x)

An object that returns the cube root of the value of `x`.

Use

Use `cubert(x)` to generate the cube root of a number `x`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For PComplex, the `cubert(x)` is defined as the cube root of the magnitude and one third the phase. Complex is converted to PComplex before the function is applied. Int32 arguments will return a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Power` ⟹ `cubert(x)`

Example

`cubert( (8, @90) )` will return `(2, @30)`.

`cubert( (1,1) )` will return `(1.08, 0.291)`.

Notes

Mappings are retained in the result.

See Also

`^ (exponent)`, `Power`, `sq(x)`, and `sqrt(x)`.

Data Filtering

A menu item.

Use

Use `Data Filtering` to apply filtering algorithms to the data to smooth it.

- `polySmooth(x)`
- `meanSmooth(x,numPts)`
- `movingAvg(x,numPts)`
- `clipUpper(x,a)`
- `clipLower(x,a)`
- `minIndex(x)`
- `maxIndex(x)`
- `minX(x)`
- `maxX(x)`

Location

`AdvMath` $\Longrightarrow$ `Data Filtering` $\Longrightarrow$

See Also

`Regression` and `Signal Processing`.

defIntegral(x,a,b)

An object used to calculate the integral of those values of the input data which are between points `a` and `b` (inclusive) using Simpson's 1/3 rule.

Use

Use `defIntegral(x,a,b)` to calculate the numerical approximation of the integral of a set of ordered, equally-spaced values (`x`) between points `a` and `b`. The result is a scalar value of type Real.

Location

`AdvMath` ⟹ `Calculus` ⟹ `defIntegral(x,a,b)`

Example

`defIntegral([ 0 1 2 3 4 4 3 2 1 0 ], 2, 5)` returns `10.12`.

Notes

The `defIntegral(x,a,b)` function is applicable to one-dimensional arrays of simple numeric values (Int32, Real, Waveform) and lists of two-dimensional (two field) coordinates which represent equally spaced ordered data. For unmapped arrays, the values `a` and `b` represent indices in the array `(0 to N-1)`. For coordinate lists, the values `a` and `b` represent real values between the x-values of the first and last point in the coordinate list (inclusive). For mapped arrays, the values `a` and `b` represent real values between the `Xmin` mapping and `Xmax-dx` (inclusive), where `dx` is the interval between consecutive points: `(Xmax-Xmin)/N`.

For unmapped arrays, it is assumed that the data is equally spaced and ordered, and a value of `1` is assumed for the interval `dx` between points. For mapped arrays, the operation is performed with the interval value `dx` equal to `(Xmax-Xmin)/N` and is thus automatically scaled appropriately. For a coordinate list, the independent (first field) values are first checked to be sure that they are ordered and equally spaced, then the operation is performed on the dependent (second field) values using the previously determined spacing as the value for `dx`.

If an unmapped array is used, but it is known otherwise that the interval between points is some value `dx`, then the correct value for the operation can be obtained by multiplying the result of `defIntegral(x,a,b)` by the known `dx` value.

When the values represent noisy data, the result of `defIntegral(x)` can often be improved (in a pragmatic sense) by first filtering or smoothing the data. Smoothing removes much of the unwanted noise which is superimposed on the underlying values. See the `polySmooth(x)` function and related smoothing functions for examples.

See Also

`integral(x)`.

deriv(x,order)

An object used to calculate the derivative of order `order` across all values of the input data using a sliding fourth-order (five-point) polynomial.

Use

Use `deriv(x,order)` to calculate the numerical approximation of the derivative of order `order` across all points of a set of ordered equally spaced values (`x`). The result is an array of the same size and shape as the input `x`. The result is the same type as `x` except for Int32. Int32 returns a Real and has the same mappings (if any) as the input `x`.

Location

`AdvMath` ⟹ `Calculus` ⟹ `deriv(x,order)`

Example

`deriv([0 1 2 3 3 2 1 0], 1)` returns
`[1.25 0.9167 1.083 0.5833 -0.5833 -1.083 -0.9167 -1.25]`.

Notes

The `deriv(x,order)` function is applicable to one-dimensional arrays of simple numeric values (Int32, Real, Waveform) and lists of two-dimensional (two field) coordinates that represent equally spaced ordered data. For mapped arrays, the mappings of the resultant array will be identical to those of `x`. For coordinate lists, the values `a` and `b` represent real values between the x-values of the first and last point in the coordinate list (inclusive). For mapped arrays, the values `a` and `b` represent real values between the `Xmin` mapping and `Xmax-dx` (inclusive), where `dx` is the interval between consecutive points: `(Xmax-Xmin)/N`.

For unmapped arrays, it is assumed that the data is equally spaced and ordered, and a value of `1` is assumed for the interval `dx` between points. For mapped arrays, the operation is performed with the interval value `dx` equal to `(Xmax-Xmin)/N` and is thus automatically scaled appropriately. For a coordinate list, the independent (first field) values are first checked to be sure

that they are ordered and equally spaced, then the operation is performed on the dependent (second field) values using the previously determined spacing as the value for `dx`.

If an unmapped array is used, but it is known otherwise that the interval between points is some value `dx`, then the correct value for the operation can be obtained by dividing the result of `deriv(x,order)` by the known `dx` value.

When the values represent noisy data, the result of `deriv(x,order)` can often be improved (in a pragmatic sense) by first filtering or smoothing the data. Smoothing removes much of the unwanted noise which is superimposed on the underlying values. See the `polySmooth(x)` function and related smoothing functions for examples.

See Also

`derivAt(x,order,pt)`.

derivAt(x,order,pt)

An object used to calculate the derivative of order `order` at value `pt` of the input data using a sliding fourth-order (five-point) polynomial.

Use

Use `derivAt(x,order,pt)` to calculate the numerical approximation of the derivative of order `order` at some point `pt` in the set of ordered equally spaced values (`x`). The result is a scalar value of type Real, except for Coord, which returns Coord.

Location

`AdvMath` ⟹ `Calculus` ⟹ `derivAt(x,order,pt)`

Example

`derivAt([ 0 1 2 3 3 2 1 0 ], 1, 3)` returns `0.5833`.

Notes

The `derivAt(x,order,pt)` function is applicable to one-dimensional arrays of simple numeric values (Int32, Real, Waveform) and lists of two-dimensional (two field) coordinates that represent equally spaced ordered data. For unmapped arrays, the value `a` represents an index in the array `(0 to N-1)`. For coordinate lists, the values `a` and `b` represent real values between the x-values of the first and last point in the coordinate list (inclusive). For mapped arrays, the values `a` and `b` represent real values between the `Xmin` mapping and `Xmax-dx` (inclusive), where `dx` is the interval between consecutive points: `(Xmax-Xmin)/N`.

For unmapped arrays, it is assumed that the data is equally spaced and ordered, and a value of `1` is assumed for the interval `dx` between points. For mapped arrays, the operation is performed with the interval value `dx` equal to `(Xmax-Xmin)/N` and is thus automatically scaled appropriately. For a coordinate list, the independent (first field) values are first checked to be sure that they are ordered and equally spaced. Then the operation is performed on

the dependent (second field) values using the previously determined spacing as the value for `dx`.

If an unmapped array is used, but it is known otherwise that the interval between points is some value `dx`, then the correct value for the operation can be obtained by dividing the result of `derivAt(x,order,pt)` by the known `dx` value.

When the values represent noisy data, the result of `derivAt(x,order,pt)` can often be improved (in a pragmatic sense) by first filtering or smoothing the data. Smoothing remove much of the unwanted noise which is superimposed on the underlying values. See the `polySmooth(x)` function and related smoothing functions for examples.

See Also

`deriv(x,order)`.

det(x)

An object used to calculate the determinant of a square matrix.

Use

Use `det(x)` to calculate the determinant of the square matrix `x`. The `x` input must be a square matrix shape and of the type Int32, Real, Complex or PComplex. For `x` input of all types, the same output type is returned and is Scalar in shape, except Int32, which returns a Real. A square matrix has the same number of rows as columns.

Location

`AdvMath` $\Longrightarrow$ `Matrix` $\Longrightarrow$ `det(x)`

Example

Where `x` is a square matrix `[ [1 2] [3 4] ]`, `det(x)` returns `-2`.

Notes

Mappings on the operand are ignored.

See Also

`identity(x)`, `inverse(x)`, and `transpose(x)`.

div (truncated division)

An object that performs an arithmetic div (truncated division) on two operands.

Use

Use `div` to determine the number of times one container divides evenly into another container, that is, the truncated division value. The two containers may be of type Int32, Real, Coord, or Waveform. The two containers may be of any shape. But if one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a container of the highest type, with the same shape as the operands.

If both operands are of type Coord, they must have their independent variable(s) match exactly or an error is returned. The div is only performed on the dependent (last) variable.

Location

`Math` ⟹ `+ - * /` ⟹ `div`

Example

A scalar div an array: `3.2 DIV [1 2 3 4]` returns `[3 1 1 0]`.

An array div a scalar: `[1 2 3] DIV 2` returns `[0 1 1]`.

Two scalars: `12.95 DIV 4` returns `3`.

Two Coord scalars: `coord(1,3) DIV coord(1,5)` returns `coord(1,0)`.

Two Coord scalars: `coord(1,3) DIV coord(2,5)` returns an error.

Notes

If either of the containers is mapped (that is, of type Waveform, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings. The return value will have the same mappings as the operands; otherwise an error is returned.

See Also

`+ (add)`, `/ (divide)`, `* (multiply)`, and `- (subtract)`.

/ (divide)

An object that performs an arithmetic division of two operands.

Use

Use / to divide the value of one container by the value of another container. The two containers may be of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The two containers may be of any shape. But if one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a container of the highest type, with the same shape as the operands.

If both operands are of type Coord, they must have their independent variable(s) match exactly or an error is returned. The division is only performed on the dependent (last) variable.

This division operation performs a parallel division on all arrays, including matrices. For a matrix divide, see the function `matDivide(numer,denom)`.

Location

`Math` ⟹ `+ - * / ` ⟹ `/`

Example

A scalar divided by an array: `3.0 / [1 2 3]` returns `[3 1.5 1]`.

Two arrays: `[4 5 6] / [1 2 3]` returns `[4 2.5 2]`.

Two PComplex scalars: `(1,@90) / (2,@30)` returns `(0.5, @60)` with Trig Mode set to Degrees.

Two Coord scalars: `coord(1,3) / coord(1,5)` returns `coord(1,0.6)`.

Two Coord scalars: `coord(1,3) / coord(2,5)` returns an error.

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings. The return value will have the same mappings as the operands; otherwise an error is returned.

See Also

`+ (add)`, `div (truncated division)`, `matDivide(numer,denom)`, `* (multiply)`, and `- (subtract)`.

dmyToDate(d,m,y)

An object that returns the value of the parameters `d`, `m`, and `y` converted to a date.

Use

Use `dmyToDate(d,m,y)` to convert three values, day, month, and year, into the date. `d`, `m`, and `y` can be any shape. If one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a container of the highest type, with the same shape as the operands. `d`, `m`, and `y` can be of type Int32, Real, Coord, or Waveform. Int32 arguments will return a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Time & Date` ⟹ `dmyToDate(d,m,y)`

Example

`dmyToDate(25, 12, 1991)` returns `62.8293999G`.

Notes

If any of the input parameters are mapped, then those mappings must be the same. The resultant container retains the mappings of the input container. If only one of the inputs is mapped, the resultant container has those mappings.

Also note that `dmy` is specified as modern Gregorian dates.

See Also

`hmsToHour(h,m,s)`, `hmsToSec(h,m,s)`, `mday(aDate)`, `month(aDate)`, `now()`, `Time & Date`, `wday(aDate)`, and `year(aDate)`.

== (equal to)

An object that performs an *is equal to* operation on two operands.

Use

Use `==` to determine whether the value(s) of one container is equal to the value(s) of another container. The two containers may be of any type and of any shape. If one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a scalar Int32 with the value `0` or `1`. If the first operand is equal to the second, the value of the result is `1`; otherwise the value is `0`.

If both operands are of type Coord, they must have all their independent variable(s) and dependent variables match exactly for the the result to be `1`. If independent variables do not match, an error is returned. Complex, PComplex, and Spectrum containers must have both parts match for the operation to return `1`. Enums are converted to Text for the comparison.

Arrays must have all the respective values of both containers equal for the operation to return `1`.

Location

`Math` ⟹ `Relational` ⟹ `==`

Example

A scalar and an array: `3.0 == [3 3 3]` returns `1`.

A scalar and an array: `3.0 == [3 1 3]` returns `0`.

Two arrays: `[1 2 3] == [1 2 4]` returns `0`.

Two PComplex scalars: `(1,@90) == (1,@85)` returns `0`.

Two Complex scalars: `(2,3) == (2,4)` returns `0`.

Two Complex scalars: `(2,3) == (2,3)` returns `1`.

Two Coord scalars: `coord(1,3) == coord(1,5)` returns `0`.

Two Coord scalars: `coord(1,3) == coord(2,3)` returns `Values for independent variables must match`.

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings; otherwise an error is returned.

See Also

`~= (almost equal to)`, `AND`, `> (greater than)`, `>= (greater than or equal to)`, `< (less than)`, `<= (less than or equal to)`, `NOT`, `!= (not equal to)`, `OR`, `Relational`, and `XOR`.

`Comparator`, `Conditional`, and `If/Then/Else` in the "General Reference" chapter.

erf(x)

An object used to calculate the error function of the input `x`.

Use

Use `erf(x)` to calculate the error function of `x`. The `erf` function is defined as:

`(2/SQRT(PI) ) * [ the integral of ( e^(-t^2)) * dt ]`

where the integral is evaluated over the range from zero to `x`.

The `x` input may be of any shape and size and of the type Int32, Real, Coord, or Waveform. For `x` input of all types the same output type is returned, except for Int32 which returns a Real type.

Location

`AdvMath` $\Longrightarrow$ `Probability` $\Longrightarrow$ `erf(x)`

Example

`erf(.5)` returns `0.52049987781305`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`erfc(x)`.

erfc(x)

An object used to calculate the complementary error function of the input `x`.

Use

Use `erfc(x)` to calculate the complementary error function of `x`. The erfc function is defined as:

`1-(2/SQRT(PI)) * [the integral of ( e^(-t^2)) * dt]`

where the integral is evaluated over the range from zero to `x`.

The `x` input may be of any shape and size and of the type Int32, Real, Coord, or Waveform. For `x` input of all types the same output type is returned, except for Int32 which returns a Real type.

Location

`AdvMath` ⟹ `Probability` ⟹ `erfc(x)`

Example

`erfc(.5)` returns `0.47950012218695`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

The `erfc(x)` function, which returns `1.0 - erf(x)`, is provided because of the extreme loss of relative accuracy if a large `x` is called for in `erf(x)`, the result is subtracted from `1.0` (for example, `x=5`, twelve places are lost).

See Also

`erf(x)`.

exp(x)

An object that returns the exponential of the value of `x`.

Use

Use `exp(x)` to generate the exponential, or natural antilogarithm (base e), of a number `x`; that is, `e` raised to the `x` power. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. Int32 arguments will return a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Power` ⟹ `exp(x)`

Example

`exp(10)` returns `22.0264657948k`.

`exp((1,2))` returns `(-1.131204384, 2.471726672)`.

Notes

Mappings are retained in the result.

See Also

`exp10(x)`, `^ (exponent)`, `log(x)`, `log10(x)`, and `Power`.

exp10(x)

An object that returns the exponential base 10 of the value of `x`.

Use

Use `exp10(x)` to generate the common antilogarithm (base 10) of a number `x`; that is, `10` raised to the `x` power. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. Int32 arguments will return a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Power` ⟹ `exp10(x)`

Example

`exp10(10)` returns `10G`.

`exp10((1, 2))` returns `(-4.16146837, 9.09297427)`.

Notes

Mappings are retained in the result.

See Also

`exp(x)`, `^ (exponent)`, `log(x)`, `log10(x)`, and `Power`.

^ (exponent)

An object that performs an arithmetic exponentiation of two operands.

Use

Use ^ to raise one number to the power of the other number. The two numbers may be of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The two containers may be of any shape. But if one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a container of the highest type, with the same shape as the operands.

If both operands are of type Coord, they must have their independent variable(s) match exactly or an error is returned. The exponentiation is only performed on the dependent (last) variable.

Location

`Math` ⟹ `+ - * /` ⟹ `^`

Example

A scalar raised to an array: `3.0 ^ [1 2 3]` returns `[3 9 27]`.

An array raised to a scalar: `[1 2 3] ^ 3` returns `[1 8 27]`.

Two arrays: `[4 5 6] ^ [1 2 3]` returns `[4 25 216]`.

Two Complex scalars: `(2,3) ^ (4,5)` returns `(-0.753046, -0.986429)`.

Two Coord scalars: `coord(4,3) ^ coord(4,2)` returns `coord(4,9)`.

Two Coord scalars: `coord(1,3) ^ coord(2,5)` returns an error.

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings. The return value will have the same mappings as the operands; otherwise an error is returned.

See Also

`exp(x)` and `log(x)`.

exponential regression

An object used to fit an exponential curve to `(x,y)` data.

Use

Use the exponential regression to fit the data to the equation `y = C0 + exp(C1*x)`, where `x` is the `x` coordinate and `C0` and `C1` are calculated coefficients. The exp function is the transcendental number `e` raised to the `(C1*x)` power.

The exponential regression object expects an array of Coord type of input with one independent variable, that is, an `(x,y)` pair. If the input array is not a Coord type, an attempt is made to convert it to Coord. If the input data is mapped (Waveform, Spectrum, or a mapped array) then the conversion to Coord uses the mapping information to create the `x` part of the `(x,y)` pairs. If the input data is not mapped (for example, an array), then the `x` part of the `(x,y)` pair is implicitly generated from its position in the array.

Location

`AdvMath` $\Longrightarrow$ `Regression` $\Longrightarrow$ `exponential`

Open View Parameters

The `Fit Type` field on the open view is used to change the regression type to `linear, logarithmic, exponential, power curve` or `polynomial regression`. Clicking on the field will bring up a list of the different regression types.

Example

See `Regression` for a Coord conversion example.

Notes

See `Regression` for general notes on regression.

See Also

`linear regression`, `logarithmic regression`, `meanSmooth(x,numPts)`, `movingAvg(x,numPts)`, `polynomial regression`, `polySmooth(x)`, and `power curve regression`.

`Build Coord` in the "General Reference" chapter.

factorial(n)

An object used to calculate the factorial of the input number `n`.

Use

Use `factorial(n)` to calculate the factorial of the natural number `n`. The factorial is defined for any natural number `n` as `n! = (n) * (n-1) * (n-2) ... (3) * (2) * (1).`

The `n` input may be of any shape and size and of the type Int32, Real, Coord, or Waveform. For `n` input of all types the same output type is returned, except for Int32 which returns a Real type. The `n` input must not be less than zero.

The `factorial(n)` operation is only defined for integer operands so the input numbers, while of unique type, are converted to Int32 type before the calculation is done.

Location

`AdvMath` $\Longrightarrow$ `Probability` $\Longrightarrow$ `factorial(n)`

Example

`factorial(6)` returns `720` from the multiplication of `6*5*4*3*2*1`.

`factorial(6.7)` returns `720` from the multiplication of `6*5*4*3*2*1`. The rule about converting Real to Int32 forces `6.7` to `6`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`beta(x,y)`, `binomial(a,b)`, `comb(n,r)`, `gamma(x)`, and `perm(n,r)`.

fft(x)

An object to compute the Discrete Fourier Transform on the input data.

Use

Use `fft(x)` to calculate the Discrete Fourier Transform of the values in `x`. The result is a `Complex` array for `Int` and `Real` inputs, or a `Spectrum` for a `Waveform` input, and represents the positive half of the transformed values. The size of the resultant array will be `(N DIV 2)+1`, where `N` is the size of the input array.

The input `x` must represent an array of ordered, equally-spaced data. For non-Waveform inputs, the output will be an unmapped array of `Complex` values. For `Waveform` inputs, the output will be mapped from a minimum frequency of `0` to a maximum frequency of `(1+(N DIV 2))/(Tmax-Tmin)`, where `N` is the size of the input array, and `Tmin` and `Tmax` are the mappings of the input `Waveform`. Also note that the output will not be normalized.

Location

`AdvMath` ⟹ `Signal Processing` ⟹ `fft(x)`

Example

```
fft([0 1 2 3 4 5 6 7])
```

returns

```
[(28,0) (-4,9.657) (-4,4) (-4,1.657) (-4,0)]
```

Notes

1. Truncation and Periodicity: A basic assumption in the use of the discrete Fourier transform is that the given waveform `h(t)` is periodic. If this is not the case, then the waveform must be truncated and represented as a periodic function. For a band-limited periodic function `h(t)`, the waveform must be truncated at an integer multiple of the period; otherwise, the resulting discrete and continuous Fourier transform will differ considerably in that sidelobes, frequency leakage, and sharp discontinuities can appear. These difficulties are

often reduced through the use of windows. If the waveform is not band-limited, one generally must adjust (increase) the truncation interval and use windows `[0,L]` to reduce time-domain truncation errors. `L` is defined as the period of the waveform.

2. Aliasing: The problem of aliasing (frequency folding) occurs when the sampling interval `T` is chosen too large. One can expect a degradation in the performance of the discrete Fourier transform when the sampling rate is close to the `Nyquist sampling rate`. If the waveform is not periodic or not band-limited, increasing the truncation interval`[0,L]` and decreasing the sampling interval `T` can help to control truncation errors and aliasing.

A reference on the fft can be found in the book: E. Oran Brigham, *The Fast Fourier Transform and Its Applications*, Second Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988. ISBN #0-13-307505-2.

See Also

`convolve(a,b)` and `ifft(x)`.

floor(x)

An object that returns the rounded-down value of `x` to the nearest integer of a value.

Use

`x` may be any shape and of the types Int32, Real, Coord, or Waveform. The `floor(x)` function returns the largest integer (as the same type) less than or equal to `x`; that is, the value of `x` rounded to the next smaller integer, the integer nearest negative infinity.

Location

`Math` ⟹ `Real Parts` ⟹ `floor(x)`

Example

`floor([23.0 23.1 23.9 23.5 (-23.5)])` returns `[23 23 23 23 -24]`.

Notes

Mappings are retained in the result.

See Also

`abs(x)`, `ceil(x)`, `Complex Parts`, `fracPart(x)`, `intPart(x)`, `Real Parts`, and `round(x)`.

fracPart(x)

An object that returns the value of the fractional part of `x`.

Use

Use `fracPart(x)` to obtain the fractional part of a container. `x` may be any shape and of the type Int32, Real, Coord, or Waveform. The `fracPart(x)` function returns the fractional part of `x` with the same sign. It returns `x - intPart(x)`.

Location

`Math` ⟹ `Real Parts` ⟹ `fracPart(x)`

Example

`fracPart([23.0 23.1 23.9 23.5 (-23.5)])` returns `[0 0.1 0.9 0.5 -0.5]`.

Notes

Mappings are retained in the result.

See Also

`abs(x)`, `ceil(x)`, `Complex Parts`, `floor(x)`, `intPart(x)`, `Real Parts`, and `round(x)`.

Freq Distribution

A menu item.

Use

Use `Freq Distribution` to calculate the distribution of values over a range into a fixed number of sub-ranges.

- `magDist(x,from,thru,step)`
- `logMagDist(x,from,thru,logStep)`

Location

`AdvMath` ⟹ `Freq Distribution` ⟹

gamma(x)

An object used to calculate the gamma function of the input `x`.

Use

Use `gamma(x)` to calculate the gamma function of `x`. The gamma function is defined as:

`(the integral of [ t^(x-1) * e^(-t) * dt ])`

where the integral is evaluated over the range from 0 to infinity.

The `x` input may be of any shape and size and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, the same output type is returned, except for Int32 which returns a Real type.

Location

`AdvMath` $\Longrightarrow$ `Probability` $\Longrightarrow$ `gamma(x)`

Examples

`gamma(8)` returns `(8-1)! = 5040`.

`gamma(-1)` returns an error.

`gamma(3.7)` returns `= 4.17065178379660`.

`gamma(-3.7)` returns `= 0.251643995902422`.

Notes

The `gamma(x)` function is not defined for non-positive integral values of `x`. When `x` is a positive integer, `gamma(x)` is defined as `(x-1)!` (factorial).

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`beta(x,y)`, `comb(n,r)`, `factorial(x)`, and `perm(n,r)`.

Generate

A menu item.

Use

Use `Generate` to access the following objects which generate Real one-dimensional arrays of data with specific values.

- `ramp(numElem,from,thru)`
- `logRamp(numElem,from,thru)`
- `xramp(numElem,from,thru)`
- `xlogRamp(numElem,from,thru)`

Location

`Math` ⟹ `Generate` ⟹

> (greater than)

An object that performs an *is greater than* operation on two operands.

Use

Use `>` to determine whether the value(s) of one container is greater than the value(s) of another container. The two containers may be of type Int32, Real, Coord, Waveform, Text, or Enum. The two containers may be of any shape. But if one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a scalar Int32, with the value `0` or `1`. If the first operand is greater than the second, the value of the result is `1`; otherwise the value is `0`.

If both operands are of type Coord, their independent variable(s) must be identical before the operation is even attempted; if not, an error is returned. Enums are converted to Text for the comparison. Text is compared using ASCII lexical ordering.

When two arrays are compared to each other, each pair of elements must satisfy the relational operator for the operation to return `1`.

Location

`Math` ⟹ `Relational` ⟹ `>`

Example

A scalar and an array: `3.0 > [1 3 9]` returns `0`.

An array and a scalar: `[5 9 7] > 4` returns `1`.

Two arrays: `[3 4 7] > [2 3 5]` returns `1`.

Two arrays: `[3 4 3] > [2 3 5]` returns `0`.

Two Coord scalars: `coord(1,3) > coord(1,5)` returns `0`.

Two Coord scalars: `coord(1,3) > coord(2,5)` returns `Values for independent variables must match`.

Two Text scalars: `"too" > "zoo"` returns `0`.

Notes

If either of the containers is mapped (that is, of type Waveform, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings; otherwise an error is returned.

See Also

`~= (almost equal to)`, `AND`, `== (equal to)`, `>= (greater than or equal to)`, `< (less than)`, `<= (less than or equal to)`, `NOT`, `!= (not equal to)`, `OR`, `Relational`, and `XOR`.

`Comparator`, `Conditional`, and `If/Then/Else` in the "General Reference" chapter.

>= (greater than or equal to)

An object that performs an *is greater than or equal to* operation on two operands.

Use

Use `>=` to determine whether the value(s) of one container is greater than or equal to the value(s) of another container. The two containers may be of type Int32, Real, Coord, Waveform, Text, or Enum. The two containers may be of any shape. But if one of the containers is an array, the other must be either be a scalar or an array of the same size and shape. The result is a scalar Int32, with the value `0` or `1`. If the first operand is greater than or equal to the second, the value of the result is `1`; otherwise the value is `0`.

If both operands are of type Coord, their independent variable(s) must be identical before the operation is even attempted; if not, an error is returned. Enums are converted to Text for the comparison. Text is compared using ASCII lexical ordering.

When two arrays are mapped against each other, each set of elements must satisfy the relational operator for the operation to return `1`.

Location

`Math` ⟹ `Relational` ⟹ `>=`

Example

A scalar and an array: `3.0 >= [1 3 9]` returns `0`.

An array and a scalar: `[4 5 6] >= 4` returns `1`.

Two arrays: `[3 3 6] >= [2 3 5]` returns `1`.

Two arrays: `[1 4 3] >= [2 3 5]` returns `0`.

Two Coord scalars: `coord(1,3) >= coord(1,5)` returns `0`.

Two Coord scalars: `coord(1,3) >= coord(2,5)` returns `Values for independent variables must match`.

Two Text scalars: `"too" >= "zoo"` returns `0`.

Notes

If either of the containers is mapped (that is, of type Waveform, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings; otherwise an error is returned.

See Also

`~= (almost equal to)`, `AND`, `== (equal to)`, `> (greater than)`, `< (less than)`, `<= (less than or equal to)`, `NOT`, `!= (not equal to)`, `OR`, `Relational`, and `XOR`.

`Comparator`, `Conditional`, and `If/Then/Else` in the "General Reference" chapter.

hamming(x)

An object used to apply a Hamming window to a time series of values.

Use

Use `hamming(x)` to filter the values in `x` in the same manner as convolving `x` with the spectral transform of the Hamming function. This has the effect of suppressing some of the noise due to the tails of the input sequence and the potential discontinuities they represent when sampling periodic signals.

The input `x` must be an Array 1D of type Int32, Real, Coord, or a Waveform, or a Spectrum. The same type is returned, except for Int32, which returns Real.

If `x` is a Spectrum, it is first converted to a Waveform using an `ifft(x)` before the window is applied. The result of the window is then converted back to type Spectrum using an `fft(x)`. A Spectrum is returned.

Location

`AdvMath` ⟹ `Signal Processing` ⟹ `hamming(x)`

Example

`hamming([1 1 1 1 1 1 1 1])` returns
`[0.115 0.364 0.716 0.965 0.965 0.716 0.364 0.115]`.

Notes

The `Hamming` function is represented in the time domain as `0.54-0.46*cos(2*PI*(n+0.5)/N)`, where `n` is the position (index) in the array, and `N` is the size of the array. The result will be an array of the same type as `x` and will have the same mappings as `x` (if any).

For a discussion of sidelobe levels and coherent gains, see: Ziemer, Tranter, and Fannin, *Signals and Systems*, Macmillan Publishing, New York, NY, 1983. ISBN #0-02-431650-4.

See Also

`bartlet(x)`, `blackman(x)`, `convolve(a,b)`, `fft(x)`, `hanning(x)`, `ifft(x)`, and `rect(x)`.

hanning(x)

An object used to apply a Hanning window to a time series of values.

Use

Use `hanning(x)` to filter the values in `x` in the same manner as convolving `x` with the spectral transform of the Hanning function. This has the effect of suppressing some of the noise due to the tails of the input sequence and the potential discontinuities they represent when sampling periodic signals.

The input `x` must be an Array 1D of type Int32, Real, Coord, or a Waveform, or a Spectrum. The same type is returned, except for Int32, which returns Real.

If `x` is a Spectrum, it is first converted to a Waveform using an `ifft(x)` before the window is applied. The result of the window is then converted back to type Spectrum using an `fft(x)`. A Spectrum is returned.

Location

`AdvMath` ⟹ `Signal Processing` ⟹ `hanning(x)`

Example

`hanning([1 1 1 1 1 1 1 1])` returns
`[0.038 0.309 0.691 0.962 0.962 0.691 0.309 0.038]`.

Notes

The `Hanning` function is represented in the time domain as `0.5*(1-cos(2*PI*(n+0.5)/N))`, where `n` is the position (index) in the array, and `N` is the size of the array. The result will be an array of the same type as `x` and will have the same mappings as `x` (if any).

For a discussion of sidelobe levels and coherent gains, see: Ziemer, Tranter, and Fannin, *Signals and Systems*, Macmillan Publishing, New York, NY, 1983. ISBN #0-02-431650-4.

See Also

`bartlet(x)`, `blackman(x)`, `convolve(a,b)`, `fft(x)`, `hamming(x)`, `ifft(x)`, and `rect(x)`.

hmsToHour(h,m,s)

An object that returns the value of the parameters `h`, `m`, and `s` converted to hours.

Use

Use `hmsToHour(h,m,s)` to convert three values, hours, minutes, and seconds, into hours. `h`, `m`, and `s` can be any shape. If one of the containers is an array, the others must be either scalar or arrays of the same size and shape. The result is a container of the highest type, with the same shape as the operands. `h`, `m`, and `s` can be of type Int32, Real, Coord, or Waveform. Int32 arguments will return a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` $\Longrightarrow$ `Time & Date` $\Longrightarrow$ `hmsToHour(h,m,s)`

Example

`hmsToHour(1, [0 12 60], 36)` returns `[1.01 1.21 2.01]`.

Notes

If any of the input parameters are mapped, then those mappings must be the same. The resultant container retains the mappings of the input container. If only one of the inputs is mapped, the resultant container has those mappings.

See Also

`dmyToDate(d,m,y)`, `hmsToSec(h,m,s)`, `mday(aDate)`, `month(aDate)`, `now()`, `Time & Date`, `wday(aDate)`, and `year(aDate)`.

hmsToSec(h,m,s)

An object that returns the value of the parameters `h`, `m`, and `s` converted to seconds.

Use

Use `hmsToSec(h,m,s)` to convert three values, hours, minutes, and seconds, into seconds. `h`, `m`, and `s` can be any shape. If one of the containers is an array, the others must be either scalar or arrays of the same size and shape. The result is a container of the highest type, with the same shape as the operands. `h`, `m`, and `s` can be of type Int32, Real, Coord, or Waveform. Int32 arguments will return a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Time & Date` ⟹ `hmsToSec(h,m,s)`

Example

`hmsToSec(1,[2 4 6],3)` returns `[3723, 3843, 3963]`.

Notes

If any of the input parameters are mapped, then those mappings must be the same. The resultant container retains the mappings of the input container. If only one of the inputs is mapped, the resultant container has those mappings.

See Also

`dmyToDate(d,m,y)`, `hmsToHour(h,m,s)`, `mday(aDate)`, `month(aDate)`, `now()`, `Time & Date`, `wday(aDate)`, and `year(aDate)`.

Hyper Bessel

A menu item.

Use

Use `Hyper Bessel` to calculate the modified (hyperbolic) bessel function of the input data.

- `i0(x)`
- `i1(x)`
- `k0(x)`
- `k1(x)`

Location

`AdvMath` $\Longrightarrow$ `Hyper Bessel` $\Longrightarrow$

See Also

`Bessel`.

Hyper Trig

A menu item.

Use

Use `Hyper Trig` to access the following objects which perform hyperbolic trigonometric functions on data.

- `sinh(x)`
- `cosh(x)`
- `tanh(x)`
- `coth(x)`
- `asinh(x)`
- `acosh(x)`
- `atanh(x)`
- `acoth(x)`

Location

`Math` ⟹ `Hyper Trig` ⟹

See Also

`Trig`.

`Trig Mode` in the "General Reference" chapter.

i0(x)

An object used to calculate the modified (hyperbolic) Bessel function of `x` of the first kind of order zero.

Use

Use `i0(x)` to find the modified (hyperbolic) Bessel function of `x` of the first kind of order zero. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For all `x` input types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` $\Longrightarrow$ `Hyper Bessel` $\Longrightarrow$ `i0(x)`

Example

`i0(1)` returns `1.26606587775200`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`i1(x)`, `k0(x)`, and `k1(x)`.

i1(x)

An object used to calculate the modified (hyperbolic) Bessel function of `x` of the first kind of order one.

Use

Use `i1(x)` to find the modified (hyperbolic) Bessel function of `x` of the first kind of order one. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For all `x` input types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` ⟹ `Hyper Bessel` ⟹ `i1(x)`

Example

`i1(1)` returns `0.565159103992485`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`i0(x)`, `k0(x)`, and `k1(x)`.

identity(x)

An object used to return the identity matrix.

Use

Use `identity(x)` to return the identity matrix that is the same size as the square input matrix. The `x` input must be a square matrix and of the type Int32, Real, Complex or PComplex. For all `x` input types the same output type is returned. A square matrix has the same number of rows as columns.

Location

`AdvMath` $\Longrightarrow$ `Matrix` $\Longrightarrow$ `identity(x)`

Example

Where `x` is a square matrix `[ [1 2] [3 4] ]`, `identity(x)` returns `[ [1 0] [0 1] ]`.

Where `x` is the complex square matrix `[ [(1,1) (2,2)] [(3,3) (4,4)] ]`, `identity(x)` returns `[ [(1,0) (0,0)] [(0,0) (1,0)] ]`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`det(x)`, `inverse(x)`, and `transpose(x)`.

ifft(x)

An object to compute the Inverse Discrete Fourier Transform on the input data.

Use

Use `ifft(x)` to calculate the Inverse Discrete Fourier Transform of the values in `x`. The input `x` should be the positive half of a symmetric Complex, PComplex, or Spectral sequence. The result is a `Real` array for `Complex` or `PComplex` inputs, or a `Waveform`, for a `Spectrum` input. The size of the resultant array will be `2*(N-1)`, where `N` is the size of the input array (which should be an odd-count sequence).

For non-`Spectrum` inputs, the output will be an unmapped array of `Real` values. For `Spectrum` inputs, the mapping must be linear (not log), and `Fmin` must be `0`. The output will be mapped from a minimum time of `0` to a maximum time of `N/(Fmax-Fmin)`, where `N` is the size of the input array, and `Fmin` and `Fmax` are the mappings of the input `Spectrum`. Also note that the output will not be normalized.

It is assumed that the values of `x` represent data that has not been normalized.

Location

`AdvMath` ⟹ `Signal Processing` ⟹ `ifft(x)`

Example

`ifft([(3.5,0) (-1,2.414) (-1,1) (-1,0.414) (-1,0)])` returns `[0 1 2 3 4 5 6 7]`.

Notes

1. Truncation and Periodicity: A basic assumption in the use of the discrete Fourier transform is that the given waveform `h(t)` is periodic. If this is not the case, then the waveform must be truncated and represented as a periodic function. For a band-limited periodic function `h(t)`, the waveform must be truncated at an integer multiple of the period; otherwise, the resulting discrete and continuous Fourier transform will differ considerably since sidelobes,

frequency leakage, and sharp discontinuities can appear. These difficulties are often reduced through the use of windows. If the waveform is not band-limited, you generally must increase the truncation interval and use windows `[0,L]` to reduce time-domain truncation errors. `L` is defined as the period of the waveform.

2. Aliasing: The problem of aliasing (frequency folding) occurs when the sampling interval `T` is chosen too large. One can expect a degradation in the performance of the discrete Fourier transform when the sampling rate is close to the `Nyquist sampling rate`. If the waveform is not periodic or not band-limited, increasing the truncation interval `[0,L]` and decreasing the sampling interval `T` can help to control truncation errors and aliasing.

A reference on the ifft can be found in the book: E. Oran Brigham, *The Fast Fourier Transform and Its Applications*, Second Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988. ISBN #0-13-307505-2.

See Also

`fft(x)`.

im(x)

An object that returns the imaginary part of a Complex number `x`.

Use

Use `im(x)` to extract the imaginary part of a Complex number `x`. `x` can be any shape and of types Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For types Int32, Real, Coord, and Waveform, the same type with value(s) of zero is returned. For types Complex, PComplex, and Spectrum, `im(x)` returns a Real container with the value of the imaginary part of the Complex number. (PComplex is first converted to Complex.)

Location

`Math` ⟹ `Complex Parts` ⟹ `im(x)`

Example

`im( (1,2) )` returns 2.

`im( (1,@90) )`, with `Trig Mode` in Degrees, returns 1.

Notes

Mappings are retained in the result.

See Also

`conj(x)`, `j(x)`, `mag(x)`, `phase(x)`, `re(x)`, and `Real Parts`.

`Build Complex` and `UnBuild Complex` in the "General Reference" chapter.

init(x,val)

An object used to initialize an array to a specific value.

Use

Use `init(x,val)` to initialize an input array `x` to a value of `val`. The `x` input may be of any size and shape and of the type Int32, Real, Coord, Waveform, Complex, PComplex, Spectrum, or Text. For all `x` input types, the same output type is returned. The second `val` parameter must be of the same type or be able to be converted to the same type as the `x` input value.

If the input `x` is an array then the `val` init value parameter can be a Scalar value or an array. If `val` is an array, it must have the same number of dimensions and size as the `x` input. That is, if `x` is an Array 1D then `val` can either be a Scalar or an Array 1D of the same size.

Location

`AdvMath` $\Longrightarrow$ `Array` $\Longrightarrow$ `init(x,val)`

Example

`init(a,5)` initializes the input array to a value of 5.

`init(a,[1 2 3])` initializes an input array of three elements to the values `[1 2 3]`.

Notes

The init value must be the same type as the input array or be able to be converted to the same type. That is, you cannot initialize a real array to a complex initial value because a complex number cannot be converted to a real number.

The mapping on the output container is the same as the input `val`.

See Also

`ramp(numElem,from,thru)` and `randomize(x,low,high)`.

`Alloc Array` in the "General Reference" chapter.

integral(x)

An object used to calculate the integral across all values of the input data using Simpson's 1/3 rule.

Use

Use `integral(x)` to calculate the numerical approximation of the indefinite integral of a set of ordered, equally-spaced values (`x`). The result is an array of the same size and shape as the input `x`. The result is the same type as `x` except for Int32. Int32 returns a Real and has the same mappings (if any) as the input `x`.

Location

`AdvMath` ⟹ `Calculus` ⟹ `integral(x)`

Example

`integral([ 0 1 2 3 3 2 1 0 ])` returns
`[ 0 0.5 2 4.5 7.667 10.17 11.67 12.17 ]`.

Notes

The `integral(x)` function is applicable to one-dimensional arrays of simple numeric values (Int32, Real, Waveform) and lists of two-dimensional (two field) coordinates that represent equally spaced ordered data. For mapped arrays, the mappings of the resultant array will be identical to those of `x`. For coordinate lists, the values of the independent field (first field) of each coordinate will be identical to those of the corresponding values in `x`.

For unmapped arrays, it is assumed that the data is equally spaced and ordered, and a value of `1` is assumed for the interval `dx` between points. For mapped arrays, the operation is performed with the interval value `dx` equal to `(Xmax-Xmin)/N` and is thus automatically scaled appropriately. For a coordinate list, the independent (first field) values are first checked to be sure that they are ordered and equally spaced, then the operation is performed on the dependent (second field) values using the previously determined spacing as the value for `dx`.

If an unmapped array is used, but it is known otherwise that the interval between points is some value `dx`, then the correct value for the operation can be obtained by multiplying the result of `integral(x)` by the known `dx` value.

When the values represent noisy data, the result of `integral(x)` can often be improved (in a pragmatic sense) by first filtering or smoothing the data. Smoothing removes much of the unwanted noise which is superimposed on the underlying values. See the `polySmooth(x)` function and related smoothing functions for examples.

See Also

`defIntegral(x,a,b)`.

intPart(x)

An object that returns the value of the integer part of `x`.

Use

Use `intPart(x)` to obtain the integer part of a container. `x` may be any shape and of the type Int32, Real, Coord, or Waveform. The `intPart(x)` function returns the integer portion of the data. If the data is less than zero, it returns the `ceil(x)`; otherwise it returns the `floor(x)`. The result will have the same mappings as `x`.

Location

`Math` ⟹ `Real Parts` ⟹ `intPart(x)`

Example

`intPart([23.0 23.1 23.9 23.5 (-23.5)])` returns `[23 23 23 23 -23]`.

Notes

Mappings are retained in the result.

See Also

`abs(x)`, `ceil(x)`, `Complex Parts`, `floor(x)`, `fracPart(x)`, `Real Parts`, and `round(x)`.

inverse(x)

An object used to calculate the inverse of a square matrix.

Use

Use `inverse(x)` to calculate the inverse of the square matrix `x`. The `x` input must be a square matrix shape and of the type Int32, Real, Complex or PComplex. For `x` input of all types, the same output type and shape is returned, except for Int32, which returns a Real. A square matrix has the same number of rows as columns.

Location

`AdvMath` ⟹ `Matrix` ⟹ `inverse(x)`

Example

Where `x` is a square matrix `[ [1 2] [3 4] ]`, `inverse(x)` returns `[ [-2 1] [1.5 -0.5] ]`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`det(x)`, `identity(x)`, and `transpose(x)`.

j(x)

An object that returns a Complex number with `x` as the imaginary part.

Use

Use `j(x)` to generate a Complex number of the same shape as `x`, with `x` as the imaginary part and `0` as the real part. `x` may be any shape and of the type Int32, Real, or Waveform.

Location

`Math` ⟹ `Complex Parts` ⟹ `j(x)`

Example

`j(5)` returns the Complex number `(0,5)`.

Notes

Mappings are retained in the result. To return a complex number with `x` as the real part, use the type promotions on input terminals to promote Real to Complex.

See Also

`Complex Parts`, `conj(x)`, `im(x)`, `mag(x)`, `phase(x)`, `re(x)`, and `Real Parts`.

`Build Complex` and `UnBuild Complex` in the "General Reference" chapter.

j0(x)

An object used to calculate the Bessel function of `x` of the first kind of order zero.

Use

Use `j0(x)` to find the Bessel function of `x` of the first kind of order zero. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` $\Longrightarrow$ `Bessel` $\Longrightarrow$ `j0(x)`

Example

`j0(10)` returns `-0.245935764451348`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`Ai(x)`, `Bi(x)`, `j1(x)`, `jn(x,n)`, `y0(x)`, `y1(x)`, and `yn(x,n)`.

j1(x)

An object used to calculate the Bessel function of `x` of the first kind of order one.

Use

Use `j1(x)` to find the Bessel function of `x` of the first kind of order one. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` $\Longrightarrow$ `Bessel` $\Longrightarrow$ `j1(x)`

Example

`j1(1)` returns `0.440050585744933`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

See Also

`Ai(x)`, `Bi(x)`, `j0(x)`, `jn(x,n)`, `y0(x)`, `y1(x)`, and `yn(x,n)`.

jn(x,n)

An object used to calculate the Bessel function of `x` of the first kind of order `n`.

Use

Use `jn(x,n)` to find the Bessel function of `x` of the first kind of order `n`. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

The `n` parameter must be of Int32 type or be able to be converted to Int32. The `n` parameter also has to be a Scalar in shape or the same shape as `x`.

Location

`AdvMath` ⟹ `Bessel` ⟹ `jn(x,n)`

Example

`jn(1,2)` returns `0.114903484931900`.

Notes

If both of the inputs are mapped, then the mappings must be the same. The return value has the same mappings as the input if either input is mapped. If neither of the inputs is mapped, then the output is unmapped.

See Also

`Ai(x)`, `Bi(x)`, `j0(x)`, `j1(x)`, `y0(x)`, `y1(x)`, and `yn(x,n)`.

k0(x)

An object used to calculate the modified (hyperbolic) Bessel function of `x` of the second kind of order zero.

Use

Use `k0(x)` to find the modified (hyperbolic) Bessel function of `x` of the second kind of order zero. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For all `x` input types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` $\Longrightarrow$ `Hyper Bessel` $\Longrightarrow$ `k0(x)`

Example

`k0(1)` returns `0.421024438240708`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

The `x` argument must be positive.

See Also

`i0(x)`, `i1(x)`, and `k1(x)`.

k1(x)

An object used to calculate the modified (hyperbolic) Bessel function of `x` of the second kind of order one.

Use

Use `k1(x)` to find the modified (hyperbolic) Bessel function of `x` of the second kind of order one. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For all `x` input types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` ⟹ `Hyper Bessel` ⟹ `k1(x)`

Example

`k1(1)` returns `0.601907230197234`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

The `x` argument must be positive.

See Also

`i0(x)`, `i1(x)`, and `k0(x)`.

< (less than)

An object that performs an *is less than* operation on two operands.

Use

Use `<` to determine whether the value(s) of one container is less than the value(s) of another container. The two containers may be of type Int32, Real, Coord, Waveform, Text, or Enum. The two containers may be of any shape. But if one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a scalar Int32, with the value `0` or `1`. If the first operand is less than the second, the value of the result is `1`; otherwise the value is `0`.

If both operands are of type Coord, their independent variable(s) must be identical before the operation is even attempted; if not, an error is returned. Enums are converted to Text for the comparison. Text is compared using ASCII lexical ordering.

When two arrays are compared to each other, each pair of elements must satisfy the relational operator for the operation to return `1`.

Location

`Math` ⟹ `Relational` ⟹ `<`

Example

A scalar and an array: `3.0 < [1 3 9]` returns `0`.

An array and a scalar: `[1 2 3] < 4` returns `1`.

Two arrays: `[1 2 3] < [2 3 5]` returns `1`.

Two arrays: `[1 4 3] < [2 3 5]` returns `0`.

Two Coord scalars: `coord(1,3) < coord(1,5)` returns `1`.

Two Coord scalars: `coord(1,3) < coord(2,5)` returns `Values for independent variables must match`.

Two Text scalars: `"too" < "zoo"` returns `1`.

Notes

If either of the containers is mapped (that is, of type Waveform, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings; otherwise an error is returned.

See Also

`~= (almost equal to)`, `AND`, `== (equal to)`, `> (greater than)`, `>= (greater than or equal to)`, `<= (less than or equal to)`, `NOT`, `!= (not equal to)`, `OR`, `Relational`, and `XOR`.

`Comparator`, `Conditional`, and `If/Then/Else` in the "General Reference" chapter.

<= (less than or equal to)

An object that performs an *is less than or equal to* operation on two operands.

Use

Use `<=` to determine whether the value(s) of one container is less than or equal to the value(s) of another container. The two containers may be of type Int32, Real, Coord, Waveform, Text, or Enum. The two containers may be of any shape. But if one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a scalar Int32, with the value `0` or `1`. If the first operand is less than or equal to the second, the value of the result is `1`; otherwise the value is `0`.

If both operands are of type Coord, their independent variable(s) must be identical before the operation is even attempted; if not, an error is returned. Enums are converted to Text for the comparison. Text is compared using ASCII lexical ordering.

When two arrays are mapped against each other, each set of elements must satisfy the relational operator for the operation to return `1`.

Location

`Math` ⟹ `Relational` ⟹ `<=`

Example

A scalar and an array: `3.0 <= [1 3 9]` returns `0`.

An array and a scalar: `[1 2 4] <= 4` returns `1`.

Two arrays: `[1 2 3] <= [2 3 3]` returns `1`.

Two arrays: `[1 4 3] <= [2 3 5]` returns `0`.

Two Coord scalars: `coord(1,3) <= coord(1,5)` returns `1`.

Two Coord scalars: `coord(1,3) <= coord(2,5)` returns `Values for independent variables must match`.

Two Text scalars: `"too" <= "zoo"` returns `1`.

Notes

If either of the containers is mapped (that is, of type Waveform, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings; otherwise an error is returned.

See Also

`~= (almost equal to)`, `AND`, `== (equal to)`, `> (greater than)`, `>= (greater than or equal to)`, `< (less than)`, `NOT`, `!= (not equal to)`, `OR`, `Relational`, and `XOR`.

`Comparator`, `Conditional`, and `If/Then/Else` in the "General Reference" chapter.

linear regression

An object used to fit a linear regression line to `(x,y)` data.

Use

Use the linear regression to fit a straight line to the data using the equation `y = C0 + C1*x`, where `x` is the `x` coordinate and `C0` and `C1` are calculated coefficients. This type of regression should be thought of as fitting the best straight line through the data.

The linear regression object expects an array of Coord type of input with one independent variable, that is, an `(x,y)` pair. If the input array is not a Coord type, an attempt is made to convert it to Coord. If the input data is mapped (Waveform, Spectrum, or a mapped array) then the conversion to Coord uses the mapping information to create the `x` part of the `(x,y)` pairs. If the input data is not mapped (for example, an array), then the `x` part of the `(x,y)` pair is implicitly generated from its position in the array.

Location

`AdvMath` $\Longrightarrow$ `Regression` $\Longrightarrow$ `linear`

Open View Parameters

The `Fit Type` field on the open view is used to change the regression type to `linear`, `logarithmic`, `exponential`, `power curve` or `polynomial regression`. Clicking on the field will bring up a list of the different regression types.

Example

See `Regression` for a Coord conversion example.

Notes

See `Regression` for general notes on regression.

See Also

`exponential regression`, `logarithmic regression`, `meanSmooth(x,numPts)`, `movingAvg(x,numPts)`, `polynomial regression`, `polySmooth(x)`, and `power curve regression`.

`Build Coord` in the "General Reference" chapter.

log(x)

An object that returns the natural logarithm of the value of `x`.

Use

Use `log(x)` to generate the natural logarithm (base e) of a number `x`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. Int32 arguments will return a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Power` ⟹ `log(x)`

Example

`log([10 2])` will return `[2.302585092994 0.693147180559945]`.

`log((1,2))` will return `(0.804718956, 1.10714872)`.

Notes

Mappings are retained in the result.

See Also

`exp(x)`, `exp10(x)`, `^ (exponent)`, `log10(x)`, and `Power`.

log10(x)

An object that returns the common logarithm of the value of `x`.

Use

Use `log10(x)` to generate the common logarithm (base 10) of a number `x`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. Int32 arguments will return a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Power` ⟹ `log10(x)`

Example

`log10([10 2])` will return `[1 0.301029995663981]`.

`log10((1,2))` will return `[0.349485, 0.480828579]`.

Notes

Mappings are retained in the result.

See Also

`exp(x)`, `exp10(x)`, `^ (exponent)`, `log(x)`, and `Power`.

logarithmic regression

An object used to fit a logarithmic curve to data.

Use

Use the logarithmic regression to fit the data to the equation:

```
y = C0 + C1*log(x)
```

where `x` is the `x` coordinate and `C0` and `C1` are calculated coefficients. The log function is the natural logarithm. The input data `(x)` must always be greater than zero.

The logarithmic regression object expects an array of Coord type of input with one independent variable, that is, an `(x,y)` pair. If the input array is not a Coord type, an attempt is made to convert it to Coord. If the input data is mapped (Spectrum or a mapped array) then the conversion to Coord uses the mapping information to create the `x` part of the `(x,y)` pairs. If the input data is not mapped (for example, an array), then the `x` part of the `(x,y)` pair is implicitly generated from its position in the array.

Location

`AdvMath` $\Longrightarrow$ `Regression` $\Longrightarrow$ `logarithmic`

Open View Parameters

The `Fit Type` field on the open view is used to change the regression type to `linear`, `logarithmic`, `exponential`, `power curve` or `polynomial regression`. Clicking on the field will bring up a list of the different regression types.

Example

See `Regression` for a Coord conversion example.

Notes

See `Regression` for general notes on regression.

This function will not work for unmapped arrays. Unmapped arrays are converted to Coord starting with `x=0`, which will fail in the log function. This function will not work with a Waveform for the same reason.

See Also

`exponential regression`, `meanSmooth(x,numPts)`, `movingAvg(x,numPts)`, `polynomial regression`, `polySmooth(x)`, and `power curve regression`.

`Build Coord` in the "General Reference" chapter.

Logical

A menu item.

Use

Use `Logical` to access the following objects which perform logical operations on operands:

- `AND`
- `OR`
- `XOR`
- `NOT`

Location

`Math` ⟹ `Logical` ⟹

Notes

If logic requirements are met, a `1` is returned. If logic requirements are not met, a `0` is returned.

Note that the return value is of type Int32 and is the same shape as the operands. This is different than the conditionals, such as `==`, that always return a scalar.

The logical operators are defined for type Text only in the sense of whether the string is null or not. That is, `"zoo"` AND `""` (null string) is logically false since the second string is null. Remember that when comparing a Text type to a non-string type, the latter is promoted to a Text type. This means that `"zoo"` AND `0` is true since the Real `0` is promoted to the string `"0"` and, since both strings are non-null, the AND expression is true and returns `1`.

See Also

`Relational`.

`Conditional` and `If/Then/Else` in the "General Reference" chapter.

logMagDist(x, from,thru,logStep)

An object used to calculate the distribution of numbers in the input container given a logarithmic distribution.

Use

Use `logMagDist(x,from,thru,logStep)` to calculate a distribution of the log10 values of the values in the input array. The distribution is done in the range of `from` to `thru` by a size of `logStep`. The `logStep` parameter is the sub-range size in the log10 domain into which the log values are sorted (for example, a `logStep` of `0.2` would be equivalent to 5 sub-ranges per decade).

The `x` input may be of any size and of the type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For Complex, PComplex, and Spectrum type inputs, the magnitude of the complex number is calculated first, then that magnitude is used in determining the frequency distribution. The `x` input shape must be an Array.

Since `log10(x)` is not defined for values of `x<=0`, the input values should be positive.

For input of all types, a Real Array 1D is returned. Each array element represents how many of the input value will fit in each distribution range.

The `from`, `thru` and `logStep` parameters must be Scalar in shape and of Real type or be able to be converted to Real.

The `from` parameter must be less than the `thru` parameter.

The `logStep` parameter must be greater than zero.

Location

`AdvMath` ⟹ `Freq Distribution` ⟹ `logMagDist(x,from,thru,logStep)`

Example

`logMagDist(x,1,100,1)`, where `x` is the array:

```
[1 2 3 10 40 50 100 100 100]
```

returns the array `[3 6]`. In this example there are 2 sub-ranges ranging from 1 to 2 by 1. The output array shows how many of the input numbers fall into each range.

Notes

The sub-ranges for the distribution are determined in the following manner. Note that the log10 of "`from`" and "`thru`" parameters is calculated before determining the range distribution.

Distribution of Numbers Accepted

Range	Range Minimum	range	Range Maximum
1.	"`from`" parm	`<= range1`	`<` "`from`" parm + "`step`" parm (`range1limit`)
2.	`range1limit`	`<= range2`	`<` `range1limit` + "`step`" parm (`range2limit`)
3.	`range2limit`	`<= range3`	`<` `range2limit` + "`step`" parm (`range3limit`)
...	...	...	...
n-1	`range(n-2)limit`	`<= range(n-1)`	`<=` "`thru`" parm

Mappings on the operands are ignored and the return value is not mapped.

See Also

`magDist(x,from,thru,step)`.

logRamp (numElem,from, thru)

An object that generates a logarithmically ramped array.

Use

Use `logRamp(numElem,from,thru)` to generate a Real one-dimensional array of length `numElem`, with the values logarithmically ramped from `from` to `thru`. `numElem` must be a scalar container which is, or can be converted to, Int32 and with a value greater than zero. `from` and `thru` must be scalar containers which are, or can be converted to, Real. If `from` is less than `thru` the ramping is positive; otherwise it automatically ramps negatively. Both `from` and `thru` must have values greater than zero.

Location

`Math` ⟹ `Generate` ⟹ `logRamp(numElem,from,thru)`

Example

`logRamp(3, 1, 100)` returns a 1D Array with values `[1 10 100]`.

Notes

The return value has no mappings.

The algorithm for generating values is:

```
Y[I]=exp10(log10(from)+I*((log10(thru)-log10(from))/numElem-1))
```

```
for I=0..numElem-1
```

This has the effect that the last element in the resultant array has the value `thru`.

See Also

`Alloc Real` and `ramp(numElem,from,thru)`.

mag(x)

An object that returns the magnitude of a PComplex number `x`.

Use

Use `mag(x)` to extract the magnitude of a PComplex number `x`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For types Int32, Real, Coord, and Waveform, the same type with the same value(s) is returned. For types Complex, PComplex, and Spectrum, `mag(x)` returns a Real container with the value of the magnitude of the PComplex number. (Complex is first converted to PComplex.)

Location

`Math` ⟹ `Complex Parts` ⟹ `mag(x)`

Example

`mag( (1,@45) )` returns `1`.

Notes

Mappings are retained in the result.

See Also

`conj(x)`, `im(x)`, `j(x)`, `phase(x)`, `re(x)`, and `Real Parts`.

`Build Complex`, `Trig Mode`, and `UnBuild Complex` in the "General Reference" chapter.

magDist(x,from, thru,step)

An object used to calculate the distribution of numbers in the input container.

Use

Use `magDist(x,from,thru,step)` to calculate a distribution of numbers in the input array. The distribution is done in the range of `from` to `thru` by a size of `step`. The `step` parameter is the size of the range of the sub-array that the numbers will go into.

The `x` input may be of any size and of the type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For Complex, PComplex, and Spectrum type inputs, the magnitude of the complex number is calculated first, then that magnitude is used to calculate the frequency distribution. The `x` input shape must be an Array.

For input of all types, a Real Array 1D is returned. Each array element represents how many of the input values will fit in each distribution range.

The `from`, `thru` and `step` parameters must be Scalar in shape and of Real type or be able to be converted to Real.

The `from` parameter must be less than the `thru` parameter.

The `step` parameter must be greater than zero.

Location

`AdvMath` ⟹ `Freq Distribution` ⟹ `magDist(x,from,thru,step)`

Example

`magDist(x,1,5,1)`, where `x` is the array `[1 2 3 4 5 5 3 3 3 2 2 2]`, returns the array `[1 4 4 3]`. In this example there are four sub-ranges ranging from 1 to 5 by 1. The output array shows how many of the input numbers fall into each sub-range.

Notes

The sub-ranges for the distribution are determined in the following manner:

Distribution of Numbers Accepted

Range	Range Minimum	range	Range Maximum
1.	"from" parm	<= range1	< "from" parm + "step" parm (range1limit)
2.	range1limit	<= range2	< range1limit + "step" parm (range2limit)
3.	range2limit	<= range3	< range2limit + "step" parm (range3limit)
...	...	...	...
n-1	range(n-2)limit	<= range(n-1)	<= "thru" parm

Mappings on the operands are ignored and the return value is not mapped.

See Also

logMagDist(x,from,thru,logStep).

matDivide(numer, denom)

This object will divide two matrices `A` and `B`. The `matDivide(numer,denom)` operation is defined in algebraic terms as `A/B`; but in matrix (linear algebraic) terms, it is defined as `inv(B)*A`.

Use

Use the `matDivide(numer,denom)` to divide one matrix into another. Given the linear system `BX=A`, where the `B` and `A` matrices are known, solve for the solution matrix `X`. In algebraic terms the solution is to solve for `X` by dividing `A` by `B`: `X = A/B`. But, by solving the system of equations with linear algebraic matrix rules, the operation is really `X = inv(B)*A`. This operation is useful to find the solution of a linear system of `n` equations in `n` unknowns.

If `B` is an `n` x `n` (square) matrix, then the linear system `BX = A` is a system of `n` equations in `n` unknowns. Suppose that `B` is non-singular. Then `inv(B)` (inverse of `B`) exists and multiplying `BX = A` by `inv(B)` on both sides, obtains `X` = `inv(B)`*A.

The solution to the multiplication of `inv(B)`*A is the solution to the given linear system. If `B` is nonsingular, then there is a unique solution to the system.

Location

`AdvMath` ⟹ `Matrix` ⟹ `matDivide(numer,denom)`

Example

`matDivide(numer,denom)`, where `A` is an 3x3 matrix
`[ [1 1 1] [0 2 3] [5 5 1] ]` and `B` is a 3x1 matrix `[ [8] [24] [8] ]`,
returns a 3x1 matrix `[ [0] [0] [8] ]`.

Notes

Mappings on the operand are ignored and the return value is not mapped.

`matDivide(numer,denom)` uses the inverse of `B`; hence, `B` must be a square matrix. The rules for `matMultiply(A,B)` apply to the `inv(B) * A` operation.

See Also

/ (divide) and matMultiply(A,B).

matMultiply(A,B)

An object used to multiply two matrices together according to linear Algebra rules.

Use

Use `matMultiply(A,B)` to multiply together the two input matrices `A` and `B`. Let `A` be an `m` x `p` matrix and `B` be a `p` x `n` matrix. The matMultiply of A and B is a new matrix, `Z`, that is an `m` x `n` matrix. Notice that the multiplication is defined if and only if the matrices are of the sizes where `A` is an `m` x `p` matrix and `B` is a `p` x `n` matrix (that is, where the number of rows of the second matrix equals the number of columns of the first).

The input matrices must be of matrix shape and of the type Int32, Real, Complex or PComplex. For all input types the same output type is returned, except for Int32 which returns a Real type. If the two matrices are not of the same type, they lower type matrix is promoted to the higher type and the return value is of the higher type.

Location

`AdvMath` $\Longrightarrow$ `Matrix` $\Longrightarrow$ `matMultiply(A,B)`

Example

`matMultiply(A,B)`, where `A` is an 2x1 matrix `[ [3] [1] ]` and `B` is a 1x2 matrix `[2 4]`, returns a 2x2 matrix `[ [6 12] [2 4] ]`.

Notes

Mappings on the operand are ignored and the return value is not mapped.

See Also

`matDivide(numer,denom)` and `* (multiply)`.

Matrix

A menu item.

Use

Use the `Matrix` operations to calculate common linear algebra operations on matrices.

- `det(x)`
- `inverse(x)`
- `transpose(x)`
- `identity(x)`
- `minor(x,row,col)`
- `cofactor(x,row,col)`
- `matMultiply(A,B)`
- `matDivide(numer,denom)`

Location

`AdvMath` $\Longrightarrow$ `Matrix` $\Longrightarrow$

max(x)

An object used to return the maximum value in the input container.

Use

Use `max(x)` to return the maximum value in the container.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, Waveform, or Text. For `x` input of all types except Waveform, the same output type is returned, but is Scalar in shape. A Waveform input type returns a Real Scalar. For Coord input types, the operation is done on the dependent variable. For Text input types, the `max(x)` text string is the one that is the highest lexically ordered string.

Location

`AdvMath` ⟹ `Statistics` ⟹ `max(x)`

Example

`max(x)`, where `x` is the array `[5 34 1 54 6 2 9 7 16 42]`, returns `54`.

Notes

Mappings on the operand are ignored.

See Also

`maxIndex(x)`, `maxX(x)`, `mean(x)`, `median(x)`, `min(x)`, `mode(x)`, `rmx(x)`, `sdev(x)`, and `vari(x)`.

maxIndex(x)

An object used to return the index in the input array of the largest element.

Use

Use `maxIndex(x)` to return the index of the largest element in a 1D array.

The `x` input must be of Array 1D shape and of the type Int32, Real, Coord, Waveform, or Text. For all `x` input types, an Int32 number representing the index of the largest element is returned.

Location

`AdvMath` $\Longrightarrow$ `Data Filtering` $\Longrightarrow$ `maxIndex(x)`

Example

Applying the `maxIndex(x)` to an array, where `x` is the array `[.1, .4, .6, .9]` returns `3` because `.9` is the largest element of the array.

`maxIndex(x)` of `["Ken" "Sue" "Randy" "Bill" "Doug"]` returns `1` because `"Sue"` is the highest lexically ordered term.

Notes

Note that array indices start with zero.

Mappings on the operand are ignored.

See Also

`max(x)`, `maxX(x)`, `min(x)`, `minIndex(x)`, and `minX(x)`.

maxX(x)

An object used to return the `x` value of a maximum point of the input Array 1D where the array is mapped.

Use

Use `maxX(x)` to return the `Xindex` of the largest `y` element in a 1D (linear) array. The `x` input must be of Array 1D shape and of the type Int32, Real, Coord, Waveform, or Text. For `x` input of all types, a Real Scalar container is returned.

If the data is mapped, the value returned is the `x` value corresponding to the maximum `y` value of the input data. The `x` value is calculated using the array mappings. If the data is not mapped, then the value returned is the index in the array of the maximum `y` value. That is, it will return the (implicit) position in the array of maximum `x` value.

The Coord type returns the independent variable `x` at the maximum `y` value.

This function is only useful for mapped data or Coord types, as it is the same as `maxIndex(x)` for unmapped data.

Location

`AdvMath` $\Longrightarrow$ `Data Filtering` $\Longrightarrow$ `maxX(x)`

Example

Take the `maxX(x)` of a sine waveform type that has 256 points and is mapped over `0` to `20` milliseconds. The return value is `.009219`.

Take the `maxX(x)` of an array, where `x` is the unmapped array. `[.1 .4 .6 .9]` returns `3` because `.9` is the largest element of the array. This is identical to `maxIndex(x)`.

See Also

`max(x)`, `maxIndex(x)`, `min(x)`, `minIndex(x)`, and `minX(x)`.

`Get Mappings` and `Set Mappings` in the "General Reference" chapter.

mday(aDate)

An object that returns the day of the month of the time `x`.

Use

Use `mday(aDate)` to transform the given time `x` into a container of the same shape with value(s) `1 - 31` corresponding to the day of the month. `x` must be of type Int32, Real, Coord, or Waveform of any shape. Int32 returns Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Time & Date` ⟹ `mday(aDate)`

Example

`mday(dmyToDate(25,12,1991))` returns `25`, the 25th day of the month.

Notes

`mday(aDate)` returns `1-31`, not `0-30`. This is one of the few non-zero based functions in HP VEE. Mappings are retained in the return value.

See Also

`dmyToDate(d,m,y)`, `hmsToHour(h,m,s)`, `hmsToSec(h,m,s)`, `month(aDate)`, `now()`, `Time & Date`, `wday(aDate)`, and `year(aDate)`.

mean(x)

An object used to calculate the mean value of the data in the input container.

Use

Use `mean(x)` to return the mean (average) value of the container. The numbers in the input container are added, then the sum is divided by the number of elements in the input.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, a Real Scalar container is returned. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` ⟹ `Statistics` ⟹ `mean(x)`

Example

Where `x` is the array `[1 2 3 11 12 34]`, `mean(x)` returns `10.5`.

Notes

Mappings on the operand are ignored.

See Also

`max(x)`, `median(x)`, `min(x)`, `mode(x)`, `rms(x)`, `sdev(x)`, and `vari(x)`.

meanSmooth(x, numPts)

An object used to smooth input data using the mean of a specified number of data points surrounding the data point of interest to calculate the smoothed data point.

Use

Use `meanSmooth(x,numPts)` to perform a data smoothing similar to the `polySmooth(x)` routine. The algorithm calculates an arithmetic mean of the `n` data points around the point to be smoothed. For example, if `n` is 5, take the mean of: 2 points before the current point, the current point, and two points after the current point. If `n` is even, the algorithm will add one more to the number of points to make it odd.

The `x` input must be of Array 1D shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, the same output type is returned, except for Int32 which returns a Real type. The independent variables of a Coord input type must be equidistant, that is, the x-interval between adjacent points must be a constant. The smoothing is only done on the dependent variable (the `y` data of the Coord).

The noise suppression capability of this technique is a function of the size `n` of the averaging which you specify. Large values of `n` will suppress noise to a much greater degree than small values of `n`. Quite often this function can be used to dramatically improve the results of other functions (like `deriv(x,order)`) if used to preprocess the data.

Location

`AdvMath` $\Longrightarrow$ `Data Filtering` $\Longrightarrow$ `meanSmooth(x,numPts)`

Notes

Mappings on input parameters are ignored and the output has the same mappings as the input parameter `x`.

See Also

movingAvg(x,numPts), polySmooth(x), Regression types, and Signal Processing.

median(x)

An object used to calculate the median value of the input container.

Use

Use `median(x)` to return the middle value of the container.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, a Real Scalar is returned. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` ⟹ `Statistics` ⟹ `median(x)`

Example

`median([1 2 3 8 20])` returns `3`.

`median(x)`, where `x` is the array `[1 2 5 11 12 33]`, returns `8`. The median value is the one in the middle of the range of numbers. In the case where there are an even number of numbers, the median is calculated as the average between the two middle numbers. In this case, the median value is the average of `5` and `11`, which is `8`.

`median([20 5 10])` returns `10`.

Notes

Mappings on the operand are ignored.

See Also

`max(x)`, `mean(x)`, `min(x)`, `mode(x)`, `rms(x)`, `sdev(x)`, and `vari(x)`.

min(x)

An object used to return the minimum value in the input container.

Use

Use `min(x)` to return the minimum value in the container.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, Waveform, or Text. For `x` input of all types except Waveform, the same output type is returned, but is Scalar in shape. A Waveform input type returns a Real Scalar. For Coord input types, the operation is done on the dependent variable. For Text input types, the `min(x)` text string is the one that is the lowest lexically ordered string.

Location

`AdvMath` ⟹ `Statistics` ⟹ `min(x)`

Example

`min(x)`, where `x` is the array `[5 34 1 54 6 2 9 7 16 42]`, returns `1`.

Notes

Mappings on the operand are ignored.

See Also

`max(x)`, `mean(x)`, `median(x)`, `minIndex(x)`, `minX(x)`, `mode(x)`, `rms(x)`, `sdev(x)`, and `vari(x)`.

minIndex(x)

An object used to return the index in the input array of the smallest element.

Use

Use `minIndex(x)` to return the index of the smallest element in a 1D array.

The `x` input must be of Array 1D shape and of the type Int32, Real, Coord, Waveform, or Text. For all `x` input types, an Int32 Scalar number representing the index of the smallest element is returned.

Location

`AdvMath` ⟹ `Data Filtering` ⟹ `minIndex(x)`

Example

Applying the `minIndex(x)` to an array, where `x` is the array `[.1 .4 .6 .9]` returns `0` because `.1` is the smallest element of the array.

`minIndex(x)` of `["Ken" "Randy" "Doug" "Bill" "Sue"]` returns `3` because `"Bill"` is the lowest lexically ordered term.

Notes

Note that array indices start with zero.

Mappings on the operand are ignored.

See Also

`max(x)`, `maxIndex(x)`, `maxX(x)`, `min(x)`, and `minX(x)`.

minor(x,row,col)

An object used to calculate the minor of a matrix `x` at row `r` and column `c`.

Use

Use `minor(x,row,col)` to calculate the minor of the square input matrix `x`. The minor of matrix is defined as the determinant of the submatrix of the input matrix `x` obtained by deleting the `r`th row and `c`th column. The `x` input must be a square matrix shape and of the type Int32, Real, Complex or PComplex. For `x` input of all types, output of the same type is returned and is Scalar in shape, except for Int32 which returns a Real. The input for the row and column to delete, `r` and `c`, must be Int32 Scalar or be able to be converted to Int32 type.

A square matrix has the same number of rows as columns.

Location

`AdvMath` ⟹ `Matrix` ⟹ `minor(x,row,col)`

Example

`minor(a,1,2)`, where `a` is a matrix `[ [3 -1 2] [4 5 6] [7 1 2] ]`, returns `-34`.

Notes

Mappings on the operand are ignored.

The `r` and `c` inputs are expected to be of type `Int32` or be able to be converted to that type. The rows and columns of the matrix are numbered from `0` to `n-1` so be careful when specifying which row and column to use.

See Also

`cofactor(x,row,col)` and `det(x)`.

minX(x)

An object used to return the `x` value of the minimum point of the input Array 1D where the array is mapped.

Use

Use `minX(x)` to return the `Xindex` of the smallest `y` element in an 1D (linear) array. The `x` input must be of Array 1D shape and of the type Int32, Real, Coord, Waveform, or Text. For `x` input of all types a Real Scalar container is returned.

If the data is mapped, the value returned is the `x` value corresponding to the minimum `y` value of the input data. The `x` value is calculated using the array mappings. If the data is not mapped, then the value returned is the index in the array of the minimum `y` value. That is, it will return the (implicit) position in the array of minimum `x` value.

The Coord type returns the independent variable `x` at the minimum `y` value.

This function is only useful for mapped data or Coord types. It is the same as `minIndex(x)` for unmapped data.

Location

`AdvMath` $\Longrightarrow$ `Data Filtering` $\Longrightarrow$ `minX(x)`

Example

Take the `minX(x)` of a sine waveform that has 256 points and is mapped over `0` to `20` milliseconds. The return value is `.0107`.

Take the `minX(x)` of an array, where `x` is the unmapped array. `[.1 .4 .6 .9]` returns `0` because `.1` is the smallest element of the array. This is identical to `minIndex(x)`.

See Also

`max(x)`, `maxIndex(x)`, `maxX(x)`, `min(x)`, and `minIndex(x)`.

`Get Mappings` and `Set Mappings` in the "General Reference" chapter.

mod (modulo)

An object that performs an arithmetic modulo (remainder) of two operands.

Use

Use `mod` to determine the remainder of the division of two containers. The two containers may be of type Int32, Real, Coord, or Waveform. The two containers may be of any shape. But if one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a container of the highest type, with the same shape as the operands.

If both operands are of type Coord, they must have their independent variable(s) match exactly or an error is returned. The modulo is only performed on the dependent (last) variable.

Location

`Math` ⟹ `+ - * /` ⟹ `mod`

Example

A scalar modulo an array: `3.2 MOD [1 2 3]` returns `[0.2 1.2 0.2]`.

An array modulo a scalar: `[1 2 3] MOD 2` returns `[1 0 1]`.

Two Coord scalars: `coord(1,3) MOD coord(1,5)` returns `coord(1,2)`.

Two Coord scalars: `coord(1,3) MOD coord(2,5)` returns an error.

Two scalars: `12.95 mod 2` returns `0.95`.

Two scalars: `12.95 mod 2.1` returns `0.32`.

Notes

If either of the containers is mapped (that is, of type Waveform, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings. The return value will have the same mappings as the operands; otherwise an error is returned.

See Also

`+ (add)`, `div (truncated division)`, `/ (divide)`, `* (multiply)`, and `- (subtract)`.

mode(x)

An object used to calculate the mode of the data in the input container.

Use

Use `mode(x)` to return the mode of the value in the container. The mode value is the one which is found most often in the data. In case of bimodal data (two numbers represented the same number of times) or higher modes, `mode(x)` returns the lowest valued mode found in the array container.

In the case where there is no mode value (that is, every number is unique), the first value in the input container is returned.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types except Int32, a Real Scalar is returned. For Coord input types, the operation is done on the dependent variable. An Int32 input type returns an Int32.

Location

`AdvMath` ⟹ `Statistics` ⟹ `mode(x)`

Example

Single mode: where `x` is the array `[2 1 12 11 2 33]`, `mode(x)` returns 2.

Bimodal: where `x` is the array `[2 1 2 12 11 12]`, `mode(x)` returns 2.

No mode: where `x` is the array `[3 2 1 11 12 33]`, `mode(x)` returns 3.

Notes

Mappings on the operand are ignored.

See Also

`max(x)`, `mean(x)`, `median(x)`, `min(x)`, `rms(x)`, `sdev(x)`, and `vari(x)`.

month(aDate)

An object that returns the month of the year of the time `x`.

Use

Use `month(aDate)` to transform the given time `x` into a container of the same shape with value(s) `1 - 12` corresponding to the month of the year. `x` must be of type Int32, Real, Coord, or Waveform of any shape. Int32 returns Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Time & Date` ⟹ `month(aDate)`

Example

`month(dmyToDate(25,12,1991))` returns `12`, the 12th month of the year.

Notes

`month(aDate)` returns `1-12`, not `0-11`. This is one of the few non-zero based functions in HP VEE. Mappings are retained in the return value.

See Also

`dmyToDate(d,m,y)`, `hmsToHour(h,m,s)`, `hmsToSec(h,m,s)`, `mday(aDate)`, `now()`, `Time & Date`, `wday(aDate)`, and `year(aDate)`.

movingAvg(x, numPts)

An object used to smooth the input data using the average of a specified number of data points preceding the data point of interest to calculate the smoothed data point.

Use

Use `movingAvg(x,numPts)` to perform a data smoothing similar to the `meanSmooth(x,numPts)` routine. The algorithm calculates an arithmetic mean of `n` previous data points.

The `x` input must be of Array 1D shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types the same output type is returned, except for Int32 which returns a Real type. The independent variables of a Coord input type must be equidistant, that is, the x-interval between adjacent points must be a constant. The smoothing is only done on the dependent variable (the `y` data of the Coord).

This technique tends to represent more of a "historical" view of the data since, for any point in the array, the average value is calculated only from earlier points. This is in contrast to the average value being calculated from points on both sides of the point in question, as is the case for `polySmooth(x)` and `meanSmooth(x,numPts)`.

Location

`AdvMath` $\Longrightarrow$ `Data Filtering` $\Longrightarrow$ `movingAvg(x,numPts)`

Example

If `n` is 5, the average of the 4 previous points and the point of interest are used to determine what the smoothed point should be.

Notes

Mappings on input parameters are ignored and the output has the same mappings as the input parameter `x`.

See Also

`meanSmooth(x,numPts)`, `polySmooth(x)`, `Regression` types, and `Signal Processing`.

* (multiply)

An object that performs an arithmetic multiplication on two operands.

Use

Use * to multiply the values of two containers. The two containers may be of any type and of any shape. If one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a container of the highest type, with the same shape as the operands.

If both operands are of type Coord, they must have their independent variable(s) match exactly or an error is returned. The multiplication is only performed on the dependent (last) variable.

If one of the containers is Text, the other must be an Int32. Text multiplication consists of repeating the string the number of times given by the value of the Int32. Enums are converted to Text for multiplication.

This multiplication operation performs a parallel multiplication on all elements of the arrays, including matrices. For a matrix multiply, see the function `matMultiply(A,B)`.

Location

`Math` ⟹ `+ - * / ` ⟹ `*`

Example

Array times a scalar: `[1 2 3] * 3` returns `[3 6 9]`.

Two arrays: `[4 5 6] * [1 2 3]` returns `[4 10 18]`.

Two PComplex scalars: `(2,@45) * (3,@90)` returns `(6,@135)` with Trig Mode set to Degrees.

Two Coord scalars: `coord(1,3) * coord(1,5)` returns `coord(1,15)`.

Two Coord scalars: `coord(1,3) * coord(2,5)` returns an error.

Text times a scalar: `"hello" * 3` returns `"hellohellohello"`.

*** (multiply)**

A Text array times an array: `["hello" "b"] * [3 2]` returns `["hellohellohello" "bb"]`.

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings. The return value will have the same mappings as the operands; otherwise an error is returned.

See Also

`+ (add)`, `/ (divide)`, `matMultiply(A,B)`, and `- (subtract)`.

NOT

An object that performs a logical NOT operation on one operand.

Use

Use `NOT` to determine the logical opposite of the value(s) of a container. The container may be of any type and of any shape. The result is an Int32 of the same shape as the operand, with value(s) `0` or `1`. If the operand is false (zero), the value of the `NOT` operator is `1`; otherwise the value is `0`.

For Coord containers, only the dependent (last) variable is considered for the `NOT` operation.

For Complex, PComplex, and Spectrum containers, the value of the operand is true if either part is non-zero. Text is true if non-null. Enums are converted to Text for the operation.

Location

`Math` ⟹ `Logical` ⟹ `NOT`

Example

A scalar: `NOT 3` returns `0`.

A scalar: `NOT 0` returns `1`.

An array: `NOT [-3 0 3]` returns `[0 1 0]`.

A PComplex scalar: `NOT (1,@90)` returns `0`.

A Complex scalar: `NOT (0,1)` returns `0`.

A Complex scalar: `NOT (0,0)` returns `1`.

A Coord scalar: `NOT coord(1,3)` returns `0`.

A Coord scalar: `NOT coord(1,0)` returns `1`.

A Text scalar: `NOT "too"` returns `0`.

A Text scalar: `NOT ""` returns `1`.

Notes

The result has the same mapping as the operand.

Note that the `If/Then/Else` device requires the expression(s) inside it to evaluate to either a scalar or an array, which is either all zeros or all ones.

See Also

`Conditional` and `If/Then/Else` in the "General Reference" chapter.

!= (not equal to)

An object that performs an *is not equal to* operation on two operands.

Use

Use `!=` to determine whether the value(s) of one container is not equal to the value(s) of another container. The two containers may be of any type and of any shape. If one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a scalar Int32 with the value `0` or `1`. If the first operand is not equal to the second, the value of the result is `1`; otherwise the value is `0`.

For Coord containers, if any of the independent variable(s) do not match exactly, an error is returned. For Complex, PComplex, and Spectrum containers, if either part or both parts do not match, the result is `1`. Enums are converted to Text for the comparison.

Arrays must have one or more of their respective values not equal for the result to be `1`.

Location

`Math` ⟹ `Relational` ⟹ `!=`

Example

A scalar and an array: `3.0 != [3 3 3]` returns `0`.

A scalar and an array: `3.0 != [3 1 3]` returns `1`.

Two arrays: `[1 2 3] != [1 2 4]` returns `1`.

Two PComplex scalars: `(1,@90) != (1,@85)` returns `1`.

Two Complex scalars: `(2,3) != (2,4)` returns `1`.

Two Complex scalars: `(2,3) != (2,3)` returns `0`.

Two Coord scalars: `coord(1,3) != coord(1,5)` returns `1`.

Two Coord scalars: `coord(1,3) != coord(2,3)` returns `Values for independent variables must match`.

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings; otherwise an error is returned.

See Also

`~= (almost equal to)`, `AND`, `== (equal to)`, `> (greater than)`, `>= (greater than or equal to)`, `< (less than)`, `<= (less than or equal to)`, `NOT`, `OR`, `Relational`, and `XOR`.

`Comparator` and `Conditional` in the "General Reference" chapter.

now()

An object that returns the current time.

Use

Use `now()` to generate a Real scalar container with the value of the current time. The value of the Real is the number of seconds since `00:00 hours 1 January 0001 AD` given in UTC.

Location

`Math` ⟹ `Time & Date` ⟹ `now()`

Example

`now()` returns around `62.802G`.

See Also

`dmyToDate(d,m,y)`, `hmsToHour(h,m,s)`, `hmsToSec(h,m,s)`, `mday(aDate)`, `month(aDate)`, `Time & Date`, `wday(aDate)`, and `year(aDate)`.

OR

An object that performs a logical `OR` operation on two operands.

Use

Use `OR` to determine whether the value(s) of either of two containers is logically true (non-zero). The two containers may be of any type and of any shape. If one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is an Int32 of the same shape as the operands, with value(s) `0` or `1`. If either or both operands are non-zero, the value of the `OR` operation is `1`; otherwise the value is `0`.

If both operands are of type Coord, they must have their independent variable(s) match exactly or an error is returned. Only the dependent (last) variable is considered for the `OR` operation.

For Complex, PComplex, and Spectrum containers, the value of the operand is true if either part is non-zero. Text is true if non-null. Enums are converted to Text for the operation.

Location

`Math` ⟹ `Logical` ⟹ `OR`

Example

A scalar and an array: `3 OR [3 3 3]` returns `[1 1 1]`.

A scalar and an array: `0 OR [-3 0 3]` returns `[1 0 1]`.

Two arrays: `[1 0 0] OR [0 1 (-1)]` returns `[1 1 1]`.

Two PComplex scalars: `(1,@90) OR (1,@85)` returns `1`.

Two Complex scalars: `(0,0) OR (0,1)` returns `1`.

Two Complex scalars: `(0,1) OR (1,0)` returns `1`.

Two Complex scalars: `(0,0) OR (0,0)` returns `0`.

Two Coord scalars: `coord(1,3) OR coord(1,5)` returns `1`.

Two Coord scalars: `coord(1,3) OR coord(2,3)` returns an error.

A Text scalar and a scalar number: `"too" OR 0` returns `1`.

A Text scalar and a scalar number: `"" OR 0` returns `1` because the `0` is promoted to the string `"0"`, which is non-null.

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings. The return value will have the same mappings as the operands; otherwise an error is returned.

Note that the `If/Then/Else` device requires the expression(s) inside it to evaluate to either a scalar or an array, which is either all zeros or all ones.

See Also

`AND`, `NOT`, `Relational`, and `XOR`.

`Conditional` and `If/Then/Else` in the "General Reference" chapter.

ordinal(x)

An object that returns the ordinal value of `x`.

Use

Use `ordinal(x)` to obtain the ordinal value of a container. `x` may be any shape and of the types Int32, Real, Coord, Waveform, or Enum. For Int32, Real, Coord, and Waveform, the result is the same as the argument with the same mappings. For Enum, the result is an Int32 of the same shape as `x` with the value of the ordinal value of the Enum. Enums created from the Enum Constant have their ordinal values starting at `0` and going to `n-1`, where `n` is the number of Enum values in the list.

Location

`Math` ⟹ `Real Parts` ⟹ `ordinal(x)`

Example

If an `Enum` constant is created with the strings ordered `"a"`, `"b"`, and `"c"`, and the current value `"a"` is selected, `ordinal(x)` on that container would return `0` since the first value is selected. Remember that HP VEE is zero-based. If, with the the same `Enum` constant, the currently selected value is `"c"`, then `ordinal(x)` on that container would return `2`, the offset of the currently selected value.

Notes

Mappings are retained in the result.

See Also

`Complex Parts`, `Enum`, and `Real Parts`.

perm(n,r)

An object used to calculate the permutations of `n` numbers taken `r` at a time.

Use

Use `perm(n,r)` to calculate the number of permutations of `n` things taken `r` at a time using the formula:

```
Perm(n, r) = n! / (n - r)!
```

The `!` symbol means factorial. The `n` input may be of any shape and size and of the type Int32, Real, Coord, or Waveform. For `n` input of all types the same output type is returned, except for Int32 which returns a Real type. The second `r` parameter must be of the same type or be able to be converted to the same type as the `n` input value. If both of the inputs are arrays, they must be of exactly the same shape and size.

The `perm(n, r)` operation is only defined for integer operands so the input values, while of unique type, are converted to Int32 type before the calculation is done.

Location

`AdvMath` ⟹ `Probability` ⟹ `perm(n, r)`

Example

`perm(10,3)` will return 720 as given by the formula
`10!/(10-3)! = 10!/7! = 10*9*8 = 720`.

`perm(10.4,3.9)` will return 720 as given by the formula
`10!/(10-3)! = 10!/7! = 10*9*8 = 720`. The rule about converting Real to Int32 forces `10.4` to `10` and `3.9` to `3`.

Notes

Both `n` and `r` must be positive and `n` must be greater than `r`. If both of the inputs are mapped, then the mappings must be the same. The return value has the same mappings as the input if either input is mapped. If neither of the inputs is mapped, then the output is unmapped.

See Also

`beta(x,y)`, `binomial(a,b)`, `comb(n,r)`, `factorial(n)`, and `gamma(x)`.

phase(x)

An object that returns the phase of a PComplex number `x`.

Use

Use `phase(x)` to extract the phase of a PComplex number `x`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For types Int32, Real, Coord, and Waveform, the same type with value(s) of zero is returned. For types Complex, PComplex, and Spectrum, `phase(x)` returns a Real container with the value of the polar phase angle of the PComplex number. (Complex is first converted to PComplex.) The `phase(x)` value is returned in the units of the current `Trig Mode`.

Location

`Math` ⟹ `Complex Parts` ⟹ `phase(x)`

Example

`phase((1,@45))` returns `45`.

Notes

Mappings are retained in the result.

See Also

`conj(x)`, `im(x)`, `j(x)`, `mag(x)`, `re(x)`, and `Real Parts`.

`Build PComplex`, `Trig Mode`, and `UnBuild PComplex` in the "General Reference" chapter.

poly(x,vec)

An object that returns the polynomial value of the values of `x` and `vec`.

Use

Use `poly(x,vec)` to generate the polynomial result of the `x` data and `vec` coefficients. `x` can be any shape and of type Int32 or Real. `vec` must be either a scalar or a one-dimensional array. `poly(x,vec)` takes the elements of `vec` as the polynomial coefficients, and `x` as the data. `vec` starts with `a0`, then `a1`, and so on, out to the length of `vec`. The return value is Real and the same shape as `x`.

The polynomial equation computes:

```
a0 + a1*x + a2*x*x + a3*x*x*x + ...
```

Location

`Math` ⟹ `Polynomial` ⟹ `poly(x,vec)`

Example

`poly([10 2], [3 2 4])` generates `[423 23]`.

Notes

Mappings of the resultant container are the same as the input container `x`.

See Also

`ramp(numElem,from,thru)`.

Polynomial

A menu item.

Use

Use `Polynomial` to access the following object which generates a Real polynomial result from the data and coefficient values.

- `poly(x,vec)`

Location

`Math` ⟹ `Polynomial` ⟹

See Also

`Generate`.

polynomial regression

An object used to fit a polynomial curve of arbitrary degree to `(x,y)` data.

Use

Use the polynomial regression to fit the data to the equation `y = C0 + C1*x + C2*x^2 + ...` , where `x` is the `x` coordinate and `C0`, `C1`, and so forth, are calculated coefficients. Polynomial curve fitting requires the degree of the polynomial to be set on the Order input field on the regression object.

The polynomial regression object expects an array of Coord type of input with one independent variable, that is, an `(x,y)` pair. If the input array is not a Coord type, an attempt is made to convert it to Coord. If the input data is mapped (Waveform, Spectrum, or a mapped array), then the conversion to Coord uses the mapping information to create the `x` part of the `(x,y)` pairs. If the input data is not mapped (for example, an array), then the `x` part of the `(x,y)` pair is implicitly generated from its position in the array.

Location

`AdvMath` $\Longrightarrow$ `Regression` $\Longrightarrow$ `polynomial`

Open View Parameters

The `Fit Type` field on the open view is used to change the regression type to `linear`, `logarithmic`, `exponential`, `power curve` or `polynomial regression`. Clicking on the button will bring up a list of the different regression types.

The Order input field is used to set the order of the polynomial to which the data is fitted. The order of the polynomial fit must be greater than or equal to one. A polynomial fit of order one fits the data to a straight line and is the same as the `linear regression` choice.

Example

See `Regression` for a Coord conversion example.

Notes

See `Regression` for general notes on regression.

See Also

`exponential regression`, `linear regression`, `logarithmic regression`, `meanSmooth(x,numPts)`, `movingAvg(x,numPts)`, `polySmooth(x)`, and `power curve regression`.

`Build Coord` in the "General Reference" chapter.

polySmooth(x)

An object used to smooth data using a polynomial fit of the data.

Use

Use `polySmooth(x)` to smooth out data. A fourth order polynomial is fitted to the data and a new data point is calculated based on the curve fit using the polynomial. To do the 4th order curve fit, 5 points are used, two on either side of the data point being fitted and the data point itself, with special consideration for the two end points.

The `x` input must be of Array 1D shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types the same output type is returned, except for Int32 which returns a Real type. The independent variables of a Coord input type must be equidistant. That is, the x-interval between adjacent points must be a constant. The smoothing is only done on the dependent variable (the `y` data of the Coord).

Location

`AdvMath` $\Longrightarrow$ `Data Filtering` $\Longrightarrow$ `polySmooth(x)`

Notes

The method used is to take a 5-element sliding window and apply the standard formula for a 4th-degree Stirling collocation polynomial. Refer to Francis Scheid, *Schaum's Outline of Theory and Problems of Numerical Analysis*, McGraw Hill, New York, NY, 1968. ISBN #07-055197-9.

Mappings on the operand are ignored and the output container has the same mappings as the input.

The filtering operation which results from this technique has a fixed "frequency response" which suppresses only the noisiest part of the data. If more noise suppression is required, see the `meanSmooth(x,numPts)` function, which you can use to suppress much more noise.

See Also

`meanSmooth(x,numPts)`, `movingAvg(x,numPts)`, `Regression` types, and `Signal Processing`.

Power

A menu item.

Use

Use `Power` to access the following objects which perform power functions such as square root, square, log, exp, and log base 10.

- `sq(x)`
- `sqrt(x)`
- `cubert(x)`
- `recip(x)`
- `log(x)`
- `log10(x)`
- `exp(x)`
- `exp10(x)`

Location

`Math` ⟹ `Power` ⟹

See Also

`+ (add)`, `/ (divide)`, `^ (exponent)`, `* (multiply)`, and `- (subtract)`.

power curve regression

An object used to fit a power curve to `(x,y)` data.

Use

Use the power regression to fit the data to the equation `y = C0 * x^C1`, where `x` is the `x` coordinate and `C0` and `C1` are calculated coefficients.

The power curve regression object expects an array of Coord type of input with one independent variable. That is, an `(x,y)` pair. If the input array is not a Coord type an attempt is made to convert it to Coord. If the input data is mapped (Waveform, Spectrum, or a mapped array), then the conversion to Coord uses the mapping information to create the `x` part of the `(x,y)` pairs. If the input data is not mapped (for example, an array), then the `x` part of the `(x,y)` pair is implicitly generated from its position in the array.

Location

`AdvMath` ⟹ `Regression` ⟹ `power curve`

Open View Parameters

The `Fit Type` field on the open view is used to change the regression type to `linear`, `logarithmic`, `exponential`, `power curve` or `polynomial regression`. Clicking on the field will bring up a list of the different regression types.

Example

See `Regression` for a Coord conversion example.

Notes

See `Regression` for general notes on regression.

This function will not work for unmapped arrays. Unmapped arrays are converted to Coord starting with `x=0`, which fails in the conversion in the log function. The log function is used in the internal regression calculation.

power curve regression

See Also

`exponential regression`, `linear regression`, `logarithmic regression`, `meanSmooth(x,numPts)`, `movingAvg(x,numPts)`, `polynomial regression`, and `polySmooth(x)`.

`Build Coord` in the "General Reference" chapter.

Probability

A menu item.

Use

Use these `Probability` functions in problems dealing with random numbers and probabilities.

- `random(low,high)`
- `randomize(x,low,high)`
- `randomSeed(seed)`
- `perm(n,r)`
- `comb(n,r)`
- `gamma(x)`
- `beta(x,y)`
- `factorial(n)`
- `binomial(a,b)`
- `erf(x)`
- `erfc(x)`

Location

`AdvMath` $\Longrightarrow$ `Probability` $\Longrightarrow$

See Also

`Random Number` and `Random Seed` in the "General Reference" chapter.

prod(x)

An object used to multiply all the elements of the input array.

Use

Use `prod(x)` to multiply together all the elements of the input array. The `x` input may be of any size and shape and of the type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For `x` input of all types, except Waveform, Coord, and Spectrum, output of the same type as the input is returned. An input of type Waveform or Coord returns a Real Scalar. An input of type Spectrum returns a PComplex Scalar.

The output of this object is the arithmetic product of all the data contained in the input container and is Scalar in shape.

Location

`AdvMath` $\Longrightarrow$ `Array` $\Longrightarrow$ `prod(x)`

Example

Where `x` is an array `[1 2 3 4]`, `prod(x)` returns `24`.

Notes

Mappings on the operand are ignored.

See Also

`* (multiply)` and `sum(x)`.

ramp(numElem, from,thru)

An object that generates a linearly ramped array.

Use

Use `ramp(numElem,from,thru)` to generate a Real one-dimensional array of length `numElem`, with the values linearly ramped from `from` to `thru`. `numElem` must be a scalar container that is, or can be converted to, Int32 and with a value greater than zero. `from` and `thru` must be scalar containers that are, or can be converted to, Real.

If `from` is less than `thru`, the ramping is positive; otherwise it automatically ramps negatively.

Location

`Math` ⟹ `Generate` ⟹ `ramp(numElem,from,thru)`

Example

`ramp(3, 0.5, 0)` returns a one-dimensional array with values `[0.5 0.25 0]`.

`ramp(5, 2, 10)` returns a one-dimensional array with values `[2 4 6 8 10]`.

Notes

The return value has no mappings.

The algorithm for generating values is:

```
Y[I] = from + I*(thru-from/numElem-1)
```

```
for I=0..numElem-1
```

This has the effect that the last element in the resultant array has the value `thru`.

See Also

`Alloc Real` and `logRamp(numElem,from,thru)`.

random(low,high)

An object used to generate random numbers between a low and high value. This function generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.

Use

Use `random(low,high)` to generate pseudo-random numbers between the `low` and `high` value. The numbers generated are in the interval including the low value, but not including the high value, which is denoted as `"[l h)"`.

The `l` and `h` inputs must be Scalar in shape and of the type Int32 or Real. A Real Scalar is returned in all cases that are in the range of `"[l h)"`.

This function only outputs a single Scalar value. Use the `randomize(x,low,high)` function with an array input to generate an array of scaled random numbers.

If the `l` value is greater than the `h` value, then the range of random numbers returned is in the interval `"(l h]"`.

Location

`AdvMath` ⟹ `Probability` ⟹ `random(low,high)`

Example

`random(0,1)` will generate real numbers in the range (0.0 1.0). That is, including 0.0, but excluding 1.0.

Notes

`random(low,high)` will return a Real Scalar container in the range `"(0.0 1.0)"`.

Also, `random(low,high)` will return a different number each time it is used in a model or if it is activated several times by an iterator object.

See Also

`randomize(x,low,high)` and `randomSeed(seed)`.

`Random Number` and `Random Seed` in the "General Reference" chapter.

randomize(x, low,high)

An object used to randomize the values of a numeric input container in the range of `[low high)`. The numbers generated are in the interval including `low`, but not including `high`, which is denoted as "`[low high)`". This function generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.

This object can also be used to randomize strings in `Text` input containers within the range of `[low high]`.

Use

For numeric values:

Use `randomize(x,low,high)` to generate pseudo-random numbers between `low` and `high`, that is, between the low and high value. The numbers generated are in the interval including `low`, but not including `high`, which is denoted as "`[low high)`". All of the data in the input container, `x` is overwritten with the pseudo-random numbers.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For `x` input of all types, the same output type is returned, except for PComplex or Spectrum. For PComplex and Spectrum types, only the magnitude of the number is in the range `[low high)`. The phase is random in the range `(-PI to PI)`. For type Coord, only the dependent variable is randomized.

The `low` and `high` inputs must be Scalar in shape and of the type Int32, Real, or of a type that can be converted to Int32 or Real. The random numbers generated are in the range of `[low high)`.

If the `low` value is greater than the `high` value, then the range of random numbers returned is in the interval `(low high]`.

For string values:

To randomize a string, use `randomize(x,low,high)` with input values of the `Text` data type. The input string (the data container on the `x` input) will be overwritten by an output string of ASCII characters in pseudo-random order, within the range `[low high]`, where `low` is the first character in the `low` input container and `high` is the first character in the `high` input container. (Note

that the range is *inclusive* of both `low` and `high`, which is different than for numeric values.) The length of the output string is the same as that of the input string.

The default values for `low` and `high` are the space (" ", ASCII 32) and the tilde ("~", ASCII 126).

Note

Default values are used for strings when the input data for `low` or `high` is "" (null string).

Location

`AdvMath` ⟹ `Probability` ⟹ `randomize(x,low,high)`

Examples

Numeric example:

If the container on the `x` input is of the `Real` data type, `randomize(x,0,10)` will randomize all the values of the input container `x` to real numbers in the range `(0.0 10.0)`. That is, including 0.0, but excluding `10.0`.

String examples:

`randomize("abcdef","a","b")` might return "`abbaba`".

`randomize("abcabcabcab","a","z")` might return "`pterodactyl`". (Well—it might!)

`randomize("abcabc","","")` might return "`Al@z~#`". (With null strings for `low` and `high`, the default limits—ASCII 32 through 126—are used.)

`randomize("aaaaa","go","stop")` might return "`smjpg`". (Only the first characters in `low` and `high` count in determining the range.)

`randomize("aaaaa","97","99")` returns "`99999`". (The `low` and `high` values are interpreted as strings, not ASCII decimal values.)

Notes

Mappings on input parameters are ignored and the output has the same mappings as the input parameter `x`.

For a numeric `x` input, `randomize(x)` will return the a container the same size and shape as `x`, but with the data randomized in the range `(0.0 - 1.0)`.

For string operations, the character set for output cannot be defined outside the limits `low` = ASCII 32 and `high` = ASCII 126. That is, the ASCII characters below 32 and above 126 are not allowed for output in the randomized string.

See Also

`random(low,high)` and `randomSeed(seed)`.

`Random Number` and `Random Seed` in the "General Reference" chapter.

randomSeed(seed)

An object used to set initial values for the `random(low,high)` and the `randomize(x,low,high)` functions.

Use

Use `randomSeed(seed)` to input a random seed entry point for the linear congruential algorithm used by the `random(low,high)` and the `randomize(x,low,high)` functions.

The seed input `s` must be Scalar in shape and of the type Int32 or of a type able to be converted to Int32. The return value of this function is the `s` seed value.

Location

`AdvMath` $\Longrightarrow$ `Probability` $\Longrightarrow$ `randomSeed(seed)`

Example

`randomSeed(97)` initializes the pseudo-random number algorithm.

Notes

This function is the same as the `Random Seed` object.

See Also

`random(low,high)` and `randomize(x,low,high)`.

`Random Number` and `Random Seed` in the "General Reference" chapter.

re(x)

An object that returns the real part of a Complex number `x`.

Use

Use `re(x)` to extract the real part of a Complex number `x`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For types Int32, Real, Coord, and Waveform, the same type with the same value(s) is returned. For types Complex, PComplex, and Spectrum, `re(x)` returns a Real container with the value of the real part of the Complex number. (PComplex is first converted to Complex.)

Location

`Math` ⟹ `Complex Parts` ⟹ `re(x)`

Example

`re( (1,2) )` returns `1`.

`re(1,@180)`, with `Trig Mode` in Degrees, returns `-1`.

Notes

Mappings are retained in the result.

See Also

`Complex Parts`, `conj(x)`, `im(x)`, `j(x)`, `mag(x)`, `phase(x)`, and `Real Parts`.

`Build Complex` and `UnBuild Complex` in the "General Reference" chapter.

Real Parts

A menu item.

Use

Use `Real Parts` to access the following objects which return different real parts of a container.

- `abs(x)`
- `signof(x)`
- `ordinal(x)`
- `round(x)`
- `floor(x)`
- `ceil(x)`
- `intPart(x)`
- `fracPart(x)`

Location

`Math` ⟹ `Real Parts` ⟹

See Also

`Complex Parts`.

recip(x)

An object that returns the reciprocal of the value of `x`.

Use

Use `recip(x)` to generate the reciprocal of a number; that is, one divided by the number `x`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For PComplex, the `recip(x)` is defined as the reciprocal of the magnitude, and the phase's sign is reversed. Complex is converted to PComplex before the function is applied. Int32 arguments will return a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Power` ⟹ `recip(x)`

Example

`recip( (5, @45) )` will return `(0.2, @-45)`.

`recip( (1,-1)) )` will return `(0.5, 0.5)`.

Notes

Mappings are retained in the result.

See Also

`/ (divide)`, `Power`, `sq(x)`, and `sqrt(x)`.

rect(x)

An object used to apply a rectangular window to a time series of values.

Use

Use `rect(x)` to filter the values in `x` in the same manner as convolving `x` with the spectral transform of the rectangular function. This has the effect of suppressing some of the noise due to the tails of the input sequence and the potential discontinuities they represent when sampling periodic signals.

The input `x` must be an Array 1D of type Int32, Real, Coord, or a Waveform, or a Spectrum. The same type is returned, except for Int32, which returns Real.

If `x` is a Spectrum, it is first converted to a Waveform using an `ifft(x)` before the window is applied. The result of the window is then converted back to type Spectrum using an `fft(x)`. A Spectrum is returned.

Location

`AdvMath` ⟹ `Signal Processing` ⟹ `rect(x)`

Example

`rect([1 1 1 1 1 1 1 1])` returns `[1 1 1 1 1 1 1 1]`.

Notes

The `rect` function is represented in the time domain as `1` for all values, where `0 <= n <= N -1`, where `n` is the position (index) in the array, and `N` is the size of the array. The result will be an array of the same type as `x` and will have the same mappings as `x` (if any).

For a discussion of sidelobe levels and coherent gains, see: Ziemer, Tranter, and Fannin, *Signals and Systems*, Macmillan Publishing, New York, NY, 1983. ISBN #0-02-431650-4.

See Also

`bartlet(x)`, `blackman(x)`, `convolve(a,b)`, `fft(x)`, `hamming(x)`, and `hanning(x)`.

Regression

A menu item.

Use

Use `Regression` to fit various types of regression equations to data.

- `linear`
- `logarithmic`
- `exponential`
- `power curve`
- `polynomial`

Location

`AdvMath` ⟹ `Regression` ⟹

Notes

All the Regression objects expect an array of Coord type for input. If the input array is not a Coord type it is converted to Coord. The conversion to Coord type follows standard type conversion rules summarized here. If the input array is not mapped, then an implicit `x` parameter is generated from the array indices. That is, the regression must be done on `(x,y)` pairs. The `x` values will start at zero and range to `n - 1`, where `n` is the number of data points. If the input array is mapped then the generated `x` data will range from the low value of the mapping to the high value of the mapping.

The regression gives as output an array of data fitted to the equation chosen. The number of points of the output array is the same as in the input data array. It also outputs an array of coefficients that corresponds to the derived constants for each regression type. See the individual regression type to determine to which equation the data is fitted. The third output is the R-squared or "goodness of fit" coefficient. It ranges between -1 and 1. A fit of -1 or 1 is an exact fit and numbers in between -1 and 1 represent varying degrees of goodness of fit.

Example

You have an unmapped array input `[10 20 30 40 50]`. It would be converted to Coord array:

`[ (0, 10) (1, 20) (2, 30) (3, 40) (4, 50) ]`.

If an input array is the array `[1 2 3 4 5]`, and it is mapped from 1 to 2 (seconds for instance), it would be converted to Coord array:

`[ (1, 1) (1.25, 2) (1.5, 3) (1.75, 4) (2, 5) ]`.

See Also

`meanSmooth(x,numPts)`, `movingAvg(x,numPts)`, and `polySmooth(x)`.

Relational

A menu item.

Use

Use `Relational` to access the following objects which perform relational operations on two operands:

- `==`
- `!=`
- `<`
- `>`
- `<=`
- `>=`

Location

`Math` ⟹ `Relational` ⟹

Notes

All of these operations return a scalar Int32 with the value of `0` or `1`, which corresponds to whether the operation is true or false.

It is possible for two operands to not be relational, that is, the two operands are not less than, greater than, or equal to each other. Examples are Coords with mismatched mappings and arrays. For example, `[1 2 6] != [2 3 4]` is true, but the two operands are neither less than nor greater than each other, they are simply not equal. That is, `[1 2 6] < [2 3 4]` is false, and `[1 2 6] > [2 3 4]` is false because each element in the first array is not `<` or `>` respectively to each element in the second array.

Also note the difference between menu items under `Relational` and under `Conditional`. `Relationals` are formulas with output `0` or `1`. `Conditionals` are `If/Then/Else` and have two outputs, of which one activates.

Note that the return value is of type Int32 and is the same shape as the operands. This is different than the conditionals such as `==` that always return a scalar.

Relational

The logical operators are defined for type Text only in the sense of whether the string is null or not. That is, `"zoo"` AND `""` (null string) is logically false since the second string is null. Remember that when comparing a Text type to a non-string type, the latter is promoted to a Text type. This means that `"zoo"` AND `0` is true since the Real `0` is promoted to the string `"0"` and, since both strings are non-null, the AND expression is true, returning `1`.

See Also

`~= (almost equal to)`, `AND`, `== (equal to)`, `> (greater than)`, `>= (greater than or equal to)`, `< (less than)`, `<= (less than or equal to)`, `NOT`, `!= (not equal to)`, `OR`, and `XOR`.

`Comparator`, `Conditional`, and `If/Then/Else` in the "General Reference" chapter.

rms(x)

An object used to return the `rms` (Root Mean Square) value of the data in the input container.

Use

Use `rms(x)` to calculate the Root Mean Square value of the input container. The `rms` value is defined as the square root of `[(The sum of x^2)/N]`, where `N` is the number of the points.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For all `x` input types, a Real Scalar container is returned. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` ⟹ `Statistics` ⟹ `rms(x)`

Example

Where `x` is the array `[1 2 3 4 5 11 12 13 14 34]`, `rms(x)` returns `13.319159`.

Notes

Mappings on the operand are ignored.

See Also

`max(x)`, `mean(x)`, `median(x)`, `min(x)`, `mode(x)`, `sdev(x)`, and `vari(x)`.

rotate(x,numElem)

An object used to rotate elements in an array.

Use

Use `rotate(x,numElem)` to rotate the elements of the input array `x` by `n` positions. The `x` input may be of any size and shape and of the type Int32, Real, Coord, Waveform, Complex, PComplex, Spectrum, or Text. For all `x` input types the same output type is returned. The input `n` must be a scalar Int32 or be able to be converted to Int32 type. If the `n` rotate value is a positive value, the rotation direction is down the array (forward). For example, if `n` was 2, then the first element of the array would end up in the third position (`x[2]`) , the second element ends up in the fourth position (`x[3]`), and so on. If the `n` value is negative, the array elements are rotated up the array (backward).

If the rotate number `n` is larger than the number of elements in the array, then the number of places to rotate is calculated by taking the modulo of the rotate number by the number of array elements. For example, if the input array has 5 elements and you want to rotate the array 13 places, the modulo operation `(13 MOD 5 = 3)` determines that the array should be rotated 3 places.

Location

`AdvMath` $\Longrightarrow$ `Array` $\Longrightarrow$ `rotate(x,numElem)`

Example

`rotate(x,5)` rotates the input array elements 5 positions further to the right in the array.

`rotate(x,15)`, where the input array `x` has 15 elements, does nothing because you have rotated the array back on top of itself.

`rotate([1 2 3 4 5], 1)` returns `[5 1 2 3 4]`.

Notes

You might think of the array as a circular list when using the rotate function, meaning that the last element in the array is connected to the first array element. When you rotate array elements past the end of the array, they wrap back around to the top of the list. For example, if the array is 10 elements long and you do a `rotate(A,5)`, then the tenth array element number will end up in fifth position in the array.

Mappings on input parameters are ignored and the output has the same mappings as the input parameter `x`.

See Also

`concat(x,y)` and `init(x,y)`.

round(x)

An object that returns the rounded value of `x`.

Use

Use `round(x)` to obtain the rounded value, to the nearest integer, of a container. `x` may be any shape and of the types Int32, Real, Coord, or Waveform. The `round(x)` function returns the closest integer (as the same type) to `x`. The dividing line is at `0.5`, which rounds up.

Location

`Math` ⟹ `Real Parts` ⟹ `round(x)`

Example

`round([23.0 23.1 23.9 23.5 (-23.5)])` returns `[23 23 24 24 -23]`.

Notes

Mappings are retained in the result.

See Also

`abs(x)`, `ceil(x)`, `Complex Parts`, `floor(x)`, `fracPart(x)`, `intPart(x)`, and `Real Parts`.

sdev(x)

An object used to return the standard deviation of the data in the input container.

Use

Use `sdev(x)` to calculate the standard deviation of the container.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For all `x` input types, a Real Scalar container is returned. For Coord input types, the operation is done on the dependent variable.

The `sdev(x)` is defined as the square root of the `vari(x)`.

Location

`AdvMath` ⟹ `Statistics` ⟹ `sdev(x)`

Example

Where `x` is the array `[1 2 3 4 5 11 12 13 14 34]`, `sdev(x)` returns `9.50789`.

Notes

Mappings on the operand are ignored.

See Also

`max(x)`, `mean(x)`, `median(x)`, `min(x)`, `mode(x)`, `rms(x)`, and `vari(x)`.

setBit(x,n)

An object that returns `x` with the `n`th bit set to `1`.

Use

Use `setBit(x,n)` to set a particular binary digit of a container `x` to `1`. `x` can be any shape and of type Int32, Real, Coord, or Waveform. If `x` is not of type Int32, it is converted to Int32, retaining shape. The return value is of type Int32. `n` must be a container which is, or can be converted to, Int32 and have a value between `0` and `31` inclusive. `n` must be either scalar or match the shape of `x`.

Location

`Math` ⟹ `Bitwise` ⟹ `setBit(x,n)`

Example

`setBit(8,0)` returns `9`.

`setBit(8.24,0.9)` also returns `9`.

Notes

The mappings of the return value are the same as the `x` parameter.

The least significant bit is on the right, and the most significant bit is on the left.

See Also

`bit(x,n)`, `bits(str)`, `Bitwise`, and `clearBit(x,n)`.

Signal Processing

A menu item.

Use

Use `Signal Processing` to access one of several signal processing functions that can be applied to one-dimensional arrays (`Int32`, `Real`, `Complex`, `PComplex`, `Waveform`, `Spectrum`) for some functions, and lists of two-dimensional (two field) coordinates.

- `fft(x)`
- `ifft(x)`
- `convolve(a,b)`
- `xcorrelate(a,b)`
- `bartlet(x)`
- `hamming(x)`
- `hanning(x)`
- `blackman(x)`
- `rect(x)`

Location

`AdvMath` ⟹ `Signal Processing` ⟹

Notes

The functions under `Signal Processing` are applicable to one-dimensional arrays of values that represent ordered, equally spaced data.

signof(x)

An object that returns, as an Int32, the sign of `x`.

Use

Use `signof(x)` to obtain the sign of a container. `x` may be any shape and of the types Int32, Real, Coord, or Waveform. The return value will be an Int32 of the same shape as `x`, with value(s) of `-1`, `0`, or `+1`, depending on whether the `x` value is negative, zero, or positive. If `x` is negative, `-1` is returned; if `x` is zero, `0` is returned; if `x` is positive then `+1` is returned.

Location

`Math` ⟹ `Real Parts` ⟹ `signof(x)`

Example

`signof([23 0 -1])` returns `[1 0 -1]`.

Notes

Mappings are retained in the result.

See Also

`abs(x)`, `Complex Parts`, and `mag(x)`.

sin(x)

An object that returns the sine of `x`.

Use

Use `sin(x)` to generate the sine of the `x` data. `x` is assumed to be in the current `Trig Mode` units. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Trig` ⟹ `sin(x)`

Example

`sin([0 PI/2])` returns `[0 1]` with `Trig Mode` set to Radians.

`sin((1, @PI))` returns `(0.84147, @3.14159)` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`asin(x)`, `cos(x)`, `tan(x)`, `sinh(x)`, and `Trig`.

`Trig Mode` in the "General Reference" chapter.

sinh(x)

An object that returns the hyperbolic sine of `x`.

Use

Use `sinh(x)` to generate the hyperbolic sine of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. `x` is assumed to be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Hyper Trig` ⟹ `sinh(x)`

Example

`sinh(1.2)` returns `1.509` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`asinh(x)`, `cosh(x)`, `Hyper Trig`, `sin(x)`, and `tanh(x)`.

`Trig Mode` in the "General Reference" chapter.

sort(x,direction,field)

A function that sorts an array of data (any data type).

Use

Use the `sort` function in a `Formula` object to sort an array of data on input `x`. All data types are allowed, and by default the data is sorted in ascending order. If you use `sort` in a `Formula` object the syntax is:

`sort(x` [`,`*direction* [`,`*field*]] `)`

where:

- *direction* (optional) specifies the direction of sort. The default is ascending order, and any non-zero value will specify ascending order. A value of `0` specifies a sort in descending order.
- *field* (optional) specifies the field on which to sort the array:
  - For the Real, Integer, and Text data types, this parameter is ignored.
  - For the Complex data type, specify `0` or `"real"` to sort on the real field. Specify `1` or `"imag"` to sort on the imaginary field. All three parameters must be specified.
  - For the PComplex data type, specify `0` or `"mag"` to sort on the magnitude field. Specify `1` or `"phase"` to sort on the phase field. All three parameters must be specified.
  - For the Coordinate data type, the array will be sorted by the dependent field by default (`"y"` in an x-y pair, `"z"` in an x-y-z triplet). Specify `0` or `"x"` to sort on the x field, `1` or `"y" to sort on the y` field, and so forth.
  - For the Record data type, the array will be sorted by the first field by default. Or you can specify the field by number (`0`, `1`, and so forth) or by name (`"operator"`, `"date"`, etc.).

Note

For `Coordinate` data type, the field names will *always* be `x`,`y`, `z`,`w`, `v`,`u`, `t`,`s`, `r`,`q` even though you may change the input terminal names.

sort(x,direction,field)

Location

`AdvMath` ⟹ `Array` ⟹ `sort`

Examples

`sort(x)` will sort a Real array `x` in ascending order.

`sort(x,0)` will sort a Real array `x` in descending order.

`sort(x,1,"real")` or `sort(x,1,0)` will sort a Complex array `x` by the real field in ascending order.

`sort(x,0,"phase")` or `sort(x,0,1)` will sort a PComplex array `x` by the phase field in descending order.

`sort(x,1,"A")` or `sort(x,1,0)` will sort a Coordinate array `x` by the "A" field in ascending order.

`sort(x,1,1)` will sort a Record array `x` by the second field in ascending order.

Notes

Text data is sorted by the standard ASCII character-code order. Thus, uppercase letters come before lowercase letters.

See Also

`Formula` in the "General Reference", Chapter 2, and String Functions.

sq(x)

An object that returns the square of the value of `x`.

Use

Use `sq(x)` to perform the squaring of a number `x`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The `sq(x)` function multiplies the value of `x` times itself. The return value is of the same type and shape as `x`.

Location

`Math` ⟹ `Power` ⟹ `sq(x)`

Example

`sq([1 4])` returns `[1 16]`.

`sq((1,2))` returns the complex number `(-3,4)`.

Notes

Mappings are retained in the result.

See Also

`Power` and `sqrt(x)`.

sqrt(x)

An object that returns the square root of the value of `x`.

Use

Use `sqrt(x)` to generate the square root of a number `x`. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. For PComplex, the `sqrt(x)` is defined as the square root of the magnitude and half the phase. Complex is converted to PComplex before the function is applied. Int32 arguments will return a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Power` ⟹ `sqrt(x)`

Example

`sqrt( (16, @90) )` will return `(4, @45)`.

`sqrt( (1,1) )` will return `(1.1, 0.455)`.

`sqrt([4 9 64])` will return `[2 3 8]`.

Notes

Mappings are retained in the result.

See Also

`cubert(x)`, `^ (exponent)`, `Power`, and `sq(x)`.

Statistics

A menu item.

Use

Use the `Statistics` functions to calculate these common statistical parameters on data.

- `min(x)`
- `max(x)`
- `median(x)`
- `mode(x)`
- `mean(x)`
- `sdev(x)`
- `vari(x)`
- `rms(x)`

Location

`AdvMath` ⟹ `Statistics` ⟹

See Also

`Matrix`.

strDown(str)

An object that changes all uppercase alphabetic characters in a string to lowercase characters.

Use

Use `strDown(str)` to change the case of an input string `str` to lowercase. All uppercase alphabetic characters ("`A`" through "`Z`") are converted to the corresponding lowercase characters ("`a`" through "`z`"). Non-alphabetic ASCII characters are left unchanged.

Location

`Math` ⟹ `String` ⟹ `strDown(str)`

Examples

`strDown("ABCdefg%+#")` returns "`abcdefg%+#`"

`strDown("123456")` returns "`123456`"

See Also

`strRev(str)`, `strTrim(str)`, `strUp(str)`, and `String`.

strFromLen(str,from,len)

An object that returns a substring of a specified length and starting point.

Use

Use `strFromLen(str,from,len)` to return a substring from the input string `str`, beginning at index position `from` and of length `len` in bytes. If `from` = 0, the substring starts at the beginning of the input string. Base 0 indexing of the string is used.

Location

`Math` ⟹ `String` ⟹ `strFromLen(str,from,len)`

Notes

If the value of `len` is less than or equal to 0, a nil string will be returned.

If the value of `from` is greater than the length of the input string `str`, a nil string will be returned.

If the value of `len` is greater than the length of the input string `str`, the returned string stops at the end of the input string.

Examples

`strFromLen("Now is the time",0,6)` returns "`Now is`"

`strFromLen("Now is the time",6,9)` returns " `the time`"

`strFromLen("Now is the time",0,30)` returns "`Now is the time`"

See Also

`strFromThru(str,from,thru)`, `strPosChar(str,char)`, `strPosStr(str1,str2)`, and `String`.

strFromThru(str,from,thru)

An object that returns a substring of a specified starting point and ending point.

Use

Use `strFromThru(str,from,thru)` to return a substring from the input string `str`, beginning at index position `from` and ending at index position `thru`. If `from` = `0`, the substring starts at the beginning of the input string. Base 0 indexing of the string is used.

Location

`Math` ⟹ `String` ⟹ `strFromThru(str,from,thru)`

Notes

Negative values of `from` or `thru` are converted to `0`.

Values of `from` or `thru` that are greater than the length of the input string `str` are converted to the index position of the last character in the input string.

If the value of `from` is greater than the length of the input string `str`, or greater than `thru`, a nil string is returned.

Examples

`strFromThru("Now is the time",0,6)` returns "`Now is `"

`strFromThru("Now is the time",0,0)` returns "`N`"

`strFromThru("Now is the time",6,9)` returns "` the`"

`strFromThru("Now is the time",0,30)` returns "`Now is the time`"

See Also

`strFromLen(str,from,len)`, `strPosChar(str,char)`, `strPosStr(str1,str2)`, and `String`.

String

A menu item.

Use

Use the `String` functions to perform these operations on string data:

- `strDown(str)`
- `strFromLen(str,from,len)`
- `strFromThru(str,from,thru)`
- `strLen(str)`
- `strPosChar(str,char)`
- `strPosStr(str1,str2)`
- `strRev(str)`
- `strTrim(str)`
- `strUp(str)`

Location

`Math` ⟹ `String` ⟹

strLen(str)

An object used to determine the length of a string in bytes.

Use

Use `strLen(str)` to determine the length of an input string `str`. The return value is an Int32 Scalar giving the number of bytes present. Kanji characters and other 16-bit characters count as two bytes each.

Location

`Math` ⟹ `String` ⟹ `strLen(str)`

Examples

`strLen("ABCdefg%+#")` returns `10`

`strLen("123456")` returns `6`

See Also

`String`.

strPosChar(str,char)

An object that returns the index within the input string `str` of any character in the character list `char`.

Use

Use `strPosChar(str,char)` to return the index within the input string `str` of the first occurrence of any character in character list `char`. Base 0 indexing of the string is used.

Location

`Math` ⟹ `String` ⟹ `strPosChar(str,char)`

Notes

If there is more than one occurrence of more that one listed character, only the index of the first listed character found is returned.

If none of the characters listed in `char` are found in the input string `str`, a value of `-1` is returned.

If either `str` or `char` is the nil string, a value of `-1` is returned.

Note that the character match is case sensitive.

Examples

`strPosChar("Now is the time","Nwme")` returns `0`

`strPosChar("Now is the time","nwme")` returns `2`

`strPosChar("Now is the time","x yz ")` returns `3`

`strPosChar("Now is the time","")` returns `-1`

`strPosChar("Now is the time","xyz")` returns `-1`

See Also

`strFromLen(str,from,len)`, `strFromThru(str,from,thru)`, `strPosStr(str1,str2)`, and `String`.

strPosStr(str1,str2)

An object that returns the index of string `str2` within string `str1`.

Use

Use `strPosStr(str1,str2)` to return the index of the first occurrence of string `str2` within string `str1`—that is, the index position within `str1` where `str2` begins. Base 0 indexing is used.

Location

`Math` ⟹ `String` ⟹ `strPosStr(str1,str2)`

Notes

If `str2` occurs more than once in `str1`, only the index of the first occurrence is returned.

If an occurrence of `str2` is not found within `str1`, a value of `-1` is returned.

If either `str1` or `str2` is the nil string, a value of `-1` is returned.

Note that the character match is case sensitive.

Examples

`strPosStr("Now is the time","Now")` returns `0`

`strPosStr("Now is the time","is the")` returns `4`

`strPosStr("Now is the time"," ")` returns `3`

`strPosStr("Now is the time","")` returns `-1`

`strPosStr("Now is the time","is The")` returns `-1`

See Also

`strFromLen(str,from,len)`, `strFromThru(str,from,thru)`, `strPosChar(str,char)`, and `String`.

strRev(str)

An object used to reverse the order of a string.

Use

Use `strRev(str)` to reverse the order in which the characters in a string appear.

Location

`Math` ⟹ `String` ⟹ `strRev(str)`

Examples

`strRev("ABCdefg%+#")` returns "`#+%gfedCBA`"

`strRev("123456")` returns "`654321`"

See Also

`strDown(str)`, `strTrim(str)`, `strUp(str)`, and `String`.

strTrim(str,trimlist)

An object used to trim spaces and tab characters from the front and back of a string.

Use

Use the `strTrim(str)` object to trim leading and trailing spaces and tab characters from an input string `str`. All spaces and tab characters that precede the first non-space/non-tab character, or follow the last non-space/non-tab character, are trimmed. In a `Formula` object the complete syntax is:

`strTrim(str` [`,`*trimlist*] `)`

Use `strTrim(str,trimlist)` to trim off any characters from `str` that are contained in `trimlist`. Note that these characters are trimmed only if they are leading or trailing characters in `str`.

Location

`Math` ⟹ `String` ⟹ `strTrim(str)`

Notes

You can extend the syntax of this function to `strTrim(str,trimlist)` by using it in a `Formula` object with two inputs: `str` and `trimlist`. In this case, any characters listed in the `trimlist` string, if leading or trailing, are trimmed from the `str` input string.

Examples

`strTrim("    ABCdefg%+#")` returns "`ABCdefg%+#`"

`strTrim("eeeABCdefg%+#","eg#")` returns "`ABCdefg%+`"

See Also

`strDown(str)`, `strRev(str)`, `strUp(str)`, and `String`.

strUp(str)

An object used to change all lowercase alphabetic characters in a string to uppercase characters.

Use

Use `strUp(str)` to change the case of an input string `str` to uppercase. All lowercase alphabetic characters ("`a`" through "`z`") are converted to the corresponding uppercase characters ("`A`" through "`Z`"). Non-alphabetic ASCII characters are left unchanged.

Location

`Math` ⟹ `String` ⟹ `strUp(str)`

Examples

`strUp("ABCdefg%+#")` returns "`ABCDEFG%+#`"

`strUp("123456")` returns "`123456`"

See Also

`strDown(str)`, `strRev(str)`, `strTrim(str)`, and `String`.

- (subtract)

An object that performs an arithmetic subtraction on two operands.

Use

Use - to subtract the value of one container from another container. The two containers may be of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. The two containers may be any shape. But if one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is a container of the highest type, with the same shape as the operands.

If both operands are of type Coord, they must have their independent variable(s) match exactly or an error is returned. The subtraction is only performed on the dependent (last) variable.

Location

Math ⟹ + - * / ⟹ -

Example

Array minus a scalar: `[1 2 3] - 3` returns `[-2 -1 0]`.

Scalar minus an array: `3 - [1 2 3]` returns `[2 1 0]`.

Two arrays: `[4 5 6] - [3 2 1]` returns `[1 3 5]`.

Two Complex scalars: `(2,4) - (1,3)` returns `(1,1)`.

Two Coord scalars: `coord(1,3) - coord(1,5)` returns `coord(1,-2)`.

Two Coord scalars: `coord(1,3) - coord(2,5)` returns an error.

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings. The return value will have the same mappings as the operands; otherwise an error is returned.

See Also

`+ (add)`, `/ (divide)`, and `* (multiply)`.

sum(x)

An object used to sum up (add) all the elements of the input array.

Use

Use `sum(x)` to add together all the elements of the input array. The `x` input may be of any size and shape and of the type Int32, Real, Coord, Waveform, Coord, Complex, PComplex, Spectrum, or Text. For `x` input of all types, except Waveform and Spectrum, output of the same type as the input is returned. An input of a Waveform or Coord returns a Real scalar. An input of Spectrum returns a PComplex.

The output of this object (for numeric types) is the arithmetic sum of all the data contained in the input container and is a Scalar. For Text type input, the output is a Scalar Text container and contains the concatenation of all of the Text elements together to form one large string.

The `sum(x)` object obeys the same rules for Text string addition as the dyadic operator `+`.

Location

`AdvMath` ⟹ `Array` ⟹ `sum(x)`

Example

Where `x` is an array `[1 2 3 4]`, `sum(x)` returns `10`.

Where `x` is an array `["first" "second" "third"]`, `sum(x)` returns `"firstsecondthird"`.

Notes

Mappings on the operand are ignored.

Note the difference between the `concat(x,y)` and `sum(x)` functions on text inputs. The `concat(x,y)` creates an Array 1D in all cases. The `sum(x)` function will simply link all the text strings together to form one large output string. That is, `concat(x,y)`, where `x` is the Scalar Text value `"a"` and `y` is the Scalar Text value `"b"`, yields the Array 1D `["a","b"]`. On the other hand,

sum(x)

`sum(["a","b"])` will return the Scalar container with the Text value of `"ab"` in it.

See Also

`+ (add)`, `concat(x,y)`, and `prod(x)`.

tan(x)

An object that returns the tangent of `x`.

Use

Use `tan(x)` to generate the tangent of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. `x` is assumed to be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Trig` ⟹ `tan(x)`

Example

`tan([0 45])` returns `[0 1]` with `Trig Mode` set to Degrees.

`tan((1, 2))` returns `(33.8128m, 1.01479)` with `Trig Mode` set to Radians.

`tan((3, @PI/2))` returns `(0.99505, @1.57079)` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`atan(x)`, `atan2(y,x)`, `cos(x)`, `cot(x)`, `sin(x)`, `tanh(x)`, and `Trig`.

`Trig Mode` in the "General Reference" chapter.

tanh(x)

An object that returns the hyperbolic tangent of `x`.

Use

Use `tanh(x)` to generate the hyperbolic tangent of the `x` data. `x` can be any shape and of type Int32, Real, Coord, Waveform, Complex, PComplex, or Spectrum. `x` is assumed to be in the current `Trig Mode` units. Int32 returns a Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Hyper Trig` ⟹ `tanh(x)`

Example

`tanh(0.6)` returns `0.537` with `Trig Mode` set to Radians.

Notes

Mappings are retained in the result. Using `Trig Mode` set to anything except Radians may result in accuracy errors beyond the 12th significant digit.

See Also

`atanh(x)`, `cosh(x)`, `Hyper Trig`, `sinh(x)`, and `tan(x)`.

`Trig Mode` in the "General Reference" chapter.

Time & Date

A menu item.

Use

Use `Time & Date` to access the following objects which perform date and time conversions.

- `now()`
- `wday(aDate)`
- `mday(aDate)`
- `month(aDate)`
- `year(aDate)`
- `dmyToDate(d,m,y)`
- `hmsToSec(h,m,s)`
- `hmsToHour(h,m,s)`

Location

`Math` ⟹ `Time & Date` ⟹

Notes

The `now()` and `dmyToDate(d,m,y)` functions will output a time as the number of seconds since the beginning of the Epoch (00:00 hours 1 January 0001 AD) and is stored in UTC (Universal Coordinated Time). That is, a correction is applied for the time zone difference between your local time and GMT (Greenwich Mean Time). Since the `wday()`, `mday()`, `month()`, and `year()` functions expect a UTC type of time as input (from the `now()`, `dmyToDate,()`, `Time Stamp` or `Date Time` constant), they will remove the correction before doing the calculation.

The `hmsToSec(h,m,s)` and `hmsToHour(h,m,s)` are only doing conversion to delta (elapsed) time and do not apply any UTC correction. You may notice that output from one of these functions displayed in an alpha display (in some Time format) is off by the number of hours between your time zone and UTC. You will need to display this information with the `Delta Time:` time selection on the alpha `Time Stamp` display on transactions.

See Also

`Time Stamp` and `To String` in the "General Reference" chapter.

totSize(x)

An object that returns an integer value that is the total size of a container.

Use

Use the `totSize(x)` function in a `Formula` object to determine the size of the container on input `x`. If `x` is a scalar or an array with one element, `totSize(x)` will return a value of 1.

Location

`AdvMath` ⟹ `Array` ⟹ `TotSize`

Examples

For `x`= scalar: `totSize(x)` returns `1`.

For `x`= [1]: `totSize(x)` returns `1`.

For `x`= [1 2]: `totSize(x)` returns `2`.

For `x`= [[1 2] [3 4]]: `totSize(x)` returns `4`.

Notes

The `totSize(x)` function performs the same function as the `TotSize` pin in the `Get Values` object.

See Also

`Formula`, `Get Values`.

transpose(x)

An object used to calculate the transpose of a 2D array (a matrix).

Use

Use `transpose(x)` to return the transpose of a matrix. The `x` input must be of matrix shape and of the type Int32, Real, Complex, PComplex, or Text. For `x` input of all types, the same output type is returned. The matrix shape required for this function is any 2-Dimensional matrix. It does not have to be square.

Location

`AdvMath` $\Longrightarrow$ `Matrix` $\Longrightarrow$ `transpose(x)`

Example

Where `x` is a square matrix `[ [1 2] [3 4] ]`, `transpose(x)` returns `[ [1 3] [2 4] ]`.

Where `x` is a matrix `[ [1 2 3] [4 5 6] ]`, `transpose(x)` returns `[ [1 4] [2 5] [3 6] ]`.

Notes

Mappings on the operand are ignored and the output container has its mappings transposed.

See Also

`det(x)`, `identity(x)`, and `inverse(x)`.

Triadic Operator

An operator that returns the result of one expression if a condition is true, but returns the result of another expression if that condition is false.

Use

Use the triadic operator in a `Formula` object with the following syntax:

(*condition* `?` *expression* `:` *expression*)

- If the *condition* is true, the result of the first *expression* is returned.
- If the *condition* is false, the result of the second *expression* is returned.

Location

Must be used within a `Formula` object.

Example

`(A<10 ? A*2 : A*3)` returns the result of `A*2` if `A<10` is true, but returns the result of `A*3` if `A<10` is false.

Notes

The enclosing parentheses are required.

The triadic operator is useful in setting up "IF/THEN/ELSE" conditional tests.

See Also

`Formula`, `If/Then/Else`.

Trig

A menu item.

Use

Use `Trig` to access the following objects which perform trigonometric functions on data.

- `sin(x)`
- `cos(x)`
- `tan(x)`
- `cot(x)`
- `asin(x)`
- `acos(x)`
- `atan(x)`
- `acot(x)`
- `atan2(y,x)`

Location

`Math` ⟹ `Trig` ⟹

Notes

All `Trig` operations are performed in Radians. If `Trig Mode` is set to Degrees, all containers must be converted from Degrees to Radians before the function can be performed. Therefore, for the best performance, use `Trig Mode` set to Radians.

See Also

`Hyper Trig`.

`Trig Mode` in the "General Reference" chapter.

vari(x)

An object used to return the variance of the data in the input container.

Use

Use `vari(x)` to calculate the variance of the data in the input container.

The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For all `x` input types, a Real Scalar container is returned. For Coord input types, the operation is done on the dependent variable.

The `vari(x)` is defined as:

`[The SUM of (Xi - Xmean)^2] divided by (numPts - 1)`

where `Xi` is the individual data point, `Xmean` is the mean of the data points, and `numPts` is the number of elements in the input array `x`.

Location

`AdvMath` ⟹ `Statistics` ⟹ `vari(x)`

Example

Where `x` is the array `[1 2 3 4 5 11 12 13 14 34]`, `vari(x)` returns `90.4`.

Notes

Mappings on the operand are ignored.

See Also

`max(x)`, `mean(x)`, `median(x)`, `min(x)`, `mode(x)`, `rms(x)`, and `sdev(x)`.

wday(aDate)

An object that returns the weekday of the time `x`.

Use

Use `wday(aDate)` to transform the given time `x` into a container of the same shape with value(s) `0 - 6`, corresponding to the day of the week. The value `0` corresponds to Sunday, `1` to Monday, and so on, with `6` as Saturday. `x` must be of type Int32, Real, Coord, or Waveform any shape. Int32 returns Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Time & Date` ⟹ `wday(aDate)`

Example

`wday(dmyToDate(25,12,1991))` returns `3`, so Dec 25, 1991 is a Wednesday.

Notes

Mappings are retained in the return value.

See Also

`dmyToDate(d,m,y)`, `hmsToHour(h,m,s)`, `hmsToSec(h,m,s)`, `mday(aDate)`, `month(aDate)`, `now()`, `Time & Date`, and `year(aDate)`.

xcorrelate(a,b)

An object used to cross-correlate two arrays of data.

Use

Use `xcorrelate(a,b)` to calculate the discrete cross-correlation of two 1D arrays `a` and `b`. The result will be an array of the same type and of size `(Na+Nb)-1`, where `Na` is the size of input array `a`, and `Nb` is the size of input array `b`.

The input values `a` and `b` must be an Array 1D of type Int32, Real, Coord, or Waveform. The return type is the same as the highest type of the inputs, except Int32, which returns Real. `a` and `b` do not have to be the same size. The resultant values are not normalized.

Location

`AdvMath` ⟹ `Signal Processing` ⟹ `xcorrelate(a,b)`

Example

`xcorrelate([1 2 3 4 5],[3 2 1])` returns `[1 4 10 16 22 22 15]`.

Notes

The inputs `a` and `b` must represent equally spaced data. In addition, the interval between any two values of `a` must be the same as that between any two values of `b`. For two unmapped arrays, it is assumed that the interval is always `1`. For mapped arrays, the interval is `(Xmax-Xmin)/N`, where `Xmax` and `Xmin` are the mappings and `N` is the size of the array. When one input is mapped and the other is not, the unmapped input is assumed to be sampled at the same frequency as the mapped input. The resultant values will not be normalized.

Note that the `xcorrelate(a,b)` algorithm is essentially identical to the standard algorithm for discrete convolution, except that the second operand is not sequence-reversed as it is in convolution.

See Also

`convolve(a,b)`.

xlogRamp (numElem,from, thru)

An object that generates a logarithmically ramped array in the same manner used to generate `x` values for coordinate conversions of log mapped 1D Arrays.

Use

Use `xlogRamp(numElem,from,thru)` to generate a Real one-dimensional array of length `numElem`, with the values logarithmically ramped from `from` to `thru`. `numElem` must be a scalar container which is, or can be converted to, Int32 and with a value greater than zero. `from` and `thru` must be scalar containers which are, or can be converted to, Real. If `from` is less than `thru` the ramping is positive; otherwise it automatically ramps negatively. Both `from` and `thru` must have values greater than zero.

Location

`Math` ⟹ `Generate` ⟹ `xlogRamp(numElem,from,thru)`

Example

`xlogRamp(3, 1, 100)` returns a 1D Array with values `[1 10 100]`.

Notes

The return value has no mappings.

The algorithm for generating values is:

```
Y[I]=exp10(log10(from)+I*((log10(thru)-log10(from))/numElem))
```

```
for I=0..numElem-1
```

This has the effect that the last element in the resultant array has a value which is less than `thru` by:

```
exp10(log10(thru)-log10(from)/numElem).
```

See Also

`Alloc Real`, `logRamp(numElem,from,thru)`, and `xramp(numElem,from,thru)`.

XOR

An object that performs a logical exclusive OR operation on two operands.

Use

Use `XOR` to determine whether one and only one of the value(s) of two containers is logically true (non-zero). The two containers may be of any type and of any shape. If one of the containers is an array, the other must be either a scalar or an array of the same size and shape. The result is an Int32 of the same shape as the operands, with value(s) `0` or `1`. If either operand is non-zero, but not both, the value of the `XOR` operation is `1`; otherwise the value is `0`.

If both operands are of type Coord, they must have their independent variable(s) match exactly or an error is returned. Only the dependent (last) variable is considered for the `XOR` operation.

For Complex, PComplex, and Spectrum containers, the value of the operand is true if either part is non-zero. Text is true if non-null. Enums are converted to Text for the operation.

Location

`Math` ⟹ `Logical` ⟹ `XOR`

Example

A scalar and an array: `3 XOR [3 3 3]` returns `[0 0 0]`.

A scalar and an array: `3 XOR [-3 0 3]` returns `[0 1 0]`.

Two arrays: `[1 2 3] XOR [0 1 (-1)]` returns `[1 0 0]`.

Two PComplex scalars: `(1,@90) XOR (1,@85)` returns `0`.

Two Complex scalars: `(0,0) XOR (0,1)` returns `1`.

Two Complex scalars: `(0,1) XOR (1,0)` returns `0`.

Two Complex scalars: `(0,0) XOR (0,0)` returns `0`.

Two Coord scalars: `coord(1,3) XOR coord(1,5)` returns `0`.

Two Coord scalars: `coord(1,3) XOR coord(2,3)` returns an error.

A Text scalar and a scalar number: `"too" XOR 0` returns `1`.

A Text scalar and a scalar number: `"" XOR 0` returns `1` because the real `0` is promoted to `"0"`, which is a non-null string.

Notes

If either of the containers is mapped (that is, of type Waveform, Spectrum, Coord, or a mapped array of any other type), the other container must be unmapped or have identical mappings. The return value will have the same mappings as the operands; otherwise an error is returned.

Note that the `If/Then/Else` device requires the expression(s) inside it to evaluate to either a scalar or an array, which is either all zeros or all ones.

See Also

`AND`, `NOT`, `OR`, and `Relational`.

`Conditional` and `If/Then/Else` in the "General Reference" chapter.

xramp(numElem, from,thru)

An object that generates a linearly ramped array in the same manner used to generate `x` values for coordinate conversions of mapped 1D Arrays (for example, Waveform).

Use

Use `xramp(numElem,from,thru)` to generate a Real one-dimensional array of length `numElem`, with the values linearly ramped from `from` to `thru`. `numElem` must be a scalar container that is, or can be converted to, Int32 and with a value greater than zero. `from` and `thru` must be scalar containers that are, or can be converted to, Real.

If `from` is less than `thru`, the ramping is positive; otherwise it automatically ramps negatively.

Location

`Math` ⟹ `Generate` ⟹ `xramp(numElem,from,thru)`

Example

`xramp(3, 0.5, 0)` returns a one-dimensional array with values `[0.5 0.25 0]`.

`xramp(5, 2, 10)` returns a one-dimensional array with values `[2 4 6 8 10]`.

Notes

The return value has no mappings.

The algorithm for generating values is:

```
Y[I] = from + I*(thru-from/numElem)
```

```
for I=0..numElem-1
```

This has the effect that the last element in the resultant array has a value which is less than `thru` by the size of the sampling interval: `(thru-from)/numElem`.

See Also

`Alloc Real`, `ramp(numElem,from,thru)`, and `xlogRamp(numElem,from,thru)`.

y0(x)

An object used to calculate the Bessel function of `x` of the second kind of order zero.

Use

Use `y0(x)` to find the Bessel function of `x` of the second kind of order zero. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` $\Longrightarrow$ `Bessel` $\Longrightarrow$ `y0(x)`

Example

`y0(2)` returns `0.510375672649745`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

The value of `x` must be positive.

See Also

`Ai(x)`, `Bi(x)`, `j0(x)`, `j1(x)`, `jn(x,n)`, `y1(x)`, and `yn(x,n)`.

y1(x)

An object used to calculate the Bessel function of `x` of the second kind of order one.

Use

Use `y1(x)` to find the Bessel function of `x` of the second kind of order one. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

Location

`AdvMath` $\Longrightarrow$ `Bessel` $\Longrightarrow$ `y1(x)`

Example

`y1(9)` returns the value `0.104314575196715`.

Notes

Mappings on the operand are ignored and the output container has the same mappings as the input.

The value of `x` must be positive.

See Also

`Ai(x)`, `Bi(x)`, `j0(x)`, `j1(x)`, `jn(x,n)`, `y0(x)`, and `yn(x,n)`.

year(aDate)

An object that returns the year of the time `x`.

Use

Use `year(aDate)` to transform the given time `x` into a container of the same shape with the value of the year of the time `x`. `x` must be an Int32, Real, Coord, or Waveform any shape. Int32 returns Real; all others will return the same type. All will return the same shape as `x`.

Location

`Math` ⟹ `Time & Date` ⟹ `year(aDate)`

Example

`year(dmyToDate(25,12,1991))` returns `1991`.

Notes

Mappings are retained in the return value.

See Also

`dmyToDate(d,m,y)`, `hmsToHour(h,m,s)`, `hmsToSec(h,m,s)`, `mday(aDate)`, `now()`, `month(aDate)`, `Time & Date`, and `wday(aDate)`.

yn(x,n)

An object used to calculate the Bessel function of `x` of the second kind of order `n`.

Use

Use `yn(x,n)` to find the Bessel function of `x` of the second kind of order `n`. The `x` input may be of any size and shape and of the type Int32, Real, Coord, or Waveform. For `x` input of all types, the same output type is returned, except for Int32 which returns a Real type. For Coord input types, the operation is done on the dependent variable.

The `n` parameter must be of Int32 type or be able to be converted to Int32. The `n` parameter also has to be a Scalar in shape or the same shape as `x`.

Location

`AdvMath` ⟹ `Bessel` ⟹ `yn(x,n)`

Example

`yn(10,4)` returns `-0.144949511868093`.

Notes

If both of the inputs are mapped, then the mappings must be the same. The return value has the same mappings as the input if either input is mapped. If neither of the inputs is mapped, then the output is unmapped.

The value of `x` must be positive.

See Also

`Ai(x)`, `Bi(x)`, `j0(x)`, `j1(x)`, `jn(x,n)`, `y0(x)`, and `y1(x)`.

A

Data Type Conversions

This appendix contains reference information about the data type conversions which occur on input terminals in HP VEE.

In conventional programming languages, you manually convert between data types. HP VEE automatically converts between most data types. These conversions are discussed in the "Formula (Math and AdvMath) Reference" chapter.

Note

Data shapes are not converted on input terminals, but data types and shapes may be automatically converted when used in math functions. These conversions are discussed in the "Formula (Math and AdvMath) Reference" chapter.

Most objects accept any data type, but a few need a particular data type or shape input. For these objects, their data input terminal automatically tries to convert the container to have the desired data type.

For example, a `Magnitude Spectrum` display needs Spectrum data. If the output of a `Function Generator` (a Waveform) is connected to the `Magnitude Spectrum` display, the input terminal of the `Magnitude Spectrum` automatically does an FFT to convert time-domain data to frequency-domain data (Waveform to a Spectrum).

The conversion can be a promotion or demotion. A promotion is the conversion from a data type with less information to one with more. For example, a conversion from an Int32 to Real is a promotion.

A demotion is a conversion that loses data. For example, the conversion from a Real to an Int32 is a demotion because the fractional part of the Real number is lost. Demotion only generally occurs when you have specified a certain data type for an input. HP VEE objects generally accept any data type. For

example, if you change the input on a `Formula` object to Int32, and you try to input a Real number (such as `28.2`), the value that will be input to the object will be `28`.

When the conversion can't be done, HP VEE returns an error. The following table shows when conversion is automatic (yes) or when HP VEE returns an error (no).

Table A-1. Promotion and `Demotion` of Types In Input Terminals

To ▶ ▼ From	Int32	Real	Complex	PComplex	Waveform	Spectrum	Coord	Enum	Text
Int32	n/a	yes	yes(1)	yes(1)	no	no	yes(2)	no	yes
Real	yes(3)	n/a	yes(1)	yes(1)	no	no	yes(2)	no	yes
Complex	no	no(4)	n/a	yes	no	no	no	no	yes
PComplex	no	no(4)	yes	n/a	no	no	no	no	yes
Waveform	yes(3)	yes(8)	no	no	n/a	yes(5)	yes	no	yes
Spectrum	no	no	yes(8)	yes(8)	yes(5)	n/a	yes	no	yes
Coord	no	no	no	no	no	no	n/a	no	yes
Enum	no(6)	no	no	no	no	no	no	n/a	yes
Text	yes(7)	yes(7)	yes(7)	yes(7)	no	no	yes(7)	no	n/a

Notes:

n/a = Not applicable.

(1) An Int32, or Real *value* promotes to Complex (*value*, `0`) or to PComplex (*value*, `@0`).

(2) The independent component(s), which are the first `n-1` field(s) of an `n`-field Coord, are the array indexes of the value unless the array is mapped. If the array is mapped, the independent component(s) are derived from the mappings of each dimension. The dependent component, `y`, is the array element. If the container is a Scalar (non-array), conversion fails with an error.

(3) These demotions will cause an error if the value is out of range for the destination type.

A-2 Data Type Conversions

(4) This demotion is not done automatically, but can be done with the `re(x)`, `im(x)`, `mag(x)`, and `phase(x)` objects or the `Build/UnBuild` $\Longrightarrow$ objects.

(5) An FFT or inverse FFT is automatically done.

(6) This demotion is not done automatically, but can be done with the `ordinal(x)` object.

(7) This demotion causes an error if the text value is not a number (such as `34` or `42.6`) or is not in an acceptable numerical format. The acceptable formats are as follows (spaces, except within each number, are ignored):

- Text that is demoted to an Int32 or Real type may also include:
  - A preceding sign. For example, `-34`.
  - A suffix of `e` or `E` followed by an optional sign or space and an integer. For example, `42.6E-3`.
- Text demoted to Complex must be in the following format: *(number, number).*
- Text demoted to PComplex must be in the following format: *(number, @number).* The phase (the second component) is considered to be radians for this conversion, regardless of the `Trig Mode` setting.
- Text demoted to a Coord type must be in the following format: *(number, number, ...).*

(8) These demotions keep the Waveform and Spectrum mappings.

Note

The `Record` data type has the highest precedence. But non-record data cannot be automatically converted to or from `Record` data type by promotion or demotion. Use the `Build Record` and `UnBuild Record` objects to accomplish this task.

B

ASCII Table

This appendix contains reference tables of ASCII 7-bit codes.

ASCII 7-bit Codes

	Binary	Oct	Hex	Dec	HP-IB Msg
NUL	0000000	000	00	0	
SOH	0000001	001	01	1	GTL
STX	0000010	002	02	2	
ETX	0000011	003	03	3	
EOT	0000100	004	04	4	SDC
ENQ	0000101	005	05	5	PPC
ACK	0000110	006	06	6	
BEL	0000111	007	07	7	
BS	0001000	010	08	8	GET
HT	0001001	011	09	9	TCT
LF	0001010	012	0A	10	
VT	0001011	013	0B	11	
FF	0001100	014	0C	12	
CR	0001101	015	0D	13	
SO	0001110	016	0E	14	
SI	0001111	017	0F	15	
DLE	0010000	020	10	16	
DC1	0010001	021	11	17	LLO
DC2	0010010	022	12	18	
DC3	0010011	023	13	19	
DC4	0010100	024	14	20	DCL
NAK	0010101	025	15	21	PPU
SYN	0010110	026	16	22	
ETB	0010111	027	17	23	

	Binary	Oct	Hex	Dec	HP-IB Msg
CAN	0011000	030	18	24	SPE
EM	0011001	031	19	25	SPD
SUB	0011010	032	1A	26	
ESC	0011011	033	1B	27	
FS	0011100	034	1C	28	
GS	0011101	035	1D	29	
RS	0011110	036	1E	30	
US	0011111	037	1F	31	
space	0100000	040	20	32	listen addr 0
!	0100001	041	21	33	listen addr 1
"	0100010	042	22	34	listen addr 2
#	0100011	043	23	35	listen addr 3
$	0100100	044	24	36	listen addr 4
%	0100101	045	25	37	listen addr 5
&	0100110	046	26	38	listen addr 6
'	0100111	047	27	39	listen addr 7
(	0101000	050	28	40	listen addr 8
)	0101001	051	29	41	listen addr 9
*	0101010	052	2A	42	listen addr 10
+	0101011	053	2B	43	listen addr 11
,	0101100	054	2C	44	listen addr 12
-	0101101	055	2D	45	listen addr 13
.	0101110	056	2E	46	listen addr 14
/	0101111	057	2F	47	listen addr 15

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
0	0110000	060	30	48	listen addr 16
1	0110001	061	31	49	listen addr 17
2	0110010	062	32	50	listen addr 18
3	0110011	063	33	51	listen addr 19
4	0110100	064	34	52	listen addr 20
5	0110101	065	35	53	listen addr 21
6	0110110	066	36	54	listen addr 22
7	0110111	067	37	55	listen addr 23
8	0111000	070	38	56	listen addr 24
9	0111001	071	39	57	listen addr 25
:	0111010	072	3A	58	listen addr 26
;	0111011	073	3B	59	listen addr 27
<	0111100	074	3C	60	listen addr 28
=	0111101	075	3D	61	listen addr 29
>	0111110	076	3E	62	listen addr 30
?	0111111	077	3F	63	UNL
@	1000000	100	40	64	talk addr 0
A	1000001	101	41	65	talk addr 1
B	1000010	102	42	66	talk addr 2
C	1000011	103	43	67	talk addr 3
D	1000100	104	44	68	talk addr 4
E	1000101	105	45	69	talk addr 5
F	1000110	106	46	70	talk addr 6
G	1000111	107	47	71	talk addr 7
H	1001000	110	48	72	talk addr 8

	Binary	Oct	Hex	Dec	HP-IB Msg
I	1001001	111	49	73	talk addr 9
J	1001010	112	4A	74	talk addr 10
K	1001011	113	4B	75	talk addr 11
L	1001100	114	4C	76	talk addr 12
M	1001101	115	4D	77	talk addr 13
N	1001110	116	4E	78	talk addr 14
O	1001111	117	4F	79	talk addr 15
P	1010000	120	50	80	talk addr 16
Q	1010001	121	51	81	talk addr 17
R	1010010	122	52	82	talk addr 18
S	1010011	123	53	83	talk addr 19
T	1010100	124	54	84	talk addr 20
U	1010101	125	55	85	talk addr 21
V	1010110	126	56	86	talk addr 22
W	1010111	127	57	87	talk addr 23
X	1011000	130	58	88	talk addr 24
Y	1011001	131	59	89	talk addr 25
Z	1011010	132	5A	90	talk addr 26
[	1011011	133	5B	91	talk addr 27
\	1011100	134	5C	92	talk addr 28
]	1011101	135	5D	93	talk addr 29
^	1011110	136	5E	94	talk addr 30
_	1011111	137	5F	95	UNT
‘	1100000	140	60	96	secondary addr 0

ASCII 7-bit Codes (continued)

	Binary	Oct	Hex	Dec	HP-IB Msg
a	1100001	141	61	97	secondary addr 1
b	1100010	142	62	98	secondary addr 2
c	1100011	143	63	99	secondary addr 3
d	1100100	144	64	100	secondary addr 4
e	1100101	145	65	101	secondary addr 5
f	1100110	146	66	102	secondary addr 6
g	1100111	147	67	103	secondary addr 7
h	1101000	150	68	104	secondary addr 8
i	1101001	151	69	105	secondary addr 9
j	1101010	152	6A	106	secondary addr 10
k	1101011	153	6B	107	secondary addr 11
l	1101100	154	6C	108	secondary addr 12
m	1101101	155	6D	109	secondary addr 13
n	1101110	156	6E	110	secondary addr 14
o	1101111	157	6F	111	secondary addr 15
p	1110000	160	70	112	secondary addr 16
q	1110001	161	71	113	secondary addr 17
r	1110010	162	72	114	secondary addr 18
s	1110011	163	73	115	secondary addr 19
t	1110100	164	74	116	secondary addr 20
u	1110101	165	75	117	secondary addr 21
v	1110110	166	76	118	secondary addr 22
w	1110111	167	77	119	secondary addr 23
x	1111000	170	78	120	secondary addr 24
y	1111001	171	79	121	secondary addr 25
z	1111010	172	7A	122	secondary addr 26
{	1111011	173	7B	123	secondary addr 27
|	1111100	174	7C	124	secondary addr 28
}	1111101	175	7D	125	secondary addr 29
~	1111110	176	7E	126	secondary addr 30
[del]	1111111	177	7F	127	

Glossary

This Glossary contains the terms and definitions used to name or describe graphical objects and processes in the HP VEE software, as well as some hardware items related to installing and using HP VEE.

Activate

1. To send a container to a terminal. See also **Container** and **Terminal**.

2. The action that resets the context of a `UserObject` before it operates each time. See also **Context** and **PreRun**.

Application

A software program that completes work directly for the user. For example, HP VEE-Engine is a general purpose engineering application, and HP VEE-Test is a test and measurement application.

Array

A data shape that contains a systematic arrangement of data items in one or more dimensions. The data items are accessed via indexes. See also **Data Shape**.

Arrow

1. An arrow-shaped pointer. See **Pointer**.

2. A scroll arrow that is a part of a scroll bar and is used either to scroll a list box, or to pan the work area.

Asynchronous

A method of operating without a common signal to synchronize events; rather, the events occur at unspecified times. Control pins in HP VEE are asynchronous.

Auto Execute
An option on the object menus of the data constant objects. When `Auto Execute` is set, the object operates when its value is edited.

Bitmap
A bit pattern or picture. In HP VEE you can display a bitmap on an icon.

Buffer
An area where information is stored temporarily.

Button
1. A button on a mouse. See **Mouse Button**.

2. A graphical object in HP VEE that simulates a real-life pushbutton and appears to pop out from your screen. When a button is "pressed" in HP VEE, by clicking on it with a mouse, an action occurs.

Cascading Menu
A submenu on a pull-down or pop-up menu that provides additional selections to a menu selection (feature).

Checkbox
A recessed square box on HP VEE menus and dialog boxes that allows you to select a setting. To select a setting, click on the box and a checkmark appears in the box to indicate a selection has been made. To cancel the setting, simply click on the box again to remove the checkmark.

Click
To press and release a mouse button quickly. Clicking usually selects a menu feature or object in the HP VEE window. See also **Double-Click** and **Drag**.

Compiled Function
A user-defined function created by dynamically linking a program, written in a programming language such as C, into the HP VEE process. The user must create a shared library file and a definition file for the program to be linked. The `Import Library` object attaches the shared library to the HP VEE process and parses the definition file declarations. The Compiled Function can then be called with the `Call Function` object, or from certain expressions. See also **User Function** and **Remote Function**.

Component
A single instrument function or measurement value in an HP VEE-Test `State Driver` or `Component Driver`. For example, a voltmeter driver contains components that record the range, trigger source, and latest reading. See also **Component Driver**, **Driver Files**, **State**, and **State Driver**.

Component Driver
An instrument control object that reads and writes values to components you specifically select. Use `Component Drivers` to control an instrument using a driver by setting the values of only a few components at a time. See also **Component**, **Driver Files**, and **State Driver**.

Composite Data Type
A data type that has an associated shape. See also **Data Shape** and **Data Type**.

Configure
To arrange or modify software, hardware, or both in a computer system. In HP VEE, a menu selection with which you may change transaction array formats, the number of elements in a constant, and so forth.

Container
The package that is transmitted over lines and is processed by objects. Each container contains data, the data type, and the data shape. See also **Data Shape** and **Data Type**.

Context
A level of the work area that can contain other levels of work areas (such as nested `UserObjects`) but is independent of them. See also **UserObject**.

Control Pin
An asynchronous input pin that transmits data to the object without waiting for the object's other input pins to contain data. For example, control pins in HP VEE are commonly used to clear or autoscale a display.

Coupling
The interrelationship of certain functions in a test and measurement instrument. If in a state or component driver, functions A and B are coupled, changing the value of A may automatically change the value of B, even though you do not change B explicitly.

Crosshairs

A cross-shaped pointer in HP VEE that indicates that the software is waiting for your action. For example, when you see the crosshairs, you can select an object in the work area to perform some action on it, select a menu feature, or select any of the window controls (such as the scroll arrows, minimize or maximize buttons, and so forth).

Cursor

A white rectangular pointer in an entry field that shows where alphanumeric data will appear when you type information from the keyboard.

Data Field

The field within a transaction specification in which you specify either the expression to be written (WRITE transactions), or the variable to receive data that is read (READ transactions). See also **Transactions**.

Data Flow

The direction in which data moves in HP VEE. Data flows from left to right through objects and propagation has continued as far as it can in this way. Propagation continues through the sequence input and output pins.

Data Input Pin

A connection point on the left side of an object that permits data to flow into the object.

Data Output Pin

A connection point on the right side of an object that propagates data flow to the next object and passes the results of the first object's operation on to the next object.

DataSet

A collection of **Record** containers saved into a file for later retrieval. The `To DataSet` object collects Record data on its input and writes that data to a named file (the DataSet). The `From DataSet` object retrieves Record data from the named file (the DataSet) and outputs that data as Record containers on its `Rec` output pin. See also **Record**.

Data Shape
A pre-defined structure that defines how data is grouped together. See also **Array**, **Container**, and **Data Type**.

Data Type
A pre-defined structure that determines how data is organized and treated by HP VEE and supports common engineering constructs. See also **Container** and **Data Shape**.

Default
A value or action that HP VEE automatically selects.

Default Button
The button in a dialog box who's action is performed by default if Return is pressed or the selection is double-clicked. The default button has a recessed border.

Demote
To convert from a data type that contains more information to one that contains less information. See also **Data Type** and **Promote**.

Detail View
A view of a model in HP VEE that shows all the objects and the lines between them. Compare with **Panel View**.

Device
An instrument attached to or plugged into an HP-IB, RS-232, GPIO or VXI interface. Specific HP VEE-Test objects such as the `Direct I/O` object send and receive information to a device.

Device Driver
See **Interface Driver**.

Dialog Box
A secondary window displayed when HP VEE requires information from you before it can continue. For example, a dialog box may contain a list of files from which you must choose a file before HP VEE can perform a particular operation.

Directory
A collection of files. For example, your `/users` directory usually contains all the files you have created. See **$HOME** and **Startup Directory**.

Double-Click
To press and release a mouse button twice in rapid succession. Double-clicking is usually a short-cut to selecting and performing an action.

Drag
To press *and continue to hold down* a mouse button while moving the mouse. Dragging moves something (for example, an object or scroll slider) within the HP VEE window. A drag ends when you release the mouse button.

Driver
Software that allows a computer to communicate with other software or hardware more easily than with raw reads and writes. See also **Component Driver**, **Driver Files**, **Interface Driver**, and **State Driver**.

Driver Files
A set of files included with HP VEE-Test that contains the information needed to create `State Driver` and `Component Driver` objects for instrument control. These files (`.cid` files) are copied to your system's hard disk automatically when you install HP VEE-Test.

Edit
To make changes in a file or entry field containing text or data.

Entry Field
A field that is typically part of a dialog box or an editable object and is used for text entry. An entry field appears recessed. For example, the open view of the `For Range` object has entry fields where you type values that specify the beginning, ending, and step values.

Error Message
Information that appears in a special type of dialog box in the HP VEE window and explains that a problem has occurred.

Error Pin
A pin that traps any errors that occur in an object. Instead of getting an error message, the error number is output on the error pin. When an error is generated, the data output pins are not activated.

Execute
The action of a model, or parts of a model, running.

Execution Flow
The order in which objects operate. See also **Data Flow**.

Expression
An equation in an entry field that can contain the input terminal names and any `Math` or `AdvMath` functions. An expression is evaluated at run-time. Expressions are only allowed in the `Formula`, `If/Then/Else`, `Get Values`, and I/O transaction objects.

Feature
An item on a menu that you select to cause a particular action to occur (for example, to open a file), or to get a particular object.

Feedback
A continuous thread path of sequence and/or data lines that uses values from the previous execution to change values in the current execution.

File
A set of information (such as a model or data) that is stored in an area of computer storage.

Flow
See **Data Flow** and **Execution Flow**.

Function
The name and action of objects where the output is a function of the input. These objects are located under `Math` or `AdvMath` menus and may be used in the `Formula` object. For example `sqrt(x)` is a function; `+` is not.

Global Variable
A named variable that is set globally, and which can be used by name in any context of an HP VEE model. For example, a global variable can be set

with `Set Global` in the root context of the model, and can be accessed by name with `Get Global` or from certain expressions within the context of a `UserObject`. However, a local variable with the same name as the global variable takes precedence in an expression.

Grayed Feature
A menu feature that is not currently available for use. For example, `Move Object` is grayed when no objects are selected.

Highlight
1. The colored band or shadow around an object that provides a visual cue to the status of the object.

2. The change of color on a menu feature that indicates you are pointing to that feature.

Host
To begin a thread or subthread. For example, the subthread that is hosted by `For Count` is the subthread that iterates.

$HOME
Your home directory (usually `/users`).

HP-UX
Hewlett-Packard Company's enhanced version of the UNIX™ operating system. (UNIX is a trademark of AT&T Bell Laboratories.)

Icon
The small, graphical representation of an HP VEE object, such as the representation of an instrument, a control, or a display. Compare with **Open View**.

Instrument Driver
See **Driver Files**, **Component Driver**, and **State Driver**.

Interface
HP-IB, RS-232, GPIO, and VXI are referred to as interfaces used for I/O. Specific HP VEE-Test objects, such as the `Interface Event` object can only send commands to an interface.

Interface Driver
Software that allows a computer to communicate with a hardware interface, such as HP-IB or RS-232. Also called *device driver* in the UNIX™ operating system, interface drivers are configured into the kernel of the operating system.

Interrupt
A signal that requires immediate attention that may suspend a process, such as the execution of a computer program. An interrupt is usually caused by an event external to that process; after the interrupt is serviced, the process may be resumed.

Label
The text area or name on an icon or button that identifies that object or button.

Library
A collection of often-used objects or small models grouped together for easy access.

Line
A link between two objects in HP VEE that transmits data containers to be processed. See also **Subthread** and **Thread**.

Log In
The process of typing in a valid user name and its associated password (if one exists) to gain access to a computer system.

Login
A valid name and password (if one exists) that you use to log in to a computer system.

Main Menu
The menus located in the HP VEE menu bar. The main menus may be opened by clicking or dragging on the menu titles in the menu bar.

Main Work Area
The area where you create a model. The main work area is the parent context of all other contexts.

Mapping
To associate a set of independent values with an array, when the array is a function of the values.

Maximize
To enlarge a window to its maximum size. You maximize a window by selecting the square button on the right side of the window's title bar. In HP VEE, the `UserObject` has a maximize button.

Menu
A collection of features that are presented in a list. See also **Cascading Menu**, **Main Menu**, **Object Menu**, **Pop-Up Menu**, and **Pull-Down Menu**.

Menu Bar
A rectangular bar at the top of the HP VEE window that contains titles of the pull-down, main menus from which you select features.

Menu Title
The name of a menu within the HP VEE menu bar. For example, `File` or `Edit`.

Minimize
1. To reduce an open view of an object to its smallest size—an icon.
2. To reduce an X11 window to its smallest size—an icon.

Model
In HP VEE, a set of objects connected with lines that simulates an engineering problem and, when run, provides a solution to that problem. You build, modify, and run models in HP VEE by selecting objects from menus and connecting them in the work area. A model can contain multiple threads. You load a model into the HP VEE work area with `Open`. A model includes both the detail and panel views and all related contexts.

Mouse
A pointing device that you move across a surface to move a pointer within the HP VEE window.

Mouse Button
One of the buttons on a mouse that you can click, double-click, or drag to

perform a particular action with the corresponding pointer in the HP VEE window.

Object
A graphical representation of an element in a model, such as an instrument, control, display, or mathematical operator. An object is placed on the work area and connected to other objects to create a model. Objects can be displayed as icons or as open views. See **Icon** and **Open View**.

Object Menu
The menu associated with an object that contains features that operate on the object such as moving, sizing, copying, deleting, and adding inputs to the object. It is accessed by clicking on the upper-left corner of an open view or clicking the right mouse button on any non-field area on the object.

Open
To start an action or begin working with a text, data, or graphics file. When you select `Open` from HP VEE, a model is loaded into the work area.

Open View
The representation of an HP VEE object that is more detailed than an icon. Within the open view, you can modify the operation of the object and change the object's title. Compare with **Icon**. See also **Object**.

Operate
The action of an object processing data and outputting a result. An object operates when its data and sequence input pins have been activated. See **Activate**.

Outline Box
A box that represents the outer edges of an object or set of objects and indicates where the object(s) will be placed in the work area.

Network
A group of computers and peripherals linked together to allow the sharing of data and work loads.

Palette
A set of colors and fonts that is supplied with HP VEE and used in your HP VEE environment.

Panel

Information displayed in the center of the object's open view. In a `UserObject`, the panel contains a work area. In a `For Count` object, the panel contains an entry field. Compare with **Panel View**.

Panel View

The view of a model in HP VEE that shows only those objects needed for the user to run the model and view the resultant data. You create a panel view to meet the needs of your users. Compare with **Detail View** and **Panel**.

Pin

An external connection point on an object to which you can attach a line. See also **Control Pin**, **Data Input Pin**, **Data Output Pin**, **Error Pin**, **Sequence Input Pin**, **Sequence Output Pin**, **Terminal**, and **XEQ Pin**.

Pointer

The graphical image that maps to the movement of the mouse. A pointer allows you to make selections and provides you feedback on a particular process underway. HP VEE has pointers of different shapes that correspond to process modes, such as an arrow, crosshairs, and hourglass. See also **Arrow** and **Crosshairs**. Compare with **Cursor**.

Pop-Up Menu

A menu that provides no visual cue to its presence, but simply pops up when you perform a particular action. For example, the `Edit` menu in HP VEE pops up when you position the pointer in the work area and then click the right mouse button.

PostRun

The set of actions that are performed when the model is stopped.

PreRun

The set of actions that resets the model and checks for errors before the model starts to run.

Priority Thread

A priority thread executes to completion blocking all other parallel threads from executing. Certain of the I/O objects for devices and interfaces will host a priority thread.

Promote
To convert from a data type that contains less information to one that contains more information. See also **Data Type** and **Demote**.

Propagation
The rules that objects and models follow when they operate or run. See also **Data Flow** and **Execution Flow**.

Pterodactyl
Any of various extinct flying reptiles of the order Pterosauria of the Jurassic and Cretaceous periods. Pterodactl are characterized by wings consisting of a flap of skin supported by the very long fourth digit on each front leg.

Pull-Down Menu
A menu that is pulled down from the menu bar when you position the pointer over a menu title and click or drag the left mouse button.

Radio Button
A diamond-shaped button in HP VEE dialog boxes that allows you to select a setting that is mutually exclusive with other radio buttons in that dialog box. To select a setting, click on the radio button. To remove the setting, click on another radio button in the same dialog box.

Record
A data type that has named data fields which can contain multiple values. Each field can contain another Record container, a Scalar, or an Array. The Record data type has the highest precedence of all HP VEE data types. However, data cannot be converted to and from the Record data type through the automatic promotion/demotion process. Records must be built/unbuilt using the using `Build Record` and `UnBuild Record` objects.

Remote Function
A User Function running on a remote host computer, which is callable from the local host. The `Import Library` object starts the process on the remote host and loads the Remote File into the HP VEE process on the local host. You can then call the Remote Function with the `Call Function` object, or from certain expressions. See also **User Function** and **Compiled Function**.

Resource Manager
A program which exists on VXI controllers that runs at start-up and after a VXI system reset. This program initializes and manages the instruments in a VXI card cage.

Restore
To return a minimized window or an icon to its full size as a window or open view by double-clicking on it.

Run
To start the objects on a model or thread operating.

Save
To write a file to a storage device, such as a hard disk, for safekeeping.

Scalar
A data shape that contains a single value. See also **Data Shape**.

Schema
The structure or framework used to define a data record. This includes each field's name, type, shape (and dimension sizes) and mapping.

Screen Dump
A graphical printout of a window or part of a window.

Scroll
The act of using a scroll bar either to move through a list of data files or other choices in a dialog box or to pan the work area.

Scroll Arrow
An arrow that is part of a scroll bar and, when clicked on, moves you through a list of data files or other choices in a dialog box or pans the work area.

Scroll Bar
A graphical device used either to move through a list of data files or other choices in a dialog box or to pan the work area. A scroll bar consists of one or more scroll sliders and scroll arrows.

Scroll Slider
A rectangular bar that is part of a scroll bar and, when dragged, moves you through a list of data files or other choices in a dialog box or pans the work area.

Select
To choose an object, an action to be performed, or a menu item. Usually you select by clicking with your mouse.

Select Code
A number used to identify the logical address of a hardware interface. For example, the factory default select code for most HP-IB interfaces is 7.

Selection
1. A menu selection (feature).
2. An object or action you have selected in the HP VEE window.

Sequence Input Pin
The top pin of an object. When connected, this input pin must be activated before the object will operate.

Sequence Output Pin
The bottom pin of an object. When connected, this output pin is activated when the object and all data propagation from that object finishes executing.

Sequencer
An object that controls execution flow through a series of sequence transactions, each of which may call a **User Function**, **Compiled Function**, or **Remote Function**. The sequencer is normally used to perform a series of tests by specifying a series of sequence transactions.

Shell
The program that interfaces between the user and the operating system.

Shell Prompt
The character or characters that denote the place where you type commands while at the operating system shell level. The prompt you see displayed depends upon the type of shell you are running, such as a **#** prompt for the Bourne shell.

Sleep
An object sleeps during execution when it is waiting for an operation or time interval to complete or for an event to occur. A sleeping object will allow other parallel threads to run concurrently. Once the event, time interval, or operation occurs, the object will execute, allowing execution to continue.

Startup Directory
The directory from which you type `veeengine` or `veetest`. This directory determines the default paths for most file actions including `Save` and `Open`.

State
A particular set of values for all of the components related to an HP VEE-Test instrument driver which represents the measurement state of an instrument. For example, a digital multimeter uses one state for high-speed voltage readings and a different state for high-precision resistance measurements. See also **State Driver**.

State Driver
An instrument control object that forces all the function settings in the corresponding physical instrument to match the settings in the control panel displayed in the open view of the object. See also **Component Driver**, **Driver Files**, and **State**.

Step
The action of operating one object at a time. An arrow points to the object that will operate next.

Submenu
See **Cascading Menu**.

Subthread
A portion of a thread.

Synchronous
A method of execution that requires all events to occur before operation.

Terminal
The internal representation of a pin that displays information about the pin

and the data container held by the pin. Double-click on the terminal to view the container information.

Thread
A set of objects connected by solid lines in an HP VEE model. A model with multiple threads can run all threads simultaneously.

Title Bar
The rectangular bar at the top of the HP VEE window or the object's open view where the model's or object's name is shown.

Transactions
The specifications for input and output (I/O) used by certain objects in HP VEE, such as `To File` and `From File`, as well as by `Direct I/O` and `Sequencer`. Transactions appear as English-like phrases listed in the open view of I/O objects.

User-Defined Function
HP VEE allows three types of user-defined functions: the **User Function**, **Compiled Function**, and **Remote Function**. See also **Function**, **User Function**, **Compiled Function**, and **Remote Function**.

User Function
A user-defined function created from a **UserObject** by executing `Make UserFunction`. The User Function exists in background, but provides the same functionality as the original UserObject. You can call a User Function with the `Call Function` object, or from certain expressions. A User Function can be created and called locally, or it can be saved in a library and imported into an HP VEE model with `Import Library`. See also **Compiled Function**, **Remote Function**, and **UserObject**.

User Interface
The part of an application that permits a user and the application to communicate with each other to perform certain tasks. HP VEE uses a graphical user interface, which includes windows, menus, dialog boxes, and objects.

UserObject
An object that can encapsulate a group of objects that perform a particular function. A `UserObject` allows you to use top-down design techniques when

building a model and to build user-defined objects that can be saved in a library and reused. See also **Context**.

View
See **Detail View**, **Icon**, **Open View**, and **Panel View**.

Wait
See Sleep.

Window
A rectangular area on the screen that contains a particular application program, such as HP VEE or the shell.

Window Frame
The area surrounding a window that contains a resize border, window menu button, minimize and maximize buttons, and a title area.

Work Area
The area within the HP VEE window or the open view of a `UserObject` where you group objects together. When you `Open` a model, it is loaded into the main work area. The panel of a `UserObject` is a work area. See also **Panel**.

X Window System (X11)
An industry-standard windowing system used on UNIX™ computer systems. See also **X11 Resources**.

X11 Resources
A file or set of files that define your X11 environment.

XEQ Pin
A pin that forces the operation of the object, even if the data or sequence input pins have not been activated. See also **Control Pin**, **Data Input Pin**, and **Sequence Input Pin**.

Index

B

C

D

E

H

I

N

O

P

R

S

T

U