VIAVI Solutions

VIAVI Instrument Programming Tool (VIP Tool™) User Guide

Contents

Introduction to the VIP Tool

The VIAVI Instrument Programming Tool (VIP Tool) is an easy-to-use programming tool that is embedded into a Microsoft® Excel workbook. The VIP Tool provides a large range of capabilities that allow a user to write simple or complex programs in a scripting language designed to be conducive to radio test methodology. The VIP Tool is an easy to use programming tool that is embedded into a Microsoft Excel workbook. The VIP Tool allows the end user to automate operation of a VIAVI instrument by taking advantage of the RCI capabilities of the instrument. Custom programs can be written to automate common functions or to create test capabilities that only automation can provide.

Overview

At its heart, the VIP Tool is a means of communicating with the instrument over an Ethernet connection. In and of itself, this capability is certainly not rare; many programs possess that capability. However, the ability to send a remote command or receive query data is of limited use if there is no means to organize the commands into a sequence, perform decisions based upon conditions, and collate test data in a storable form.

The VIP Tool uses a scripting language called the VIAVI Instrument Programming Language. This language allows the operator to branch programs based on conditional statements, perform repetitive tasks through looping, and create test reports with a sequence of commands placed within the spreadsheet. This capability combines the power of a scripting language with Microsoft Excel that allows even an entry level technician to create programs that can make his job more efficient and profitable.

Intended User

The VIP Tool is an easy-to-use tool that is intended for personnel with little or no programming experience.

System Requirements

Users require the following to use the VIP tool:

- Microsoft Excel is required to operate the VIP Tool.
- VIP Tool Option 847 installed on VIAVI instrument.

Installing VIP Tool

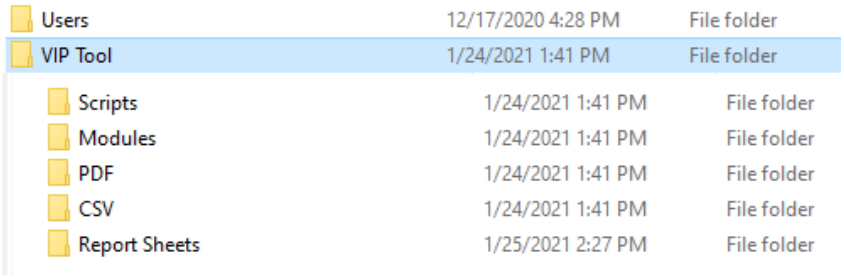
Installing the VIP Tool is as simple as copying the VIP Tool workbook to your computer; the VIP Tool program is embedded in the workbook. All that is needed to run VIP Tool is to have access to Microsoft Excel and a VIAVI instrument with Option 847 installed. . No special drivers are required.

A Sample VIP Tool Directory Structure

The VIP Tool Setup sheet provides the means to configure paths for the following file types:

- Script Files
- Module Files
- PDF Report Files
- CSV Report Files (Please check the format between this bullet and the next. The space looks bigger)
- Export / Import Report Sheets

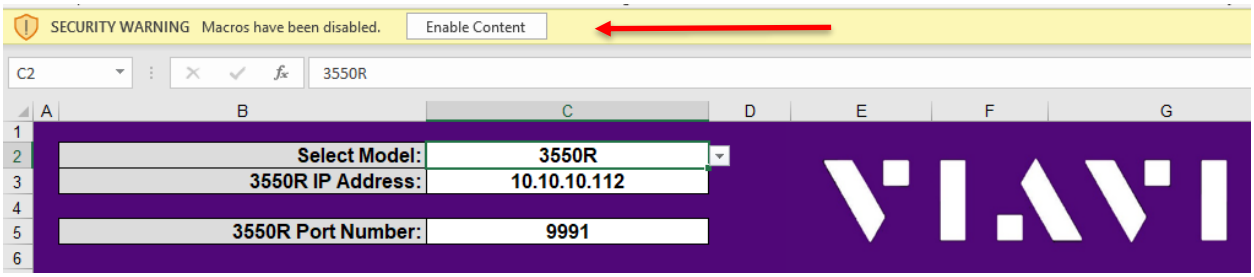
A sample setup may look something like this:



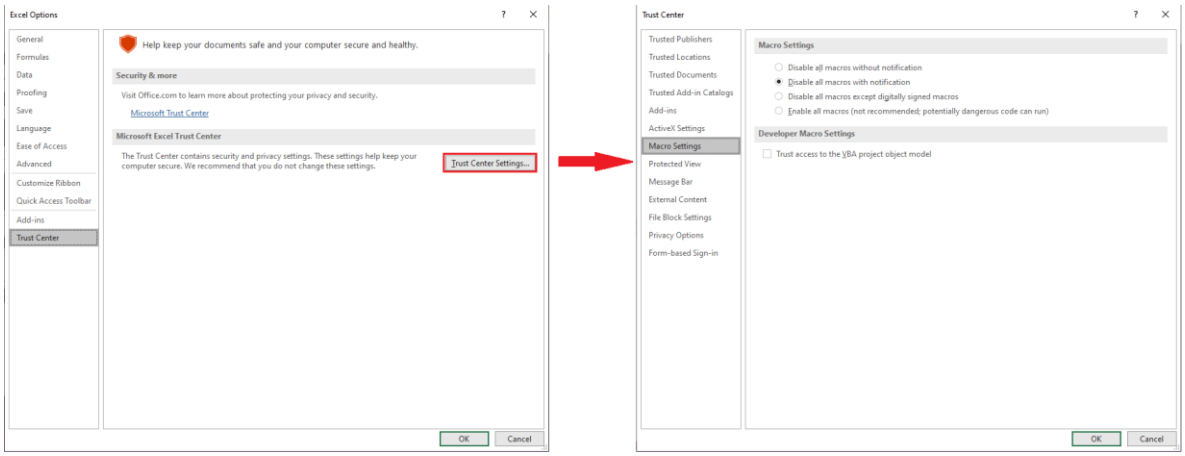
The VIP Tool workbook can be copied to the VIP Tool folder, and the sub folders will hold the Script, Module, PDF and report template files.

Enabling the VIP Tool

The VIP Tool requires that macros be enabled in Microsoft Excel. Depending on the configuration of Excel, this message may be displayed when the VIP Tool workbook is opened.



If macros cannot be enabled in the spreadsheet, review the settings in Excel and Trust Center.



Select Macro Settings and ensure the settings allow macros. Select, at minimum ‘Disable all macros with notification’. This setting causes Excel to notify the user that the workbook has macros and allows the operator to enable them. If configuring macro settings has been disabled, contact your IT administrator.

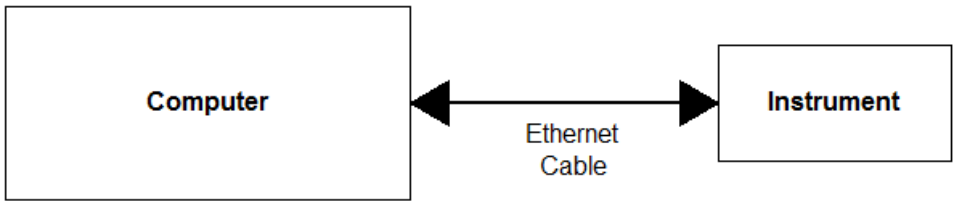
Establishing an Ethernet Connection to the Instrument

Before the VIP Tool can be used, the computer on which Excel is running must be connected to the VIAVI instrument through an Ethernet connection. This section of the manual provides guidance on establishing and verifying that a valid connection has been made between the computer and the instrument.

Connection Types

Direct Connection

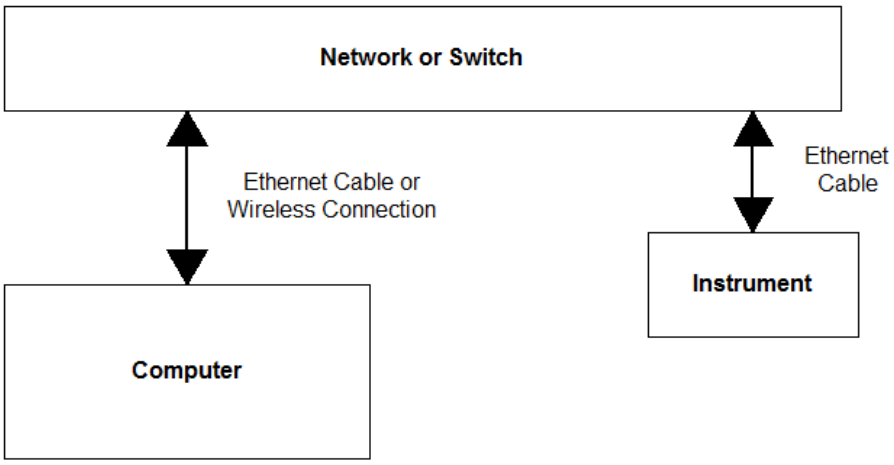
There are a few different ways to connect the computer to the instrument. One method is a direct connection from the computer to the instrument.



The direct connection consists of connecting an Ethernet cable from the computer to the instrument. Generally, a common Ethernet cable can be used, though if the computer is an older model, it may be necessary to use a crossover Ethernet cable to make the connection.

In most cases, a direct connection implies that a static IP connection will be used.

Network Connection



Another method of connection is that the computer is connected to a network or a switch, and the instrument is also connected to the network or switch. Again, a common Ethernet cable can be used. to connect the instrument to the network.

In the case of a switch, the connection type will likely be a static IP connection, though more advanced switches may have DHCP capability. In the case of a network connection, the connection type will most commonly be DHCP.

Configuring the Instrument's Network Connection

Configuring the 8800 Series Connection

The 8800 Series network configuration screen can be reached by selecting the Utilities tab, then selecting the Software button from the dropdown menu, and selecting System from the sub menu that appears. Once in the System Config menu, press the Remote tab.

The example above shows a Static IP configuration. The IP Address is set to 10.10.10.110. This IP address is the same address to enter in the IP Address field on the VIP Tool Setup sheet.

If a DHCP connection is desired, set the Network Mode to 'DHCP'. When the 8800 is connected to a network, the network will assign an address in the IP address field.

The 8800 TCP/IP port, which is set automatically by the VIP Tool when the 88XX model is selected on the Setup Sheet, is 9991.

NOTE:	It should be noted that only one device at a time can be connected to port 9991 of the 8800. If port 9991 is occupied by some other program or device, the VIP Tool will not be able to connect to the 8800 until the port is released by the other device or program.

Configuring the 3550 Series Connection

The 3550 Series network configuration screen can be reached by selecting the System tab, then selecting the System Config button from the dropdown menu. Once in the System Config menu, press the Remote tab.

The example above is of a Static IP configuration. The IP Address is set to 10.10.10.112. That IP address is the same address to enter in the IP Address field on the VIP Tool Setup sheet.

If a DHCP connection is desired, set the Network Mode to 'DHCP'. When the 8800 is connected to a network, the network will assign an address in the IP address field.

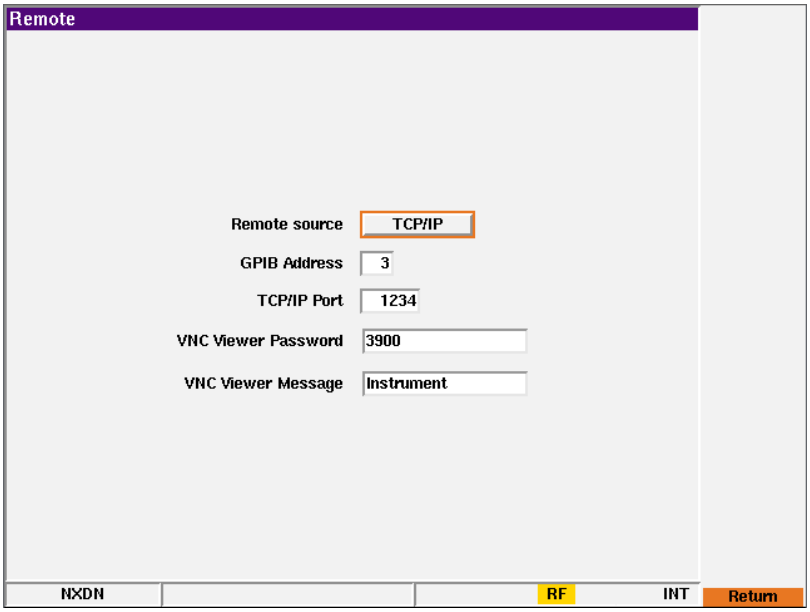
The 3550 TCP/IP port, which is set automatically by the VIP Tool when the 3550R model is selected on the Setup Sheet, is 9991.

NOTE:	It should be noted that only one device at a time can be connected to port 9991 of the 3550. If port 9991 is occupied by some other program or device, the VIP Tool will not be able to connect to the 3550 until the port is released by the other device or program.

Configuring the 3900 Series Connection

The 3900 Series setup involves selecting TCP/IP as the remote connection type, then setting up the network settings.

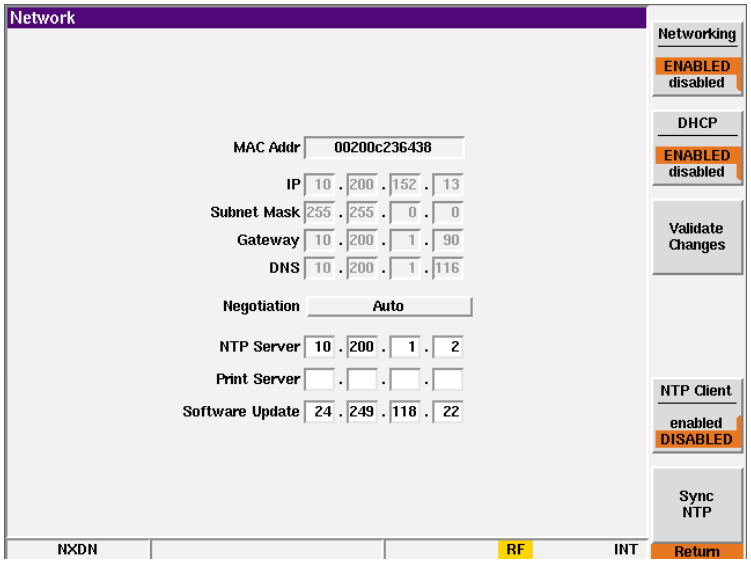
To access the remote connection type setting, select the Utilities menu, then select Hardware Settings and select Remote from the dropdown menu.



Once on the Remote screen, set the Remote source field to 'TCP/IP'.

NOTE:	Note that the TCP/IP port on the 3900 Series is configurable. The default port number is 1234. When connecting to the VIP Tool, verify that the entry in the Setup sheet 'Port Number' field is set to match the number entered on this screen.

After setting the Remote source, set up the network connection. To access the network settings, select the Utilities menu, then select Hardware Settings and select Network from the dropdown menu.



The example above is of a DHCP configuration. The IP Address is set to 10.200.152.13. That IP address is the same address to enter in the IP Address field on the VIP Tool Setup sheet.

NOTE:	Please note that with some Ethernet routers and switches, it may be necessary to select a Negotiation setting other than Auto, such as “100Base Half-Duplex” to avoid negotiation conflicts with the Ethernet router or switch.

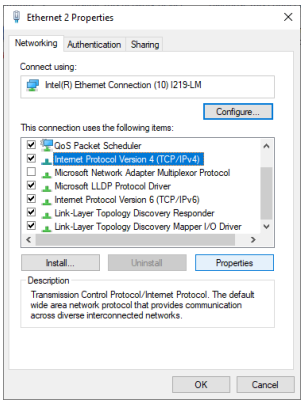
If a Static IP connection is desired, press the DHCP soft key so that DHCP is disabled. Enter the desired IP address and subnet mask and, after entering the IP address, press the Validate Changes soft key.

The 3900 Series can host multiple connections on its TCP/IP port. The VIP Tool is capable of connecting to the 3900 Series even if another device or program is connected to the instrument.

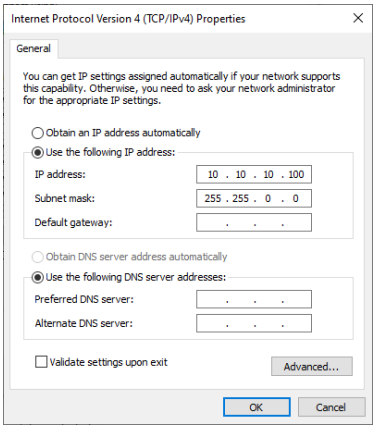
Setting up the Computer for Static IP Connection

To use a direct connection with a Static IP connection, the computer must be configured to access the Static IP of the instrument.

1. On the computer, access the Ethernet properties of the computer’s Ethernet card. Accessing this menu may vary from computer to computer.



2. Once the Ethernet properties menu is open, select ‘Internet Protocol Version 4 (TCP/IPv4) field. After selecting the field, press the Properties button.



3. Select ‘Use the following IP Address:’.
4. In the IP Address Field, enter the IP address that you wish to use. The instrument will need to share the first three elements of the IP address, and have a different number on the fourth address element. For

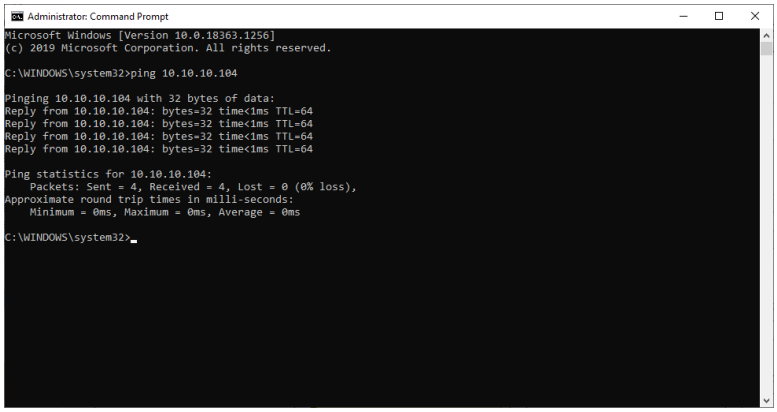
example, this computer's Static IP is set to 10.10.10.100. Therefore, an instrument that is connected to it can have the address 10.10.10.110.

5. Set the subnet mask to 255.255.0.0. Ensure the subnet mask on your instrument is set to this same setting.
6. When finished, press the OK key on this menu, and the OK key on the first menu to close out the operation.

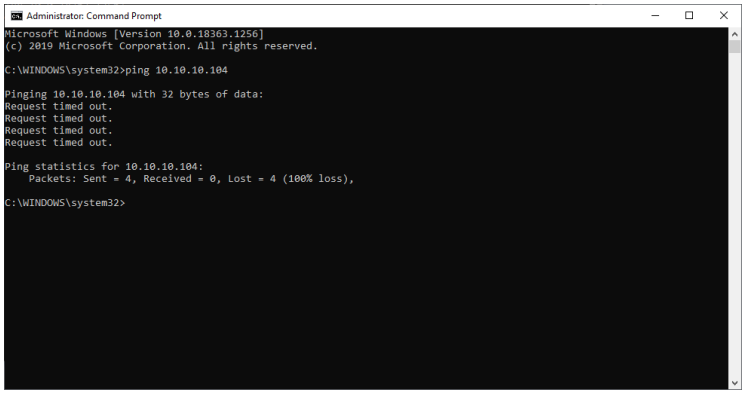
Testing the Connection

Before connecting to the VIP Tool, it is recommended that the Ethernet connection to the instrument be verified. A good way of verifying the connection is to see if the instrument can be pinged from the computer.

1. On the computer, open a command prompt.
2. In the command prompt, type 'ping' followed by the IP address the instrument is set to. For example, if the instrument IP address is 10.10.10.104, then one would type: ping 10.10.10.104.



- If the connection is successful, then the text on the command window will appear like the example above. Each ping receives a reply from the instrument, and no timeout messages appear. The instrument and the VIP Tool will be able to communicate back and forth.



- If the connection is unsuccessful, then the text on the command window will appear like the example above. No pings have been returned by the instrument; each request has timed out. This is indicative of a communications failure between the computer and the instrument. The VIP Tool will not be able to communicate with the instrument until successful communication has been established.

The VIP Tool Workbook

The VIP Tool workbook contains the VIP Tool program and several embedded sheets that provide the functionality of the tool. Additional sheets for creating reports or performing calculations can be added to and removed from the VIP Tool workbook. The VIP Tool cannot interact with sheets that are not a part of the VIP Tool workbook.

The workbook also contains the VIP Tool ribbon which provides a selection of buttons specific to the currently selected sheet.

This section of the manual describes the purpose and functionality of each sheet and the ribbon buttons that apply to each sheet.

The VIP Tool Ribbon

The VIP Tool ribbon is synchronized to the sheet selection. When a sheet is selected, the VIP Tool ribbon will display the buttons relevant to the sheet that is selected.

The VIP Tool ribbon does not automatically re-synchronize if the VIP Tool is saved with a different name after clicking on ‘File’ and ‘Save As’. The ribbon is still accessible but will not update to a newly selected sheet if a different sheet is selected; instead, the VIP Tool ribbon will need to be manually selected. To restore synchronization, close the newly ‘Saved As’ workbook, then re-open it. After re-opening the workbook, the ribbon will synchronize with the different sheets as they are selected.

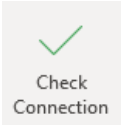
The Getting Started Sheet

The Getting Started sheet provides information about the VIP Tool workbook that the user can access directly from the workbook.

The Getting Started Sheet Ribbon

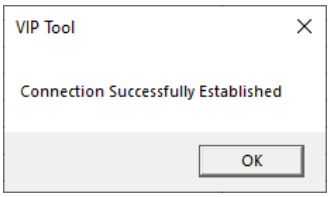
Results Data Sheet Ribbon I/O Group

The Results Data sheet ribbon I/O group consists of only one button, the ‘Check Connection’ button.

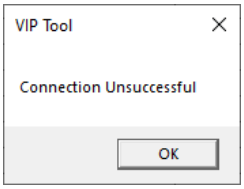
Check Connection	
	Opens a socket and checks to see if VIP Tool has successfully connected to the instrument. Closes the socket after the check has completed.

The ‘Check Connection’ button is used to verify if the instrument is connected to the VIP Tool. This button shows up on several different sheets of the VIP Tool, and on each sheet the functionality of the button is the same. It is on the Setup sheet that the VIP Tool is configured to communicate with the instrument through an Ethernet socket.

'Check Connection' attempts to open a socket to the instrument, based on the settings entered in the Setup Sheet. If a valid connection to the instrument is detected, a message indicating successful connection is displayed:



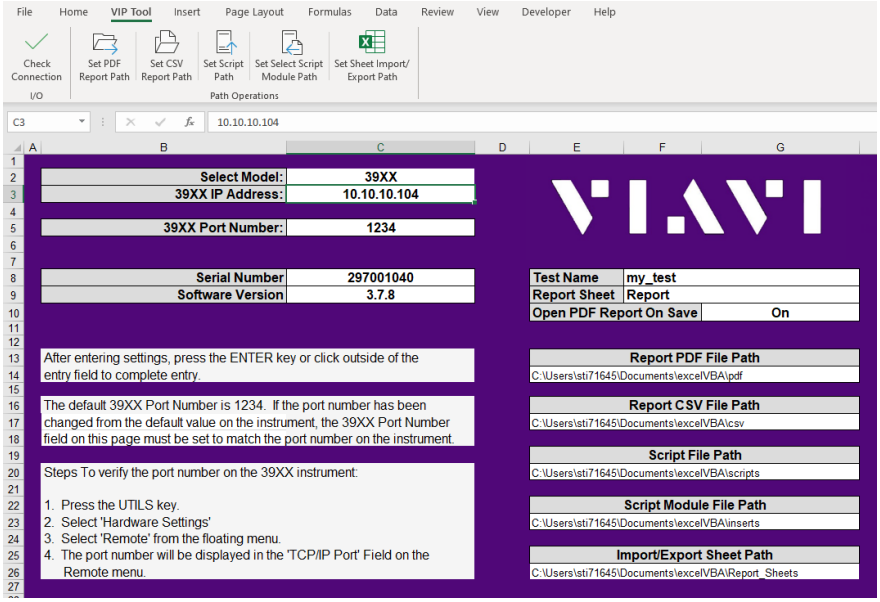
If, however, the VIP Tool cannot communicate with the instrument, a message indicating unsuccessful connection is displayed:



If the connection is unsuccessful, select the Setup Sheet and configure the model and IP address of the instrument to establish a successful connection. The VIP Tool will not execute script commands if there is not a valid connection to the instrument.

The Setup Sheet

The VIP Tool Setup sheet is used to set up the Ethernet socket that connects the VIP Tool to the instrument.

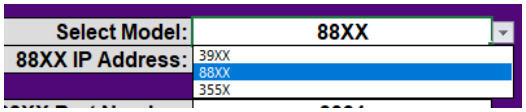


The Setup sheet is also used to set paths for saving various types of files, select which sheet is to be saved as a report, what name to give the report, and to select whether a PDF report is saved in the background or is opened when saved.

Setup Sheet Fields

The Setup Sheet contains user editable fields that are used to set Ethernet communication parameters, to select default file paths, and to determine whether a PDF file is automatically opened when saved either by a script command or when using a ribbon button.

The 'Select Model' Field

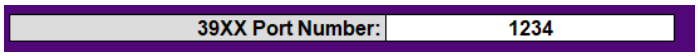


The 'Select Model' field allows the user to select which model the VIP Tool will connect to. This field is a drop-down menu that the provides a list of the instruments the VIP Tool can connect with. Selecting the model automatically sets the 'Port Address' field setting to the appropriate port address of the selected instrument.

The 'IP Address' Field

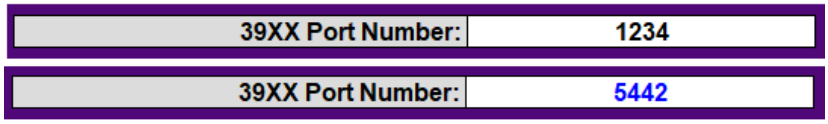
The VIP Tool must have a valid socket connection to the instrument to communicate with the instrument. The 'IP Address' field is used to point the VIP Tool to the IP address of the instrument it is connected to, therefore the instrument IP address is entered in this field.

The 'Port Number' Field



Each VIAVI instrument has a port number that must be addressed to establish a TCP/IP connection with the instrument through an Ethernet socket. The VIP Tool will determine the correct port number based upon the model selected in the 'Select Model' field.

In the case of 39XX instruments (3901, 3902, 3920, and 3920B variants), the instrument defaults to the port number of 1234. However, the 39XX instruments contain a feature that allows the user to set a port number other than 1234 for TCP/IP communications. When '39XX' is selected as the model, the VIP Tool will initially default to a port number selection of 1234. However, if the 39XX instrument is set to a different port number, the actual port number must be entered in the Port Number field in the Setup Sheet.



If the user-selected port number entry into the VIP Tool port number field is not the default value of 1234, then the port number will be displayed in blue so that the operator can tell immediately if at-a-glance the port number is not set up to a default number if changing instruments from one 39XX to a different 39XX series instrument.

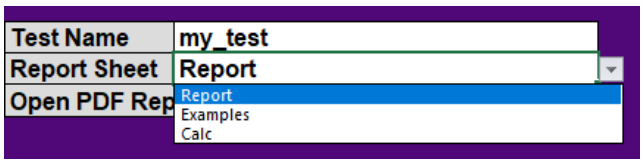
The Test Name Field

The VIP Tool automatically saves PDF and CSV report files with a date and time stamp. The VIP Tool scripting language allows the user to programmatically assign a name to the report that is included with the date and time stamp data. If the script does not assign a name to the report by using a specific command, the name will default to the value entered in the Test Name field, along with the date and time stamp data. If a report is printed using the 'Save Report as PDF' or 'Save Report as CSV' buttons found on report page ribbons, this name will be added to the date and time stamp information when the report is saved using this method.

If a script is loaded with a test name command, VIP Tool will detect the command and place the name of the report it sees in the script into this field.

The Report Sheet Field

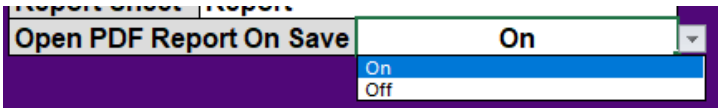
The VIP Tool command language allows the programmer to specify which report sheet to save when the command is executed. If no such command is in the script, the output of the report will default to the sheet specified in this field.



The field provides the available saveable sheets found within the VIP Tool worksheet as a drop-down menu. Only sheets that can be saved as a report are available on this drop-down menu.

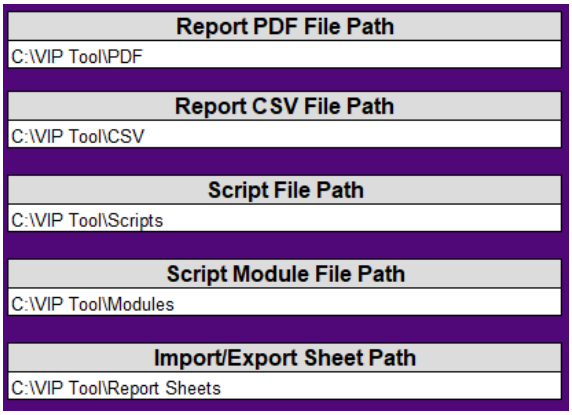
The Open PDF Report on Save Field

The 'Open PDF Report on Save Field' determines if a PDF viewer is opened immediately upon a manual (by pressing the 'Save as PDF' button) save operation or programmatic PDF save function is executed. The field provides a dropdown 'On' or 'Off' selection. If 'On' is selected, the PDF report will be opened by the program on the user's computer that is used to view PDF files. If ''Off' is selected, the PDF file will be saved, but not opened immediately.



File Path Fields

Five fields on the Setup Sheet are used for setting file paths for various files imported and exported by the VIP Tool. The purpose of the file path fields is to allow the user to create a directory structure that organizes the various types of files used and generated by the VIP Tool. If the file path fields are left blank, the path of the files will default to the path of the VIP Tool workbook itself.



A path can be directly entered in a file path field, or the buttons in the Setup Sheet ribbon Path Operations group can be used to select the path for a particular type of file using a file dialog.

Report PDF File Path

The Report PDF File Path field sets the default path for PDF reports that are generated either by script commands or pressing the ‘Save Report as PDF’ button found on report pages.

Report CSV File Path

The Report CSV File Path field sets the default path for CSV formatted reports that are generated either by script commands or pressing the ‘Save Report as CSV’ button found on report pages.

Script File Path

The Script File Path field sets the default path for saving and loading Script files when using the ‘Save Script’ and ‘Load Script’ buttons found on the Script sheet.

Script Module File Path

The Script Module File Path field sets the default path for saving and loading Script module files when using the ‘Insert Module Here’ and ‘Save Selection as Module’ buttons found on the Script sheet.

Import / Export Field Sheet Path

The Import / Export File Sheet Path sets the path that a Report sheet is exported to when pressing the ‘Export Sheet to XLS’ button found on report pages and sets the path for importing sheets when using the ‘Import XLS Sheet’ button found on the Script sheet.

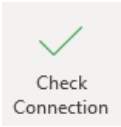
The Setup Sheet Ribbon

When the Setup Sheet is selected, the Setup Sheet ribbon is brought into focus. The Setup Sheet ribbon contains functions specific to checking the Ethernet connection and for setting paths for the various files used and produced by the VIP Tool. The Setup Sheet ribbon is divided into two groups:

- I/O
- Path Operations

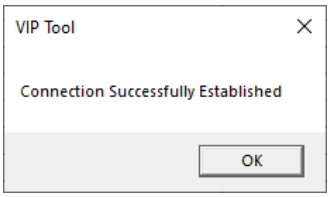
Setup Sheet Ribbon I/O Group

The Setup Sheet Ribbon I/O group consists of only one button, the ‘Check Connection’ button.

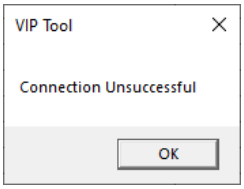
Check Connection	
	Opens a socket and checks to see if VIP Tool has successfully connected to the instrument. Closes the socket after the check has completed.

The ‘Check Connection’ button is used to verify if the instrument is connected to the VIP Tool. This button shows up on several different sheets of the VIP Tool, and on each sheet the functionality of the button is the same. It is on the Setup Sheet where the VIP Tool is configured to communicate with the instrument through an Ethernet socket.

‘Check Connection’ attempts to open a socket to the instrument, based on the settings entered in the Setup Sheet. If a valid connection to the instrument is detected, a message indicating successful connection is displayed:



If the VIP Tool cannot communicate with the instrument, a message indicating unsuccessful connection is displayed:



If the connection is unsuccessful, select the Setup Sheet and configure the model and IP address of the instrument to establish a successful connection. The VIP Tool will not execute a script if there is not a valid connection to the instrument.

Setup Sheet Path Operations Group

The Setup Sheet Path Operations Group contains a set of buttons that allow the user to set default paths for the various files the VIP Tool uses to perform tests and save results.

Set PDF Report Path	
	Opens a file dialog and allows the user to set a default path for saved PDF files. The selected path populates the ‘Report PDF File Path’ field.

Pressing the ‘Set PDF Report Path’ button opens a dialog that allows the user to set the default path for PDF reports that are generated either by script commands or pressing the ‘Save Report as PDF’ button found on report pages. The path that is selected populates the Setup Sheet ‘Report PDF File Path’ field.

Set CSV Report Path	
	Opens a file dialog and allows the user to set a default path for saved CSV report files. The selected path populates the ‘Report CSV File Path’ field.

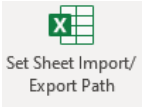
Pressing the ‘Set CSV Report Path’ button opens a dialog that allows the user to set the default path for CSV formatted reports that are generated either by script commands or pressing the ‘Save Report as CSV’ button found on report pages. The path that is selected populates the Setup sheet ‘Report CSV File Path’ field.

Set Script Path	
	Opens a file dialog and allows the user to set a default path for saved script files. The selected path populates the ‘Script File Path’ field.

Pressing the ‘Set Script Path’ button opens a dialog that allows the user to set the default path for saving and loading Script files when using the ‘Save Script’ and ‘Load Script’ buttons found on the Script sheet. The path that is selected populates the Setup Sheet ‘Script File Path’ field.

Set Select Script Module Path	
	Opens a file dialog and allows the user to set a default path for saved script module files. The selected path populates the ‘Script Module File Path’ field.

Pressing the ‘Set Select Script Module Path’ button opens a dialog that allows the user to set the default path for saving and loading Script module files when using the ‘Insert Module Here’ and ‘Save Selection as Module’ buttons found on the Script sheet. The path that is selected populates the Setup Sheet ‘Script Module File Path’ field.

Set Sheet Import / Export Path	
	Opens a file dialog and allows the user to set a default path for importing and exporting saved report sheets. The selected path populates the ‘Import / Export Sheet Path’ field.

Pressing the ‘Set Sheet Import / Export’ button opens a dialog that allows the user to set the default path for exporting and importing saved Report sheets. This setting sets the path that a Report sheet is exported to when pressing the ‘Export Sheet to XLS’ button found on report pages and sets the path for importing sheets when using the ‘Import XLS Sheet’ button found on the Script sheet. The path that is selected populates the Setup Sheet ‘Import / Export XLS Sheet’ field.

The Script Sheet

The VIP Tool Script sheet is where all programming, edit and debug operations happen. It is the VIP Tool's programming environment. It is on this sheet that the script is built.

A script is a sequence of commands that perform the function desired by the programmer. In its simplest form, a script can be a series of RCI commands sent to the instrument. However, in most applications it is desirable to control the sequence of commands sent to the instrument, and it is also desirable to be able to record the outcome or results of the operation programmed into the script. To organize the logical flow of the task, the VIP Tool provides a scripting language that can be used in conjunction with functions built into Microsoft Excel to realize the goal the script is meant to accomplish.

The Script

A script is the sequence of commands and arguments that are used to program the VIP Tool's task. The VIP Tool will automatically execute the selected row of a script, then move to the next row and execute that row. If the 'Run Script' button is used to start script execution, the script will automatically start on row 2 of the Script sheet.

The VIP Tool will continue to execute each line of a script until it encounters either a blank line, the keyword END, or an error is detected either by the instrument, Excel, or the VIP Tool program.

A line in the script consists of the contents of the Command and Argument cells on a single row of the script page. Therefore, a 'line' in a script is often referred to in this document as a 'row'. If a script is executed while the Script page is being viewed, the current row of execution will be highlighted in green, though sometimes the script may move through the line so quickly, the green color will be so brief as to be invisible. The Script page will automatically scroll to keep the current row of execution in focus.

A script or module can be stored and recalled to the Script sheet at a later time. A script or module file is simply a text file that stores the contents of a script or a module. Certain characters are used by the script to store information such as line indentation. A script or module text file can be viewed in any text editor.

Script vs. Module

Both scripts and modules are essentially the same thing: a series of commands that control the flow of a series of tasks. The only differences between a script and a module are size and purpose. A script can be considered the entire program used to accomplish a task. A module can be considered a part of a script that can be re-used and inserted into other scripts. For example, the programmer may find portions of certain scripts contain identical subroutines. A subroutine can be stored as a module and inserted into other scripts when that subroutine is needed, saving time and reducing duplication of effort.

Organization of the Script Sheet

	Command Column Accepts Entry No Spaces Allowed	Argument Column Accepts Entry May be empty, depending on command	Reply Column Locked, does not accept entry Displays information generated by queries	Info Message Column Locked, does not accept entry Displays information generated by script
1	Command	Argument	Reply	Info Message
2	Sub	Begin		
3	*idn?	[TestUnit]=Reply3	AEROFLEX,3920,1000595557,3.7.8,2	[TESTUNIT] = 1000595557
4	close_socket			
5	# Store Current Test Selections and Radio SN			
6	For	1 to 10		For 1 to 10

The Script Sheet consists of four columns. Two columns, Command and Argument, allow user entry in order to build a script. The remaining two columns, Reply and Info Message, are protected against user entry. These two columns serve to provide feedback to the programmer as a script is executed.

Commands and Arguments

RCI commands and queries, as well as VIP Scripting Language keywords, will always require an entry into the command column. Arguments may or may not be required, depending on the RCI command or the keywords. The argument for any command is always placed in the Argument column on the same row as its command.

The Command Column

The command column cell for a row holds the command for the line operation. That command can be an RCI command or query such as '*IDN?', or it can be a VIP Scripting Language keyword or variable. If a command does not require an argument, the argument column on the same row as the command can be blank.

Any command or keyword placed in a command column cell cannot contain a blank space. Remarks, which are defined by the '#' symbol, however, may contain spaces.

If a blank space in a command cell is detected by the VIP Tool during script execution, and the contents of the cell are not a remark, the blank space will automatically be removed. If an RCI command that requires a preceding colon is detected in a command column cell, the VIP Tool will automatically prepend the command with a colon if the preceding colon is missing, thereby ensuring the instrument can process the command properly.

If a command column cell is left blank and script operation steps to the blank column cell, script operation will cease at that row.

The Argument Column

The argument column is used to complete the command structure if the command requires an argument. For example, the RCI command :RF:Gen:Freq, which is used to set the RF generator frequency in a 3920B, would be placed in the command column, and it must have the argument defining the frequency placed in the corresponding cell of the argument column. The argument column can contain spaces, depending on the rules of the argument being used for a particular command.

The Reply Column

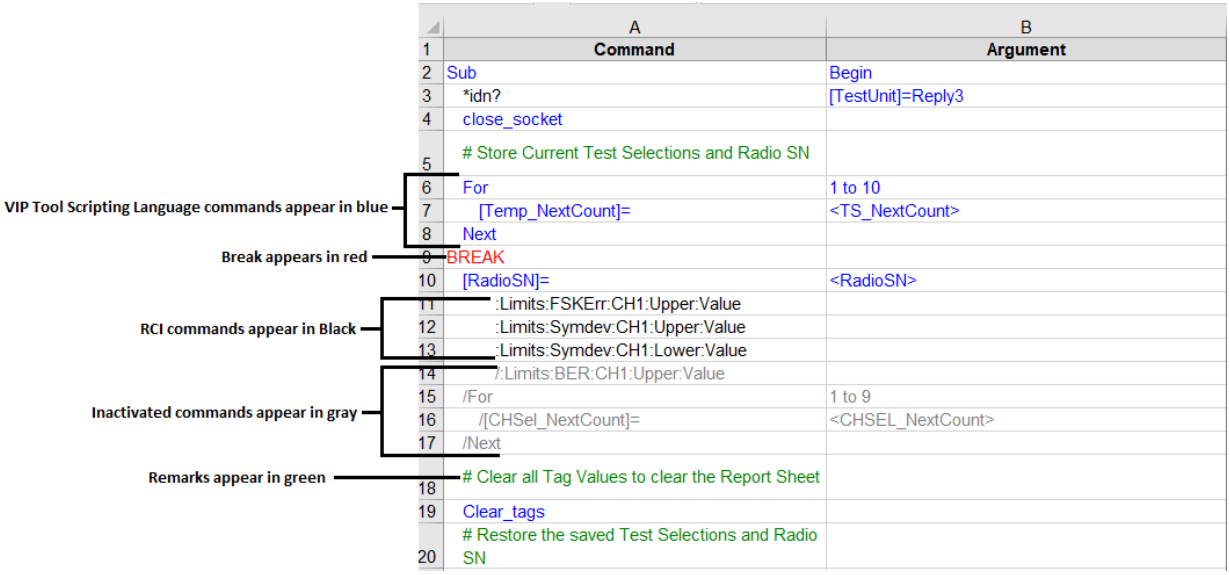
A Reply Column cell will display any information generated by an RCI query or a VIP Tool Scripting Language keyword on its row. The purpose of the Reply column is to provide feedback to the operator as the Script is being programmed. The cells of the Reply Column are read-only; they are protected against entry by the operator.

The Info Message Column

An Info Message Column cell will display any information generated by the VIP Tool Programming Language concerning the specific command executed on that row, if the command provides feedback information. For example, the cell may provide information as to the value or destination of a variable.

The Info Message Column is also used to display error messages generated by the instrument, Excel or the VIP Tool program.

VIP Tool Script Color Coding



	A	B
1	**Command**	**Argument**
2	Sub	Begin
3	*idn?	[TestUnit]=Reply3
4	close_socket	
5	# Store Current Test Selections and Radio SN	
6	For	1 to 10
7	[Temp_NextCount]=	<TS_NextCount>
8	Next	
9	BREAK	
10	[RadioSN]=	<RadioSN>
11	:Limits:FSKErr:CH1:Upper:Value	
12	:Limits:Symdev:CH1:Upper:Value	
13	:Limits:Symdev:CH1:Lower:Value	
14	/:Limits:BER:CH1:Upper:Value	
15	/For	1 to 9
16	/[CHSel_NextCount]=	<CHSEL_NextCount>
17	/Next	
18	# Clear all Tag Values to clear the Report Sheet	
19	Clear_tags	
20	# Restore the saved Test Selections and Radio SN	

The Script Sheet automatically color codes command and argument text based upon the function of the command. The function of the color coding is to aid in reviewing and editing a script by providing a means to distinguish between RCI commands, the VIP Tool scripting commands, remarks, and disabled commands.

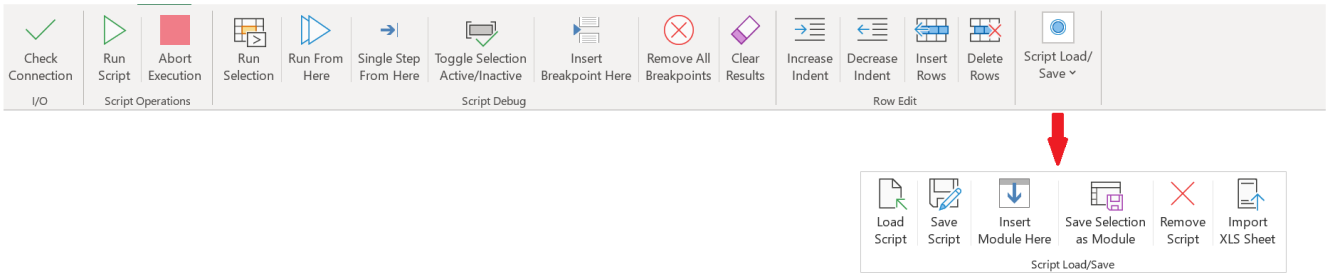
RCI commands, which are transmitted to the instrument through the Ethernet connection appear in a black font. The VIP Tool scripting keywords are displayed in a blue font. If an RCI command or a scripting keyword is disabled by the '/' symbol placed in the command column, the command and argument cell content is displayed in a gray font. If a row is denoted as a remark by placing the '#' symbol in the command field, the command and argument font color is green. And, finally, if a breakpoint is entered in the command field by using the keyword 'break', the command column font is red.

The Script Sheet Ribbon

When the Script Sheet is selected, the Script Sheet Ribbon is brought into focus.

Depending on the scaling settings of the computer the VIP Tool is displayed on, all the buttons will be displayed at once, or there may be buttons collapsed on the right side of the display underneath a button that will expand the collapsed buttons. The Script Sheet Ribbonribbon button graphics used in this manual were taken with the

display scaling set at 100%. If the display scaling is set to a higher number, this may hide the buttons on the right side of the ribbon.



The illustration above is of the Script sheet ribbon displayed on a computer that has the graphic scaling set at 175%. In this case, the Script Load/Save group buttons are only accessible after pressing the button labelled “Script Load/Save”.

The Script sheet ribbon contains functions specific to creating, saving, restoring, and debugging a script. The Script Sheet Ribbon is divided into five groups:

- I/O
- Script Operations
- Script Debug
- Row Edit
- Script Load / Save

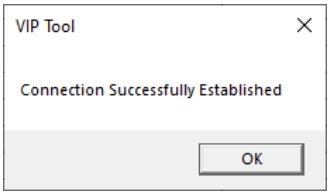
Script Sheet Ribbon I/O Group

The Script sheet ribbon I/O group consists of only one button, the ‘Check Connection’ button.

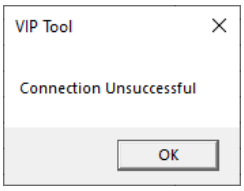
Check Connection	
	Opens a socket and checks to see if the VIP Tool has successfully connected to the instrument. Closes the socket after the check has completed.

The ‘Check Connection’ button is used to verify if the instrument is connected to the VIP Tool. This button shows up on several different sheets of the VIP Tool and on each sheet the functionality of the button is the same.

‘Check Connection’ attempts to open a socket to the instrument, based on the settings entered on the Setup Sheet. If a valid connection to the instrument is detected, a message indicating successful connection is displayed:



If, however, the VIP Tool cannot communicate with the instrument, a message indicating unsuccessful connection is displayed:



If the connection is unsuccessful, select the Setup Sheet and configure the model and IP address of the instrument to establish a successful connection. The VIP Tool will not execute script commands if there is not a valid connection to the instrument.

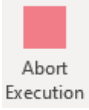
Script Sheet Ribbon Script Operations Group

The Script Sheet Ribbon Script Operations group consists of two buttons, the ‘Run Script’ button and the ‘Abort Execution’ button. These buttons also show up on user editable sheets, such as report sheets, sheets used to store data, etc. On each sheet these buttons appear on the functionality of the buttons is the same.

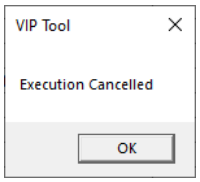
Run Script	
	Executes the entire script, starting at row 2 of the script page.

The ‘Run Script’ button is the main button used to begin execution of a full script. Pressing the ‘Run Script’ button will direct the VIP Tool to begin on row 2 of the script and continue execution until a blank command column cell is encountered or the ‘end’ command is encountered. It is of paramount importance when writing a script and executing with the ‘Run Script’ button that a command or remark be present in row 2 of the script page. If a command is in the command column of row 2, it can be active or inactive. If the command column of row 2 is empty, pressing the ‘Run Script’ button will have no effect.

When the ‘Run Script’ button is pressed, the script starts executing with a ‘clean slate’. This means all previous results and trace data are deleted from the VIP Tool work sheet. Any previous variables and variable values are also cleared.

Abort Execution	
	When pressed while a script is running will stop execution at the current row.

When the ‘Abort Execution’ button is pressed, script execution will immediately halt at its current operation.



When script execution is aborted, a message box will appear. Pressing the ‘OK’ button will clear this message box.

41	do			
42	:rf:gen:ch1:freq	151.1		:rf:gen:ch1:freq 151.1
43	:rf:gen:ch1:freq?	[genfreq]=reply	151100000	[GENFREQ] = 151100000
44	docount		2	
45	exitdo	[genfreq] != 151100000		If 151100000 != 151100000 = False
46	exitdo	docount = 3		User interrupt occurred

The Info Message column will present the message ‘User interrupt occurred’ at the section of the script where script operation halted.

NOTE:	It should be noted that the Abort button will have no effect if a static message is currently displayed. For example, if the ‘Pause’ command message is displayed, it must be closed before the ‘Abort Execution’ button can be used.

Script Sheet Ribbon Script Debug Group

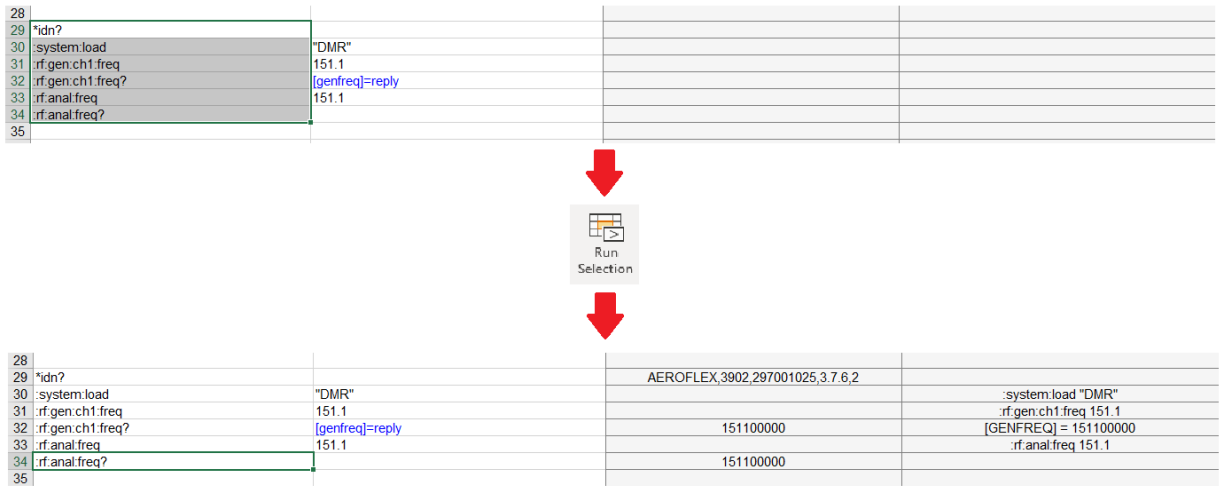
The Script Sheet Ribbon Script Debug group consists of a group of seven buttons that are provided as tools for debugging a script. This group of buttons is available only on the Script Sheet. The functions provided by the Script Debug group allow for running sections of a script instead of the entire script, for pausing the script at a breakpoint, and for activating or inactivating single or multiple rows of the script.

Run Selection	
	Executes all contiguously selected rows of a script. Can run only one row or multiple rows, but it will not execute any row not selected in a contiguous block of rows.

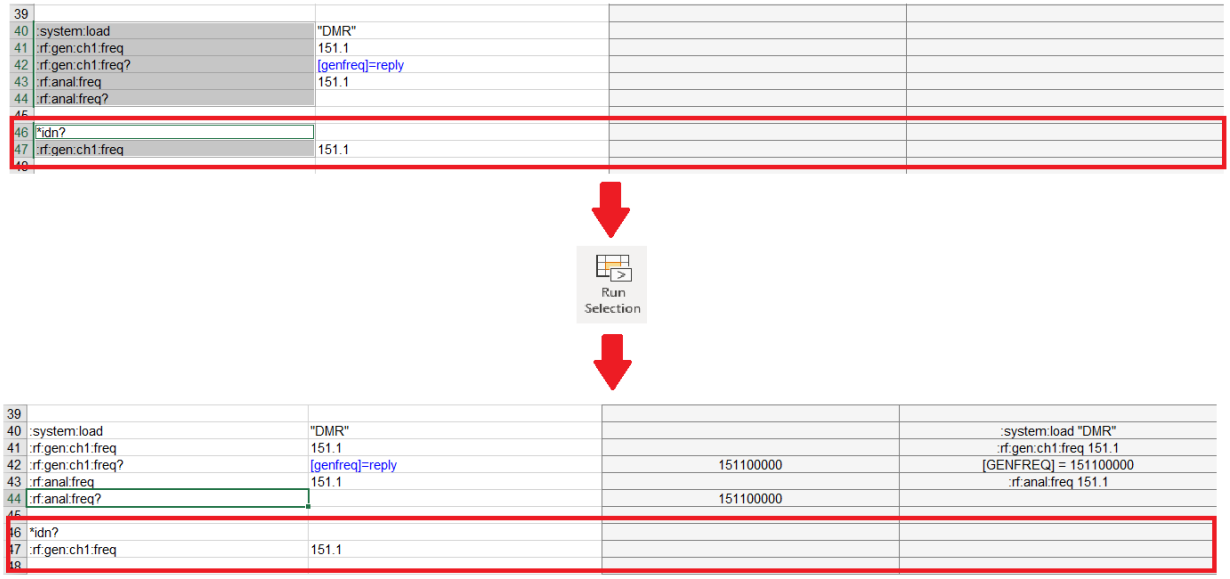
When pressed, the ‘Run Selection’ button will execute *selected* script rows. If only one row is selected by the cursor, pressing ‘Run Selection’ will execute that one row. If multiple rows are selected, and there are no blank spaces in the selected block of rows, pressing the ‘Run Selection’ button will execute all of the selected rows. Thus, the ‘Run Selection’ button is ideal for testing if a single command or a series of commands in a range.

28				
29	*idn?			
30	:system:load	"DMR"		
31	:rf:gen:ch1:freq	151.1		:rf:gen:ch1:freq 151.1
32	:rf:gen:ch1:freq?	[genfreq]=reply		
33	:rf:anal:freq	151.1		
34	:rf:anal:freq?			
35				

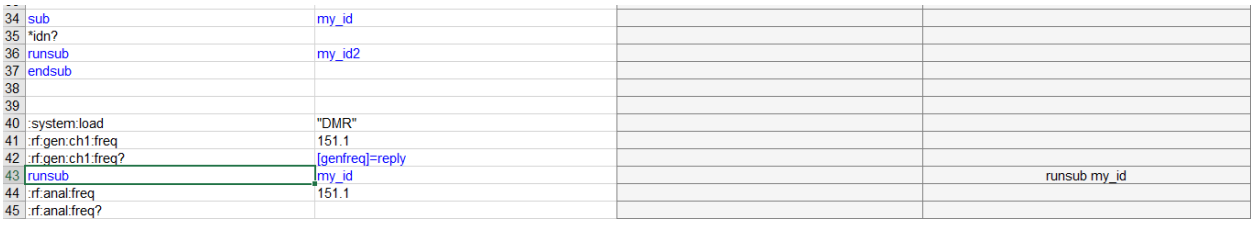
For example, after the programmer enters an RCI command, selecting that command and pressing ‘Run Selection’ will run just that one command. At that point, if no error codes are generated, and the instrument reacts to the command in the expected manner, the programmer can be assured that the entered command will work when the script is executed.



After selecting multiple contiguous lines and pressing the 'Run Selection' button, the programmer can determine if a block of code will work as expected.

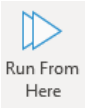


If non-contiguous rows (in other words, rows that have an empty row between them) are selected, pressing the 'Run Selection' button will result in only the first set of contiguous rows to be executed.



There are instances when pressing the 'Run Selection' button is not appropriate. For example, executing a single line that directs the script to run a subroutine will not cause the subroutine to be run, simply because the

subroutine is not a part of the selection. In that case, the ‘Run From Here’ or ‘Single Step From Here’ buttons should be used.

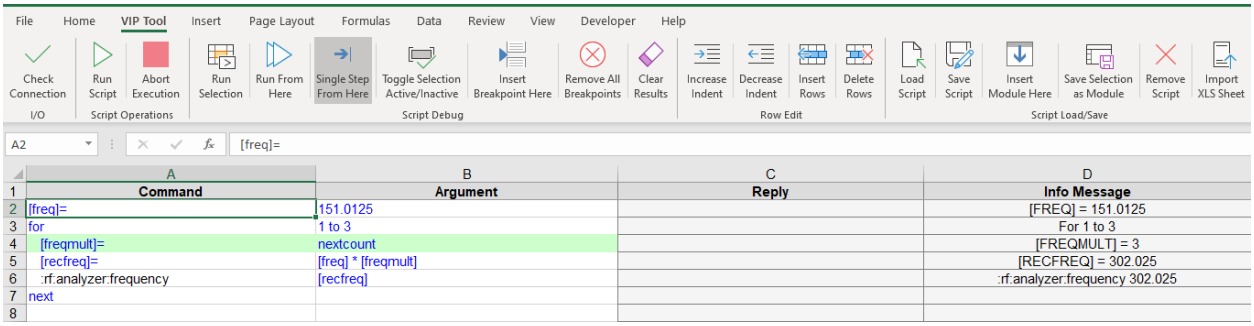
Run From Here	
	Begins execution at the currently selected row and runs until a stop point is encountered.

When the ‘Run From Here’ button is pressed, the VIP Tool will begin execution of the script sequence automatically, beginning at the current cursor position instead of automatically returning to row 2 to run the script. When the script reaches an end point, focus will return to the cursor position. This functionality is useful for running sections of non-contiguous commands without starting the entire script over again. It is particularly useful when used in conjunction with the break command. In the case of use with the break command, the script will pause at the break command. To continue running the script, pressing the ‘Run From Here’ button will cause the script to resume executing from the current break command position.

NOTE:	It should be noted that starting a script from a specific position after the value of a variable has been assigned may result in an error, as the variable will be empty. Therefore, if the section of the script that will execute when the ‘Run From Here’ button is pressed contains variables, be sure to start at a point before the variable is assigned a value.

Single Step from Here	
	Begins execution at the currently selected row and runs one line each time it is pressed. Continues to step until a stop point is encountered.

When the ‘Single Step From Here’ button is pressed, the VIP Tool will begin execution of the script sequence beginning at the current cursor position instead of automatically returning to row 2 to run the script. Pressing the ‘Single Step From Here’ button will execute only one row of the script each time it is pressed. When the script reaches an end point, focus will return to the cursor position. This functionality is useful for running sections of non-contiguous commands without starting the entire script over again. It provides the ability to evaluate each step of the script as it happens. It is particularly useful when used in conjunction with the break command. In the case of use with the break command, the script will pause at the break command. Pressing the ‘Single Step From Here’ button will cause the script to resume on the next step from the break command, moving forward in the script one row for each time the ‘Single Step From Here’ button is pressed. Pressing the ‘Run From Here’ button will cause the script to resume automatic execution of the script from the current position of the script selected while using ‘Single Step From Here’.



When the ‘Single Step From Here’ button is used, the background color of the row being executed will remain blue while the command is being processed. Once the command has been fully executed, the background color of the row will turn to green. It is important to remember that the script is still executing, and either the ‘Abort’ button must be pressed to end execution of the script, or the ‘Run From Here’ button must be pressed to resume automatic execution of the script.

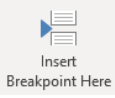
Starting a script from a specific position after the value of a variable has been assigned may result in an error, as the variable will be empty. Therefore, if the section of the script that will execute when the ‘Single Step From Here’ button is pressed contains variables, be sure to start at a point before the variable is assigned a value.

Toggle Selection Active / InActive	
	Acts upon a single or multiple selected rows. When pressed, if the contents of a selected row are active, it will set that row to inactive. If the contents of a row are inactive, it will set that row to active.

The ‘Toggle Selection Active / InActive’ button provides a means to inactivate or activate a row or multiple rows in a selection. This control simply provides a means of inserting or removing the ‘/’ symbol in the command column. This control is useful for disabling commands that are suspected of causing problems within a script. It can also be used to activate or inactivate entire portions of a script when a certain task is required or not required.

When the ‘/’ symbol is detected in a command column cell, the command in the cell is inactivated. The contents of the command and argument columns will appear in a gray font when the symbol is detected in the command column. The program will not step through the row, but will step over or ‘skip’ the row.

To activate or inactivate a specific row, select the row and press the ‘Toggle Selection Active / Inactive’ button. To activate or inactivate multiple rows, select a range of rows and press the ‘Toggle Selection Active / Inactive’ button.

Insert Breakpoint Here	
	Creates a row at cursor position and inserts a breakpoint in the new row.

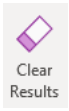
The ‘Insert Breakpoint Here’ button provides a means to easily insert the break keyword into any point within a script.

When a script encounters the break command, program operation pauses (but execution does not end; the script program is still running) on that row. Pressing the 'Run From Here' or 'Single Step' buttons on the VIP Tool ribbon will allow the script to resume, either in full automatic mode, or single step mode, respectively.

When the script is not running, selecting a row and pressing the 'Insert Breakpoint Here' button will insert a row and place the break command in the new row. Break can also be typed into any command column row and will provide the same outcome.

Remove All Breakpoints	
	Removes all breakpoints from the script and deletes the rows formerly occupied by the breakpoint.

Pressing the 'Remove All Breakpoints' button will remove all instances of break on the Script Sheet and will remove the row(s) formerly occupied by the break command(s).

Clear Results	
	Removes all information from the Script Sheet Reply and Info columns. Removes data from the Results Data Sheet and removes Trace Data from the Trace Data Sheet.

Pressing the 'Clear Results' button removes all results from previous script operations to be removed from the Script Sheet. In addition, all trace data are removed from the Trace Data Sheet and all results are removed from the Results Data Sheet.

Script Sheet Ribbon Row Edit Group

The Script Sheet Ribbon Row Edit group consists of four buttons. This group of buttons is available only on the Script Sheet. The functions provided by the Row Edit group replace functions that are inaccessible through normal Excel operations because the Script Sheet is a locked sheet. The functions provided allow insertion of blank rows, deletion of rows, and setting the indentation of the cell contents in the Command Column.

Increase Indent	
	Increases the indentation of Command column cell contents on all selected rows of the Script Sheet each time the button is pressed.

The 'Increase Indent' button is used to indent contents of selected command column contents. The indent function is used format the commands in the Command column so that the script can more easily be followed. For example, indenting a For Next loop or a Do loop aids in understanding what the loop is doing within multiple lines of a script. Because the Script Sheet is locked, the indent commands normally accessed within Excel are inaccessible on the Script Sheet. Therefore, the 'Increase Indent' button is made available on the Script Sheet Ribbon.

To indent a single command, select the command's row with the cursor and press the 'Increase Indent' button. To indent a range of commands, select the range of commands and press the 'Increase Indent' button. Each time the 'Increase Indent' button is pressed, the indentation is increased by one.

Decrease Indent	
	Decreases the indentation of Command column cell contents on all selected rows of the Script Sheet each time the button is pressed.

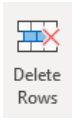
The ‘Decrease Indent’ button is used to decrease the indentation of contents of selected Command column contents. Because the Script Sheet is locked, the indent commands normally accessed within Excel are inaccessible on the Script Sheet. Therefore, the ‘Decrease Indent’ button is made available on the Script Sheet Ribbon.

To decrease the indentation of a single command, select the command’s row with the cursor and press the ‘Decrease Indent’ button. To decrease the indentation of a range of commands, select the range of commands and press the ‘Decrease Indent’ button. Each time the ‘Decrease Indent’ button is pressed, the indentation is decreased by one.

Insert Rows	
	Inserts single or multiple rows on selected rows of the Script Sheet.

Because the Script Sheet is locked, the insert row functions normally accessed within Excel are inaccessible on the Script Sheet. Therefore, the ‘Insert Rows’ button is provided on the Script Ribbon.

To insert a single row within a script, select the row in which the new row is desired. Pressing the ‘Insert Rows’ button will insert a row at that location. To insert multiple rows, select the number of rows that are desired, in the location of the script where the rows are needed. Pressing the ‘Insert Rows’ button will insert multiple rows at that point without overwriting any of the script contents,

Delete Rows	
	Deletes a single or multiple rows on selected rows of the Script Sheet.

Because the Script Sheet is locked, the delete row functions normally accessed within Excel are inaccessible on the Script Sheet. Therefore, the ‘Delete Rows’ button is provided on the Script Ribbon.

To delete a single row within a script, select the row to be deleted. Pressing the ‘Delete Rows’ button will delete that a row and all of its contents. To delete multiple rows, select the rows to be deleted. Pressing the ‘Delete Rows’ button will delete the multiple selected rows and their contents.

Script Sheet Script Load / Save Group

The Script Sheet Ribbon Row Edit Group consists of six buttons. This group of buttons is available only on the Script Sheet. The functions provided by the Script Load / Save Group provide for saving and loading script files, saving and loading module files, completely removing a script file, and for importing report sheets into the VIP Tool workbook.

Load Script	
	Opens a file dialog from which an entire script file can be loaded. Removes the current script and replaces the current script with the new script.

Pressing the ‘Load Script’ button will open a file dialog that allows the user to select a script to load into the Script Sheet. If a Script is selected, the current script and all results will be removed. The selected script will replace the current script.

Save Script	
	Opens a file dialog to save the entire current script. Allows the script to be named and stored for later recall.

Pressing the ‘Save Script’ button will open a file dialog that allows the user to select a path and name for the current script. The script is then stored as a ‘.txt’ ASCII text file in the selected path.

Insert Module Here	
	Opens a file dialog to select a script module. Creates rows and inserts the selected module at the current position of the cursor on the Script page.

A module is simply a small snippet of script that can be added to a larger script. The module may contain commands that are common to different scripts, such as certain subroutines or blocks of RCI commands. The ‘Insert Module Here’ button is used to insert the module at the desired location.

To use the ‘Insert Module Here’ button, select the desired row in which to insert the module, then press the ‘Insert Module Here’ button. A file dialog will open allowing the user to select the specific module to insert. The VIP Tool will insert the module at cursor location, creating rows to place the module within the script without overwriting any other rows in the script.

Save Selection as Module	
	Opens a file dialog to save the current selected rows to a module text file.

The ‘Save Selection as Module’ button allows the operator to select a range of commands and save them as a module that can then be imported to other scripts. For example, there may be a subroutine that is common to many scripts that the operator would like to make available when building script.

To use the ‘Save Selection as Module Button’, first select the range of rows that are to be saved. After selecting the rows to save as a module, press the ‘Save Selection as Module Button’. A file dialog will open allowing the user to assign the module a name and select the path that the module should be saved to. Modules are saved as ‘txt’ ASCII text files.

Remove Script	
	Clears results and removes the entire current script from the script page.

Pressing the ‘Remove Script’ button will cause the VIP Tool to remove the current script and delete all results from the Script, Trace Data and Results Data Sheets.

Import XLS Sheet	
	Opens a file dialog and imports the selected Excel spreadsheet into the VIP Tool. The VIP Tool assimilates any tags present on the imported sheet, provided the tags are visible on the imported sheet.

The ‘Import XLS Sheet’ button is used to import XLS sheets into the VIP Tool workbook. When this button is used, the VIP Tool internally registers the sheet as being part of the VIP Tool workbook and records the position of any tags present on the imported sheet. This button is a counterpart to the ‘Export Sheet to XLS’ button found on report pages present in the VIP Tool workbook.

When the ‘Import XLS Sheet’ button is pressed, a file dialog is opened allowing the user to select the sheet to import. After the sheet is selected, the VIP Tool automatically switches to ‘show tags’ mode, scans the imported sheet for tags, then switches back to the ‘hide tags’ mode after the sheet has been imported and scanned.

The Trace Data Sheet

Trace data is data returned from a specific trace query. Trace data consist of Comma Separated Values (CSV). Trace data can consist of oscilloscope data, spectrum or channel analyzer data, or data from various graphs such as Distribution or Power Over Time, as examples.

Trace data consist of two components: X Axis (horizontal) data, which can represent elements such as frequency or time, depending on the type of trace, and Y Axis (vertical) data, which generally represent some value of amplitude. When trace data is encountered by the program, the X axis and Y axis data are separated into two separate rows on the Trace Data worksheet. The "upper" row of a pair of trace data rows will contain the X axis values, and the "lower" row will contain the Y axis values. The trace data row pairs are separated from preceding or succeeding trace data row pairs by a blank row.

The X axis data are separated into specific columns on the Trace Data worksheet:

- Column A contains the Date the trace data were requested from the instrument.
- Column B contains the Time the trace data were requested from the instrument.
- Column C contains the number of trace data elements returned (and therefore the number of trace data columns on the worksheet).
- Column D contains the X axis designation, and horizontal units of the X axis data.

- Column E is left blank. This makes it easier to select the trace data with the Ctrl-A key combination provided by Excel.
- Columns F and higher contain the X axis trace data. The number of columns occupied by data is equal to the value in Column C.

The Y axis data are separated into specific columns on the Trace Data worksheet:

- Column A contains the Date the trace data were requested from the instrument.
- Column B contains the Time the trace data were requested from the instrument.
- Column C contains the query command that was used to access the data.
- Column D contains the Y axis designation of the X axis data.
- Column E is left blank. This makes it easier to select the trace data with the Ctrl-A key combination provided by Excel.
- Columns F and higher contain the Y axis trace data. The number of columns occupied by data is equal to number of X data values.

Trace data are automatically erased each time a script or a selected sequence of commands is executed.

NOTE:	Note that X data is generally returned by most RCI commands. When X data is not available for a specific trace query, the program calculates the X data and inserts it into the Trace Data worksheet.

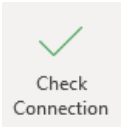
In the rare event that X data alone is available through a query, the program ignores the data and does not store it in the Trace Data worksheet when the query is sent as a command. When the Y data query is executed for the function, the program will automatically send the X data query, and store the X data results with the Y data in the format described above.

NOTE:	Note that if the argument cell is populated with a variable, tag or cell assignment command to redirect the output to Reply, then the trace data is not sent to the Trace Data sheet, but is instead assigned to the variable, tag or cell assignment and the trace data will appear in the reply column.

The Trace Data Sheet Ribbon

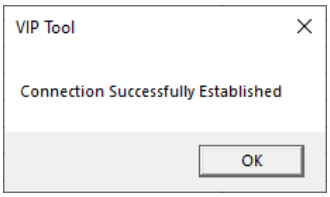
Trace Data Sheet Ribbon I/O Group

The Trace Data Sheet Ribbon I/O Group consists of only one button, the ‘Check Connection’ button.

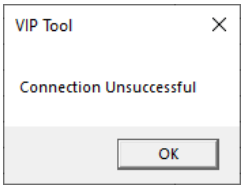
Check Connection	
	Opens a socket and checks to see if the VIP Tool has successfully connected to the instrument. Closes the socket after the check has completed.

The ‘Check Connection’ button is used to verify if the instrument is connected to the VIP Tool. This button shows up on several different sheets of the VIP Tool, and on each sheet the functionality of the button is the same. It is on the Setup Sheet that the VIP Tool is configured to communicate with the instrument through an Ethernet socket.

‘Check Connection’ attempts to open a socket to the instrument, based on the settings entered on the Setup Sheet. If a valid connection to the instrument is detected, a message indicating successful connection is displayed:



If, however, the VIP Tool cannot communicate with the instrument, a message indicating unsuccessful connection is displayed:



If the connection is unsuccessful, select the Setup Sheet and configure the model and IP address of the instrument to establish a successful connection. The VIP Tool will not execute script commands if there is not a valid connection to the instrument.

Trace Data Sheet Ribbon Script Operations Group

The Trace Data Sheet Ribbon Script Operations Group consists of two buttons, the ‘Run Script’ button and the ‘Abort Execution’ button.

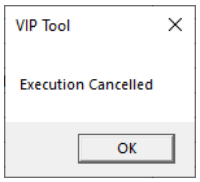
Run Script	
	Executes the entire script, starting at row 2 of the script page.

The ‘Run Script’ button is the main button used to begin execution of a full script. Pressing the ‘Run Script’ button will direct the VIP Tool to begin on row 2 of the script and continue execution until a blank command column cell is encountered or the ‘end’ command is encountered.

When the ‘Run Script’ button is pressed, the script starts executing with a ‘clear slate’. This means all previous results and trace data are deleted from the VIP Tool work sheet. Any previous variables and variable values are also cleared. If the script is programmed to output data to the currently viewed report sheet, the user can observe the data updating on the report sheet as the script runs.

Abort Execution	
	When pressed while a script is running will stop execution at the current row.

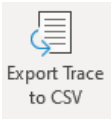
When the ‘Abort Execution’ button is pressed, script execution will immediately halt at its current operation.



When script execution is aborted, a message box will appear. Pressing the ‘OK’ button will clear this message box.

Trace Data Sheet File Operations Group

The Trace Data Sheet Ribbon Script Operations Group consists of only one button, the ‘Export Trace to CSV’ button.

Export Trace to CSV	
	Opens a file dialog to save all trace data on the Trace Data Sheet to a CSV file.

Pressing the ‘Export Trace to CSV’ button opens a file dialog that allows the user to select a path and save all trace data contained in the Trace Data sheet to an external CSV file.

NOTE:	Note that the default path for this function is not the path defined in the ‘Report CSV File’ field on the Setup Sheet.

The Results Data Sheet

Measurement reply data are the data returned by a query. On the script worksheet, the data is recorded in the reply column. Measurement reply data, except for trace data, is also recorded on the results data sheet. Argument data supplied by the info keyword will also be copied to the Results Data worksheet. Trace data is treated as a separate category of reply data that are sent to the Trace Data worksheet.

Measurement reply data can consist of a single datum, or as several data elements, separated by commas (termed Comma Separated Values or CSV). One function of the Results Data worksheet is to separate any CSV into separate columns to make evaluating a particular datum in a CSV string an easier process.

The data are separated out into specific columns on the Results Data worksheet:

- Column A contains the Date the data were requested from the instrument.
- Column B contains the Time the data were requested from the instrument.
- Column C contains the query command that was used to access the data.
- Column D will contain any argument that was used with the query.
- Column E will contain the first received datum. If the reply was a single datum, this will be the only column containing reply data.

If the reply was CSV data, Column E will contain the first datum in the CSV string.

Columns F and onward will each contain a single datum from any received CSV string, in the order of the data in the CSV string.

Following the reply data, Column E will contain the total time the script or selected commands took to complete.

If a trace query is sent, the data in Column E will indicate the number of points or pairs of data received by the trace query. Some trace data contain additional status byte or fail byte information. If the trace data contain that information, it will appear on the Results Data worksheet in addition to the number of points information.

Results data are automatically erased each time a script or a selected sequence of commands is executed.

The Results Data Sheet Ribbon

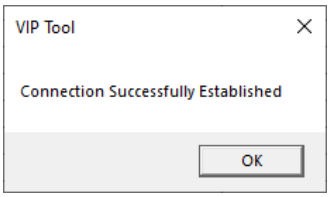
Results Data Sheet Ribbon I/O Group

The Results Data Sheet Ribbon I/O Group consists of only one button, the ‘Check Connection’ button.

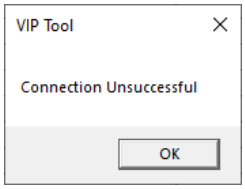
Check Connection	
	Opens a socket and checks to see if the VIP Tool has successfully connected to the instrument. Closes the socket after the check has completed.

The ‘Check Connection’ button is used to verify if the instrument is connected to the VIP Tool. This button shows up on several different sheets of the VIP Tool, and on each sheet the functionality of the button is the same. It is on the Setup Sheet that the VIP Tool is configured to communicate with the instrument through an Ethernet socket.

'Check Connection' attempts to open a socket to the instrument, based on the settings entered on the Setup Sheet. If a valid connection to the instrument is detected, a message indicating successful connection is displayed:



If, however, the VIP Tool cannot communicate with the instrument, a message indicating unsuccessful connection is displayed:



If the connection is unsuccessful, select the Setup Sheet and configure the model and IP address of the instrument to establish a successful connection. The VIP Tool will not execute script commands if there is not a valid connection to the instrument.

Results Data Sheet Ribbon Script Operations Group

The Results Data Sheet Ribbon Script Operations Group consists of two buttons, the 'Run Script' button and the 'Abort Execution' button.

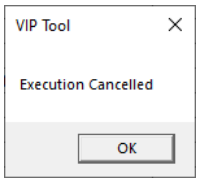
Run Script	
	Executes the entire script, starting at row 2 of the script page.

The 'Run Script' button is the main button used to begin execution of a full script. Pressing the 'Run Script' button will direct the VIP Tool to begin on row 2 of the script and continue execution until a blank command column cell is encountered or the 'end' command is encountered.

When the 'Run Script' button is pressed, the script starts executing with a 'clear slate'. This means all previous results and trace data are deleted from the VIP Tool work sheet. Any previous variables and variable values are also cleared. If the script is programmed to output data to the currently viewed report sheet, the user can observe the data updating on the report sheet as the script runs.

Abort Execution	
	When pressed while a script is running will stop execution at the current row.

When the ‘Abort Execution’ button is pressed, script execution will immediately halt at its current operation.



When script execution is aborted, a message box will appear. Pressing the ‘OK’ button will clear this message box.

Results Data Sheet File Operations Group

The Results Data Sheet Ribbon Script Operations Group consists of only one button, the ‘Export Results to CSV’ button.

Export Results to CSV	
	Opens a file dialog to save all results data on the Results Data Sheet to a CSV file.

Pressing the ‘Export Results to CSV’ button opens a file dialog that allows the user to select a path and save all results data contained in the Results Data Sheet to an external CSV file.

NOTE:	Note that the default path for this function is not the path defined in the ‘Report CSV File’ field on the Setup Sheet.

Report Sheets

A Report Sheet is any sheet in the VIP Tool workbook that is unlocked and available for full editing by Microsoft Excel. The Report Sheet must be a tab within the VIP Tool workbook; the VIP Tool cannot interface with a sheet that is not a part of the workbook. A Report Sheet can be used by the user to create reports using test data generated by the script. A Report Sheet can also be used as a supplemental sheet used for performing calculations on data generated by the script. Sheets that cannot be used as report sheets are the Getting Started Sheet, the Script Sheet, the Version Sheet, the Results Data Sheet, and the Trace Data Sheet.

The VIP Tool workbook is delivered with two sample Report Sheets included. One sheet is named ‘Calc’ and one sheet is named ‘Report’. These sheets are not write-protected and can be used or removed at the user’s discretion.

Report Sheet Rules

When adding and using a Report Sheet, these rules must be observed:

- A Report Sheet cannot be write-protected. If a sheet is write-protected, the VIP Tool cannot write to it. Attempting to write to a write-protected sheet will result in an error.
- A Report Sheet can use any name except for the already present names of other sheets that make up the VIP Tool (Getting Started, Setup, Script, Results Data, Trace Data, and Version). The name 'Utils' is also reserved and cannot be used as the name of a new sheet.

Tags

Report Sheets have the capability of having tags assigned to them. Tags provide a means of organizing the data generated by a script on the Report Sheet. When a value is assigned to a tag by the script, that value will appear anywhere the tag has been placed on a single or multiple Report Sheets. Report Sheet Ribbons will always have 'Show Tags' and 'Hide Tags' buttons, which are used for assigning tags to a Report Sheet. Report Sheets can be saved as a template, complete with tags, by pressing the 'Export Sheet to XLS' button on the Report Sheet Ribbon.

Saving Report Sheets

Report Sheet Ribbons contain the 'Save Report as PDF' and 'Save Report to CSV' buttons for saving report data as either a PDF file or a CSV worksheet. To use the 'Save Report to PDF' button, **two special tags must be present**: the **<**beginsave**>** tag is placed in the upper left cell of the area to be saved to a PDF; the **<**endsave**>** tag is placed in the lower right cell of the area to be saved to a PDF. The 'Save Report to PDF' function does not require these tags to export the report data to CSV.

The Report Sheet Ribbon

When a Report Sheet is selected, the Report Sheet Ribbon is brought into focus. The Report Sheet Ribbon contains functions specific to creating, saving, editing tags and exporting a Report Sheet template to a separate worksheet. The Report Sheet Ribbon also provides controls for running and aborting a script. The Report Sheet Ribbon is divided into four groups:

- I/O
- Script Operations
- Save Report
- Report Tools

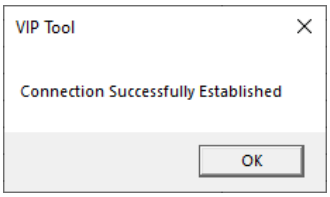
Report Sheet Ribbon I/O Group

The Report Sheet Ribbon I/O Group consists of only one button, the ‘Check Connection’ button.

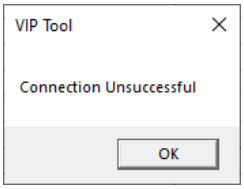
Check Connection	
	Opens a socket and checks to see if the VIP Tool has successfully connected to the instrument. Closes the socket after the check has completed.

The ‘Check Connection’ button is used to verify if the instrument is connected to the VIP Tool. This button shows up on several different sheets of the VIP Tool, and on each sheet the functionality of the button is the same. It is on the Setup Sheet that the VIP Tool is configured to communicate with the instrument through an Ethernet socket.

‘Check Connection’ attempts to open a socket to the instrument, based on the settings entered on the Setup Sheet. If a valid connection to the instrument is detected, a message indicating successful connection is displayed:



If, however, the VIP Tool cannot communicate with the instrument, a message indicating unsuccessful connection is displayed:



If the connection is unsuccessful, select the Setup Sheet and configure the model and IP address of the instrument to establish a successful connection. The VIP Tool will not execute script commands if there is not a valid connection to the instrument.

Report Sheet Ribbon Script Operations Group

The Report Sheet Ribbon Script Operations Group consists of two buttons, the ‘Run Script’ button and the ‘Abort Execution’ button.

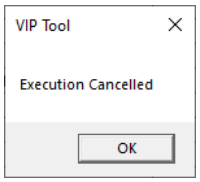
Run Script	
	Executes the entire script, starting at row 2 of the script page.

The ‘Run Script’ button is the main button used to begin execution of a full script. Pressing the ‘Run Script’ button will direct the VIP Tool to begin on row 2 of the script and continue execution until a blank command column cell is encountered or the ‘end’ command is encountered.

When the ‘Run Script’ button is pressed, the script starts executing with a ‘clear slate’. This means all previous results and trace data are deleted from the VIP Tool work sheet. Any previous variables and variable values are also cleared. If the script is programmed to output data to the currently viewed report sheet, the user can observe the data updating on the report sheet as the script runs.

Abort Execution	
	When pressed while a script is running will stop execution at the current row.

When the ‘Abort Execution’ button is pressed, script execution will immediately halt at its current operation.



When script execution is aborted, a message box will appear. Pressing the ‘OK’ button will clear this message box.

Report Sheet Ribbon Save Report Group

The Report Sheet Ribbon Save Report Group consists of two buttons, the ‘Save Report as PDF’ button and the ‘Export Report to CSV’ button.

Save Report as PDF	
	Saves the area of the report page defined by the <beginsave> and <endsave> tags to a PDF file.

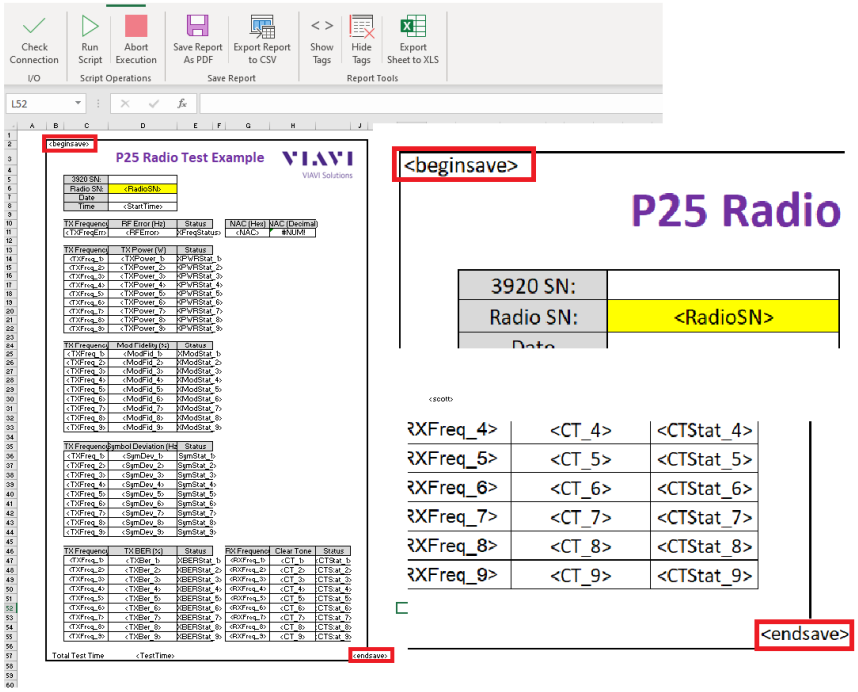
If the report page has a <beginsave> and an <endsave> tag defining the upper left and lower right cells of the print area, the ‘Save Report as PDF’ button can be used to save the currently viewed page as a PDF file. If the currently selected Report Sheet does not contain the <beginsave> and <endsave> tags, the VIP Tool will attempt to open the Report Sheet selected in the Setup Sheet ‘Report Sheet’ field. If the sheet designated in that field also does not contain the <beginsave> and <endsave> tags, no PDF report will be saved, and a message indicating that the VIP Tool could not locate the <beginsave> and <endsave> tags will be display.

The PDF report will be saved to the path designated in the Setup Sheet 'Report PDF File Path' field. If there is no designated path in that field, the PDF report will be saved to the same path the VIP Tool workbook is located in. The PDF report name will always contain a time/date stamp of when the report was saved. If the 'Test Name' field of the Setup Sheet has a name entered in it, that name will be pre-pended to the report file name date/time stamp.

If the 'Open PDF Report on Save' field of the Setup Sheet is set to 'On', then the PDF file will automatically be opened by the program designated for viewing PDF files on the user's computer.

Setting Up A Save Selection Area

The <beginsave> and <endsave> tags are used to designate the print area of the report sheet when it is saved either by using the 'Save Report to PDF' button or programmatically saving the report using the script command. This allows the user to define what part of a report sheet will appear on the saved PDF report.



Place the <beginsave> tag in the upper left cell of the report page area that is to be saved. Place the <endsave> tag in the lower right cell of the report page area to be saved. The block of cells defined by these two points will then be used as the selection area when saving the PDF report. The <beginsave> and <endsave> tags are not required for exporting reports to the CSV format.

Export Report as CSV	
Export Report to CSV	Exports the contents of the report page in Comma Separated Value (CSV) format to a separate file.

Pressing the 'Export Report as CSV' button will export the currently selected report page to a CSV file.

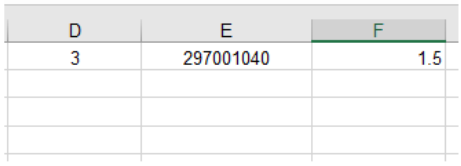
The CSV report will be saved to the path designated in the Setup Sheet ‘Report CSV File Path’ field. If there is no designated path in that field, the CSV report will be saved to the same path the VIP Tool workbook is located in. The PDF report name will always contain a time/date stamp of when the report was saved. If the ‘Test Name’ field of the Setup Sheet has a name entered in it, that name will be pre-pended to the report file name date/time stamp.

Report Sheet Ribbon Report Tools Group

The Report Sheet Ribbon Report Tools Group consists of three buttons, the ‘Show Tags’ button, the ‘Hide Tags’ button and the ‘Export Sheet to XLS’ button. These buttons address organizing Report Sheets using tags and exporting a sheet template to an external file for use in other VIP Tool workbooks.

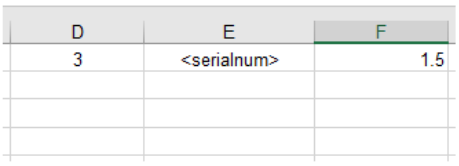
Show Tags	
< > Show Tags	Places the VIP Tool worksheet into the ‘Show Tags’ mode. This mode is required for entering and editing tags on a report sheet.

A Tag is a type of variable used by the VIP Tool. A Tag is a word enclosed in angle brackets, such as <mytag>, for example. Tags are used in the VIP Tool as a means of easily transferring data generated by the script to a report page. To enter Tags into a report page or to observe and edit Tags, the ‘Show Tags’ button needs to be pressed to place the VIP Tool into the ‘Show Tags’ mode. When in the ‘Show Tags’ mode, the Tags in a report page become visible. In this mode, Tags can be entered, deleted, copied to new locations, or moved. Once the Tags are in the position desired by the user, the position of the Tag is recorded, and the Tag is hidden when the ‘Hide Tags’ button is pressed.



D	E	F
3	297001040	1.5

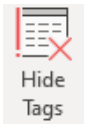
Before the 'Show Tags' button is pressed.



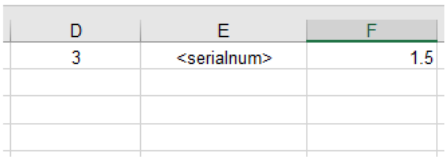
D	E	F
3	<serialnum>	1.5

After the 'Show Tags' button is pressed

In the above example, the serial number of the instrument is displayed in Column E. In this case, the script was used to send a *IDN? query, and the serial number field of the query response was assigned to the Tag <serialnum>. On the right side of the example, the ‘Show Tags’ button has been pressed and the VIP Tool is in the ‘Show Tags’ mode. Instead of showing the serial number of the instrument, the Tag that was used to assign that value to the cell (<serialnum>) is displayed instead. At this point, the Tag can be copied, moved, or deleted. Pressing ‘Hide Tags’ will remove the Tag from view, and the value assigned to the Tag will be restored at any and every position the Tag was copied or moved to.

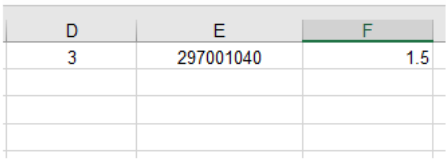
Hide Tags	
	Places the VIP Tool into the ‘Hide Tags’ mode. Pressing the button automatically saves all Tags present in the VIP Workbook.

Tags are used in the VIP Tool as a means of easily transferring data generated by the script to a report page (see above ‘Show Tags’ description). The ‘Hide Tags’ button is used to disable the ‘Show Tags’ mode of the VIP Tool. When the ‘Hide Tags’ button is pressed, the position of all Tags in the VIP Tool worksheet is recorded, and the Tags are hidden from view. If a value was assigned to a Tag before the ‘Show Tags’ button was pressed, pressing the ‘Hide Tags’ button will reassign the value to the Tag.



D	E	F
3	<serialnum>	1.5

Before the 'Hide Tags' button is pressed.



D	E	F
3	297001040	1.5

After the 'Hide Tags' button is pressed.

In the above example, the Tag the script assigns for the serial number of the instrument is displayed in Column E. If the serial number value was present when the ‘Show Tags’ button was pressed, pressing the ‘Hide Tags’ button will restore the previous value. If there was no value present, or if the Tag was newly created while in the ‘Show Tags’ mode, pressing the ‘Hide Tags’ button will record the Tag value position and hide the Tag from view.

NOTE:	It should be noted that if the VIP Tool is in the ‘Show Tags’ mode when ‘Run Script’, ‘Run Selection’, ‘Run From Here’ or ‘Single Step From Here’ buttons are pressed, the VIP Tool will automatically implement the ‘Hide Tags’ function to prevent overwriting any displayed Tags.

Export Sheet to XLS	
	Exports report template complete with Tags to a separate Excel spreadsheet.

Pressing the ‘Export Sheet to XLS’ button will export the selected Report Sheet to the path specified in the Setup Sheet ‘Import / Export Sheet Path’ field. If no path is specified in that field, then the current Report Sheet will be exported to the path the VIP Tool workbook is located in. When the Report Sheet is exported, the VIP Tool is placed in the ‘Show Tags’ mode before the sheet is exported, thus the exported sheet will have its Tags visible when it is exported. This functionality allows the sheet to be imported, complete with Tags, into other copies of the VIP Tool worksheet.

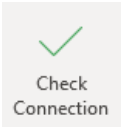
The Version Sheet

The Version Sheet will contain the current version number and release date of the VIP Tool programmed contained in the workbook. In addition, any changes from previous versions will be listed in this sheet.

The Version Sheet Ribbon

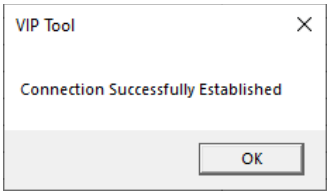
Results Data Sheet Ribbon I/O Group

The Results Data Sheet Ribbon I/O Group consists of only one button, the ‘Check Connection’ button.

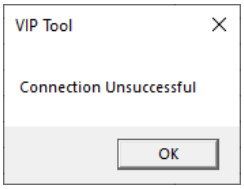
Check Connection	
	Opens a socket and checks to see if the VIP Tool has successfully connected to the instrument. Closes the socket after the check has completed.

The ‘Check Connection’ button is used to verify if the instrument is connected to the VIP Tool. This button shows up on several different sheets of the VIP Tool, and on each sheet the functionality of the button is the same. It is on the Setup Sheet that the VIP Tool is configured to communicate with the instrument through an Ethernet socket.

‘Check Connection’ attempts to open a socket to the instrument, based on the settings entered on the Setup Sheet. If a valid connection to the instrument is detected, a message indicating successful connection is displayed:



If, however, the VIP Tool cannot communicate with the instrument, a message indicating unsuccessful connection is displayed:



If the connection is unsuccessful, select the Setup Sheet and configure the model and IP address of the instrument to establish a successful connection. The VIP Tool will not execute script commands if there is not a valid connection to the instrument.

VIP Tool Programming Language Reference

The VIAVI Instrument Programming Language is used to organize, branch, debug and control the flow of a VIP Tool script. The language also provides special functions that aid in the process of testing radios and other devices.

The language provides for the use of variables and takes advantage of Microsoft Excel functions for storing data, transferring data, and creating test reports.

A major feature of the language is the use of Tags to create and organize report data. Another feature is that the language provides methods of interactively inputting numerical and alphanumerical data as the script runs.

This reference is divided up into several sections.

- Variables Tags and Cell Assignments

  Different types of variables used in the language include variables, tags and cell assignments.

- Counters and Timers

  The language provides counters and timers to control various elements of a script.

- Goto and Subroutine Functions

  Goto and subroutine functions allow scripts to be organized and compact through reuse of code.

- Loops and Conditional Statements

  Loops and conditional statements allow the flexibility of branching and iterative tasks.

- Flow Control, Messages and Forms

  Flow control consists of keywords and messages and forms to introduce delays, pauses and data input based upon user entry and detected signals from the instrument.

- Special Functions

  Special functions consist of an automated SINAD test, calculators, and a means to control what the operator can observe as a script runs.

- Report Tools

  A set of commands that allow the script to programmatically save test data.

- Debug and Notation Tools

  A set of symbols and commands that provide for remark notation, selectively disabling sections of a script, break points and the ability to print the contents of variables, tags, and cell assignments for debugging a script.

- Utilities

  Commands that can be operated outside of a script for performing tag creation and copying cell data to set up a test environment.

Variables, Tags, and Cell Assignments

The VIP Tool provides for the use of variables and variable arrays. All variables are global to the script – there is no differentiation of local and global variables.

Variables

Variable Syntax

A variable is created by placing a variable name within square brackets. For example [myvariable] represents the variable 'myvariable'. The VIP Tool will recognize any word or number enclosed in brackets as a variable.

Variable names cannot use spaces or special characters other than the '_' symbol (which is used to denote an array of variables of the same base name). Variables are not case sensitive.

A variable can use the underscore '_' symbol to designate an array of variables using the same base name. The VIP Tool utilizes the various counter functions available for assigning values to an array. For example, the For Next loop function has a built-in counter that increments or decrements with each step of the For Next loop. The For Next counter uses the keyword 'nextcount'. Creating a variable called [myvariable_nextcount] within a For Next loop will create an array using 'myvariable' as the base name. For example, if the For Next loop counts from 1 to 3, then placing a variable declaration of [myvariable_nextcount] within the loop will create the array variables [myvariable_1], [myvariable_2] and [myvariable_3]. Other counters can be used to create arrays from a base variable name. These techniques will be discussed later in this document.

Declaring Variables and Assigning Values

A variable is declared by assigning a value to the variable. The value held by a variable can be a number or text. It is possible to concatenate two text variables. Mathematical functions including addition, subtraction, multiplication, and division can be performed using two numerical variables.

When assigning a text value to a variable, it is recommended to enclose the text in double quotes, particularly if there is a space in the text value. If the text variable is to be used as an argument to an RCI command that requires quotes in the argument, the text must be enclosed in two sets of double quotes when assigning the value to a variable.

Command Column Variables

A variable can be declared and assigned a value by placing the variable in the command column, followed by the '=' sign, and placing the value in the argument column.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	[mytextvariable]=	"This is a test."		[MYTEXTVARIABLE] = This is a test.
3	[mynumbervariable]=	151.0125		[MYNUMBERVARIABLE] = 151.0125
4	[mytextvariable]=	"This is another test."		[MYTEXTVARIABLE] = This is another test.
5	[mynumbervariable]=	451.0725		[MYNUMBERVARIABLE] = 451.0725
6				

In the above example, on row 2, the value of "This is a test" is assigned to the variable [mytextvariable].

NOTE:	Note that in the command column, the '=' sign follows the variable name. In the argument column, the value "This is a test." is enclosed in double quotations because it is a text value. When a value is assigned to a variable, the Info Message column will display the uppercase form of the variable name and the value assigned to the variable.

On row 3, the numerical value of 151.0125 is assigned to the variable [mynumbervalue]. Again, in the command column, the '=' sign follows the variable name. In the argument column, the value 151.0125 is placed without double quotations because it is a numerical value.

Variable values can be reassigned at any time. On rows 4 and 5, each variable is reassigned a different value.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mysystem]=	""Analog Duplex""		[MYSYSTEM] = "Analog Duplex"
3	:SYSTem:LOAD	[mysystem]		:SYSTem:LOAD "Analog Duplex"
4				

There are instances where double quotes may be required for a text value. For example, the 39xx instrument command to load a system requires the argument to be enclosed in double quotations. Other examples may be RCI commands that are used to load saved setups or pre-sets.

In the above example, the 39xx command to load the analog duplex system is:

:SYSTem:LOAD "Analog Duplex"

The variable [mysystem] is assigned the value of "Analog Duplex" complete with the double quotations. In order to assign the value "Analog Duplex" including the double quotations, the argument column value for the variable is enclosed in two sets of double quotations: ""Analog Duplex"". The Info Message column displays the actual value assigned as "Analog Duplex". On row 3, the variable is used as the argument to the :SYSTem:LOAD command, so executing rows 2 and 3 will cause the 39xx instrument to load the Analog Duplex system.

1	Command	Argument	Reply	Info Message
2	[mynumber]=	151.0125		[MYNUMBER] = 151.0125
3	[mytext]=	"Hello World"		[MYTEXT] = Hello World
4	[myothernumber]=	[mynumber]		[MYOTHERNUMBER] = 151.0125
5	[myothertext]=	[mytext]		[MYOTHERTEXT] = Hello World
6				

Values can be passed from one variable to another. In the above example, on rows 2 and 3, the variables [mynumber] and [mytext] are each assigned a value. On rows 4 and 5, the values held by these two variables are passed to [myothernumber] and [myothertext] respectively.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[myserialnumber]=	<serialnum>		[MYSERIALNUMBER] = 297001040
3	[myfrequency]=	(examples_D1)		[MYFREQUENCY] = 202.0125
4				

Values can be passed to a variable from a tag or an Excel sheet cell using the Cell Assignment syntax. In the above example, a report page has a tag called <serialnum> that is holding the value 297001040. On another sheet within the workbook, called 'examples', cell D1 contains the value 202.0125.

On row 2, the value of the <serialnum> tag is assigned to the variable [myserialnumber]. On row 3, the value held in cell D1 of a sheet called 'examples' is assigned to the variable [myfrequency].

Command Column Placeholder Variables

Certain VIAVI Instrument Programming Language commands provide the use of introducing variables into the syntax of the command itself. These variables are referred to in this guide as 'placeholder variables'. A placeholder variable is denoted in the description of the command as being enclosed in curly brackets, such as '{variable}'. The variable syntax itself does not use curly brackets; the curly brackets are simply used as an instrument to denote them in the descriptions provided in this guide.

Placeholder variables have a separate set of syntax rules. Because they only appear in the command column, the variable name cannot have a space in it. The variable also cannot have an underscore "_" character in it. Variables and tags can be used for placeholder variables. Cell assignments are not allowed for use as a placeholder variable.

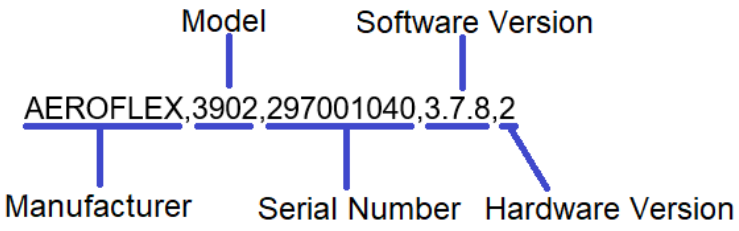
Argument Column Variables

A variable can be declared and assigned a value by placing the variable in the argument column and using an RCI query to retrieve the data to be assigned to the variable. In this case, the variable name is followed by some form of the '=reply' keyword.

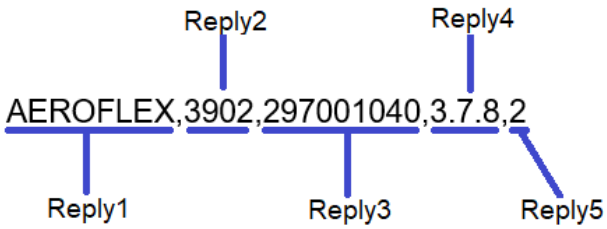
The '=reply' keyword is used to select the data from the reply that will be assigned to the variable. This structure is used so that the desired value can be retrieved from a reply that has multiple data in the Comma Separated Value (CSV) format when returned by the instrument.

A common example of such a reply would be the response to the query '*IDN?'. The '*IDN?' query returns data such as manufacturer, model number, serial number, and software version of the instrument in the CSV format. The '=reply' keyword allows the user to select which of those data elements is to be assigned to the variable.

When the '=reply' keyword contains a number, that number is used to select which element of the CSV response is to be assigned to the variable. For example, the response of a '*IDN?' query will be structured as some variation of this example:



Following the '=Reply' keyword with a number will select the field to be stored by the variable, in the order the field appears in the response, starting at 1.



In the example above, '=Reply1' will assign the variable the name 'AEROFLEX', '=Reply2' will assign the variable the model number '3902', '=Reply3' will assign the variable the serial number '297001040', '=Reply4' will assign the variable the software version'3.7.8.2', and '=Reply5' will assign the variable the hardware version '2'. If no number follows the '=Reply' keyword, the entire string will be assigned to the variable. For responses that contain only one field (in other words, not a multi-field CSV string), then '=Reply' must be used.

The number range of the '=Reply' keyword is 1 through 8, so, for example, '=Reply8' is supported, but '=Reply9' is not supported. The '=Reply' keyword is not case sensitive.

	Command	Argument	Reply	Info Message
2	*idn?	[manufacturer]=Reply1	AEROFLEX,3902,297001040,3.7.8,2	[MANUFACTURER] = AEROFLEX
3	*idn?	[model]=Reply2	AEROFLEX,3902,297001040,3.7.8,2	[MODEL] = 3902
4	*idn?	[serialnum]=Reply3	AEROFLEX,3902,297001040,3.7.8,2	[SERIALNUM] = 297001040
5	*idn?	[swver]=Reply4	AEROFLEX,3902,297001040,3.7.8,2	[SWVER] = 3.7.8
6	*idn?	[hwver]=Reply5	AEROFLEX,3902,297001040,3.7.8,2	[HWVER] = 2
7	*idn?	[wholestring]=Reply	AEROFLEX,3902,297001040,3.7.8,2	[WHOLESTRING] = AEROFLEX,3902,297001040,3.7.8,2
8				

In the above illustration, the manufacturer name field value is assigned to the variable '[manufacturer]', the model number field value is assigned to the variable '[model]', the serial number field value is assigned to the variable '[serialnum]', the software version field value is assigned to the variable '[swver]', and the hardware version field value is assigned to the variable '[hwver]'. The entire string is assigned to the variable '[wholestring]'.

Query Arguments and Variables

In some cases, an instrument query may require an argument. For example, the 39xx instrument query for broadband power uses the following syntax:

:FETCh:RF:ANALyzer:TRBPower? <units>

The query allows the user to specify the unit of measurement of the broadband power value by providing a units argument with the query. The unit of measurement can be set to Watts, dBm, or dBW using 'W', 'dBm' or 'W' respectively. The response of the broadband power meter query is CSV formatted as follows:

<statusbyte>,<failbyte>,<avgcount>,<avg>

The power reading value to be assigned to the variable is in the <avg> field, which is the fourth field of the CSV reply string, so the user will use '=Reply4' to retrieve that value and assign it to the variable. In this case, the units argument can be placed either before or after the "=Reply4" keyword.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:fetch:rf:anal:trbp?	[pow]=reply4 dBm	0,0,1,9.2	[POW] = 9.2
3	:fetch:rf:anal:trbp?	[pow]=reply4 W	0,0,1,0.0083	[POW] = 0.0083
4	:fetch:rf:anal:trbp?	[pow]=reply4 dBW	0,0,1,-20.7	[POW] = -20.7
5				

In the above illustration, the variable '[pow]' is assigned the power measurement using the broadband power query. In row 2, 'dBm' is supplied as the query argument, so the dBm value of the measurement is retrieved and assigned to '[pow]'. In row 3, 'W' is supplied as the query argument, so the Watts value of the power measurement is retrieved and assigned to '[pow]'. In row 4, 'dBW' is supplied as the query argument, so the dBW value of the measurement is retrieved and assigned to '[pow]'.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:fetch:rf:anal:trbp?	dBm [pow]=reply4	0,0,1,9.2	[POW] = 9.2
3	:fetch:rf:anal:trbp?	W [pow]=reply4	0,0,1,0.0084	[POW] = 0.0084
4	:fetch:rf:anal:trbp?	dBw [pow]=reply4	0,0,1,-20.8	[POW] = -20.8
5				

The same results will occur if the query argument is placed *before* the variable assignment in the argument column.

Creating Variable Arrays

A variable array can be constructed by creating a base variable name followed by an underscore and a counter name within the bracket. For example, '[myvariable_nextcount]' will create the variable name followed by the underscore and the current value of 'nextcount'. So, if the current value of nextcount is 2 when the variable is declared, the variable name will be [myvariable_2].

'Nextcount' is the name of the counter built into the For Next loop. There are seven different counters that can be used to create variable arrays.

Counter Name	Counter Information
nextcount	Integer up or down counter available with each instance of a For Next Loop
docount	Integer up counter available with each instance of a Do Loop
counter1	Autonomous decimal up or down counter
counter2	Autonomous decimal up or down counter
counter3	Autonomous decimal up or down counter
counter4	Autonomous decimal up or down counter
counter5	Autonomous decimal up or down counter

All counters but 'docount' can count up or down; 'docount' can only count up. All the counters can be assigned arbitrary values. Counters 1 through 5 can be assigned integer or decimal values. When using counters 1 through 5 for creating variable array names, only integers can be used.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	for	1 to 5		For 1 to 5
3	:FETCh:AF:ANALyzer:SINad?	[sinad_nextcount]=reply4	0,0,1,11.54,0.00	[SINAD_5] = 11.54
4	delay	1		delay 1
5	next			
6	print	[sinad_1]		[SINAD_1] = 11.54
7	print	[sinad_2]		[SINAD_2] = 11.40
8	print	[sinad_3]		[SINAD_3] = 11.54
9	print	[sinad_4]		[SINAD_4] = 11.34
10	print	[sinad_5]		[SINAD_5] = 11.54
11				

In the above example, a For Next loop is set to count from 1 to 5. For each iteration of the loop, a SINAD measurement is taken on row 3. The argument for row 3 is '[sinad_nextcount]=reply4'. Each time the loop steps to row 3, the SINAD measurement is taken, and 'reply4', which is the SINAD measurement taken from the return string to the query, is assigned to '[sinad_nextcount]'. 'Nextcount is the built-in counter for the For Next loop. Its current count is assigned to the name of '[sinad_nextcount] to which the current SINAD reading is assigned. So, on the first step, the variable is named '[sinad_1]', which is assigned the current reading. On the second step, the variable is named '[sinad_2]', which is assigned the next current reading. This procedure is repeated from 1 to 5, the For Next range.

'Print' is a VIP Tool debugging keyword. Placing 'print' in the Command column and the name of a variable in the Argument column will result in the Info Message column displaying the variable name and the value stored in the variable.

After the For Next loop has run, on rows 6 through 10, 'print' is used to display each name of the variable array created by the For Next loop and the value that was stored in each variable of the array.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	do			
3	:FETCh:AF:ANALyzer:SINad?	[sinad_docount]=reply4	0,0,1,11.46,0.00	[SINAD_5] = 11.46
4	exitdo	docount = 5		If 5 = 5 = True
5	delay	1		delay 1
6	loop			
7	print	[sinad_1]		[SINAD_1] = 11.55
8	print	[sinad_2]		[SINAD_2] = 11.25
9	print	[sinad_3]		[SINAD_3] = 11.54
10	print	[sinad_4]		[SINAD_4] = 11.50
11	print	[sinad_5]		[SINAD_5] = 11.46
12				

In the above illustration, 'docount' is used to set the array number. By default, 'docount' increments to 1 on the first iteration of a Do Loop and increments on each iteration of the loop. In this example, the Do Loop is set to exit when 'docount' is equal to 5 on row 4.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	set_counter1	4	4	
3	for	1 to 5		For 1 to 5
4	inc_counter1		9	
5	:FETCh:AF:ANALyzer:SINad?	[sinad_counter1]=reply4	0,0,1,11.44,0.00	[SINAD_9] = 11.44
6	delay	1		delay 1
7	next			
8	print	[sinad_5]		[SINAD_5] = 11.31
9	print	[sinad_6]		[SINAD_6] = 11.30
10	print	[sinad_7]		[SINAD_7] = 11.68
11	print	[sinad_8]		[SINAD_8] = 11.40
12	print	[sinad_9]		[SINAD_9] = 11.44
13				

The above example uses counter1 to set the array number. In this case, counter1 is set to a value of 4 before the For Next loop starts. The 'inc_counter1' command on row 4 increments counter1 by the default value of 1 each time the For Next loop iterates. Counter1 is incremented by 1 before the first SINAD query is sent on row 5, so the first array number will be 5. The For Next loop is set to iterate 5 times before the loop ends. Because the counter1 value used for the array is incremented 5 times, the array number steps from 5 to 9 for the 5 iterations of the loop. Thus, the first array name is '[sinad_5]', the second array name is '[sinad_6]' and so forth until the terminal count of the For Next loop, which sets the last array name to '[sinad_9]'.

Variable as Argument for RCI Command

A variable can be used to hold the argument for an RCI command.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:FETCh:RF:ANALyzer:FCOUNTer?	[rf_freq]=reply	151.058	[RF_FREQ] = 151.058
3	:RF:GENerator:FREQuency	[rf_freq]		:RF:GENerator:FREQuency 151.058
4				

In this example, row 2 sends a query for the RF frequency counter reading of a 39xx series instrument. The value of the frequency counter is assigned to the variable '[rf_freq]' in the argument column of row 2.

On row 3, the command to set the RF generator frequency uses the '[rf_freq]' variable value to set the RF generator to the same frequency as the value read from the frequency counter on row 2.

NOTE:	It should be noted that a variable **cannot** hold an RCI command or query for use in the command column.

Tags

The VIP Tool uses a system of tags to transfer values efficiently between the Script Sheet and Report Sheets. Tags are particularly useful for building test result reports. When using tags, the programmer does not have to use the script to designate which specific sheet and cell number to send a value to but can instead assign the value to a tag and the VIP Tool will transfer the value to any cell that is assigned that tag name. Tags can be freely moved around to arrange data on a report sheet without having to change procedures in the script.

A <tag> is a type of variable that broadcasts its value to all other tags sharing the same tag name. In a sense, a <tag> is a variable that is global to the entire VIP Tool workbook, whereas a [variable] is local to the Script Sheet.

A tag is defined as a tag name enclosed in angle brackets such as <mytag>. The value of a tag can be assigned from the command column with a specific form of syntax, or from the argument column using a different form of syntax. The value of a tag can be entered manually as well. Tag functionality is only available on sheets other than Getting Started, Setup, Trace Data, Results Data, and Version. Tag values can be assigned by direct entry, but only running a script will procedurally assign and read values of tags on other sheets.

Working with Tags

A tag is created by placing a variable name within angle brackets. For example, <mytag> represents the tag 'mytag'. The VIP Tool Script page will recognize any word or number enclosed in angle brackets as a tag. If the VIP Tool is placed in the 'Show Tags' mode by pressing the 'Show Tags' button, entering a word within angle brackets creates the tag on a report page. The tag is registered only after the 'Hide Tags' button is pressed. If the VIP Tool is not in the 'Show Tags' mode when a word within angle brackets is entered in a report page, it will **not** be recognized or registered as a tag by the VIP Tool.

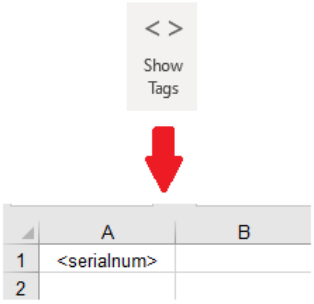
Tag names cannot use spaces or special characters other than the '_' symbol (which is used to denote an array of tags of the same base name). Tags are not case sensitive.

A tag can use the underscore '_' symbol to designate an array of tags using the same name. The VIP Tool utilizes the various counter functions available for assigning values to an array. For example, the For Next loop function has a built-in counter that increments or decrements with each step of the For Next loop. The For Next

counter uses the keyword ‘nextcount’. Creating a tag called <mytag_nextcount> within a For Next loop will create an array using ‘mytag’ as the base tag. For example, if the For Next loop counts from 1 to 3, then placing a variable declaration of <mytag_nextcount> within the loop will create the array tags <mytag_1>, <mytag_2> and <mytag_3>. Other counters can be used to create arrays from a base tag name. These techniques will be discussed later in this document.

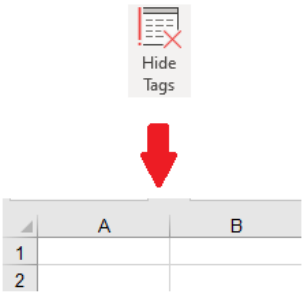
Creating a Tag on a Report Sheet

To create a tag on a Report Sheet, go to the Report Sheet and press the ‘Show Tags’ button. At that point, at tag can be entered in the Report Sheet.



In this example, the tag <serialnum> has been entered in the Report Sheet in cell A1.

To register the tag’s position with the VIP Tool, press the ‘Hide Tags’ button.



After pressing the ‘Hide Tags’ button, the tag disappears from view. Pressing the ‘Hide Tags’ button also registers the tag’s position within the VIP Tool workbook.

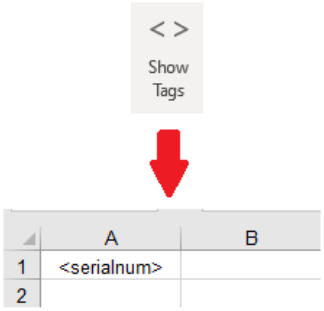
Assigning Values to Tags and Arranging Tags

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	*idn?	<serialnum>=reply3	AEROFLEX,3902,297001040,3.7.8,2	<SERIALNUM> = 297001040
3				

In the above illustration, the ‘*IDN?’ query is sent. The argument column syntax assigns the serial number field of the reply, 297001040, to the ‘<serialnum>’ tag.

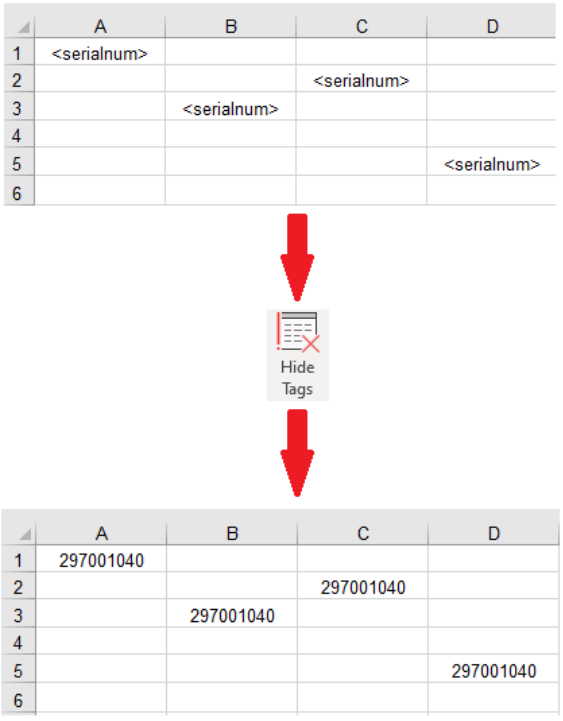
	A	B
1	297001040	
2		

After returning to the report page, the value assigned to the '<serialnum>' tag appears in the position the '<serialnum>' tag was recorded in when the 'Hide Tags' button was pressed.



If the 'Show Tags' button is pressed again, the '<serialnum>' tag can still be seen marking the position of the tag.

If the tag is moved or copied to other positions on the same Report Sheet or any other Report Sheet while in the 'Show Tags' mode, pressing the 'Hide Tags' button will register the position of all of the tags, and restore the value of the tag, if any, to the tag positions.



In the above example, the '<serialnum>' tag was copied from cell A1 to cells B3, C3, and D5. After pressing the 'Hide Tags' button, the value assigned to the '<serialnum>' tag appears in all of these locations.

Rules of Tag Use

The nature of using tags requires that they be invisible when executing a script. Creating and assigning a tag to a worksheet cell involves first *showing* the tag names to make the tags visible. This is accomplished by pressing the 'Show Tags' button on the VIP Tool ribbon. Pressing 'Show Tags' puts the VIP Tool into the 'Show Tags' mode. In this mode, all tags are made visible to the programmer. Any tag entered when not in the 'Show Tags' mode will not be registered or recognized as a tag by the VIP Tool.

A tag must first exist on a report page for the value to be transferred to or read from. If a tag does not exist, the script will still assign a value to a tag name, but the value will not be observable or retrievable. Later operations with the non-existent tag will only produce a null value, though a value was previously assigned to the non-existent tag.

Only in the 'Show Tags' mode, is it possible to create a tag. A tag is created by putting a tag name into a cell while the VIP Tool is in the 'Show Tags' mode and subsequently pressing 'Hide Tags' to save the tag.

The tag name must be enclosed in angle brackets, such as <tagname>. After placing a tag in a cell, pressing the 'Hide Tags' button will cause the tag to disappear from view in the cell. Internally, however, the VIP Tool memorizes the tag location when 'Hide Tags' is pressed.

When a tag is given a value by the script, VIP Tool will automatically copy that value to any cell that has that tag name assigned to it. If a tag has a value currently assigned to it, pressing the 'Show Tags' button will still display the tag name, and, subsequently, after pressing the 'Hide Tags' button, the value assigned to that tag, if any, will be restored to the cell.

If the VIP Tool is in 'Show Tags' mode when a script or command is executed on the script page, the VIP Tool will automatically switch to the 'Hide Tags' mode before executing any command to prevent overwriting any tag names. The action of automatically switching to 'Hide Tags' automatically saves the location information of all tags in the VIP Tool workbook.

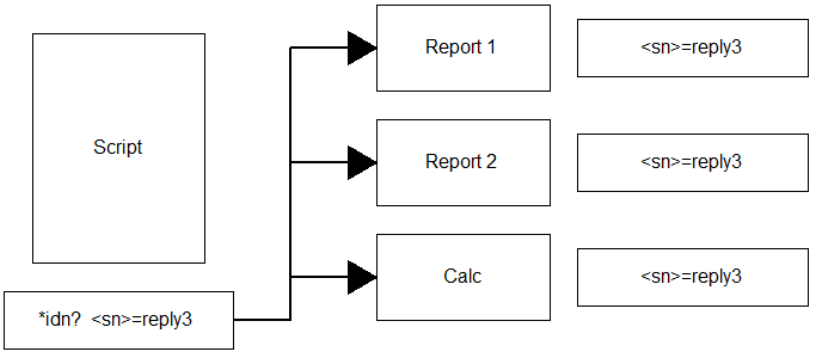
An Excel formula can occupy a cell that has been marked with a tag. When the value of a tag occupied by the formula is retrieved, the retrieved value will be the result of the formula, not the formula itself.

Unlike variables, whose values are automatically erased when a script or portion of a script is executed, tag values are not erased until they are overwritten, or until the script programmatically erases the tag values with the 'clear_tags' command.

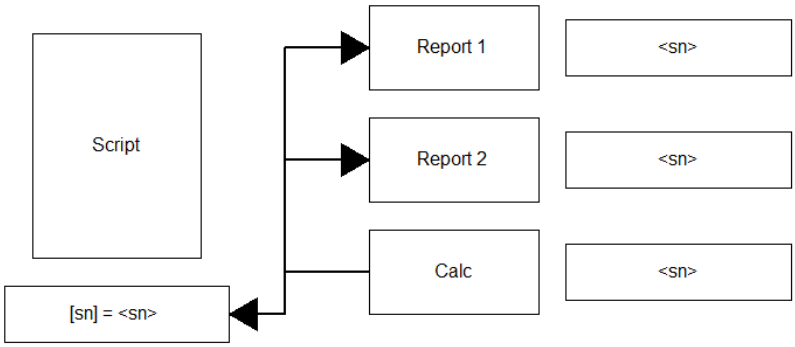
NOTE:	Note that formulas located in a cell marked by a tag are not erased to preserve the formula.

Considerations for Assigning Values to Tags

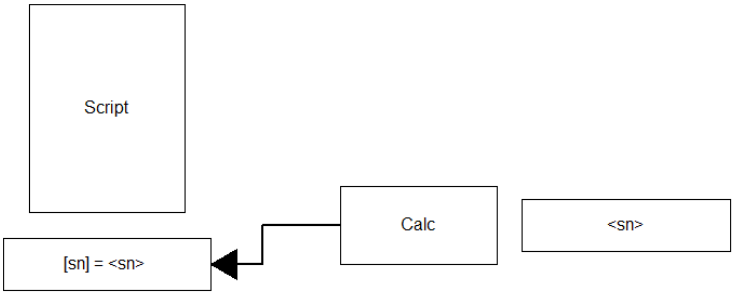
As mentioned before, tag values can be assigned procedurally from a script. A tag value can also be read by the script. For example, if on a report page, the programmer chooses to have a value entered on the report page, such as selecting a radio operating mode, for example, the script can be used to incorporate that information into what the script is meant to accomplish. A tag can also be used to transfer the result of a formula back to the script. However, it must be pointed out that if a tag is used to send information to a script, one and only one tag must be used to accomplish this. If more than one tag exists for sending information to the script, the VIP Tool will only use the last value it 'sees' from a list of multiple values a tag with the same name in different locations can potentially hold. This will generate unpredictable results.



When a script is used to assign a value to multiple tag locations designated by the same tag name, that same value is transferred to all tags.



If a value is manually entered in a cell that contains a tag that is shared on other sheets, the value 'grabbed' by the script may not be the value one expected to grab. In the example above, sheets Report 1, Report 2 and Calc all have a tag called <sn>. If, for example, a value is manually entered for <sn> on the Report 1 page, VIP Tool may finish scanning for the tag name on the Calc sheet, which may not have a value entered for <sn>. Therefore, the script may pull in a blank value rather than the intended value.



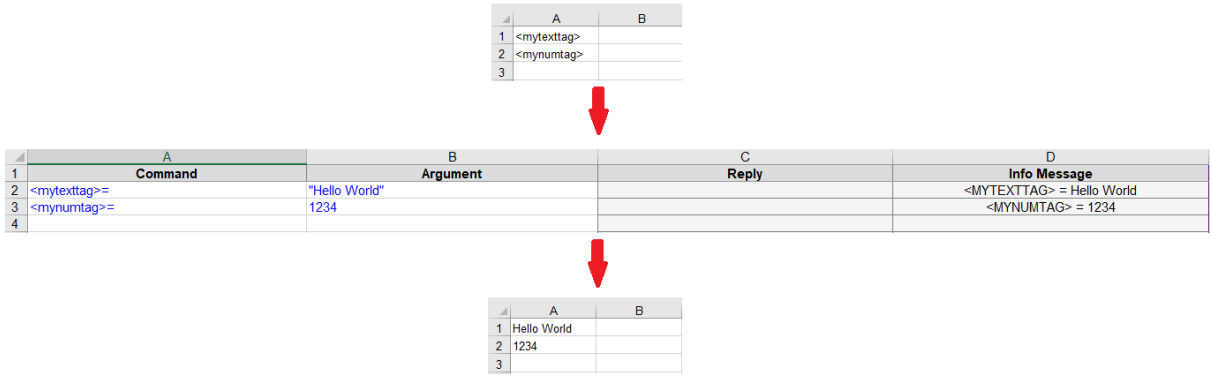
If, however, only one <sn> tag is used to transfer the value to the script, then its value is guaranteed to be the value the script 'sees'.

Tag Syntax

The syntax rules for assigning and reading values using tags generally follow rules identical to performing the same functions with variables.

Command Column Tags

A tag can be assigned a value by placing the tag in the command column, followed by the '=' sign, and placing the value in the argument column.

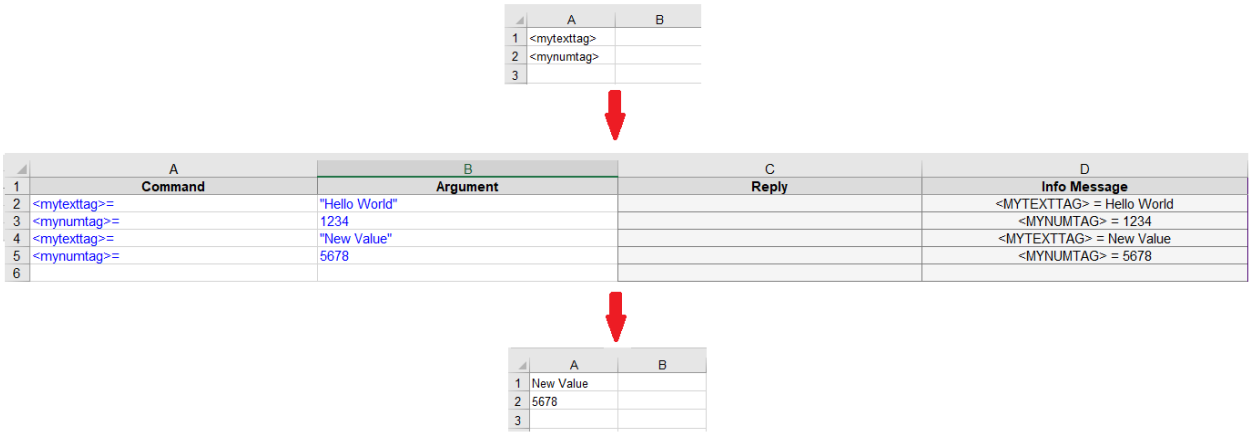


In the above example the tag '<mytexttag>' is located on a report sheet in cell A1. The tag '<mynumtag>' is located on the same report sheet in cell A2. On row 2 of the script, the value of "Hello World" is assigned to '<mytexttag>'.

NOTE:	Note that in the command column, the '=' sign follows the tag name. In the argument column, the value "Hello World." is enclosed in double quotations because it is a text value. When a value is assigned to a tag, the Info Message column will display the uppercase form of the tag name and the value assigned to the tag.

On row 3, the numerical value of 1234 is assigned to '<mynumtag>'. In the argument column, the value 1234 is placed without double quotations because it is a numerical value.

After executing rows 2 and 3 of the script, the tag values appear in the cell locations marked by the tags.



Tag values can be reassigned at any time. In the above illustration, on rows 4 and 5, each tag is tag reassigned a different value. In this case the new values appear in every location the tag name has been assigned to.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	<mytexttag>=	""Analog Duplex""		<MYTEXTTAG> = "Analog Duplex"
3				

As with variables holding text values, if the tag value must include double quotes in the actual value, enclose the value in two sets of double quotes.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	<mytag>=	150.0125		<MYTAG> = 150.0125
3	<myothertag>=	<mytag>		<MYOTHERTAG> = 150.0125
4				

Values can be passed from one tag to another. In the above example, on row 2, '<mytag>' is assigned the value 150.0125. On row 3, '<myothertag>' is assigned the same value held by '<mytag>'.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	[myfreq]=	150.0125		[MYFREQ] = 150.0125
3	<mytag>=	[myfreq]		<MYTAG> = 150.0125
4	<mytag>=	(examples_d1)		<MYTAG> = 220.0125
5				

Values can be passed to a tag from a variable or an Excel sheet cell using the VIP Tool cell assignment syntax. In the above example, the variable '[myfreq]' is assigned the value 150.0125. On row 3, the value held by '[myfreq]' is assigned to '<mytag>'.

On another sheet, called 'examples', cell D1 contains the value 220.0125. On row 4 of the script, cell assignment syntax is used to assign the value held in that cell to the tag '<mytag>'.

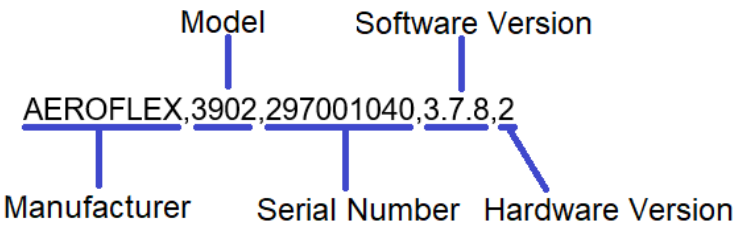
Argument Column Tags

A tag can be assigned a value by placing the tag in the argument column and using an RCI query to retrieve the data to be assigned to the tag. In this case, the tag name is followed by some form of the '=reply' keyword.

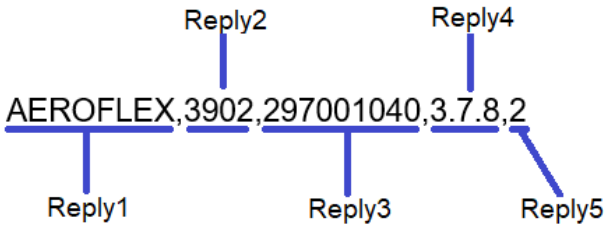
The '=reply' keyword is used to select the data from the reply that will be assigned to the tag. This structure is used so that the desired value can be retrieved from a reply that has multiple data in the Comma Separated Value (CSV) format when returned by an instrument.

A common example of such a reply would be the response to the query '*IDN?'. The '*IDN?' query returns data such as manufacturer, model number, serial number, and software version of the instrument in the CSV format. The '=reply' keyword allows the user to select which of those data elements is to be assigned to the tag.

When the '=reply' keyword contains a number, that number is used to select which element of the CSV response is to be assigned to the tag. For example, the response of a '*IDN?' query will be structured as some variation of this example:



Following the '=Reply' keyword with a number will select the field to be stored by the tag, in the order the field appears in the response, starting at 1.



In the above example, '=Reply1' will assign the tag the name 'AEROFLEX', '=Reply2' will assign the tag the model number '3902', '=Reply3' will assign the tag the serial number '297001040', '=Reply4' will assign the tag the software version'3.7.8.2', and '=Reply5' will assign the tag the hardware version '2'. If no number follows the '=Reply' keyword, the entire string will be assigned to the tag. For responses that contain only one field (in other words, not a multi-field CSV string), then '=Reply' must be used.

The number range of the '=Reply' keyword is 1 through 8, so, for example, '=Reply8' is supported, but '=Reply9' is not supported. The '=Reply' keyword is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	*idn?	<manufacturer>=Reply1	AEROFLEX,3902,297001040,3.7.8,2	<MANUFACTURER> = AEROFLEX
3	*idn?	<model>=Reply2	AEROFLEX,3902,297001040,3.7.8,2	<MODEL> = 3902
4	*idn?	<serialnum>=Reply3	AEROFLEX,3902,297001040,3.7.8,2	<SERIALNUM> = 297001040
5	*idn?	<swver>=Reply4	AEROFLEX,3902,297001040,3.7.8,2	<SWVER> = 3.7.8
6	*idn?	<hwver>=Reply5	AEROFLEX,3902,297001040,3.7.8,2	<HWVER> = 2
7	*idn?	<wholestring>=Reply	AEROFLEX,3902,297001040,3.7.8,2	<WHOLESTRING> = AEROFLEX,3902,297001040,3.7.8,2
8				

In the above illustration, the manufacturer name field value is assigned to the tag '<manufacturer>', the model number field value is assigned to the tag '<model>', the serial number field value is assigned to the tag '<serialnum>', the software version field value is assigned to the tag '<swver>', and the hardware version field value is assigned to the tag '<hwver>'. The entire string is assigned to the tag '<wholestring>'.

Query Arguments and Tags

In some cases, an instrument query may require an argument. For example, the 39xx instrument query for broadband power uses the following syntax:

:FETCh:RF:ANALyzer:TRBPower? <units>

The query allows the user to specify the unit of measurement of the broadband power value by providing a units argument with the query. The unit of measurement can be set to Watts, dBm, or dBW using 'W', 'dBm' or 'W' respectively. The response of the broadband power meter query is CSV formatted as follows:

<statusbyte>,<failbyte>,<avgcount>,<avg>

The power reading value to be assigned to the tag is in the <avg> field, which is the fourth field of the CSV reply string, so the user will use '=Reply4' to retrieve that value and assign it to the tag. In this case, the units argument can be placed either before or after the "=Reply4" keyword.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:fetch:rf:anal:trbp?	<pow>=Reply4 dBm	0,0,1,9.2	<POW> = 9.2
3	:fetch:rf:anal:trbp?	<pow>=Reply4 W	0,0,1,0.0085	<POW> = 0.0085
4	:fetch:rf:anal:trbp?	<pow>=Reply4 dBW	0,0,1,-20.7	<POW> = -20.7
5				

In the above illustration, the tag '<pow>' is assigned the power measurement using the broadband power query. In row 2, 'dBm' is supplied as the query argument, so the dBm value of the measurement is retrieved and assigned to '<pow>'. In row 3, 'W' is supplied as the query argument, so the Watts value of the power measurement is retrieved and assigned to '<pow>'. In row 4, 'dBW' is supplied as the query argument, so the dBW value of the measurement is retrieved and assigned to '<pow>'.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:fetch:rf:anal:trbp?	dBm <pow>=Reply4	0,0,1,9.4	<POW> = 9.4
3	:fetch:rf:anal:trbp?	W <pow>=Reply4	0,0,1,0.0085	<POW> = 0.0085
4	:fetch:rf:anal:trbp?	dBW <pow>=Reply4	0,0,1,-20.7	<POW> = -20.7
5				

The same results will occur if the query argument is placed *before* the tag assignment in the argument column.

Creating Tag Arrays

A tag array can be constructed by creating a base tag name followed by an underscore and a counter name within the bracket. For example '<mytag_nextcount>' will create the tag name followed by the underscore and the current value of 'nextcount'. So, if the current value of nextcount is 2 when the tag value is assigned, the tag name will be '<mytag_2>'.

'Nextcount' is the name of the counter built into the For Next loop. There are seven different counters that can be used to create tag arrays.

Counter Name	Counter Information
nextcount	Integer up or down counter available with each instance of a For Next Loop
docount	Integer up counter available with each instance of a Do Loop
counter1	Autonomous decimal up or down counter
counter2	Autonomous decimal up or down counter
counter3	Autonomous decimal up or down counter
counter4	Autonomous decimal up or down counter
counter5	Autonomous decimal up or down counter

All counters but 'docount' can count up or down; 'docount can only count up. All of the counters can be assigned arbitrary values. Counters 1 through 5 can be assigned integer or decimal values. When using counters 1 through 5 for creating tag array names, only integers can be used.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	For	1 to 5		For 1 to 5
3	:fetch:af:analyzer:sinad?	<sinad_nextcount>=reply4	0,0,1,11.27,0.00	<sinad_5> = 11.27
4	delay	1		delay 1
5	next			
6	print	<sinad_1>		<sinad_1> = 11.42
7	print	<sinad_2>		<sinad_2> = 11.52
8	print	<sinad_3>		<sinad_3> = 11.37
9	print	<sinad_4>		<sinad_4> = 11.56
10	print	<sinad_5>		<sinad_5> = 11.27
11				

In the above example, a For Next loop is set to count from 1 to 5. For each iteration of the loop, a SINAD measurement is taken on row 3. The argument for row 3 is '<sinad_nextcount>=reply4'. Each time the loop steps to row 3, the SINAD measurement is taken, and 'reply4', which is the SINAD measurement taken from the return string to the query, is assigned to '<sinad_nextcount>'. 'Nextcount is the built-in counter for the For Next loop. Its current count is assigned to the name of '<sinad_nextcount> to which the current SINAD reading is assigned. So, on the first step, the tag is named '<sinad_1>', which is assigned the current reading. On the second step, the tag is named '<sinad_2>', which is assigned the next current reading. This procedure is repeated from 1 to 5, the For Next range.

'Print' is a VIP Tool debugging keyword. Placing 'print' in the Command column and the name of a tag in the Argument column will result in the Info Message column displaying the tag name and the value stored in the tag.

After the For Next loop has run, on rows 6 through 10, 'print' is used to display each name of the tag array created by the For Next loop and the value that was stored in each tag of the array.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	do			
3	:fetch:af:analyzer:sinad?	<sinad_docount>=reply4	0,0,1,11.31,0.00	<sinad_5> = 11.31
4	exitdo	docount = 5		If 5 = 5 = True
5	delay	1		delay 1
6	loop			
7	print	<sinad_1>		<sinad_1> = 11.6
8	print	<sinad_2>		<sinad_2> = 11.45
9	print	<sinad_3>		<sinad_3> = 11.4
10	print	<sinad_4>		<sinad_4> = 11.39
11	print	<sinad_5>		<sinad_5> = 11.31
12				

In the above illustration, 'docount' is used to set the array number. By default, 'docount' increments to 1 on the first iteration of a Do Loop and increments on each iteration of the loop. In this example, the Do Loop is set to exit when 'docount' is equal to 5 on row 5.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	set_counter1	4	4	
3	for	1 to 5		For 1 to 5
4	inc_counter1		9	
5	:fetch:af:analyzer:sinad?	<sinad_counter1>=reply4	0,0,1,11.36,0.00	<sinad_9> = 11.36
6	delay	1		delay 1
7	next			
8	print	<sinad_5>		<sinad_5> = 11.44
9	print	<sinad_6>		<sinad_6> = 11.37
10	print	<sinad_7>		<sinad_7> = 11.46
11	print	<sinad_8>		<sinad_8> = 11.53
12	print	<sinad_9>		<sinad_9> = 11.36
13				

The above example uses counter1 to set the array number. In this case, counter1 is set to a value of 4 before the For Next loop starts. The 'inc_counter1' command on row 4 increments counter1 by the default value of 1 each time the For Next loop iterates. Counter1 is incremented by 1 before the first SINAD query is sent on row 5, so the first array number will be 5. The For Next loop is set to iterate 5 times before the loop ends. Because the counter1 value used for the array is incremented 5 times, the array number steps from 5 to 9 for the 5 iterations of the loop. Thus, the first array name is '<sinad_5>', the second array name is '<sinad_6>' and so forth until the terminal count of the For Next loop, which sets the last array name to '<sinad_9>'.

Tag as Argument for RCI Command

A tag can be used to hold the argument for an RCI command.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:fetch:rf:anal:fcount?	<rf_freq>=reply	151.053	<rf_freq> = 151.053
3	:rf:gen:freq	<rf_freq>		:rf:gen:freq 151.053
4				

In this example, row 2 sends a query for the RF frequency counter reading of a 39xx instrument. The value of the frequency counter is assigned to the tag '<rf_freq>' in the argument column of row 2.

On row 3, the command to set the RF generator frequency uses the '<rf_freq>' variable value to set the RF generator to the same frequency as the value read from the frequency counter on row 2.

NOTE:	It should be noted that a tag can **not** hold an RCI command or query for use in the command column.

Cell Assignments

The VIP Tool provides syntax for directly reading and assigning values to and from cells throughout the VIP Tool worksheet. This syntax serves to provide a shorthand means of transferring these values without relying on Excel formulas. It also permits an expedient method of transferring values to and from multiple cells within a loop; the same techniques used to create variable and tag arrays are used to create an array or range of addressed cells. Cell assignments can only be used in the argument column of the script sheet.

Simple Cell Assignment Syntax

A simple cell assignment combines the cell location along with the name of the sheet the cell is located in. A cell assignment is enclosed in open and closed parenthesis. It contains the worksheet tab name the cell is located in, followed by an underscore, which is followed by the column letter/row number designation of the target cell. A simple cell assignment takes this form:

(sheetname_CellColumCellRow)

An example of a simple cell assignment would be as follows:

(Report_A1)

This example specifies cell A1 on a worksheet named ‘Report’ that is included within the VIP Tool as a tab in the workbook.

The sheet name parameter may include spaces. For example, ‘(My Report_B3)’ could be used to specify cell B3 on a worksheet named ‘My Report’ that is included as a tab in the VIP Tool workbook. Neither the worksheet name nor the column letter is case sensitive. The same cell assignment could be written as ‘(my report_b3)’ and still be a valid cell assignment.

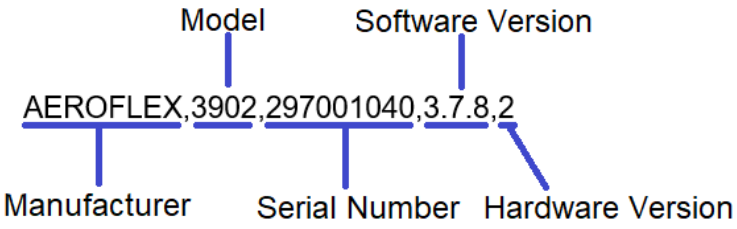
Assigning Values Using Cell Assignment

A worksheet cell can be assigned a value by placing the cell assignment in the argument column and using an RCI query to retrieve the data to be assigned to the target cell. In this case, the cell assignment is followed by some form of the ‘=reply’ keyword.

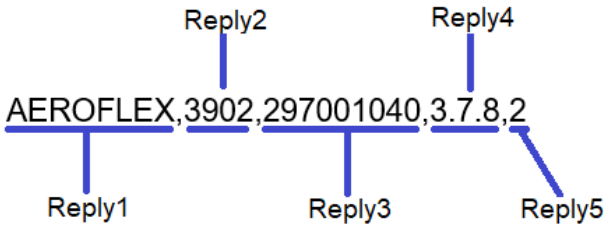
The ‘=reply’ keyword is used to select the data from the reply that will be assigned to the target cell. This structure is used so that the desired value can be retrieved from a reply that has multiple data in the Comma Separated Value (CSV) format when returned by an instrument.

A common example of such a reply would be the response to the query ‘*IDN?’. The ‘*IDN?’ query returns data such as manufacturer, model number, serial number, and software version of the instrument in the CSV format. The ‘=reply’ keyword allows the user to select which of those data elements is to be assigned to the target cell.

When the ‘=reply’ keyword contains a number, that number is used to select which element of the CSV response is to be assigned to the variable. For example, the response of a ‘*IDN?’ query will be structured as some variation of this example:



Following the '=Reply' keyword with a number will select the field to be assigned to the target cell, in the order the field appears in the response, starting at 1.



In this example, '=Reply1' will assign the target cell the name 'AEROFLEX', '=Reply2' will assign the target cell the model number '3902', '=Reply3' will assign the target cell the serial number '297001040', '=Reply4' will assign the target cell the software version'3.7.8.2', and '=Reply5' will assign the target cell the hardware version '2'. If no number follows the '=Reply' keyword, the entire string will be assigned to the target cell. For responses that contain only one field (in other words, not a multi-field CSV string), then '=Reply' must be used.

The number range of the '=Reply' keyword is 1 through 8, so, for example, '=Reply8' is supported, but '=Reply9' is not supported. The '=Reply' keyword is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	*idn?	(examples_A1)=reply1	AEROFLEX,3902,297001040,3.7.8,2	AEROFLEX copied to sheet EXAMPLES cell A1
3	*idn?	(examples_B1)=reply2	AEROFLEX,3902,297001040,3.7.8,2	3902 copied to sheet EXAMPLES cell B1
4	*idn?	(examples_C1)=reply3	AEROFLEX,3902,297001040,3.7.8,2	297001040 copied to sheet EXAMPLES cell C1
5	*idn?	(examples_D1)=reply4	AEROFLEX,3902,297001040,3.7.8,2	3.7.8 copied to sheet EXAMPLES cell D1
6	*idn?	(examples_E1)=reply5	AEROFLEX,3902,297001040,3.7.8,2	2 copied to sheet EXAMPLES cell E1
7	*idn?	(examples_F1)=reply	AEROFLEX,3902,297001040,3.7.8,2	AEROFLEX,3902,297001040,3.7.8,2 copied to sheet EXAMPLES cell F1
8				

	A	B	C	D	E	F	G
1	AEROFLEX	3902	297001040	3.7.8	2	AEROFLEX,3902,297001040,3.7.8,2	
2							

In the above illustration, the cell assignments target a report sheet tab called 'Examples', which is embedded in the VIP Tool workbook. In this example, each target cell is in a different column on the 'Examples' sheet. The manufacturer name field value is assigned to the 'Examples' sheet cell A1, the model number field value is assigned to the 'Examples' sheet cell B1, the serial number field value is assigned to the 'Examples' sheet cell C1, the software version field value is assigned to the 'Examples' sheet cell D1, and the hardware version field value is assigned to the 'Examples' sheet cell E1. The entire string is assigned to the 'Examples' sheet cell F1.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	*idn?	(examples_A1)=reply1	AEROFLEX,3902,297001040,3.7.8,2	AEROFLEX copied to sheet EXAMPLES cell A1
3	*idn?	(examples_A2)=reply2	AEROFLEX,3902,297001040,3.7.8,2	3902 copied to sheet EXAMPLES cell A2
4	*idn?	(examples_A3)=reply3	AEROFLEX,3902,297001040,3.7.8,2	297001040 copied to sheet EXAMPLES cell A3
5	*idn?	(examples_A4)=reply4	AEROFLEX,3902,297001040,3.7.8,2	3.7.8 copied to sheet EXAMPLES cell A4
6	*idn?	(examples_A5)=reply5	AEROFLEX,3902,297001040,3.7.8,2	2 copied to sheet EXAMPLES cell A5
7	*idn?	(examples_A6)=reply	AEROFLEX,3902,297001040,3.7.8,2	AEROFLEX,3902,297001040,3.7.8,2 copied to sheet EXAMPLES cell A6
8				

	A	B
1	AEROFLEX	
2	3902	
3	297001040	
4	3.7.8	
5	2	
6	AEROFLEX,3902,297001040,3.7.8,2	
7		

In this next example, the cell assignments still target a report sheet tab called ‘Examples’, but each target cell is in a different row in the same column. The manufacturer name field value is assigned to the ‘Examples’ sheet cell A1, the model number field value is assigned to the ‘Examples’ sheet cell A2, the serial number field value is assigned to the ‘Examples’ sheet cell A3, the software version field value is assigned to the ‘Examples’ sheet cell A4, and the hardware version field value is assigned to the ‘Examples’ sheet cell A5. The entire string is assigned to the ‘Examples’ sheet cell A6.

Query Arguments and Cell Assignments

In some cases, an instrument query may require an argument. For example, the 39xx instrument query for broadband power uses the following syntax:

:FETCh:RF:ANALyzer:TRBPower? <units>

The query allows the user to specify the unit of measurement of the broadband power value by providing a units argument with the query. The unit of measurement can be set to Watts, dBm, or dBW using ‘W’, ‘dBm’ or ‘W’ respectively. The response of the broadband power meter query is CSV formatted as follows:

<statusbyte>,<failbyte>,<avgcount>,<avg>

The power reading value to be assigned to the target cell is in the <avg> field, which is the fourth field of the CSV reply string, so the user will use ‘=Reply4’ to retrieve that value and assign it to the target cell. In this case, the units argument can be placed either before or after the “=Reply4” keyword.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:FETCh:RF:ANALyzer:TRBPower?	(examples_a1)=reply4 dBm	0,0,1,9.3	9.3 copied to sheet EXAMPLES cell A1
3	:FETCh:RF:ANALyzer:TRBPower?	(examples_a2)=reply4 W	0,0,1,0.0085	0.0085 copied to sheet EXAMPLES cell A2
4	:FETCh:RF:ANALyzer:TRBPower?	(examples_a3)=reply4 dBW	0,0,1,-20.8	-20.8 copied to sheet EXAMPLES cell A3
5				

	A	B
1	9.3	
2	0.0085	
3	-20.8	
4		

In the above illustration, the variable ‘[pow]’ is assigned the power measurement using the broadband power query. In row 2, ‘dBm’ is supplied as the query argument, so the dBm value of the measurement is retrieved and assigned to ‘Example’ sheet cell A1. In row 3, ‘W’ is supplied as the query argument, so the Watts value of the

power measurement is retrieved and assigned to ‘Example’ sheet cell A2. In row 3, ‘dBW’ is supplied as the query argument, so the dBW value of the measurement is retrieved and assigned to ‘Example’ sheet cell A3.

	Command	Argument	Reply	Info Message
2	:FETCh:RF:ANALyzer:TRBPower?	dbm (examples_a1)=reply4	0,0,1,9.3	9.3 copied to sheet EXAMPLES cell A1
3	:FETCh:RF:ANALyzer:TRBPower?	W (examples_a2)=reply4	0,0,1,0.0084	0.0084 copied to sheet EXAMPLES cell A2
4	:FETCh:RF:ANALyzer:TRBPower?	dBw (examples_a3)=reply4	0,0,1,-20.8	-20.8 copied to sheet EXAMPLES cell A3
5				

	A	B
1	9.3	
2	0.0084	
3	-20.8	
4		

The same results will occur if the query argument is placed *before* the variable assignment in the argument column.

Cell Assignment Arrays and Ranges

An array of cell assignments can be constructed by designating a target column on a sheet tab followed by a colon and a counter name within the cell assignment parenthesis. For example, ‘nextcount’ is the built-in counter of a For Next loop. The syntax ‘(Examples_A:nextcount)’ designates column ‘A’ of an existing worksheet within the workbook called ‘Examples’ with the row number defined by the current value of the ‘nextcount’ counter. So, if the current value of nextcount is 2 when ‘(Examples_A:nextcount)’ is encountered, the target cell will be cell A2 of the ‘Examples’ sheet.

There are seven different counters that can be used to create designate an array of cell assignments.

Counter Name	Counter Information
nextcount	Integer up or down counter available with each instance of a For Next Loop
docount	Integer up counter available with each instance of a Do Loop
counter1	Autonomous decimal up or down counter
counter2	Autonomous decimal up or down counter
counter3	Autonomous decimal up or down counter
counter4	Autonomous decimal up or down counter
counter5	Autonomous decimal up or down counter

All counters but ‘docount’ can count up or down; ‘docount’ can only count up. All of the counters can be assigned arbitrary values. Counters 1 through 5 can be assigned integer or decimal values. When using counters 1 through 5 for creating an array of cell assignments, only integers can be used.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	for	1 to 5		For 1 to 5
3	:fetch:af:analyzer:sinad?	(examples_a:nextcount)=reply4	0,0,1,11.56,0.00	11.56 copied to sheet EXAMPLES cell A5
4	delay	1		delay 1
5	next			
6	print	(examples_a1)		Sheet EXAMPLES cell A1 = 11.16
7	print	(examples_a2)		Sheet EXAMPLES cell A2 = 11.26
8	print	(examples_a3)		Sheet EXAMPLES cell A3 = 11.18
9	print	(examples_a4)		Sheet EXAMPLES cell A4 = 11.8
10	print	(examples_a5)		Sheet EXAMPLES cell A5 = 11.56
11				

	A	B
1	11.16	
2	11.26	
3	11.18	
4	11.8	
5	11.56	
6		

In the above example, a For Next loop is set to count from 1 to 5. For each iteration of the loop, a SINAD measurement is taken on row 3. The argument for row 3 is (examples_a:nextcount)=reply4'. Each time the loop steps to row 3, the SINAD measurement is taken, and 'reply4', which is the SINAD measurement taken from the return string to the query, is assigned to '(Examples_A:nextcount)'. 'Nextcount' is the built-in counter for the For Next loop. Its current count is assigned to the row number of column A of sheet embedded in the workbook called 'Examples'. Therefore, the current SINAD reading is assigned to each row number in column A of the Examples sheet enumerated by 'nextcount'. So, on the first step, the first SINAD reading is assigned to cell A1 of the 'Examples' sheet. On the second step, the second SINAD reading is assigned to cell A2 of the 'Examples' sheet. This procedure is repeated from 1 to 5, the For Next range.

'Print' is a VIP Tool debugging keyword. Placing 'print' in the Command column and the name of a cell assignment in the Argument column will result in the Info Message column displaying the target cell and the value stored in the target cell.

After the For Next loop has run on rows 6 through 10, 'print' is used to display each target cell name in the array created by the For Next loop and the value that is stored in each target cell.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	do	1 to 5		
3	:fetch:af:analyzer:sinad?	(examples_a:docount)=reply4	0,0,1,11.52,0.00	11.52 copied to sheet EXAMPLES cell A5
4	exitdo	docount = 5		If 5 = 5 = True
5	delay	1		delay 1
6	loop			
7	print	(examples_a1)		Sheet EXAMPLES cell A1 = 11.41
8	print	(examples_a2)		Sheet EXAMPLES cell A2 = 11.57
9	print	(examples_a3)		Sheet EXAMPLES cell A3 = 11.59
10	print	(examples_a4)		Sheet EXAMPLES cell A4 = 11.26
11	print	(examples_a5)		Sheet EXAMPLES cell A5 = 11.52
12				

	A	B
1	11.41	
2	11.57	
3	11.59	
4	11.26	
5	11.52	
6		

In the above illustration, 'docount' is used to set the array number. By default, 'docount' increments to 1 on the first iteration of a Do Loop and increments on each iteration of the loop. In this example, the Do Loop is set to exit when 'docount' is equal to 5 on row 5.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	set_counter1	4	4	
3	for	1 to 5		For 1 to 5
4	inc_counter1		9	
5	:fetch:af:analyzer:sinad?	(examples_a:counter1)=reply4	0,0,1,11.29,0.00	11.29 copied to sheet EXAMPLES cell A9
6	delay	1		delay 1
7	next			
8	print	(examples_a5)		Sheet EXAMPLES cell A5 = 11.59
9	print	(examples_a6)		Sheet EXAMPLES cell A6 = 11.82
10	print	(examples_a7)		Sheet EXAMPLES cell A7 = 11.19
11	print	(examples_a8)		Sheet EXAMPLES cell A8 = 11.63
12	print	(examples_a9)		Sheet EXAMPLES cell A9 = 11.29
13				

	A	B
1		
2		
3		
4		
5	11.59	
6	11.82	
7	11.19	
8	11.63	
9	11.29	
10		
11		

The above example uses counter1 to set the array number. In this case, counter1 is set to a value of 4 before the For Next loop starts. The 'inc_counter1' command on row 4 increments counter1 by the default value of 1 each time the For Next loop iterates. Counter1 is incremented by 1 before the first SINAD query is sent on row 5, so the first array number will be 5. The For Next loop is set to iterate 5 times before the loop ends. Because the counter1 value used for the array is incremented 5 times, the array number steps from 5 to 9 for the 5 iterations of the loop. Thus, the first cell assignment target cell A5 on the Examples sheet, the second target cell is A6 on the Examples sheet and so forth until the terminal count of the For Next loop, which sets the last target cell to A9 on the Examples sheet.

Cell Assignment Ranges

A range of cell assignments works much like an array of cell assignments with the difference being that the syntax specifies the row number that is the starting row number from which the array will be applied. For example, the starting row number can be specified as '5'. The first value of the array will then be assigned to row 5 and the subsequent elements in the cell assignment array will follow by assigning row 6, row 7 and so on. The syntax for a range of target cells includes the column and row number of the starting target cell, followed by a colon, and the *range* syntax for the counter to be used. Each of the counters included in the VIP Tool includes a range version of the syntax for this purpose.

Counter Name	**Cell Assignment** Range **Syntax**
nextcount	nextrange
docount	dorange
counter1	counter1range
counter2	counter2range
counter3	counter3range
counter4	counter4range
counter5	counter5range

For example, the syntax '(Examples_A5:nextrange)' designates column 'A' of an existing worksheet within the workbook called 'Examples' with the starting row number defined as '5'. If the current count of nextcount is 1, the target cell will be cell A5 of the examples sheet. The next value of nextcount will be 2, which designates the next target cell as A6 of the examples sheet, and so on.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	for	1 to 5		For 1 to 5
3	:fetch:af:analyzer:sinad?	(examples_a5:nextrange)=reply4	0,0,1,11.56,0.00	11.56 copied to sheet EXAMPLES cell A9
4	delay	1		delay 1
5	next			
6	print	(examples_a5)		Sheet EXAMPLES cell A5 = 11.68
7	print	(examples_a6)		Sheet EXAMPLES cell A6 = 11.4
8	print	(examples_a7)		Sheet EXAMPLES cell A7 = 11.29
9	print	(examples_a8)		Sheet EXAMPLES cell A8 = 11.68
10	print	(examples_a9)		Sheet EXAMPLES cell A9 = 11.56
11				

	A	B
1		
2		
3		
4		
5	11.68	
6	11.4	
7	11.29	
8	11.68	
9	11.56	
10		
11		

In the above example, a For Next loop is set to count from 1 to 5. For each iteration of the loop, a SINAD measurement is taken on row 3. The argument for row 3 is (examples_a5:nextrange)=reply4'. Each time the loop steps to row 3, the SINAD measurement is taken, and 'reply4', which is the SINAD measurement taken from the return string to the query, is assigned to '(Examples_A5:nextrange)'. 'Nextcount is the built-in counter

for the For Next loop, and ‘nextrange’ is the range variant of the counter. The first count of nextcount is assigned to row number 5 of column A of the sheet embedded in the workbook called ‘Examples’. Thereafter, the current SINAD reading is assigned to each row succeeding row number after row 5 in column A of the Examples sheet enumerated by ‘nextcount’. So, on the first step, the first SINAD reading is assigned to cell A5 of the ‘Examples’ sheet. On the second step, the second SINAD reading is assigned to cell A6 of the ‘Examples’ sheet. This procedure is repeated from 1 to 5, the For Next range.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	do	1 to 5		
3	:fetch:af:analyzer:sinad?	(examples_a5:dorange)=reply4	0,0,1,11.27,0.00	11.27 copied to sheet EXAMPLES cell A9
4	exitdo	docount = 5		If 5 = 5 = True
5	delay	1		delay 1
6	loop			
7	print	(examples_a5)		Sheet EXAMPLES cell A5 = 11.32
8	print	(examples_a6)		Sheet EXAMPLES cell A6 = 11.31
9	print	(examples_a7)		Sheet EXAMPLES cell A7 = 11.44
10	print	(examples_a8)		Sheet EXAMPLES cell A8 = 11.29
11	print	(examples_a9)		Sheet EXAMPLES cell A9 = 11.27
12				

	A	B
1		
2		
3		
4		
5	11.32	
6	11.31	
7	11.44	
8	11.29	
9	11.27	
10		

In the above illustration, ‘dorange’, the range variant of the current value of ‘docount’, is used to set the array number. By default, ‘docount’ increments to 1 on the first iteration of a Do Loop and increments on each iteration of the loop. In this example, the Do Loop is set to exit when ‘docount’ is equal to 5 on row 5. The ‘dorange’ syntax starts the count at row 5 and targets the following rows in the ascending order of the ‘docount’ value.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	for	1 to 5		For 1 to 5
3	:fetch:af:analyzer:sinad?	(examples_a5:nextrange)=reply4	0,0,1,11.56,0.00	11.56 copied to sheet EXAMPLES cell A9
4	delay	1		delay 1
5	next			
6	print	(examples_a5)		Sheet EXAMPLES cell A5 = 11.68
7	print	(examples_a6)		Sheet EXAMPLES cell A6 = 11.4
8	print	(examples_a7)		Sheet EXAMPLES cell A7 = 11.29
9	print	(examples_a8)		Sheet EXAMPLES cell A8 = 11.68
10	print	(examples_a9)		Sheet EXAMPLES cell A9 = 11.56
11				

	A	B
1		
2		
3		
4		
5	11.68	
6	11.4	
7	11.29	
8	11.68	
9	11.56	
10		
11		

In the above illustration, ‘counter1range’, the range variant of the current value of ‘counter1’, is used to set the array number. In this case, counter1 starts counting from the default value of counter1, which is zero. The ‘inc_counter1’ command on row 4 increments counter1 by the default value of 1 each time the For Next loop iterates. Counter1 is incremented by 1 before the first SINAD query is sent on row 5. The ‘counter1range’ syntax designates row 5 as the starting row, so the first array number will be 5. The For Next loop is set to

iterate 5 times before the loop ends. Because the counter1 value used for the array is incremented 5 times, the array number steps from 5 to 9 for the 5 iterations of the loop. Thus, the first cell assignment target cell A5 on the Examples sheet, the second target cell is A6 on the Examples sheet and so forth until the terminal count of the For Next loop, which sets the last target cell to A9 on the Examples sheet.

Math Functions

In addition to the ability to use Excel math formulas, the VIP Tool scripting language provides the ability to perform simple math functions using the language itself. This functionality is included to decrease reliance on using a sheet external to the script and the report page, thus enhancing the portability of the script.

The VIP Tool math functions can only be performed separately on variables, tags and cell assignments. A math function cannot be used directly when defining an argument to an RCI command. Instead, the product of a math function must be assigned to a variable or a tag, or it can be passed to a cell assignment, then that object can be used as an argument to an RCI command. The four basic math functions use the Command Column to designate the Variable or Tag that is to be assigned the outcome of the math function, and the argument column is used to perform the math function using variables, tags, cell assignment or direct entry numbers.

Simple Math Functions Supported by the VIP Tool Scripting Language

The VIP Tool supports four basic math functions.

Function	Operator	Command Column Syntax	Argument Column Syntax
Addition	+	sum=	addend + addend
Subtraction	-	difference=	minuend - subtrahend
Multiplication	*	product=	multiplicand * multiplier
Division	/	quotient=	dividend / divisor

Math Function Command Column Syntax

The outcome of a math function can only be assigned to a variable or a tag. Cell assignment syntax is not allowed in the command column.

The variable or tag must be followed by the equals sign in the command column. For example, to assign the outcome of a math function to the variable '[myvalue]', then the command column syntax is '[myvalue]='. The command column syntax for assigning the outcome of a math function to the tag '<myvalue>' is "<myvalue>=".

Math Function Argument Column Syntax

The Argument column math function can be performed using direct entry of numbers, variables, tags, cell assignments or any combination of these elements.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mysum]=	4 + 5		[MYSUM] = 9
3	[mydifference]=	4 - 5		[MYDIFFERENCE] = -1
4	[myproduct]=	4 * 5		[MYPRODUCT] = 20
5	[myquotient]=	4 / 5		[MYQUOTIENT] = 0.8
6	<mysum>=	4 + 5		<MYSUM> = 9
7	<mydifference>=	4 - 5		<MYDIFFERENCE> = -1
8	<myproduct>=	4 * 5		<MYPRODUCT> = 20
9	<myquotient>=	4 / 5		<MYQUOTIENT> = 0.8
10				

	A	B
1	<mysum>	
2	<mydifference>	
3	<myproduct>	
4	<myquotient>	
5		

	A	B
1	9	
2	-1	
3	20	
4	0.8	
5		

A variable or variable array can be assigned a value and used in a math operation. In the above example, rows 2 through 5 depict assigning the outcome of the four math functions to variables. The variable '[mysum]' is assigned the sum of '4 + 5', the variable '[mydifference]' is assigned the difference value of '4 – 5', the variable '[myproduct]' is assigned the product of '4 * 5' and the variable '[myquotient]' is assigned the quotient of '4 / 5'.

Rows 6 through 9 depict assigning the outcome to tags. The tags are assigned the outcome of the four math operations, and the value is transmitted to any cell that has the tag name assigned to it.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mynum]=	5		[MYNUM] = 5
3	[mysum]=	4 + [mynum]		[MYSUM] = 9
4	[mydifference]=	4 - [mynum]		[MYDIFFERENCE] = -1
5	[myproduct]=	4 * [mynum]		[MYPRODUCT] = 20
6	[myquotient]=	4 / [mynum]		[MYQUOTIENT] = 0.8
7	<mysum>=	4 + [mynum]		<MYSUM> = 9
8	<mydifference>=	4 - [mynum]		<MYDIFFERENCE> = -1
9	<myproduct>=	4 * [mynum]		<MYPRODUCT> = 20
10	<myquotient>=	4 / [mynum]		<MYQUOTIENT> = 0.8
11				

A variable can be assigned a value and used in a math function. In the above example, the variable '[mynum]' is assigned the value of '5' and is used to represent that number in the math operations.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mynum]=	5		[MYNUM] = 5
3	[mysum]=	<mynum> + [mynum]		[MYSUM] = 9
4	[mydifference]=	<mynum> - [mynum]		[MYDIFFERENCE] = -1
5	[myproduct]=	<mynum> * [mynum]		[MYPRODUCT] = 20
6	[myquotient]=	<mynum> / [mynum]		[MYQUOTIENT] = 0.8
7	<mysum>=	<mynum> + [mynum]		<MYSUM> = 9
8	<mydifference>=	<mynum> - [mynum]		<MYDIFFERENCE> = -1
9	<myproduct>=	<mynum> * [mynum]		<MYPRODUCT> = 20
10	<myquotient>=	<mynum> / [mynum]		<MYQUOTIENT> = 0.8
11				

	A	B	C
1	<mysum>	<mynum>	
2	<mydifference>		
3	<myproduct>		
4	<myquotient>		
5			

	A	B	C
1	9	4	
2	-1		
3	20		
4	0.8		
5			

A tag or tag array can be assigned a value and used in a math function. In the above example, the tag '<mynum>' is assigned the value of '4' and is used to represent that number in the math operations.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mynum]=	5		[MYNUM] = 5
3	[mysum]=	(examples_a1) + [mynum]		[MYSUM] = 9
4	[mydifference]=	(examples_a1) - [mynum]		[MYDIFFERENCE] = -1
5	[myproduct]=	(examples_a1) * [mynum]		[MYPRODUCT] = 20
6	[myquotient]=	(examples_a1) / [mynum]		[MYQUOTIENT] = 0.8
7	<mysum>=	(examples_a1) + [mynum]		<MYSUM> = 9
8	<mydifference>=	(examples_a1) - [mynum]		<MYDIFFERENCE> = -1
9	<myproduct>=	(examples_a1) * [mynum]		<MYPRODUCT> = 20
10	<myquotient>=	(examples_a1) / [mynum]		<MYQUOTIENT> = 0.8
11				

	A	B
1	4	
2		

A cell can be assigned a value and used in a math function using cell assignment, cell assignment array, or cell assignment range syntax. In the above example, cell A1 of an embedded sheet in the workbook called 'Examples' is assigned the value of '4' and is used to represent that number in the math operations using cell assignment syntax.

Order of Math Operations for Variables, Tags and Cell Assignments

When the value of a variable or tag is set to be assigned the value of a math operation that contains the current value of the variable or tag, the current value is the value to be evaluated in the math operation. After the math operation is completed, the variable or tag is updated with the new value rendered by the math operation.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mynum]=	5		[MYNUM] = 5
3	print	<myvalue>		<myvalue> = 4
4	[mynum]=	5 + [mynum]		[MYNUM] = 10
5	<myvalue>=	<myvalue> + 4		<MYVALUE> = 8
6	[mynum]=	[mynum] + [mynum]		[MYNUM] = 20
7	<myvalue>=	<myvalue> + <myvalue>		<MYVALUE> = 16
8				

In the above example, the variable '[mynum]' begins with the value of '5'. '[mynum]' is added to the value of 5 and assigned again to '[mynum]'. The value of '[mynum]' then holds the value of the sum, which is '10'. '[mynum]' is then added to '[mynum]' and assigned back to '[mynum]', which becomes the value of 20 (10 + 10).

The tag '<myvalue>' begins with the value of 4. '<myvalue>' is added to the value of 4 and assigned again to '<myvalue>'. The value of '<myvalue>' then holds the value of the sum, which is '8'. '<myvalue>' is then added to '<myvalue>' and assigned back to '<myvalue>', which becomes the value of 16 (8 + 8).

Using Math Functions to Generate Arguments

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[freq]=	151.0125		[FREQ] = 151.0125
3	for	1 to 3		For 1 to 3
4	[freqmult]=	nextcount		[FREQMULT] = 3
5	[recfreq]=	[freq] * [freqmult]		[RECFREQ] = 453.0375
6	:rf:analyzer:frequency	[recfreq]		:rf:analyzer:frequency 453.0375
7	next			
8				

A math function cannot be used directly as an argument to an RCI command. Instead, the outcome of a math function must be assigned to a variable, tag or cell assignment when using math functions to generate argument values. In the above example, a For Next loop is set up to set a 39xx receiver to a fundamental frequency of 151.0125, then to the second harmonic frequency of 302.025, and finally the third harmonic frequency of 453.075. This is accomplished by assigning the nextcount value to the variable '[freqmult]', then multiplying that variable with the variable holding the fundamental frequency value and assigning the result to the '[recfreq]' variable. The '[recfreq]' variable is then used as the argument to the RCI command that sets the receiver frequency.

Concatenation

The VIP Tool uses the symbol '&' for concatenating strings. A text string, or a text string held by a variable, tag or cell assignment, can be concatenated with another text string using this symbol. The VIP Tool supports concatenating only two strings at a time.

It is important to know that the VIP Tool will always remove any preceding space in a string when concatenating, so, if a space is to be preserved between the two strings when they are joined, the space must be placed at the end of the first string.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mytext]=	"Hello "		[MYTEXT] = Hello
3	[my_message]=	[mytext] & "world."		[MY_MESSAGE] = Hello world.
4				

In the above example, "Hello " is assigned to the variable '[mytext]'. On row 3, '[mytext]' is concatenated with the string "world." and the resultant string 'Hello world.' is assigned to the variable '[my_message]'.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mytext]=	"Hello "		[MYTEXT] = Hello
3	[my_secondtext]=	"world"		[MY_SECONDTEXT] = world
4	[my_message]=	[mytext] & [my_secondtext]		[MY_MESSAGE] = Hello world
5				

The above illustration demonstrates concatenating two variables that hold string values. '[mytext]' is assigned the value "Hello " and '[my_secondtext]' is assigned the value "world." On row 4, the two variables are concatenated to "Hello world." And assigned to the variable '[my_message]'.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mytext]=	"Hello "		[MYTEXT] = Hello
3	[mytext]=	[mytext] & "world"		[MYTEXT] = Hello world
4	[mytext]=	"Hello "		[MYTEXT] = Hello
5	[mytext]=	[mytext] & [mytext]		[MYTEXT] = Hello Hello
6				

A variable, tag or cell assignment value can be concatenated with another string and assigned to the same variable, tag or cell assignment. The original value of the variable, tag or cell assignment is applied to the concatenation before it assumes the new concatenated value. In the above examples, the variable '[mytext]' is assigned the value "Hello ". On row 3, '[mytext]' is then concatenated with "world", forming the string "Hello world.", which is assigned back to '[mytext]'.

On row 4, "Hello " is again assigned to '[mytext]'. On row 4, '[mytext]' is concatenated with '[mytext]' to form the string "Hello Hello", which is assigned back to '[mytext]'.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mytext]=	(examples_A1) & <mytext>		[MYTEXT] = This is a test of the emergency
3	[mytext]=	[mytext] & <mytext2>		[MYTEXT] = This is a test of the emergency broadcast system.
4				

	A	B	C
1	This is a test	<mytext>	
2		<mytext2>	
3			

	A	B	C
1	This is a test	of the emergency	
2		broadcast system.	
3			

In the above example, a worksheet called “Examples” is embedded in the VIP Tool workbook. In cell A1 of the “Examples” worksheet is the text “This is a test ”. The worksheet also contains two tags, ‘<mytext>’ and ‘<mytext2>’. ‘<mytext>’ holds the text “of the emergency “ and ‘<mytext2>’ holds the text “broadcast system.”

NOTE:	Note that when the text to be concatenated is imported from a sheet, quotations are not used to store the text on the worksheet. Only on the script sheet are quotations used to enclose text.

On row 2 of the script, ‘(examples_A1)’, which is the cell assignment for cell A1 of the examples sheet, is concatenated with the text held by the ‘<mytext>’ tag. The result, “This is a test of the emergency” is assigned to the variable ‘[mytext]’. On row 3, ‘[mytext]’ is concatenated with the contents of the tag ‘<mytext2>’ to form “This is a test of the emergency broadcast system.” This new string is assigned back to ‘[mytext]’.

Counters and Timers

Counter Functions

Counters

Command (*Argument*) Syntax:

Set_Counter1 (*initial decimal value of counter1*) Sets the initial value of Counter1. Optional; without this statement, the initial value of Counter1 defaults to 0.

Ret_Counter1 (*no argument*) When executed, causes current Counter1 value to appear in the Reply column. Can be used to assign the counter value to a variable or tag.

Inc_Counter1 (*optional decimal value, defaults to 1*) Increments the current Counter1 value by the amount specified in the argument. If no argument is applied, the increment value defaults to 1.

Dec_Counter1 (*optional decimal value, defaults to 1*) Decrements the current Counter1 value by the amount specified in the argument. If no argument is applied, the decrement value defaults to 1.

NOTE:	The above definition uses Counter1 as the example. The same syntax rules and usage apply to Counter1, Counter2, Counter3, Counter4 and Counter5.

The VIP Tool scripting language provides access to five independent counters. Unlike nextcount (the counter integral to a for/next loop) and docount (the counter integral to a do loop) the value of a counter can be a non-integer number.

The five counters are counter1, counter2, counter3, counter4 and counter5. Each counter is an independent function that can be set to a specific decimal value, incremented by a specific decimal value, and decremented

by a specific decimal value. The value of a counter can be returned to a variable, tag, or cell assignment, and can be used as an argument for SCPI commands and VIP Tool keywords. A counter value, when set to an integer value, can be used as an array index in defining and assigning or deriving values to or from a variable, tag, or cell assignment. The counter keywords are not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mynumber]=	150		[MYNUMBER] = 150
3	set_counter1	455.5	455.5	COUNTER1 = 455.5
4	inc_counter1		1	COUNTER1 = 456.5
5	set_counter2	[mynumber]	150	COUNTER2 = 150
6	ret_counter2		150	COUNTER2 = 150
7	set_counter3	<mynumber>	223.456	COUNTER3 = 223.456
8	ret_counter3		223.456	COUNTER3 = 223.456
9	set_counter4	(examples_c1)	155.425	COUNTER4 = 155.425
10	ret_counter4		155.425	COUNTER4 = 155.425
11	set_counter5	354.456	354.456	COUNTER5 = 354.456
12	ret_counter5		354.456	COUNTER5 = 354.456
13	[incnumber]=	0.0125		[INCNUMBER] = 0.0125
14	set_counter2	<mynumber>	223.456	COUNTER2 = 223.456
15	inc_counter2	[incnumber]	0.0125	COUNTER2 = 223.4685
16	inc_counter3		1	COUNTER3 = 224.456
17	inc_counter4	<incnumber>	0.025	COUNTER4 = 155.45
18	inc_counter5	(examples_c2)	600	COUNTER5 = 954.456
19	inc_counter4	25.003	25.003	COUNTER4 = 180.453
20	ret_counter3		224.456	COUNTER3 = 224.456
21	set_counter2	<mynumber>	223.456	COUNTER2 = 223.456
22	dec_counter2	[incnumber]	-0.0125	COUNTER2 = 223.4435
23	dec_counter2		-1	COUNTER2 = 222.4435
24	dec_counter3	<incnumber>	-0.025	COUNTER3 = 224.431
25	dec_counter4	(examples_c2)	-600	COUNTER4 = -419.547
26	dec_counter2	25.003	-25.003	COUNTER2 = 197.4405
27	ret_counter4		-419.547	COUNTER4 = -419.547
28				

	A	B	C	D
1		<mynumber>	155.425	
2	<incnumber>		600	
3				

	A	B	C	D
1		223.456	155.425	
2	0.025		600	
3				

The above example demonstrates the syntax and use of counter1, counter2, counter3, counter4 and counter5 commands.

Timer Functions

Timers

Command (*Argument*) Syntax:

start_timer_1 (*no argument*) Starts the timer and returns the start time in time of day format.

current_timer_1 (*no argument*) Returns the elapsed time since the timer was started in seconds.

stop_timer_1 (*no argument*) Stops the timer and returns stop time in time of day format.

total_timer_1 (*no argument*) Returns the time elapsed between start and stop in hours:minutes:seconds.

NOTE:	The above definition uses Timer_1 as the example. The same syntax rules and usage apply to Timer_1, Timer_2, Timer_3, Timer_4 and Timer_5.

The VIP Tool scripting language provides access to five independent timers. Each timer can be used to mark the beginning and end of an operation, the total elapsed time of an operation, and the current amount of time (in seconds) from when the timer was invoked.

The five timers are timer_1, timer_2, timer_3, timer_4 and timer_5. The values of a counter can be returned to a variable, tag, or cell assignment. The timer keywords are not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	start_timer_1		2:03:33 PM	
3	delay	5		delay 5
4	current_timer_1		5	
5	delay	5		delay 5
6	current_timer_1		10	
7	stop_timer_1		2:03:43 PM	
8	total_timer_1		0:00:10	
9				

The above example demonstrates the Timer syntax and response from the VIP Tool. On row 2, Start_Timer_1 begins operation for Timer_1. The Reply column displays the time of day that the timer was started. Row 3 introduces a delay of 5 seconds. After the 5 seconds of the delay have elapsed, row 4 is executed with the Current_Timer_1 command. The Reply column displays 5 seconds, because 5 seconds have elapsed since the timer was started on row 2. Row 5 introduces another delay of 5 seconds. Row 6 again uses the Current_Timer_1 command to display the time elapsed since the timer was started on row 2, now 10 seconds. On row 7, Timer_1 is stopped and the time of day the timer was stopped is displayed in the Reply column. Finally, row 8 uses the Total_Timer_1 to display the total time elapsed since the timer was started on row 2. The total time is displayed in hours:minutes:seconds format in the Reply column.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	start_timer_1	[my_start_time]=reply	2:09:04 PM	[MY_START_TIME] = 2:09:04 PM
3	delay	5		delay 5
4	current_timer_1	[my_elapsed_time]=reply	5	[MY_ELAPSED_TIME] = 5
5	delay	5		delay 5
6	current_timer_1	[my_elapsed_time]=reply	10	[MY_ELAPSED_TIME] = 10
7	stop_timer_1	[my_stop_time]=reply	2:09:14 PM	[MY_STOP_TIME] = 2:09:14 PM
8	total_timer_1	[my_total_time]=reply	0:00:10	[MY_TOTAL_TIME] = 00:00:10
9				

The responses to the various Timer commands can be assigned to variables. In the above illustration, the syntax for doing so is demonstrated.

On row 2, the Start_Timer_1 time of day response is assigned to the variable '[my_start_time]'. On rows 4 and 6, the response, in seconds, to Current_Timer_1 is assigned to the variable '[my_current_time]'. On row 7, the time of day response to the Stop_Timer_1 keyword is assigned to the variable '[my_stop_time]'. On row 8, the response to Total_Timer_1, in hours:minutes:seconds is assigned to the variable '[my_total_time]'.

The same syntax is used to assign the responses to tags or cell assignments.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	start_timer_1		2:06:31 PM	
3	[time_limit]=	30		[TIME_LIMIT] = 30
4	do			
5	*idn?		Aeroflex, 88XX, 1000000009, 2.4.4-0, Jan 12 2021, 08:55:28	
6	delay	3		delay 3
7	current_timer_1	[my_elapsed_time]=reply	32	[MY_ELAPSED_TIME] = 32
8	exitdo	[my_elapsed_time] > [time_limit]		If 32 > 30 = True
9	loop			
10				

A timer can be used to set a time limit for a loop to finish an assigned task. In the above example, before a Do Loop is started on row 4, Timer_1 is started on row 2. A time limit of 30 seconds is assigned to the '[time_limit]' variable on row 3.

The Do Loop iterates over a '*IDN?' query every 3 seconds (as set by the delay value on row 6). Each time the loop iterates, the elapsed time of Timer_1 is loaded into the '[my_elapsed_time]' variable on row 7. On row 8, each iteration of the loop evaluates whether the elapsed time held by '[my_elapsed_time]' is over the time limit of 30 seconds, which is held by the '[time_limit]' variable. When the elapsed time exceeds 30 seconds, the loop exits.

Goto and Subroutine Functions

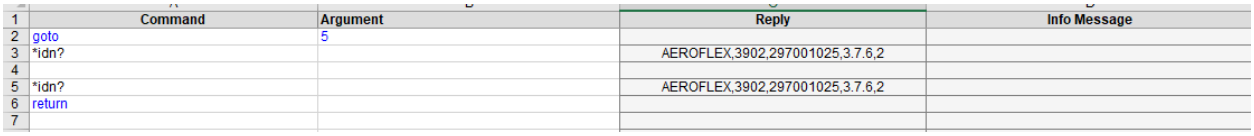
Goto Function

Goto/Return

Command (*Argument*) Syntax:

goto (*row number expressed as integer*) Row number to which execution will be directed.

return (*no argument*) Command returning execution back to the row directly following the goto row.



	Command	Argument	Reply	Info Message
2	goto	5		
3	*idn?		AEROFLEX,3902,297001025,3.7.6,2	
4				
5	*idn?		AEROFLEX,3902,297001025,3.7.6,2	
6	return			
7				

Goto/Return example

The goto function will redirect execution of a script to the line defined by its argument. The return function will return execution of the script back to the row immediately following the goto argument that was last invoked.

In the above example, row 2 has 'goto' in the command column and '5' in the argument column. This directs execution of the script to row 5. The script then executes the query '*IDN?' at row 5. The script then encounters the 'return' command on row 6. The return command returns execution back to row 3, the row immediately following the 'goto' command and executes the '*IDN?' query on that row.

Goto is not case sensitive

Return is not case sensitive

Up to 5 simultaneous goto/return loops can be used

Care should be taken with this command set. If, after a goto/return function has been defined, any rows are inserted before or between the goto/return set, the goto row number argument must be edited to define the new row number. An alternate method may be to use the sub/endSub statement and use runsub to define the beginning of the desired sequence of commands.

Subroutine Function

Sub/Endsub/Runsub

Command (*Argument*) Syntax:

Sub (*Name declaration expressed by text*) Assigns the subroutine name and defines the sub start .

Endsub (*no argument*) Defines the end of the subroutine.

Runsub (*Name of sub to go to expressed by text, variable, tag or cell assignment*) Calls the subroutine.

	Command	Argument	Reply	Info Message
2	runsub	mysub		runsub mysub
3	*idn?		AEROFLEX,3902,297001025,3.7.6,2	
4				
5				
6				
7	sub	mysub		
8	*idn?		AEROFLEX,3902,297001025,3.7.6,2	
9	*idn?		AEROFLEX,3902,297001025,3.7.6,2	
10	endsub			
11				

Sub block example

The sub block is a contiguous block of commands defined by the sub and endsub commands. A sub block begins with the keyword sub and ends with the keyword endsub. When called by the runsub command, the script will find the first instance of 'sub' that contains the name designated by the runsub command argument. The script will then execute each command placed between 'sub' and 'endsub', then return to the row number immediately following the runsub command that invoked the sub routine.

In the above example, the keyword sub appears in the command column of row 7. The argument for the sub command is 'mysub'. This gives the subroutine the name 'mysub'. Following the sub keyword are two *IDN? queries. Following the two queries, on row 10, is the keyword 'endsub'. Endsub defines the end of the sequence of commands that will run when the subroutine is called by runsub.

On row 2 is the command runsub, with the argument 'mysub'. When the script encounters this command, it is told to find the subroutine with the name 'mysub'. The script then locates 'mysub' on row 7 and begins executing the commands following the sub name. In this case, it processes two '*IDN?' queries. When the script encounters the endsub keyword on row 10, it returns to the row immediately following the runsub command that invoked the subroutine – row 3. The script then resumes execution at that point, so the '*IDN?' query on row 3 is executed after the subroutine has run.

Sub, endsub and runsub are not case sensitive

The name of the subroutine is not case sensitive

The name of the subroutine may have spaces

If a subroutine name includes quotes, the calling runsub must use quotes

If more than one subroutine with the same name is included in the script, the script will only use the subroutine that is defined at the lowest row number

Subroutines can nest within other subroutines

Up to fifty subroutines can be simultaneously nested

A subroutine can appear anywhere on the script page, before or after the runsub command that calls it

Loops and Conditional Statements

For Next Loop

For/Next/Nextcount

Command (*Argument*) Syntax:

For (*integer start count to integer end count*) Defines the number of iterations, count up or count down.

Next (*no argument*) Defines the end of the loop.

Nextcount (*optional integer argument assigns value*) Integral counter that increments once per iteration. Can also be used in the Argument column when assigning the count value to a variable or tag.

For, Next and Nextcount are not case sensitive

The For Next loop is used to iterate a sequence of commands a specific number of times. The For Next loop is defined by the For statement in the Command column. The For command must have an argument defining the number of iterations of the loop, in the form of the starting integer, the To keyword, and the ending integer. The end of the loop is defined by the Next keyword. Commands placed between the For and Next keywords will be executed in each iteration of the For Next loop.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	for	1 to 5		For 1 to 5
3	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
4	next			
5				

In the example above, For 1 To 5 defines that the loop will repeat 5 times. The '*IDN?' query is placed between the For and Next keywords, so the '*IDN?' query will be sent 5 times as the loop runs.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	for	5 to 1		For 5 to 1
3	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
4	next			
5				

A For Next loop can count down as well as count up. In the above example, the argument for the For keyword is '5 to 1'. This means the For Next loop will count down from 5 to 1.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	for	0 to -4		For 0 to -4
3	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
4	next			
5				

The For argument can contain positive or negative integers. In the above example, the For Next loop still runs 5 times, but, in this case, it counts down, starting at 0 and ending at -4.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	[startnum]=	1		[STARTNUM] = 1
3	[endnum]=	5		[ENDNUM] = 5
4	for	[startnum] to [endnum]		For 1 to 5
5	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
6	next			
7				

The For argument can be defined by the contents of variables, tags and/or cell assignments. The above illustration depicts the For Next loop running, starting at the contents of the variable '[startnum]', which has been assigned the value of 1, and ending at the value stored by '[endnum]', which has been assigned the value of 5.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	for	<startnum> to (examples_A1)		For 1 to 5
3	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
4	next			
5				

The above example accomplishes the same task using the tag <startnum>, which holds the value of 1 on a different sheet as the starting integer and the value of 5 stored in cell A1 of an embedded sheet called 'Examples' as the ending number of the loop.

Nextcount

Each For Next loop contains an integral counter called nextcount that contains the count of its For Next loop. When nextcount is placed in the Command column without an argument, the Reply column will display the current count of the For Next loop.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	for	1 to 5		For 1 to 5
3	nextcount		2	
4	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
5	next			
6				

In the above illustration, nextcount is at a value of 2, meaning that loop has so far counted from 1 to 2.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	for	0 to -4		For 0 to -4
3	nextcount		-2	
4	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
5	next			
6				

Above, nextcount reveals that the For Next loop has counted from 0 to -2.

When an integer is placed in the argument column, nextcount is set to the value of the argument. This is useful if the user wishes the For Next loop to exit when a particular condition is met. By setting the value of nextcount to the terminal count of the For Next loop, the loop will exit when it encounters the Next keyword.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[genlevel]=	-100		[GENLEVEL] = -100
3	for	1 to 20		For 1 to 20
4	nextcount		12	
5	[genlevel]=	[genlevel] - 1		[GENLEVEL] = -112
6	:rf:gen:level	[genlevel]		:rf:gen:level -112
7	delay	1		delay 1
8	:FETCh:AF:ANALyzer:SINad?	[sinad]=reply4	0,0,1,11.54,0.00	[SINAD] = 11.54
9	if	[sinad] < 12		If 11.54 < 12 = True
10	nextcount	20		Nextcount = 20
11	endif			
12	next			
13				

As an example, the For Next loop illustrated above is set to exit if the count either reaches 20 or the measured SINAD value drops below 12 dB.

Row 4 has nextcount in the Command column without an argument, so the current value of nextcount is displayed in the Reply column. Row 5 lowers the value of the '[genlevel]' variable by 1 each time the loop iterates. '[genlevel]' is applied as the argument for the 39xx generator level RCI command on row 6, so each time the loop iterates, it lowers the instrument's RF generator level by 1 dB. After a one second delay to allow the measurement to settle, the SINAD meter reading is retrieved and assigned to the '[sinad]' variable. Row 9 contains an IF statement that checks to see if the SINAD reading has fallen below 12 dB. If the SINAD reading drops below 12 dB, then on row 10, nextcount will be assigned the value of 20, the terminal count of the For Next loop. It can be seen in the Reply column of row 4 that the For Next loop counted from 1 to 12 before the SINAD reading dropped below 12 dB. When the SINAD value dropped below 12 dB, nextcount was loaded with the value of 20, and the loop terminated before it had fully counted to 20 when it encountered the Next keyword.

Nextcount can be placed in the Argument column to assign the value of nextcount to a variable or tag.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[freq]=	151.0125		[FREQ] = 151.0125
3	for	1 to 3		For 1 to 3
4	[freqmult]=	nextcount		[FREQMULT] = 3
5	[recfreq]=	[freq] * [freqmult]		[RECFREQ] = 453.0375
6	:rf:analyzer:frequency	[recfreq]		:rf:analyzer:frequency 453.0375
7	next			
8				

In this example, nextcount is placed in the Argument column on row 4. The command column contains the variable assignment '[freqmult]='. This assigns the current value of nextcount to the variable '[freqmult]'.

Do Loop

Do/Exitdo/Loop/Docount

Command (*Argument*) Syntax:

Do (*no argument*) Defines the beginning of the loop.

Exitdo (*first value condition second value*) Conditional statement used for exiting the loop.

Loop (*no argument*) Defines the end of the loop.

Docount (*optional integer argument assigns value*) Integral counter that increments once per iteration. Can also be used in the Argument column when assigning the count value to a variable or tag.

Do, exitdo, docount and loop are not case sensitive.

The Do loop is used to iterate a sequence of commands a non-specific number of times. The Do loop is defined by the Do statement in the Command column. The Do command does not use an argument. The end of the loop is defined by the Loop keyword. The Loop keyword does not use an argument. Commands placed between the Do and Loop keywords will be executed on each iteration of the Do loop. A Do loop will remain active until a condition is provided in the form of an argument for the Exitdo keyword, thereby breaking the loop. Therefore, Exitdo is a requirement for constructing a Do loop.

Exitdo requires a conditional statement as an argument. When the conditional statement is true, the loop will exit immediately from the row the Exitdo keyword that generated the true condition is placed on, and operation of the script will resume on the row immediately following the row the Loop keyword ending the Do loop is placed on. A Do loop may have more than one Exitdo condition so any number of conditions to exit the loop can be defined. Exitdo is placed in the Command column, and an argument consisting of two values separated by a conditional symbol is placed in the Argument column. There are six conditional operators that can be used for the Exitdo argument. The table below lists the conditional operators used by Exitdo.

Symbol	Conditional Statement
=	First value equal to second value is true
!=	First value not equal to second value is true
>	First value greater than second value is true
>=	First value greater than or equal to second value is true
<	First value less than second value is true
<=	First value less than or equal to second value is true

With the exception of the '=' and '!=' operators, the values evaluated by the conditional statements must be numerical. In the case of the '=' and '!=' operators, the values used for comparison can be either text or numerical.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	do			
3	*idn?	[modelnum]=reply2	AEROFLEX,3902,297001040,3.7.8,2	[MODELNUM] = 3902
4	exitdo	[modelnum] = "3902"		If 3902 = 3902 = True
5	*idn?			
6	loop			
7				

For example, the above illustration depicts the '=' operator being used to exit the loop based upon the text value of the '[modelnum]' variable.

Docount

Each Do loop contains an integral counter called docount. Docount contains the count of its do loop. Docount steps to a value of 1 when its Do keyword is executed. Docount will increment by 1 each time the Loop keyword that defines the end of its loop is encountered. When docount is placed in the Command column without an argument, the Reply column will display the current count of the Do loop.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	do			
3	docount		3	
4	exitdo	docount = 3		If 3 = 3 = True
5	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
6	loop			
7				

The above illustration depicts a simple Do loop. Row 2 defines the beginning of the loop with the do statement. Row 6 defines the end of the loop with the loop statement. On row 3, the docount keyword with no argument causes the current count value of the loop to be displayed in the Reply column. Row 4 contains the keyword exitdo. The argument for exitdo is 'docount = 3', so when the count equals 3, the loop will exit and move execution to row 7, which, in this case, is empty. Row 5 contains the RCI query '*IDN?', so for every iteration of the loop where docount is not equal to 3, the '*IDN?' query will be sent to the instrument.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	do			
3	exitdo	docount = 1		If 1 = 1 = True
4	*idn?			
5	loop			
6	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
7				

The above example depicts what happens when the loop is directed to exit when the condition for the exitdo keyword is true. In this case, the loop will exit when docount is equal to 1. In this case, when the loop begins with the do statement, docount is automatically set to 1, its starting count. On row 3, exitdo is set to cause the loop to exit when docount is equal to 1. Because docount is immediately equal to 1 at the beginning of the loop, the exitdo condition becomes true, and the loop exits. When the loop exits, execution of the script jumps to the line immediately following the loop keyword which defines the end of the loop. In this case, because the loop exits when docount is equal to 1, the '*IDN?' query on row 4 is never executed; instead, the loop exits to row 6 and executes the command on that row.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	do			
3	exitdo	docount = 2		If 2 = 2 = True
4	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
5	loop			
6	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
7				

The above example is similar to the previous example, except on row 3, exitdo is set to exit the loop when docount = 2. In this case, the loop will pass through row 4 on the first iteration of the loop because on the first iteration of the loop, docount is equal to 1. Therefore, the '*IDN?' query is sent to the instrument on the first iteration of the loop. When docount is equal to 2 on the second iteration, execution jumps to row 6 and the '*IDN?' query on that row is executed.

Multiple Loop Exit Conditions

The do loop provides the facility to perform a function the outcome of which can be based on several conditions. An example of this capability is an operation that need to generate an outcome within a specific amount of time or number of tries, such as a SINAD search. If the outcome takes an inordinate amount of time, or is not attainable at all, the do loop can be set to 'give up' after a specific amount of time has passed.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[genlevel]=	-100		[GENLEVEL] = -100
3	do			
4	[genlevel]=	[genlevel] - 1		[GENLEVEL] = -112
5	:rf:gen:level	[genlevel]		:rf:gen:level -112
6	delay	1		delay 1
7	:FETCh:AF:ANALyzer:SINad?	[sinad]=reply4	0,0,1,11.53,0.00	[SINAD] = 11.53
8	exitdo	[sinad] < 12		If 11.53 < 12 = True
9	loop			
10				

The above example depicts a simple Do Loop that lowers the generator level of an instrument until the measured SINAD level of the unit under test is less than 12 dB. The generator level is first set at -100 dBm on row 2. For each iteration of the loop, the generator level is lowered 1 dB. On row 7, a SINAD measurement is taken, and, on row 8, exitdo is set to exit the loop when the SINAD measurement is less than 12 dB.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[genlevel]=	-100		[GENLEVEL] = -100
3	do			
4	[genlevel]=	[genlevel] - 1		[GENLEVEL] = -110
5	:rf:gen:level	[genlevel]		:rf:gen:level -110
6	delay	1		delay 1
7	:FETCh:AF:ANALyzer:SINad?	[sinad]=reply4	0,0,1,13.32,0.00	[SINAD] = 13.32
8	exitdo	[sinad] < 12		If 13.32 < 12 = False
9	exitdo	docount = 10		If 10 = 10 = True
10	loop			
11				

The above Do Loop adds an additional condition for exiting the SINAD do loop. On row 10, the loop is set to exit if docount is equal to 10. In other words, this loop is set to either find the generator level that produces a SINAD measurement of less than 12 dB, or 'give up' after 10 tries and move on. In the example, the loop exits before the less than 12 dB SINAD reading is attained.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[genlevel]=	-100		[GENLEVEL] = -100
3	start_timer_1		6:57:38 AM	
4	do			
5	[genlevel]=	[genlevel] - 1		[GENLEVEL] = -105
6	:rf:gen:level	[genlevel]		:rf:gen:level -105
7	delay	1		delay 1
8	:FETCh:AF:ANALyzer:SINad?	[sinad]=reply4	0,0,1,18.22,0.00	[SINAD] = 18.22
9	current_timer_1	[elapsed_time]=reply	6	[ELAPSED_TIME] = 6
10	exitdo	[sinad] < 12		If 18.22 < 12 = False
11	exitdo	docount = 10		If 5 = 10 = False
12	exitdo	[elapsed_time] > 5		If 6 > 5 = True
13	loop			
14				

The example above adds a third condition to the SINAD Do Loop. On row 3, a timer is set before the loop begins. On row 12, an additional exitdo condition is set: if the amount of time elapsed since the loop started is greater than 5 seconds, the loop will exit. Therefore this loop sets 3 conditions for the measurement: the loop will exit if the SINAD reading is less than 12 dB, the loop will exit if it has tried 10 times to find the measurement, or the loop will exit if it has run more than 5 seconds. In this case, the loop exits after the elapsed time has exceeded 5 seconds.

IF Statement

IF/ElseIF/Else/Endif

Command (*Argument*) Syntax:

If (*first value condition second value*) Conditional statement for executing following row(s).

Elseif (*first value condition second value*) Optional conditional statement for executing following row(s).

Else (*no argument*) Optional. Executes following rows if no previous IF condition has been met.

Endif (*no argument*) Defines the end of the IF block.

IF, ELSEIF, ELSE and ENDIF are not case sensitive. These keywords are used to construct a conditional IF Block. The IF keyword defines the beginning of the block, and uses as its argument a conditional statement. If the defined condition is true, the rows following the IF statement are executed. The optional ELSEIF statement uses as its argument a conditional statement, which is a different set of conditions than those used for the IF statement. If the defined condition is true, the rows following the ELSEIF statement are executed. The optional ELSE statement defines the rows that will be executed if no previous conditional statement is true. The required ENDIF statement defines the end of the IF Block.

The IF, ELSEIF, ELSE and ENDIF keywords are not case sensitive.

The table below lists the conditional operators used by the IF and ELSEIF keywords.

Symbol	Conditional Statement
=	First value equal to second value is true
!=	First value not equal to second value is true
>	First value greater than second value is true
>=	First value greater than or equal to second value is true
<	First value less than second value is true
<=	First value less than or equal to second value is true

With the exception of the '=' and '!=' operators, the values evaluated by the conditional statements must be numerical. In the case of the '=' and '!=' operators, the values used for comparison can be either text or numerical.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:system:load?	[system]=reply	Analog Duplex	[SYSTEM] = Analog Duplex
3	if	[system] = "Analog Duplex"		If Analog Duplex = Analog Duplex = True
4	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
5	endif			
6				

The above example demonstrates using a text value in the conditional statement. On row 3, the IF statement becomes true when the contents of the variable '[system]' are equal to "Analog Duplex".

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:rf:gen:level	-112		:rf:gen:level -112
3	delay	1		delay 1
4	:FETCh:AF:ANALyzer:SINad?	[sinad]=reply4	0,0,1,11.35,0.00	[SINAD] = 11.35
5	if	[sinad] < 12		If 11.35 < 12 = True
6	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
7	endif			
8				

The above example depicts a simple IF block. The instrument's RF generator is set to an output level of -112 dBm and a SINAD measurement is taken. On row 5, the IF keyword's conditional argument checks to see if the SINAD value is less than 12 dB. Because the measured SINAD value is 11.35, less than 12, the conditional argument evaluates as true, therefore the '*IDN?' query on row 6 is executed.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:rf:gen:level	-110		:rf:gen:level -110
3	delay	1		delay 1
4	:FETCh:AF:ANALyzer:SINad?	[sinad]=reply4	0,0,1,13.44,0.00	[SINAD] = 13.44
5	if	[sinad] < 12		If 13.44 < 12 = False
6	*idn?			
7	endif			
8				

In the above example, the generator level is changed to -110 dBm. The same IF condition as the previous example is used. In this case, the SINAD reading is 13.44, greater than 12, so the IF condition is evaluated as false. Because the IF condition evaluates as false, the '*IDN?' query on row 6 is not executed. Instead, execution will jump to the line immediately following the ENDIF keyword, which defines the end of the IF Block statement.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:rf:gen:level	-110		:rf:gen:level -110
3	delay	1		delay 1
4	:FETCh:AF:ANALyzer:SINad?	[sinad]=reply4	0,0,1,13.21,0.00	[SINAD] = 13.21
5	if	[sinad] < 12		If 13.21 < 12 = False
6	*idn?			
7	else			
8	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
9	endif			
10				

The illustration above depicts the use of the ELSE keyword. If all evaluations before the ELSE keyword are false, the commands between the ELSE keyword and the ENDIF keyword will be executed. In the case above, the IF statement is evaluated as false, so the '*IDN?' query on row 8 is executed. If the IF statement on row 5 had evaluated as true, execution of the script would have executed the '*IDN?' query on row 6, then jumped to row 10 immediately following the ENDIF statement, thereby bypassing the '*IDN?' query on row 8.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:rf:gen:level	-110		:rf:gen:level -110
3	delay	1		delay 1
4	:FETCh:AF:ANALyzer:SINad?	[sinad]=reply4	0,0,1,13.16,0.00	[SINAD] = 13.16
5	if	[sinad] < 12		If 13.16 < 12 = False
6	*idn?			
7	elseif	[sinad] > 12		If 13.16 > 12 = True
8	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
9	elseif	[sinad] > 11		
10	*idn?			
11	endif			
12				

The ELSEIF keyword argument is evaluated if any previous conditional statement in the IF block evaluates as false. The ELSEIF keyword can be used multiple times, but must always follow the IF keyword, which establishes the beginning of the IF statement. In the above example, the IF statement on row 5 evaluates as false. The ELSEIF statement on row 7 evaluates as true, so the '*IDN?' query on line 8 is executed. Execution of the script then jumps to row 12, the row immediately following the ENDIF keyword.

Flow Control, Messages and Forms

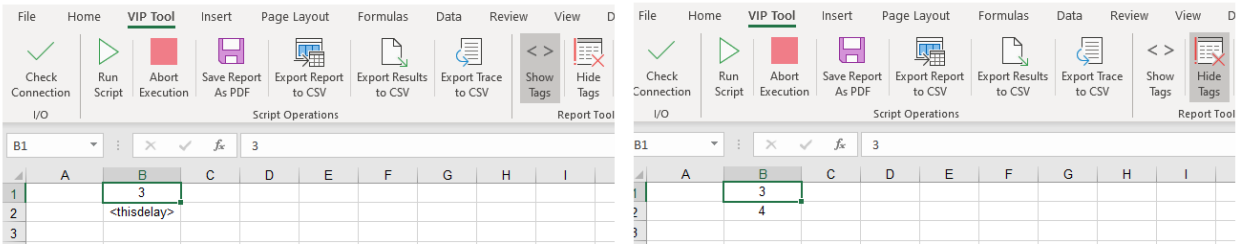
Delay Function

Delay

Command (*Argument*) Syntax:

Delay (*value in seconds*) Creates a pause in script execution for the time value defined by the argument.

	Command	Argument	Reply	Info Message
2	[mydelay]=	2		[MYDELAY] = 2
3	delay	[mydelay]		delay 2
4	delay	<thisdelay>		delay 4
5	delay	(calc_B1)		delay 3
6	delay	0.5		delay 0.5
7				
8				



The delay function provides a means to introduce a time delay between commands of a script. Generally, this function is used to allow the instrument to complete processing of a previous command before moving on to the next command in a script.

The argument for the delay function is represented in seconds. Decimal values are allowed in order to introduce delay control in milliseconds. For example, a delay argument of 0.100 will result in a delay of 100 milliseconds. The argument can be entered directly or assigned from a variable, tag or cell assignment.

In the above example, a tag called <thisdelay> is assigned the value of 4 on a sheet called ‘Calc’. Additionally, cell B1 of the ‘Calc’ sheet has a value of 3. On row 2 of the script, the variable ‘mydelay’ is assigned the value of 2. When the script is run, the delay value is obtained from the ‘mydelay’ value on row3, the delay value is obtained from the <thisdelay> tag value on row 4, the delay value is read directly from the “Calc” sheet cell B1 on row 5 of the script, and on row 6 of the script, the delay value of 0.5 seconds is entered directly.

The delay command is not case sensitive

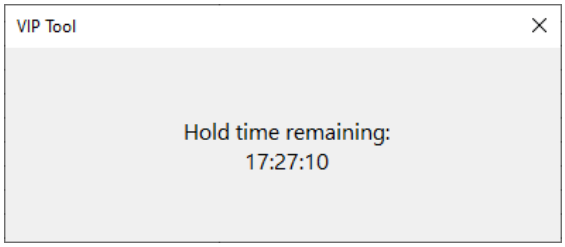
The delay value can be a positive integer or positive decimal value

Schedule Function

Schedule

Command (*Argument*) Syntax:

schedule (*target time in hours:minutes:seconds*) Pauses execution until computer time matches target time.



The schedule command is used to pause execution of a script until the computer's time matches the time entered in the argument column for schedule command. This allows the script to perform a function at a specific time of day. Schedule can pause execution for as long as 24 hours.

When the schedule command is encountered by the script, a message displaying the remaining time before execution resumes will appear, and the row color will alternate between green and purple until execution of the program resumes. The schedule keyword is not case sensitive.

End Function

End

Command (*Argument*) Syntax:

end (*no argument*) Halts execution of script when invoked.

When encountered by the script, the end command halts execution of the script; it has the same effect as a script encountering an empty row.

The end command can be placed within a script to stop execution of the script based upon a conditional statement.

The end command is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	*idn?		Aeroflex, 88XX, 1000000009, 2.4.4-0, Jan 12 2021, 08:55:28	
3	*idn?		Aeroflex, 88XX, 1000000009, 2.4.4-0, Jan 12 2021, 08:55:28	
4	end			
5	*idn?			
6				

In the above example, rows 2 and 3 are executed by the script, but the script halts execution on row 4 when the end keyword is encountered. Therefore, the command on row 5 was not executed.

Custom Dynamic Message Box

Display_Message /Message1/Message2/Close_Message

Command (*Argument*) Syntax:

Display_message (*No argument*) Opens the display_message text box.

Message1 (*message 1 as text, variable, tag or cell assignment*) Places the value of its argument as the first (top line) message on the display_message text box.

Message2 (*message 2 as text, variable, tag or cell assignment*) Places the value of its argument as the second (bottom line) message on the display_message text box.

Close_Message (*no argument*) Closes the display_message text box.

Display_message provides a method of opening a message box and dynamically displaying two lines of messages. Executing display_message opens a modeless text box. This means that script execution will continue while the message is displayed. The display_message text box will only close if is programmatically closed using the close_message keyword, or if the Abort key is pressed.

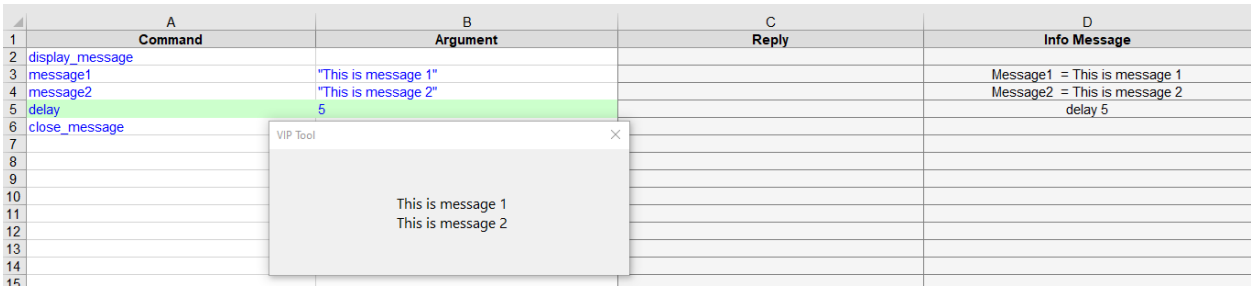
The display_message text box displays two lines of text. The keyword message1 is used to generate the first line of text, using the value held by its argument as the text value to be displayed. The argument can be in the form of direct text entry or a value held by a variable, tag, or cell assignment. The display_message text box must be first displayed by executing the display_message keyword before using message1 to define the first line of text in the display_message text box. If message1 is executed before the display_message text box is displayed, the command will generate an error.

The keyword message2 is used to generate the second line of text, using the value held by its argument as the text value to be displayed. The argument can be in the form of direct text entry or a value held by a variable, tag, or cell assignment. The display_message text box must be first displayed by executing the display_message keyword before using message2 to define the second line of text in the display_message text box. If message2 is executed before the display_message text box is displayed, the command will generate an error.

The close_message keyword is used to close the display_message text box. Close_message can only be used if a display_message text box has first been opened by executing the display_message keyword, otherwise executing close_message will generate an error.

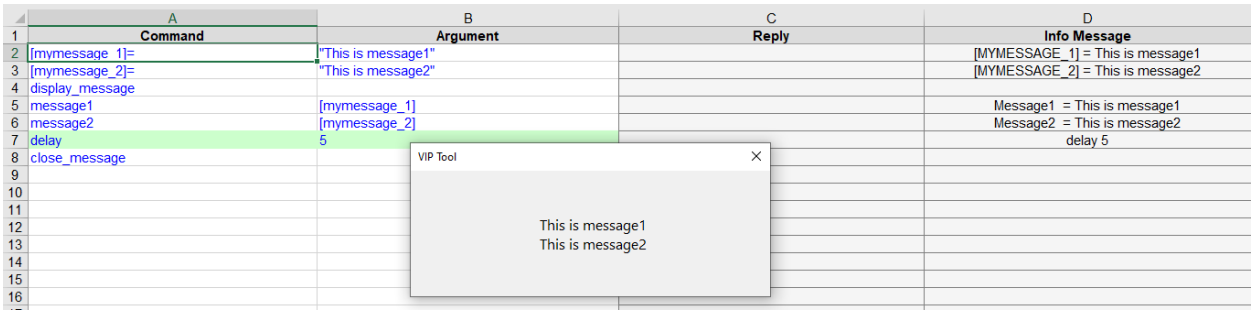
Only one display_message box can be active at any given time.

The messages displayed on the display_message text box can be dynamically changed while the display_message text box is open. For example, if the currently displayed message1 is “Hello”, executing message1 with the argument “world” will replace the first line with “world”.



The above example depicts the basic syntax for generating a display_message text box. On row 2, the display_message keyword is used to open the text box. On row 3, the message1 keyword is used to generate the first line of text with its argument “This is message 1”. On row 4, the message2 keyword generates the second line of text with its argument “This is message 2”. Row 5 introduces a delay of 5 seconds, so the message will remain displayed for 5 seconds. On row 6, the close_message keyword removes the display_message text box after the 5 second delay has elapsed.

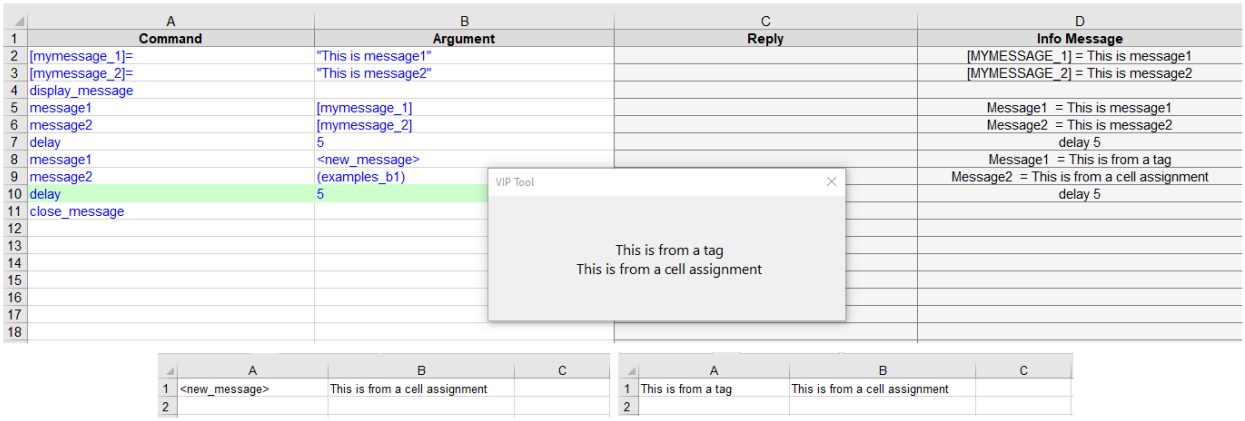
The message2 keyword can be executed without first executing message1, but, to avoid errors, the display_message keyword must always be executed before executing message1, message2 or close_message.



Above, the same result as the first example is accomplished, but this example uses variables as arguments for message1 and message2.

On row 2, the variable ‘[mymessage_1]’ is assigned the value of “This is message 1”. On row 3, the variable ‘[mymessage_2]’ is assigned the value of “This is message 2”.

On row 5, message1 uses the variable ‘[mymessage_1]’ as the argument for the message1 keyword. On row 6, message2 uses the ‘[mymessage_2]’ variable as the argument for the message2 keyword.

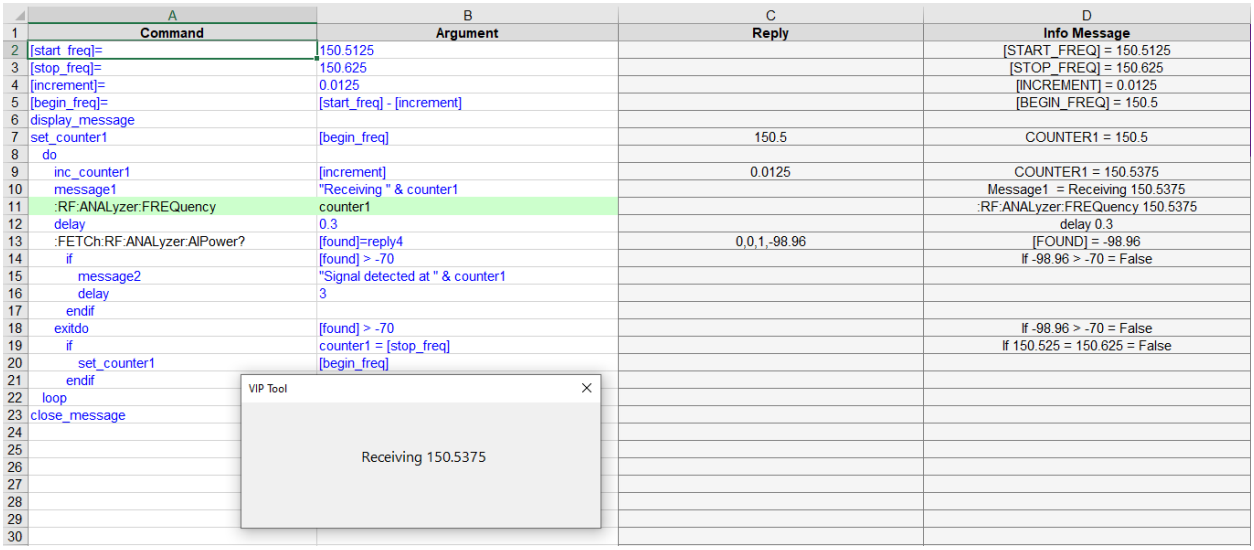


	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mymessage_1]=	"This is message1"		[MYMESSAGE_1] = This is message1
3	[mymessage_2]=	"This is message2"		[MYMESSAGE_2] = This is message2
4	display_message			
5	message1	[mymessage_1]		Message1 = This is message1
6	message2	[mymessage_2]		Message2 = This is message2
7	delay	5		delay 5
8	message1	<new_message>		Message1 = This is from a tag
9	message2	(examples_b1)		Message2 = This is from a cell assignment
10	delay	5		delay 5
11	close_message			
12				
13				
14				
15				
16				
17				
18				

	A	B	C
1	<new_message>	This is from a cell assignment	
2			

	A	B	C
1	This is from a tag	This is from a cell assignment	
2			

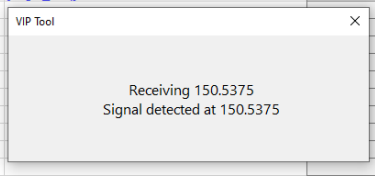
Above it can be seen that the argument for message1 or message2 can be a text value (as seen on rows 2 and 3). On rows 5 and 6, variables are used for the message arguments. Row 8 uses a tag to define the message1 argument, and row 9 uses a cell assignment to define the message2 argument. The following illustration demonstrates the use of dynamic messages in the display_message text box. The display_message keywords are not case sensitive.



	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[start_freq]=	150.5125		[START_FREQ] = 150.5125
3	[stop_freq]=	150.625		[STOP_FREQ] = 150.625
4	[increment]=	0.0125		[INCREMENT] = 0.0125
5	[begin_freq]=	[start_freq] - [increment]		[BEGIN_FREQ] = 150.5
6	display_message			
7	set_counter1	[begin_freq]	150.5	COUNTER1 = 150.5
8	do			
9	inc_counter1	[increment]	0.0125	COUNTER1 = 150.5375
10	message1	"Receiving " & counter1		Message1 = Receiving 150.5375
11	:RF:ANALyzer:FREQuency	counter1		:RF:ANALyzer:FREQuency 150.5375
12	delay	0.3		delay 0.3
13	:FETCh:RF:ANALyzer:AlPower?	[found]=reply4	0,0,1,-98.96	[FOUND] = -98.96
14	if	[found] > -70		If -98.96 > -70 = False
15	message2	"Signal detected at " & counter1		
16	delay	3		
17	endif			
18	exitdo	[found] > -70		If -98.96 > -70 = False
19	if	counter1 = [stop_freq]		If 150.525 = 150.625 = False
20	set_counter1	[begin_freq]		
21	endif			
22	loop			
23	close_message			
24				
25				
26				
27				
28				
29				
30				

The message box is opened on row 6. After the message box is opened, the script enters a Do Loop on row 8. Each time the Do Loop iterates, counter1 is incremented by .0125. The value of counter1 is assigned as the argument to the instrument RCI receiver frequency command on row 11. On row 10, this same value is concatenated with the text “Receiving “, and that concatenated value is used as the argument for the message1 keyword. The result is, every time the DO Loop iterates, the message box updates message1 with the frequency that the instrument’s receiver is tuned to.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[start_freq]=	150.5125		[START_FREQ] = 150.5125
3	[stop_freq]=	150.625		[STOP_FREQ] = 150.625
4	[increment]=	0.0125		[INCREMENT] = 0.0125
5	[begin_freq]=	[start_freq] - [increment]		[BEGIN_FREQ] = 150.5
6	display_message			
7	set_counter1	[begin_freq]	150.5	COUNTER1 = 150.5
8	do			
9	inc_counter1	[increment]	0.0125	COUNTER1 = 150.5375
10	message1	"Receiving " & counter1		Message1 = Receiving 150.5375
11	:RF:ANALyzer:FREQuency	counter1		:RF:ANALyzer:FREQuency 150.5375
12	delay	0.3		delay 0.3
13	:FETCh:RF:ANALyzer:AlPower?	[found]=reply4	0,0,1,-53.97	[FOUND] = -53.97
14	if	[found] > -70		If -53.97 > -70 = True
15	message2	"Signal detected at " & counter1		Message2 = Signal detected at 150.5375
16	delay	3		delay 3
17	endif			
18	exitdo	[found] > -70		If -99 > -70 = False
19	if	counter1 = [stop_freq]		If 150.525 = 150.625 = False
20	set_counter1	[begin_freq]		
21	endif			
22	loop			
23	close_message			
24				
25				
26				
27				
28				
29				
30				



The Do Loop contains a conditional IF statement on row 14 that checks to see if a variable that holds the value of a power reading is greater than -70 dBm. In the example above, the IF statement becomes true while the frequency value, held by counter1, is 150.5375. Because the IF condition is true, the message2 keyword on row 15 is executed. The argument for the message2 keyword is a concatenation of the phrase “Signal detected at“ and the value held by counter1. Therefore, when the message2 keyword is executed, the phrase “Signal detected at 150.5375” is displayed on the message box.

Wait for Transmit On Dynamic Message Box

Wait_For_Transmit_On

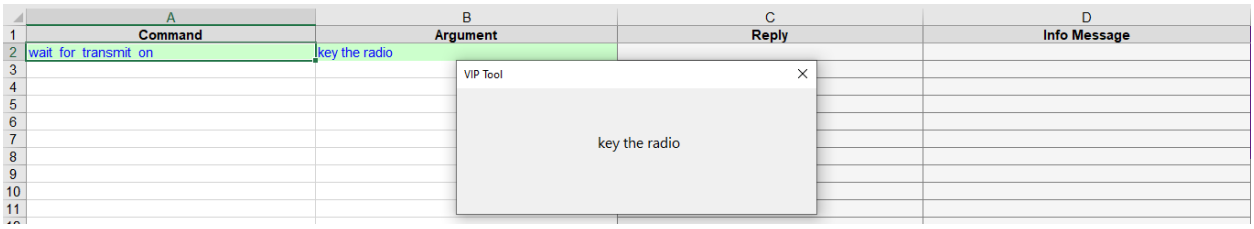
Command (*Argument*) Syntax:

Wait_for_Transmit_On (*text*, *variable, tag, or cell assignment*) Generates a message box that displays the message defined in its argument and pauses execution of the script. After the message box is generated, wait_for_transmit_on monitors the power meter of the instrument, and, when a power level over 90 mW is detected, the message box is closed and execution of the script resumes.

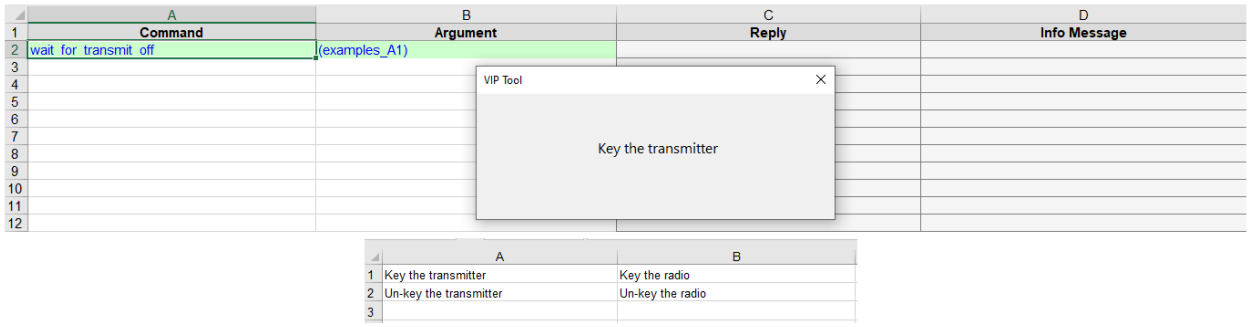
Wait_for_transmit_on is a method for pausing execution of the script and instructing the operator to key a transmitter connected to the RF power input of the instrument to allow a script to perform transmitter tests while the transmitter is transmitting. When RF power is detected by the instrument’s power meter, wait_for_transmit_on automatically un-pauses execution of the script.

The argument for wait_for_transmit_on supplies the text message for the message box. It can be entered as either text or the value held by a variable, tag or cell assignment.

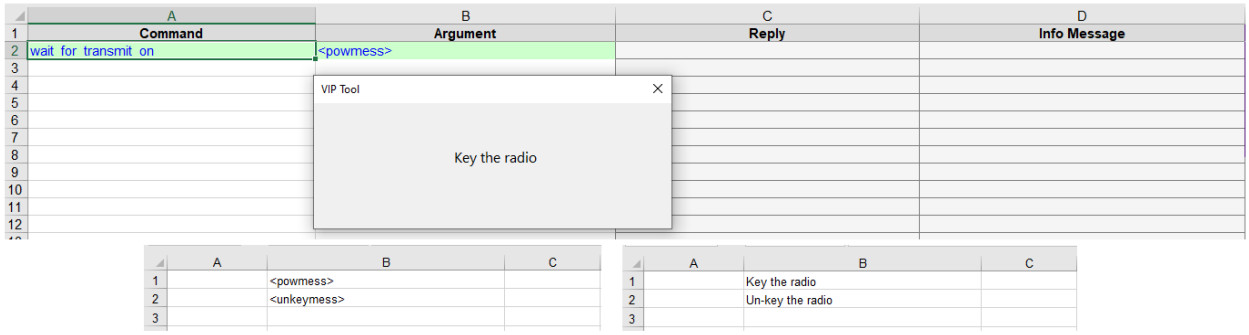
The wait_for_transmit_on keyword is not case sensitive.



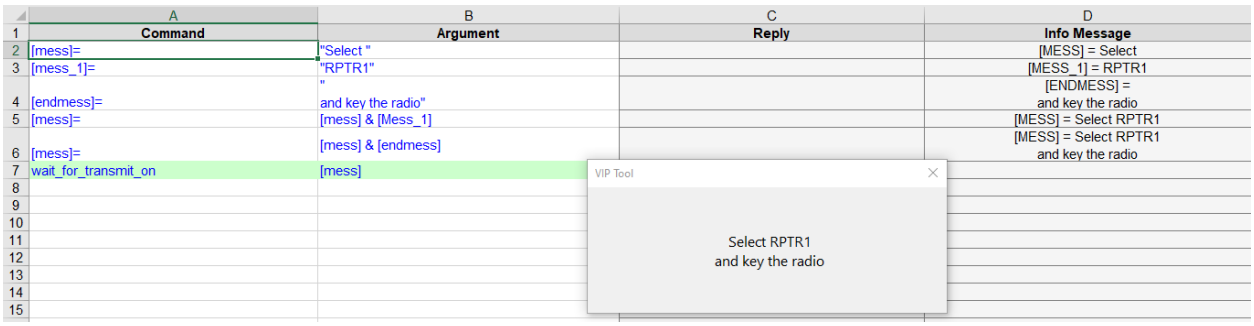
In the above example, the text “key the radio” is entered as the message for the wait_for_transmit_on message box. The text can optionally be enclosed in double quotations.



Above, a cell assignment is used to supply the text to be displayed in the wait_for_transmit_on message box. The cell assignment “(examples_A1)” points the contents in cell A1 of an embedded sheet in the VIP Tool workbook called “Examples”.



In the illustration above, the contents of a cell assigned the tag ‘<powmess>’ are assigned as the message to be displayed on the wait_for_transmit_on message box. The tag can reside on any report sheet.



The above illustration demonstrates that a variable can hold the value to be assigned as the text in the wait_for_transmit_on message box. On row 7, the variable ‘[mess]’ is used to supply the two-line text “Select

RPTR1 and key the radio". In this case, the variable '[mess]' was built up using concatenation. On row 4, the double quotations symbol was entered in the argument cell, followed by the key combination ALT-ENTER, which placed a carriage return after the first double quotes. This is followed by the text "and key the radio", followed by an ending double quotations symbol. This method is useful for adding a second line to the wait_for_transmit_on message. The rules for entering the argument to the wait_for_transmit_on keyword are identical to the rules used for entering the argument for the pause keyword; refer to the section of this manual describing the pause keyword and the section entitled 'Concatenation' for additional details.

Wait for Transmit Off Dynamic Message Box

Wait_For_Transmit_Off

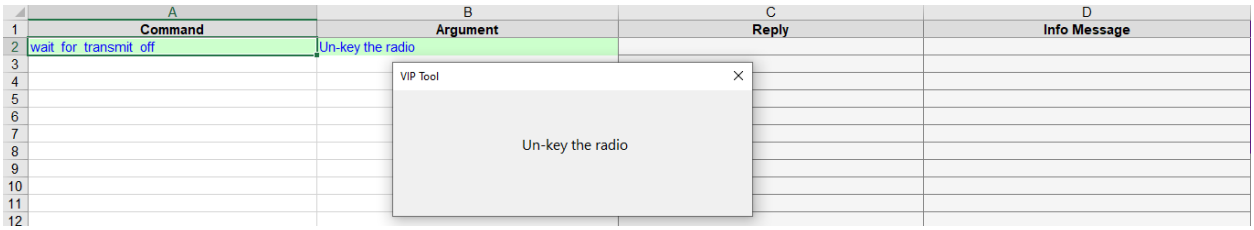
Command (*Argument*) Syntax:

Wait_for_Transmit_Off (*text*, *variable, tag, or cell assignment*) Generates a message box that displays the message defined in its argument and pauses execution of the script. After the message box is generated, wait_for_transmit_off monitors the power meter of the instrument, and, when a power level over 90 mW is detected, the message box is closed and execution of the script resumes.

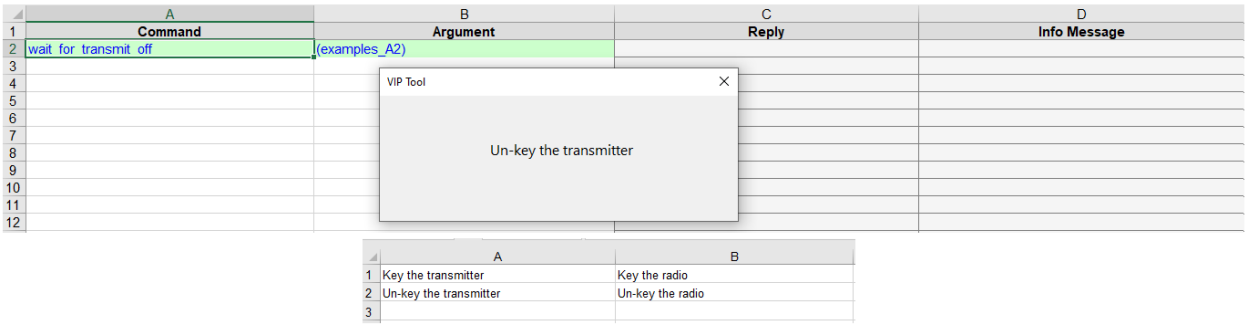
Wait_for_transmit_off is a method for pausing execution of the script and instructing the operator to key a transmitter connected to the RF power input of the instrument to allow a script to perform transmitter tests while the transmitter is transmitting. When RF power is detected by the instrument's power meter, wait_for_transmit_off automatically un-pauses execution of the script.

The argument for wait_for_transmit_off supplies the text message for the message box. It can be entered as either text or the value held by a variable, tag or cell assignment.

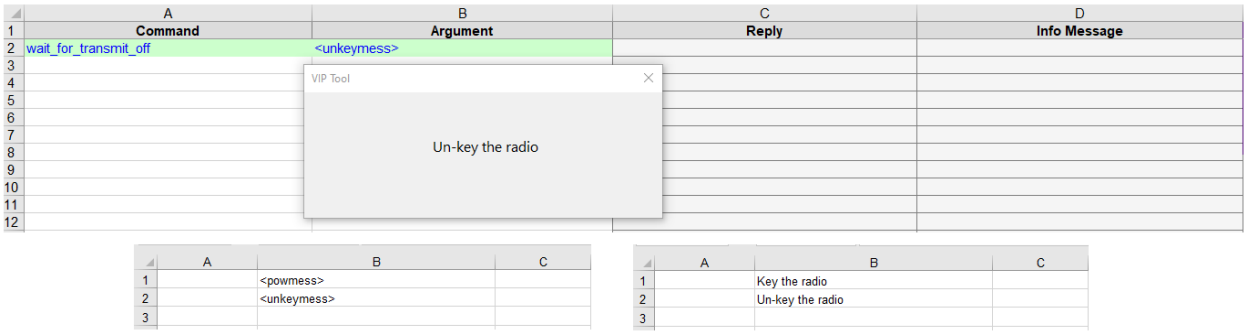
The wait_for_transmit_off keyword is not case sensitive.



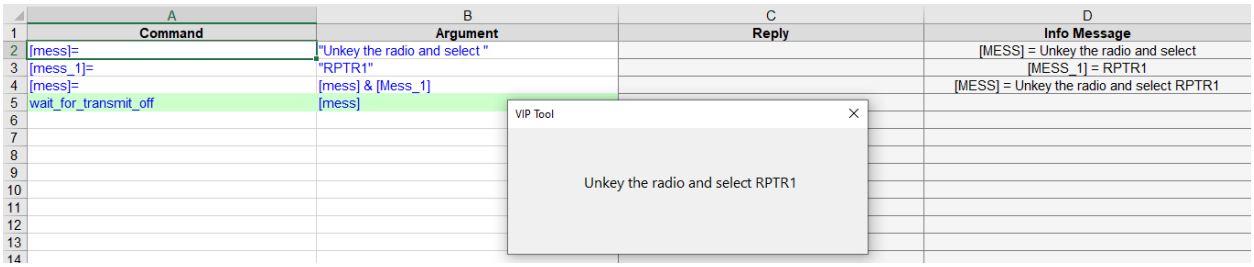
In the above example, the text "Un-key the radio" is entered as the message for the wait_for_transmit_off message box. The text can optionally be enclosed in double quotations.



Above, a cell assignment is used to supply the text to be displayed in the wait_for_transmit_off message box. The cell assignment "(examples_A2)" points the contents in cell A2 of an embedded sheet in the VIP Tool workbook called "Examples".



In the illustration above, the contents of a cell assigned the tag '<unkeymess>' are assigned as the message to be displayed on the wait_for_transmit_off message box. The tag can reside on any report sheet.



The above illustration demonstrates that a variable can hold the value to be assigned as the text in the wait_for_transmit_off message box. On row 5, the variable '[mess]' is used to supply the text "Unkey the radio and select RPTR1". In this case, the variable '[mess]' was built up using concatenation. The rules for entering the argument to the wait_for_transmit_off keyword are identical to the rules used for entering the argument for the pause keyword; refer to the section of this manual describing the pause keyword and the section entitled 'Concatenation' for additional details.

Wait for Audio Level Dynamic Message Box

Setaudio_{Units}_{Lower Limit}_{Upper Limit}

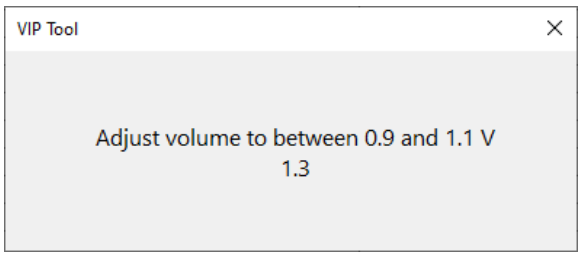
Command_{Units}_{Lower Limit} _{Upper Limit} (*Argument*) Syntax:

Setaudio_{units}_{lower limit}_{upper limit} (*optional variable, tag, or cell assignment*) Pauses execution of the script and opens a message box. The function monitors the instrument audio level meter; when the audio level meter falls within the window defined by the lower limit and upper limit in the units defined by the placeholders, the message box closes and the audio level value that was detected within the measurement window is reported in the Reply column and assigned to the optional argument variable, tag, or cell assignment. When the message box closes, the script moves to the next step.

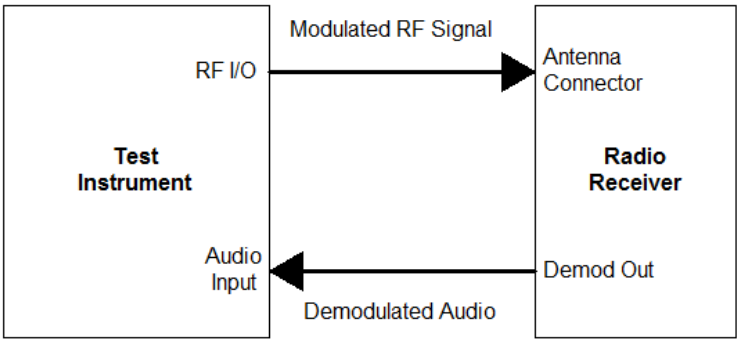
The command includes three placeholders which are defined by {units}, {lower limit} and {upper limit} in the description. The {units} placeholder holds the value for the type of units being measured, the {lower limit} placeholder holds the value for lower limit of the measurement window, and the {upper limit} placeholder holds the value for upper limit of the measurement window. The placeholder values can be entered as direct entry or assigned from a variable or tag. The argument specifies the destination of the value calculated by the keyword and can be a variable, tag, or cell assignment. This syntax causes find_sinad to find the 12 dB SINAD level between 11.5 and 12.5 dB. This example assigns the microvolt reading to the tag '<12dB_sinad_uv>'. This tag's value appears on a report sheet cell to which the tag is assigned.

Setaudio_{units}_{lower limit}_{upper limit} is not case sensitive.

Generally, measurements that rely upon the demodulated output of a radio such as SINAD and distortion require that the audio level be at a specific value. Setaudio is used as an aid to set the demodulated output of a radio to the level required for a test. When setaudio is executed, the script will pause at its current point of execution and display a message prompting the user to adjust the volume level of the Device Under Test (DUT) to a value between a lower and upper limit. The lower and upper limits provide a window that the audio level must fall into before the message box is removed from view and the script can continue moving through the test procedure.



Below the "Adjust volume" prompt, the setaudio message will provide live measurements taken by the instrument's audio level meter. This measurement will update as the audio level changes when the volume control of the radio is adjusted up or down.

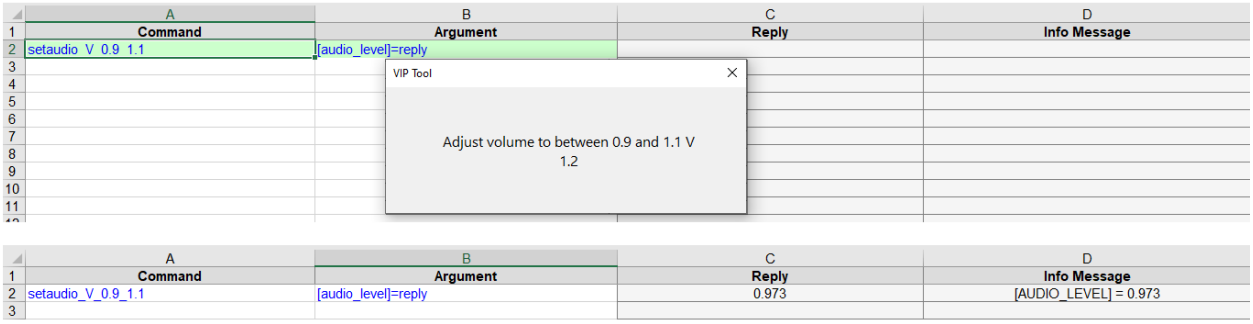


The RF I/O connector of the test instrument must be connected to the antenna connector of the radio receiver. If an Over The Air (OTA) connection is required, connect the instrument RF I/O connector to an antenna instead of making the direct connection pictured above.

The demodulated audio output of the radio must be connected to the audio input connector of the instrument. Generally, this connection is from the speaker output of the radio, and the use of a breakout box may be required to gain access to this signal.

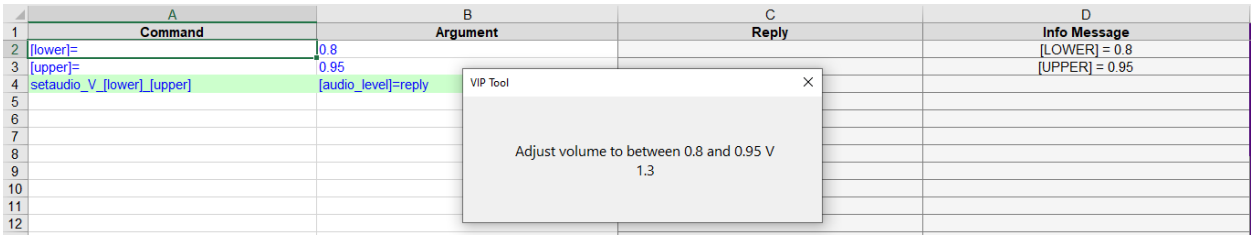
Before setaudio is used, the script must perform the following steps:

1. Set the instrument's RF generator frequency to the radio's receiver frequency
2. Set the output level of the instrument's RF generator so that the receiver can receive it clearly
3. Modulate the instrument's RF generator with the type of modulation required by the radio
4. Modulate the RF generator with any squelch tones required to un-squelch the radio receiver
5. Modulate the instrument's RF generator at the modulation level required by the test
6. Route the instrument's audio input to the instrument's audio level meter
7. Enable the instrument's RF generator

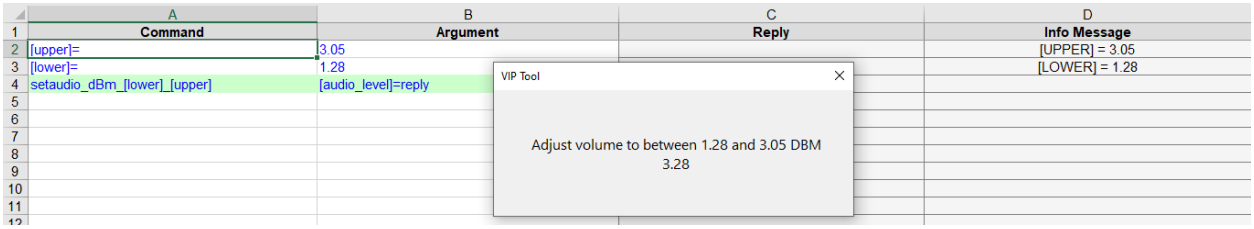


In the above example setaudio_V_0.9_1.1 sets the lower limit to 0.9 volts and the upper limit to 1.1 volts. In this example, the volume control of the radio was adjusted down until its measured value was 0.973 volts, the message box closed, and the found voltage was recorded as 0.973 volts.

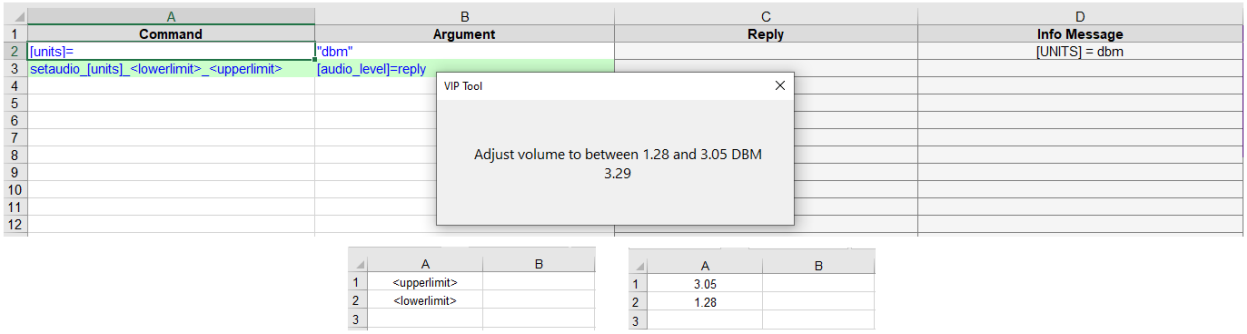
The argument assigns the found voltage to a variable called '[audio_level]'. The argument could also be a tag embedded in a report sheet, or a cell assignment, or, if the found audio level is no longer needed, the argument could be blank.



Above, the variables '[lower]' and '[upper]' are assigned '0.8' and '0.95' respectively. The variables are then used as placeholders in the setaudio command on row 4, setting the lower limit to 0.8 volts and the upper limit to 0.95 volts.



In the example above, the unit of measurement for setaudio is set to dBm. In this case, the audio level meter of the instrument must be set to measure the audio output of the radio to dBm.



Above, the variable '[units]' is assigned the value 'dbm', and is used as the placeholder that sets the units of measurement in setaudio. The cell assigned the tag '<lowerlimit>' contains the value '1.28' and the cell assigned the tag '<upperlimit>' contains the value '3.29'. Therefore, the setaudio message will close when the measured audio level is between 1.28 dBm and 3.05 dBm.

Pause Message Box

Pause

Command (Argument) Syntax:

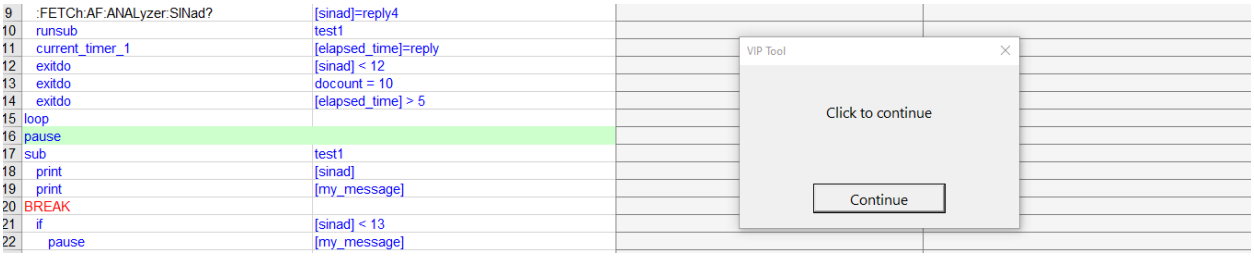
Pause (*optional pause text message*) Displays a message box that displays the text argument as the message along with a 'Continue' button; pauses script execution until the 'Continue' button is pressed.

Pause is a method for pausing execution of the script so that the operator of the script can be directed to perform a task before the script continues execution. For example, pause can be used to direct the operator to set the channel of the radio before the script runs a sequence of measurements for that channel.

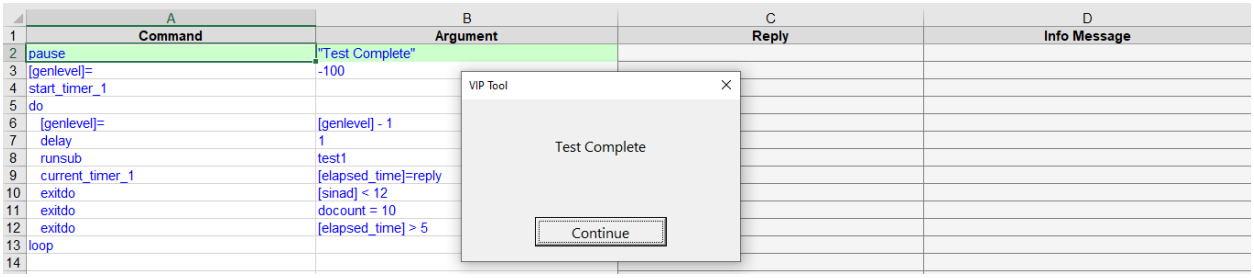
The pause message box is a modal message box. This means that script execution halts and does not continue until the 'Continue' button is pressed on the pause message box. When this message box is open, the Abort key cannot be used to stop execution of the script.

The text argument for pause is displayed as a message on the pause message box. If no argument is supplied with the pause keyword, the default message "Click to continue" is displayed.

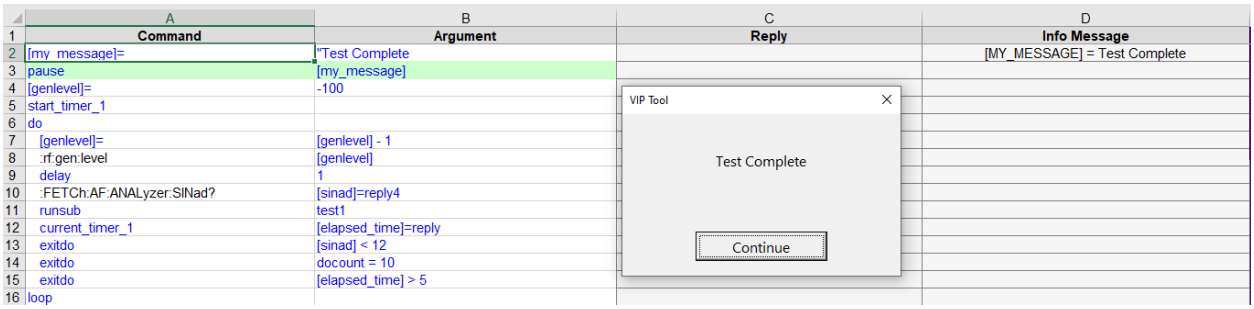
The pause keyword is not case sensitive.



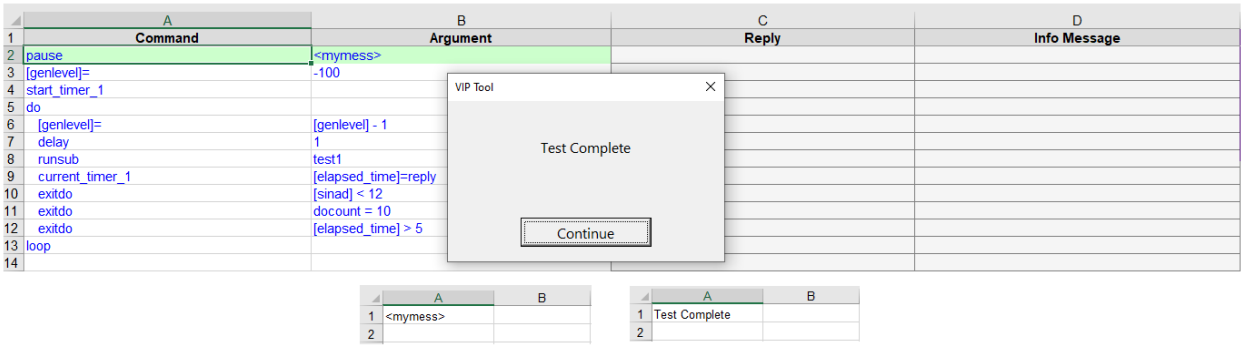
In the above illustration, no message is suppled for the pause keyword. In this case, the message 'Click to continue' is displayed. Execution is halted until the computer's Enter key is pressed, or the mouse is used to click the 'Continue' button.



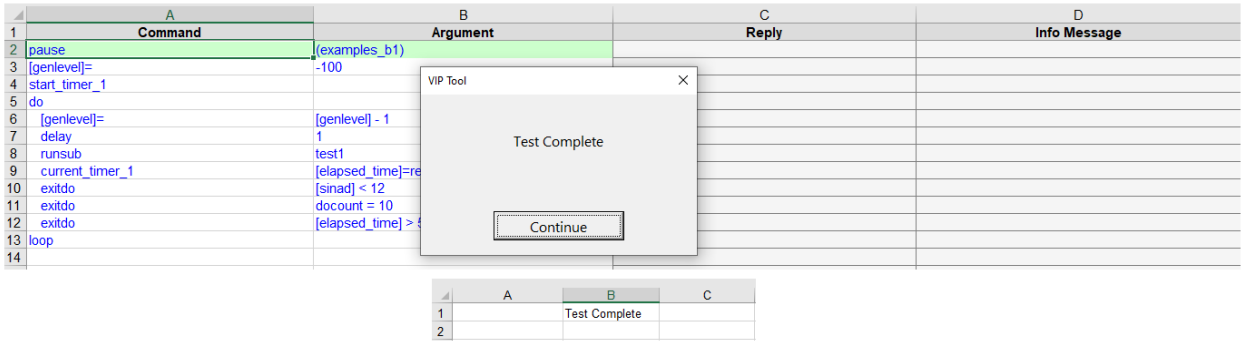
In the above example, the text "Test Complete" is entered directly as the argument for the pause keyword. This results in the text "Test Complete" to appear as the pause message.



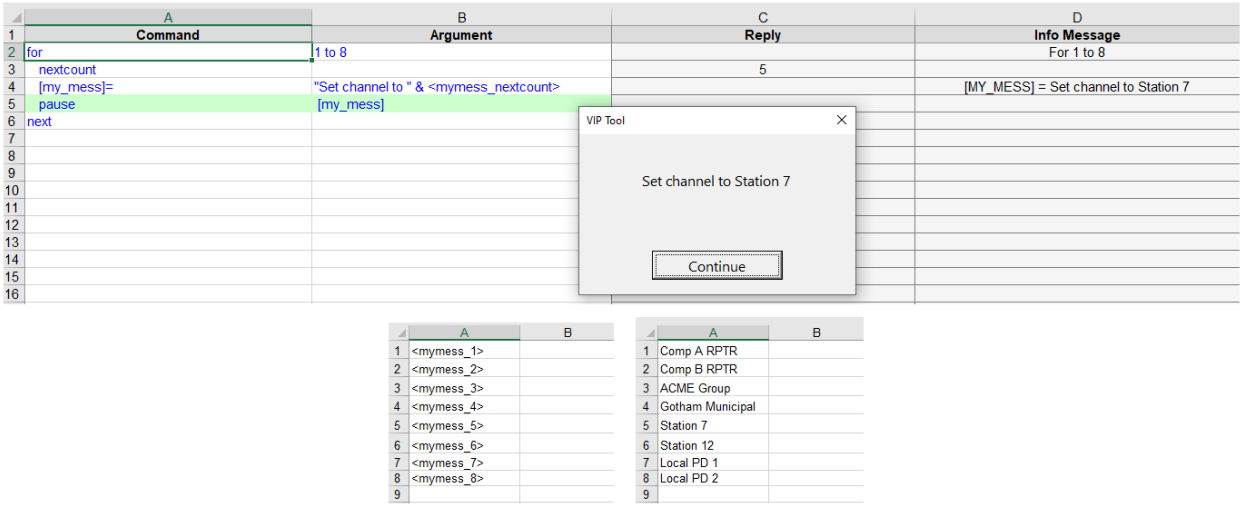
Variables can be used to hold the pause message text argument. In the above example, the variable '[my_message]' is assigned the text "Test Complete". On row three, '[my_message]' is supplied as the argument for the pause keyword. This results in "Test Complete" being displayed in the pause message box.



Tags can be used to hold the pause message text argument. In the above example, the tag '<mymess>', which exists on another sheet, is assigned the text "Test Complete". On row 2, '<mymess>' is supplied as the argument for the pause keyword. This results in "Test Complete" being displayed in the pause message box.



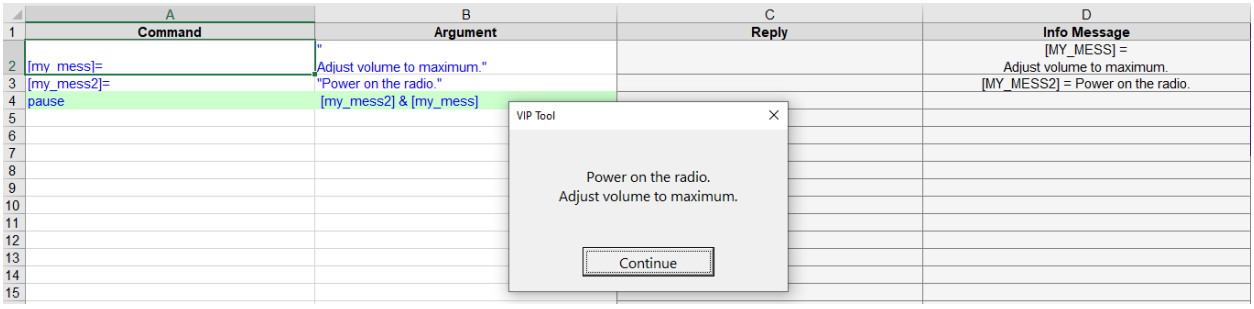
Cell assignments can be used to hold the pause message text argument. In the above example, the cell assignment '(examples_b1)', which addresses cell B1 on an embedded sheet called 'Examples', holds the text "Test Complete". On row 2, '(examples_b1)' is supplied as the argument for the pause keyword. This results in "Test Complete" being displayed in the pause message box.



The argument of the pause keyword be concatenated to generate different prompts within a loop. In the above example, a For Next loop is used to sequentially address multiple tags located on another sheet. The integral For Next counter, nextcount, is used to address which tag is used, based upon the current count of the For Next

loop. On row 4, the text “Set channel to “ is concatenated with the currently addressed tag. In the illustration, nextcount has counted up to, and holds the value of, ‘5’, therefore the tag ‘<mymess_5>’ is currently addressed. The cell ‘<mymess_5>’ is assigned to holds the text “Station 7”, so, when concatenated with the contents of ‘[my_mess]’, the outcome is “Set channel to Station 7”, which appears in the pause message box.

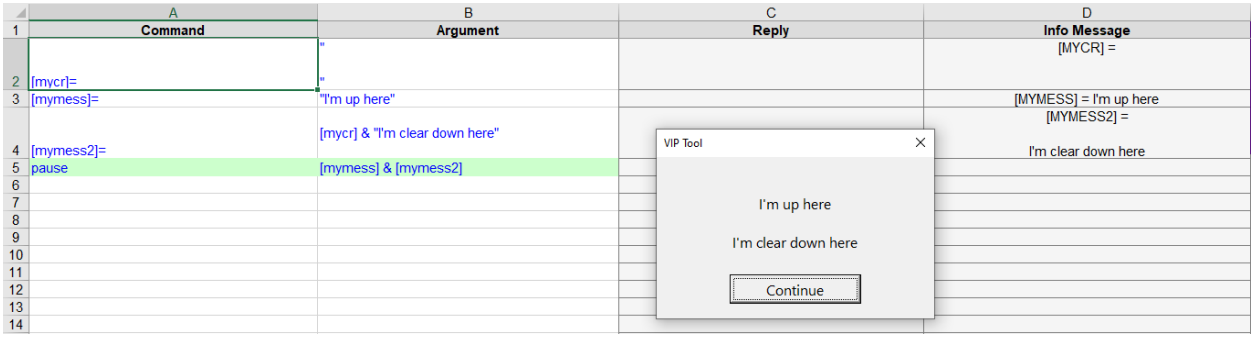
Using ALT-ENTER to Introduce Carriage Returns for Multi-Line Pause Messages



When the cell in the argument column is selected, pressing ALT-Enter on the computer keyboard will insert a carriage return into the cell. This carriage return can be used in the argument for the pause keyword or held within a text variable. On row 2, the double quotes symbol was entered. After that, ALT-ENTER was pressed to introduce a carriage return. The text “Power on the radio.” was entered afterwards, followed by the double quotes. The carriage return and text are assigned to the ‘[my_mess] variable.

The argument of the pause keyword can include concatenated text syntax. In the above example, the contents of the variable ‘[my_mess]’ and ‘[my_mess2]’ are concatenated in the argument for the pause keyword on row3.

NOTE:	Note that, because ‘[my_mess] contains the carriage return, it appears in the message box on a second line.



A variable can contain a carriage return with no accompanying text. Up to two carriage returns can be used in the pause message box. In the above example, two carriage returns are assigned to the variable ‘[mycr]’. The carriage returns were entered by selecting the argument cell, entering the double quotations symbol, then by pressing the ALT-ENTER key combination twice. After entering the carriage returns, the double quotations symbol was typed in. The variable ‘[mymess2]’ is created by concatenating the variable ‘[mycr]’, which contains the carriage returns, with the text “I’m clear down here” on row 4. On row 5, the variables ‘[mymess]’ and ‘[mymess2] are concatenated in the argument for the pause keyword to form the message displayed in the illustration.

Yes/No Selection Message Box

Query_Yes_No / Answer_Yes_No

Command (*Argument*) Syntax:

Query_yes_no (*optional query_yes_no text message*) Displays a message box that displays the text argument as the message along with 'Yes' and 'No' buttons. Pauses script execution until the 'Yes' or 'No' button is pressed. Stores either 'Yes' or 'No' answer, depending on which button is pressed.

Answer_yes_no (*variable, tag or cell assignment*) Assigns 'Yes' or 'No' result from the last query_yes_no message box executed to a variable, tag, or cell assignment.

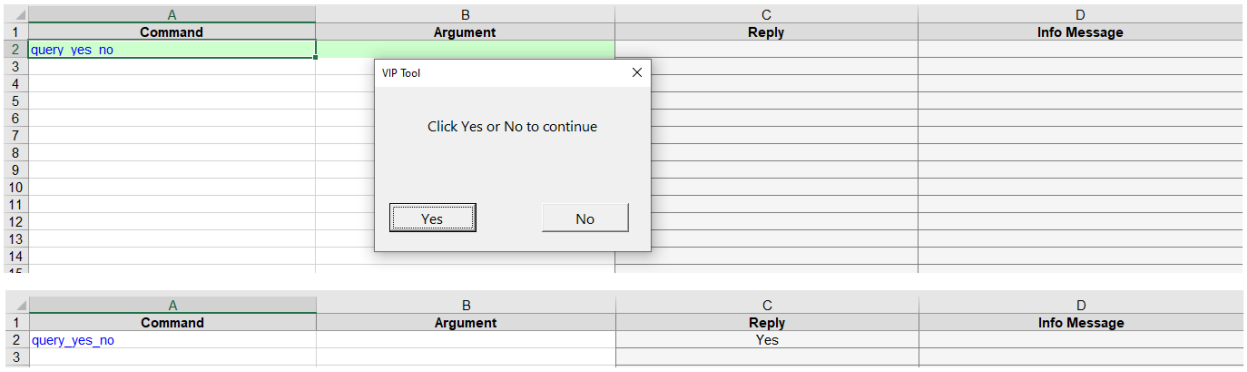
Query_yes_no is a method for pausing execution of the script and allowing the operator to enter either a 'Yes' or 'No' response. For example, query_yes_no can be used to determine if a radio is clearly demodulating a tone or not by supplying a message asking the operator if the tone can be heard clearly or not. The operator can respond to the message by clicking either the 'Yes' or 'No' button on the query_yes_no message box.

Answer_yes_no provides a method to assign the 'Yes' or 'No' response of the query_yes_no message box to a variable, tag, or cell assignment. The response can then be used to branch the script down a specific path, depending on the 'Yes' or 'No' value.

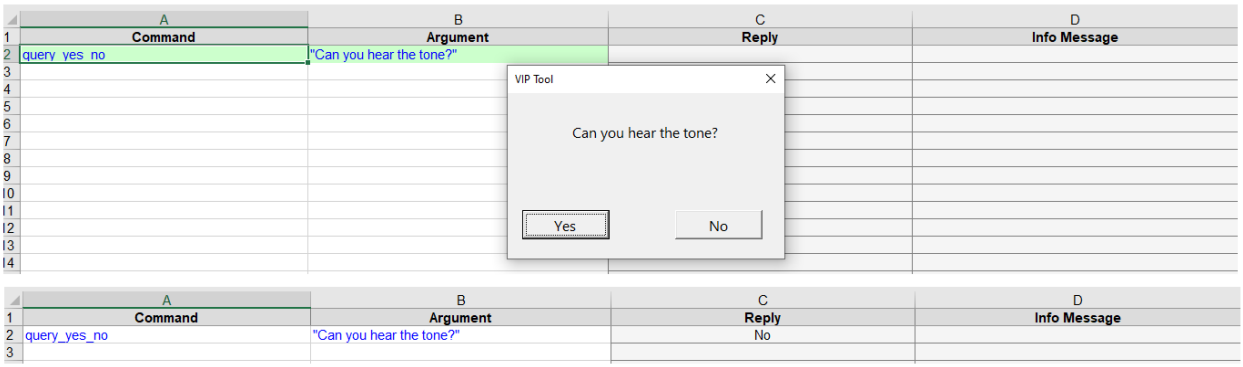
The query_yes_no message box is a modal message box. This means that script execution halts and does not continue until either the 'Yes' button or 'No' button is pressed. When this message box is open, the Abort key cannot be used to stop execution of the script.

The text argument for query_yes_no is displayed as a message on the query_yes_no message box. If no argument is supplied with the query_yes_no keyword, the default message "Click Yes or No to continue" is displayed. Query_yes_no is not case sensitive.

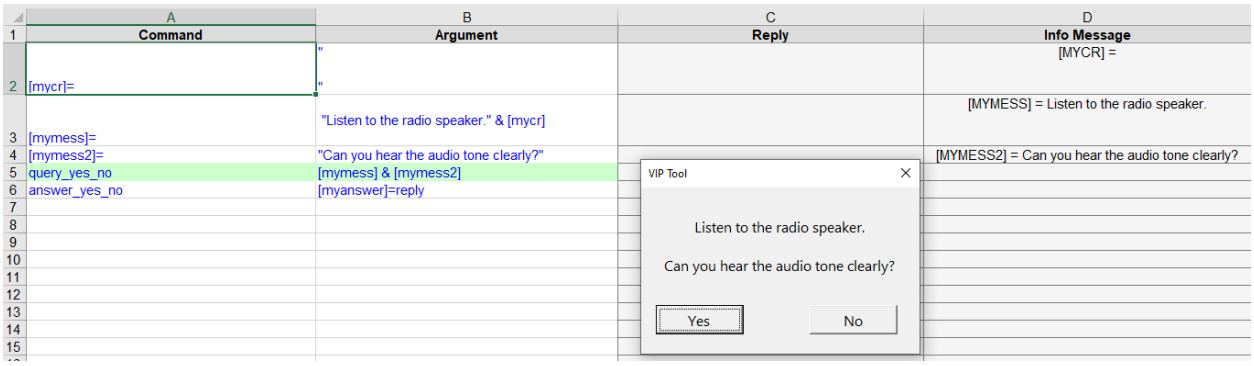
Answer_yes_no provides a method to assign the results of the last query_yes_no message executed to the variable, tag, or cell assignment that is supplied as an argument to answer_yes_no. The value generated by the query_yes_no message box is either 'Yes' or 'No'. The answer_yes_no keyword is not case sensitive, but the 'Yes' or 'No' response assigned by answer_yes_no to a variable, tag or cell assignment is strictly case sensitive: 'Yes' is capitalized and 'No' is capitalized, just as it appears in this text. This is important to remember when using either response in conditional statements.



In the above example, the query_yes_no keyword is entered without an argument on row 2, so the default message is displayed on the message box. In this example, the 'Yes' button was pressed, which closed the message box and displayed 'Yes' in the Reply column.



The illustration above depicts the query_yes_no keyword entered on row 2 with the text argument "Can you hear the tone?". When the keyword is executed, the message box displays the argument text. Pressing the 'No' button caused the message box to close and 'No' to be displayed in the Reply column.



The argument syntax for the query_yes_no keyword can take the form of entering the text directly, or a variable, tag or cell assignment holding the text value can be used. In the above example, a message is concatenated with a carriage return to form a multi-line message.

The rules for entering the argument to the query_yes_no keyword are identical to the rules used for entering the argument for the pause keyword; refer to the section of this manual describing the pause keyword and the section entitled 'Concatenation' for additional details.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	query_yes_no	"Can you hear the tone?"	Yes	
3	answer_yes_no	[myanswer]=reply		[MYANSWER] = Yes
4	answer_yes_no	<myanswer>=reply		<MYANSWER> = Yes
5	answer_yes_no	(examples_b1)=reply		Yes copied to sheet EXAMPLES cell B1
6				

Above, the use of answer_yes_no is depicted. In this case, the 'Yes' button of the query_yes_no message box was pressed. On row 3, the answer 'Yes' is assigned to the variable '[myanswer]'. On row 4, the answer 'Yes' is assigned to a tag called '<myanswer>'. On Row 5, 'Yes' is assigned to cell B1 on an embedded worksheet in the VIP Tool workbook called 'Examples'.

Applying the Result of Query_Yes_No to a Conditional Statement

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	query_yes_no	"Can you hear the tone?"	Yes	
3	answer_yes_no	[myanswer]=reply		[MYANSWER] = Yes
4	if	[myanswer] = "Yes"		If Yes = Yes = True
5	*idn?		Aeroflex, 88XX, 1000000009, 2.4.4-0, Jan 12 2021, 08:55:28	
6	elseif	[myanswer] = "No"		
7	*idn?			
8	endif			
9				

To make use of the response to a query_yes_no message, the answer is applied to a conditional statement. The response can be used in an IF or ELSEIF conditional statement, and the response can also be used as a condition for exiting a DO Loop by using it in the exitdo conditional statement.

In the example above, the response of the query_yes_no message is assigned by the answer_yes_no keyword to the variable '[myanswer]' on row 3. Rows 4 through 8 form a conditional IF block. If the answer to the query_yes_no message is 'Yes', then the '*IDN?' query on row 5 is sent to the instrument. If the answer to the query_yes_no message is 'No', then the '*IDN?' query on row 7 is sent to the instrument. In this example, the answer to the query_yes_no message was 'Yes', so the '*IDN?' query on row 5 was sent to the instrument.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	query_yes_no	"Can you hear the tone?"	No	
3	answer_yes_no	[myanswer]=reply		[MYANSWER] = No
4	if	[myanswer] = "Yes"		If No = Yes = False
5	*idn?			
6	elseif	[myanswer] = "No"		If No = No = True
7	*idn?		Aeroflex, 88XX, 1000000009, 2.4.4-0, Jan 12 2021, 08:55:28	
8	endif			
9				

The above illustration presents the same section of code used in the previous example. In this case, the answer to the query_yes_no message was 'No', so the '*IDN?' query on row 7 was sent to the instrument.

Choice Message with One Button and Cancel

Display_One_Button /Button1/Button1_Val

Command (*Argument*) Syntax:

Display_one_button (*one button message as text, variable, tag or cell assignment*) Displays a message box that contains button that is assigned a label in the script (referred to as button1), and a button labeled 'Cancel'. Pauses script execution until button1 or the 'Cancel' button is pressed.

Button1 (*button 1 label as text, variable, tag or cell assignment*) Assigns the value of its argument as the label for button1 on the display_one_button text box.

Button1_val (*variable, tag or cell assignment*) Assigns 'True' if button1 is selected or 'False' if the 'Cancel' button is selected to the variable, tag or cell assignment used as an argument for the button1_val keyword.

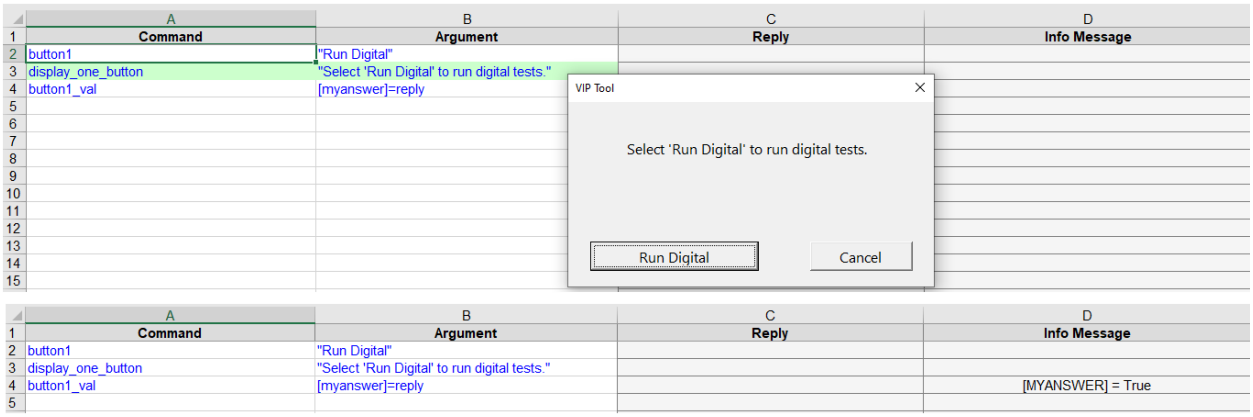
Display_one_button provides a method to assign a 'True' or 'False' text response as a means of allowing the operator to provide a decision when a script is running. The display_one_button message box provides the means to provide a message to the operator and the means to label one button with a text value determined by the programmer. This configurable button is referred to as button1. When button1 is pressed, the message box closes and the output of button1, expressed as button1_val, is set to the text value of 'True'. A 'Cancel' button is also present on the display_one_button message box. When the 'Cancel' button is pressed, the message box closes, and the output of button1 is set to the text value of 'False'.

The button1_val keyword is used to transfer the 'True' or 'False' output of the display_one_button message box to the variable, tag or cell assignment used as its argument.

The display_one_button message box is a modal message box. This means that script execution halts and does not continue until either button1 or the 'Cancel' button is pressed. When this message box is open, the Abort key cannot be used to stop execution of the script.

The text argument for display_one_button is displayed as a message on the display_one_button message box. The text argument for the button1 keyword is used to provide a label for button1. It is imperative that the button1 keyword assign a label to button1 before executing the display_one_button keyword to display the message box, otherwise button1 will appear without a label on the message box.

The display_one_button, button1, and button1_val keywords are not case sensitive. However, the 'True' or 'False' response assigned by button1_val to a variable, tag or cell assignment is strictly case sensitive: 'True' is capitalized and 'False' is capitalized, just as it appears in this text. This is important to remember when using either response in conditional statements.

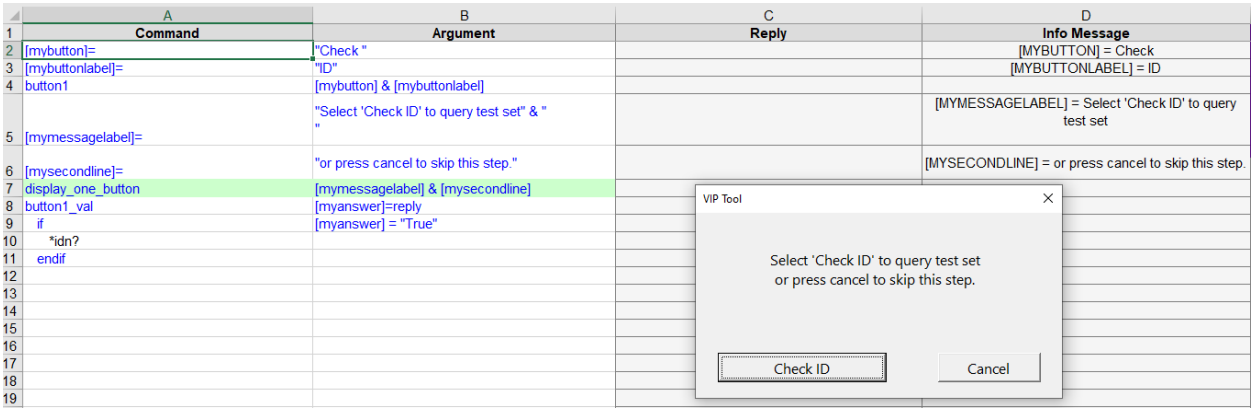


	A	B	C	D
1	Command	Argument	Reply	Info Message
2	button1	"Run Digital"		
3	display_one_button	"Select 'Run Digital' to run digital tests."		
4	button1_val	[myanswer]=reply		[MYANSWER] = True
5				

In the above illustration, the button1 keyword on row 2 is provided the text argument “Run Digital”, which labels button1 with that text. On row 3, the display_one_button keyword is provided the text argument “Select ‘Run Digital’ to run digital tests.”, which provides the message text for the display_one_button text box and opens the message box. In this example, button1 (labeled ‘Run Digital’) is pressed, so on row 4, the button1_val keyword assigns the text value of “True” to the variable ‘[myanswer]’.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	button1	"Run Digital"		
3	display_one_button	"Select 'Run Digital' to run digital tests."		
4	button1_val	[myanswer]=reply		[MYANSWER] = False
5				

In the above example, the same script exists. However, when this script was run, the ‘Cancel’ button was pressed. Therefore, the button1_val keyword assigns the text value of ‘False’ to the ‘[myanswer]’ variable.



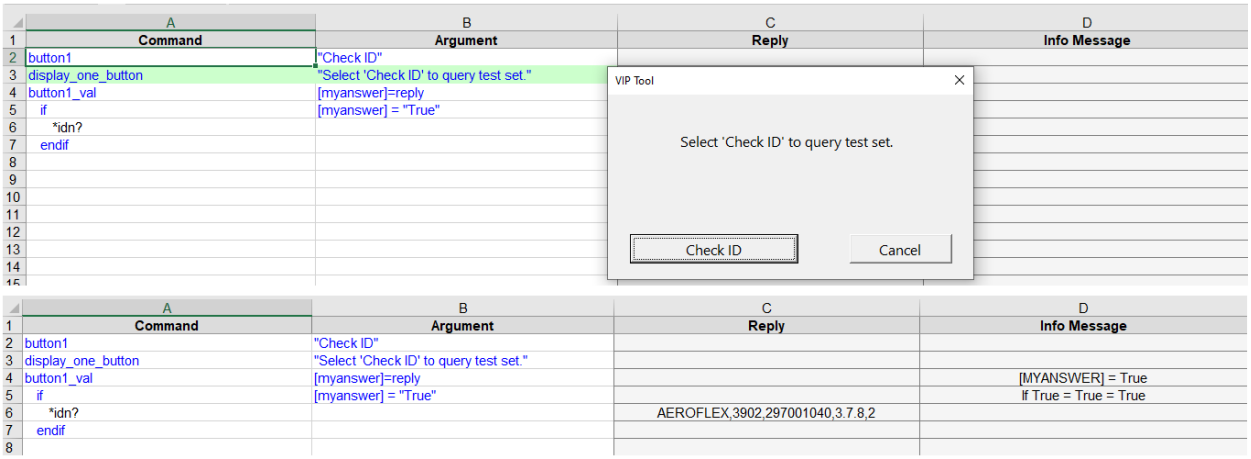
	A	B	C	D
1	Command	Argument	Reply	Info Message
2	[mybutton]=	"Check "		[MYBUTTON] = Check
3	[mybuttonlabel]=	"ID"		[MYBUTTONLABEL] = ID
4	button1	[mybutton] & [mybuttonlabel]		
5	[mymessagelabel]=	"Select 'Check ID' to query test set" & " "		[MYMESSAGELABEL] = Select 'Check ID' to query test set
6	[mysecondline]=	"or press cancel to skip this step."		[MYSECONDLINE] = or press cancel to skip this step.
7	display_one_button	[mymessagelabel] & [mysecondline]		
8	button1_val	[myanswer]=reply		
9	if	[myanswer] = "True"		
10	*idn?			
11	endif			

The argument syntax for both the button1 and display_one_button keywords can take the form of entering the text directly, or a variable, tag or cell assignment holding the text value can be used.

In the above example, on row 4 the button1 keyword uses an argument that applies concatenation in the argument column to generate the text ‘Check ID’ as the button1 label. On row 7, the display_one_button keyword also uses concatenation to generate the two line message “Select ‘Check ID’ to query test set or press cancel to skip this step.”

The rules for entering the argument for the button1 and display_one_button keywords are identical to the rules used for entering the argument for the pause keyword; refer to the section of this manual describing the pause keyword and the section entitled ‘Concatenation’ for additional details.

Applying the Result of Display_One_Button to a Conditional Statement



	A	B	C	D
1	Command	Argument	Reply	Info Message
2	button1	"Check ID"		
3	display_one_button	"Select 'Check ID' to query test set."		
4	button1_val	[myanswer]=reply		
5	if	[myanswer] = "True"		
6	*idn?			
7	endif			
8				

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	button1	"Check ID"		
3	display_one_button	"Select 'Check ID' to query test set."		
4	button1_val	[myanswer]=reply		[MYANSWER] = True
5	if	[myanswer] = "True"		If True = True = True
6	*idn?		AEROFLEX,3902,297001040,3.7.8,2	
7	endif			
8				

To make use of the response to the display_one_button message box, the answer is applied to a conditional statement. The response can be used in an IF or ELSEIF conditional statement, and the response can also be used as a condition for exiting a DO Loop by using it in the exitdo conditional statement.

In the example above, the response of the display_one_button message is assigned by the button1_val keyword to the variable '[myanswer]' on row 4. Rows 5 through 7 form a conditional IF block. If the answer to the display_one_button message is 'True', then the '*IDN?' query on row 6 is sent to the instrument. Otherwise, if the 'Cancel' button were pressed, the value passed to '[myanswer]' would be 'False', and no query would be sent to the instrument.

Choice Message with Two Buttons and Cancel

Display_Two_Button /Button1/Button2/Button1_Val/Button2_Val

Command (*Argument*) Syntax:

Display_two_button (*two button message as text, variable, tag or cell assignment*) Displays a message box that contains two button that are assigned labels in the script (referred to as button1 and button2), and a button labeled 'Cancel'. Pauses script execution until button1, button2, or the 'Cancel' button is pressed.

Button1 (*button 1 label as text, variable, tag or cell assignment*) Assigns the value of its argument as the label for button1 on the display_two_button text box.

Button2 (*button 2 label as text, variable, tag or cell assignment*) Assigns the value of its argument as the label for button2 on the display_two_button text box.

Button1_val (*variable, tag or cell assignment*) Assigns 'True' if button1 is selected, or 'False' if button2 or the 'Cancel' button is selected, to the variable, tag or cell assignment used as an argument for the button1_val keyword.

Button2_val (*variable, tag or cell assignment*) Assigns 'True' if button2 is selected, or 'False' if button1 or the 'Cancel' button is selected, to the variable, tag or cell assignment used as an argument for the button2_val keyword.

Button **Pressed**	Button1_val	Button2_val
Button1	True	False
Button2	False	True
Cancel	False	False

Display_two_button provides a method to assign a 'True' or 'False' text response as a means of allowing the operator to provide a decision when a script is running from a choice of two customizable buttons and a "Cancel" button. The display_two_button message box provides the means to provide a message to the operator and the means to label two buttons with text values determined by the programmer. These configurable buttons are referred to as button1 and button2. When button1 is pressed, the message box closes and the output of button1, expressed as button1_val, is set to the text value of 'True' and the output of button2_val is set to a text value of 'False'. When button2 is pressed, the message box closes and the output of button2, expressed as button2_val, is set to the text value of 'True' and the output of button1_val is set to a text value of 'False'. A 'Cancel' button is also present on the display_two_button message box. When the 'Cancel' button is pressed, the message box closes, and the outputs of both button1 and button2 are set to the text value of 'False'.

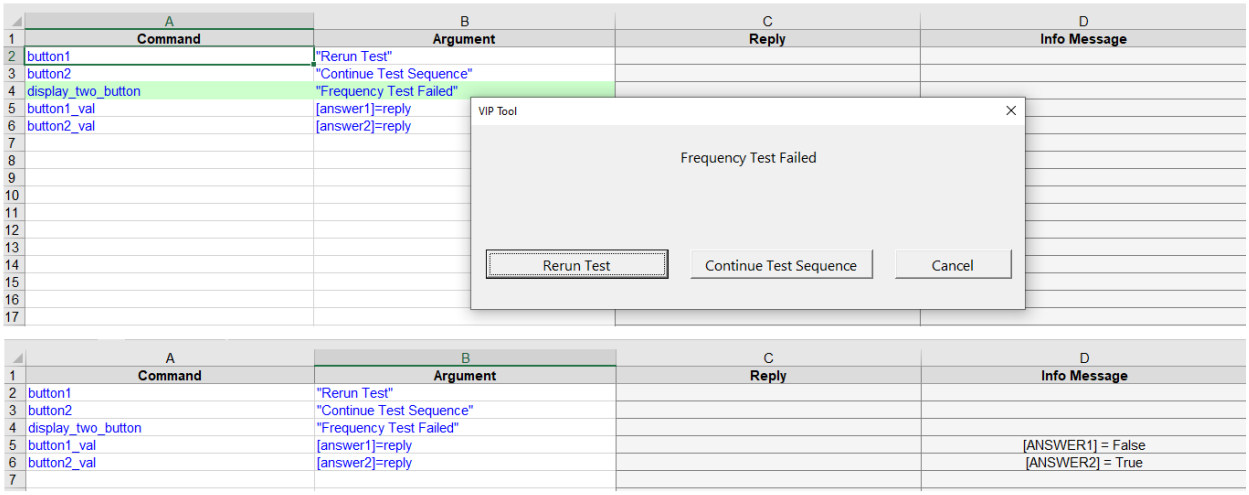
The button1_val keyword is used to transfer the 'True' or 'False' output of the display_two_button message box to the variable, tag or cell assignment used as its argument.

The button2_val keyword is used to transfer the 'True' or 'False' output of the display_two_button message box to the variable, tag or cell assignment used as its argument.

The display_two_button message box is a modal message box. This means that script execution halts and does not continue until either button1, button2, or the 'Cancel' button is pressed. When this message box is open, the Abort key cannot be used to stop execution of the script.

The text argument for display_two_button is displayed as a message on the display_two_button message box. The text argument for the button1 keyword is used to provide a label for button1. The text argument for the button2 keyword is used to provide a label for button2. It is imperative that the button1 and button2 keywords assign a label to button1 and button2 before executing the display_two_button keyword to display the message box, otherwise button1 and button2 will appear without labels on the message box.

The display_two_button, button1, button2, button1_val and button2_val keywords are not case sensitive. However, the 'True' or 'False' responses assigned by button1_val and button2_val to variables, tags or cell assignments is strictly case sensitive: 'True' is capitalized and 'False' is capitalized, just as it appears in this text. This is important to remember when using either response in conditional statements.



	A	B	C	D
1	Command	Argument	Reply	Info Message
2	button1	"Rerun Test"		
3	button2	"Continue Test Sequence"		
4	display_two_button	"Frequency Test Failed"		
5	button1_val	[answer1]=reply		[ANSWER1] = False
6	button2_val	[answer2]=reply		[ANSWER2] = True
7				

In the above illustration, the button1 keyword on row 2 is provided the text argument “Rerun Test”, which labels button1 with that text. On row 3, the button2 keyword is provided the text argument “Continue Test Sequence”, which labels button2 with that text. On row 4, the display_two_button keyword is provided the text argument “Frequency Test Failed”, which provides the message text for the display_two_button text box and opens the message box. In this example, button2 (labeled ‘Continue Test Sequence’) is pressed, so on row 6, the button2_val keyword assigns the text value of “True” to the variable ‘[answer2]’ while, on row 5, the button1_val keyword assigns the text value of “False” to the variable ‘[answer1]’.

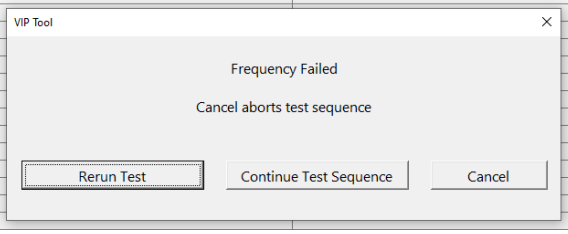
	A	B	C	D
1	Command	Argument	Reply	Info Message
2	button1	"Rerun Test"		
3	button2	"Continue Test Sequence"		
4	display_two_button	"Frequency Test Failed"		
5	button1_val	[answer1]=reply		[ANSWER1] = True
6	button2_val	[answer2]=reply		[ANSWER2] = False
7				

In the above example, the same script exists. However, when this script was run, button1 was pressed. Therefore, the button1_val keyword assigns the text value of ‘True’ to the ‘[answer1]’ variable and the button2_val keyword passes the text value of ‘False’ to the ‘[answer2]’ variable.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	button1	"Rerun Test"		
3	button2	"Continue Test Sequence"		
4	display_two_button	"Frequency Test Failed"		
5	button1_val	[answer1]=reply		[ANSWER1] = False
6	button2_val	[answer2]=reply		[ANSWER2] = False
7				

Using the same example, when the ‘Cancel’ button is pressed, both button1_val and button2_val pass the text value of ‘False’ to their respective variables.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mylabel1]=	"Rerun Test"		[MYLABEL1] = Rerun Test
3	[mylabel2]=	"Continue "		[MYLABEL2] = Continue
4	[mylabel3]=	"Test Sequence"		[MYLABEL3] = Test Sequence
5	button1	[mylabel1]		
6	button2	[mylabel2] & [mylabel3]		
7	[mymess]=	"Frequency "		[MYMESS] = Frequency
8	[mymess]=	[mymess] & "Failed"		[MYMESS] = Frequency Failed
9	[mymess]=	[mymess] & " "		[MYMESS] = Frequency Failed
10	display_two_button	[mymess] & "Cancel aborts test sequence"		
11	button1_val	[answer1]=reply		
12	button2_val	[answer2]=reply		
13	if	[Answer1] = "True"		
14	runsub	freq_test		
15	elseif	[answer2] = "True"		
16	runsub	report_freq		
17	else			
18	end			
19	endif			
20				
21				
22				



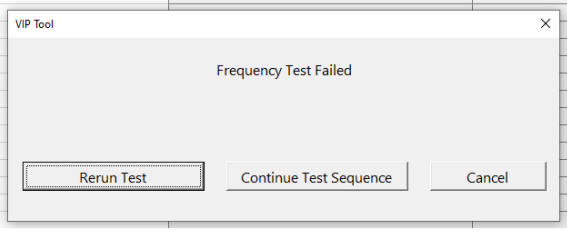
The argument syntax for the button1, button2 and display_two_button keywords can take the form of entering the text directly, or a variable, tag or cell assignment holding the text value can be used.

In the above example, on row 5 the button1 keyword uses the value held by the '[mylabel]' variable as its argument, so the label for button1 is 'Rerun Test', the value held by '[mylabel]'. On row 6 the button2 keyword uses an argument that applies concatenation in the argument column to generate the text 'Continue Test Sequence' as the button2 label. On row 10, the display_two_button keyword also uses concatenation to generate the two-line message "Frequency Failed. Cancel aborts test Sequence."

The rules for entering the argument for the button1, button2, and display_two_button keywords are identical to the rules used for entering the argument for the pause keyword; refer to the section of this manual describing the pause keyword and the section entitled 'Concatenation' for additional details.

Applying the Results of Display_Two_Button to a Conditional Statement

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	button1	"Rerun Test"		
3	button2	"Continue Test Sequence"		
4	display_two_button	"Frequency Test Failed"		
5	button1_val	[answer1]=reply		
6	button2_val	[answer2]=reply		
7	if	[Answer1] = "True"		
8	runsub	freq_test		
9	elseif	[answer2] = "True"		
10	runsub	report_freq		
11	else			
12	end			
13	endif			
14				
15				
16				
17				
18				
19				



	A	B	C	D
1	Command	Argument	Reply	Info Message
2	button1	"Rerun Test"		
3	button2	"Continue Test Sequence"		
4	display_two_button	"Frequency Test Failed"		
5	button1_val	[answer1]=reply		[ANSWER1] = False
6	button2_val	[answer2]=reply		[ANSWER2] = True
7	if	[Answer1] = "True"		If False = True = False
8	runsub	freq_test		
9	elseif	[answer2] = "True"		If True = True = True
10	runsub	report_freq		runsub report_freq
11	else			
12	end			
13	endif			
14				

To make use of the response to the display_two_button message box, the answers are applied to a conditional statement. The responses can be used in an IF or ELSEIF conditional statement, and the responses can also be used as conditions for exiting a DO Loop by using it in the exitdo conditional statement.

In the example above, the responses of the display_two_button message are assigned by the button1_val and button2_val keywords to the variables '[answer1]' and '[answer2]' on rows 5 and 6. Rows 7 through 13 form a conditional IF block. In the example, the answer to the display_two_button message was generated by button2 being pressed. Because of this, the value passed to the '[answer2]' variable on row 6 is 'True', so the ELSEIF statement on row 9 is true. This allows the subroutine 'report_freq' to be run, which is called from row 10.

If button1 were pressed, button1_val would pass the text value 'True' to the variable '[answer1]', which would cause the subroutine 'freq_test' to be called on row 8. If the 'Cancel' button were pressed, neither '[answer1]' or '[answer2]' would be assigned the value of 'True' and the script would stop at the end keyword on row 12.

Two Choice Check Box Message with Cancel

Display_Choice_Message /Choice1/Choice2/Choice1_Val/Choice2_Val

Command (Argument) Syntax:

Display_choice_message (*two button message as text, variable, tag or cell assignment*) Displays a message box that contains two selection boxes that are assigned labels in the script (referred to as choice1 and choice2), a button labeled 'OK' and a button labeled 'Cancel'. Pauses script execution until the 'OK' button or the 'Cancel' button is pressed.

Choice1 (*choice 1 label as text, variable, tag or cell assignment*) Assigns the value of its argument as the label for choice1 on the display_choice_message text box.

Choice2 (*choice 2 label as text, variable, tag or cell assignment*) Assigns the value of its argument as the label for choice2 on the display_choice_message text box.

Choice1_val (*variable, tag or cell assignment*) Assigns 'True' if choice1 is selected to the variable, tag or cell assignment used as an argument for the choice1_val keyword.

Choice2_val (*variable, tag or cell assignment*) Assigns 'True' if choice2 is selected to the variable, tag or cell assignment used as an argument for the choice2_val keyword.

Display_choice_message provides a method of assigning a 'True' or 'False' text value as a means of allowing the operator to select one, both or neither of two customizable choices presented on the message box. A button labeled 'OK' is provided to accept the choice selection and close the message box, and a button labeled 'Cancel' is provided to discard selections and close the message box.

Choice1_val is used to pass the value of the choice1 selection to a variable, tag or cell assignment. If the choice1 box was selected when the 'OK' button was pressed, choice1_val will pass the text value 'True'; if choice1 was not selected when the 'OK' button was pressed, choice1_val will pass the text value 'False'.

Choice2_val is used to pass the value of the choice2 selection to a variable, tag or cell assignment. If the choice2 box was selected when the 'OK' button was pressed, choice2_val will pass the text value 'True'; if choice2 was not selected when the 'OK' button was pressed, choice2_val will pass the text value 'False'.

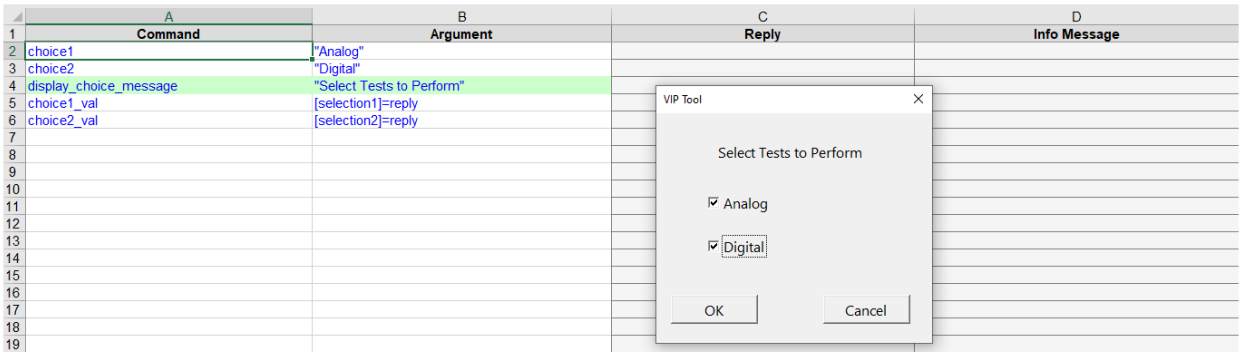
If the 'Cancel' key is pressed, choice1_val and choice2_val will pass the text value 'False', regardless of whether they are selected or not when the 'Cancel' button is pressed.

The display_choice_message message box is a modal message box. This means that script execution halts and does not continue until either the 'OK button or the 'Cancel' button is pressed. When this message box is open, the Abort key cannot be used to stop execution of the script.

The text argument for display_choice_message is displayed as a message on the message box. The text argument for the choice1 keyword is used to provide a label for choice1. The text argument for the choice2 keyword is used to provide a label for choice2. It is imperative that the choice1 and choice2 keywords assign a label to choice1 and choice 2 labels before executing the display_two_button keyword to display the message box, otherwise choice1 and choice2 will appear without labels on the message box.

The display_choice_message, choice1, choice2, choice1_val and choice2_val keywords are not case sensitive. However, the 'True' or 'False' responses assigned by choice1_val and choice2_val to variables, tags or cell

assignments is strictly case sensitive: ‘True’ is capitalized and ‘False’ is capitalized, just as it appears in this text. This is important to remember when using either response in conditional statements.



	A	B	C	D
1	Command	Argument	Reply	Info Message
2	choice1	"Analog"		
3	choice2	"Digital"		
4	display_choice_message	"Select Tests to Perform"		
5	choice1_val	[selection1]=reply		
6	choice2_val	[selection2]=reply		
7				

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	choice1	"Analog"		
3	choice2	"Digital"		
4	display_choice_message	"Select Tests to Perform"		
5	choice1_val	[selection1]=reply		[SELECTION1] = True
6	choice2_val	[selection2]=reply		[SELECTION2] = True
7				

In top portion of the above illustration, the choice1 keyword on row 2 is provided the text argument “Analog”, which labels choice1 with that text. On row 3, the choice2 keyword is provided the text argument “Digital”, which labels choice2 with that text. On row 4, the display_choice_message keyword is provided the text argument “Select Tests to Perform”, which provides the message text for the choice message and opens the message box. In this example, both choice1 and choice2 are selected and the ‘OK’ button is pressed to save the selection.

In the bottom portion of the example, the results of pressing the ‘OK’ button can be seen. On row 5, the choice1_val keyword assigns the text value of “True” to the variable ‘[selection1]’, and, on row 6, the choice2_val keyword assigns the text value of “True” to the variable ‘[selection2]’.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	choice1	"Analog"		
3	choice2	"Digital"		
4	display_choice_message	"Select Tests to Perform"		
5	choice1_val	[selection1]=reply		[SELECTION1] = True
6	choice2_val	[selection2]=reply		[SELECTION2] = True
7				

Both Choice1 and Choice2 selected. 'OK' button pressed

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	choice1	"Analog"		
3	choice2	"Digital"		
4	display_choice_message	"Select Tests to Perform"		
5	choice1_val	[selection1]=reply		[SELECTION1] = True
6	choice2_val	[selection2]=reply		[SELECTION2] = False
7				

Choice1 selected, Choice2 unselected. 'OK' button is pressed.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	choice1	"Analog"		
3	choice2	"Digital"		
4	display_choice_message	"Select Tests to Perform"		
5	choice1_val	[selection1]=reply		[SELECTION1] = False
6	choice2_val	[selection2]=reply		[SELECTION2] = True
7				

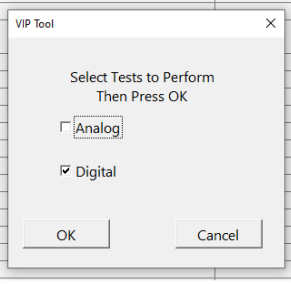
Choice1 unselected, Choice2 selected. 'OK' button is pressed.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	choice1	"Analog"		
3	choice2	"Digital"		
4	display_choice_message	"Select Tests to Perform"		
5	choice1_val	[selection1]=reply		[SELECTION1] = False
6	choice2_val	[selection2]=reply		[SELECTION2] = False
7				

If 'Cancel' button is pressed, Choice1_val and Choice2_val will be 'False', regardless of selection. If both are unselected, they will be false if the 'OK' button is pressed.

The display_choice_message function provides the facility to select more than one option for branching a test script. One, both or neither of the selections can be selected.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[choicemessage]=	"Select Tests to Perform"		[CHOICEMESSAGE] = Select Tests to Perform
3	choice1	<system1>		
4	choice2	<system2>		
5	display_choice_message	[choicemessage] & " Then Press OK"		
6	choice1_val	[selection1]=reply		
7	choice2_val	[selection2]=reply		
8	if	[selection1] = "True"		
9	runsub	run_analog		
10	endif			
11	if	[selection2] = "True"		
12	runsub	run_digital		
13	endif			
14				
15				
16				
17				
18				



C	D	E
	<system1>	
	<system2>	

C	D	E
	Analog	
	Digital	

The argument syntax for the choice1, choice2 and display_choice_message keywords can take the form of entering the text directly, or a variable, tag or cell assignment holding the text value can be used.

In the above example, on row 3 the choice keyword uses the value held by the '<system1>' tag as its argument, so the label for choice1 is 'Analog', the value held by '<system1>'. On row 4 the choice2 keyword uses the value held by the '<system2>' tag as its argument, so the label for choice2 is 'Digital', the value held by '<system2>'. On row 5, the display_choice_message keyword uses concatenation to generate the two-line message "Select Tests to Perform Then Press OK".

The rules for entering the argument for the button1, button2, and display_two_button keywords are identical to the rules used for entering the argument for the pause keyword; refer to the section of this manual describing the pause keyword and the section entitled ‘Concatenation’ for additional details.

Applying the Results of Display_Choice_Message to a Conditional Statement

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	choice1	"Analog"		
3	choice2	"Digital"		
4	display_choice_message	"Select Tests to Perform"		
5	choice1_val	[selection1]=reply		[SELECTION1] = True
6	choice2_val	[selection2]=reply		[SELECTION2] = True
7	if	[selection1] = "True"		If True = True = True
8	runsub	run_analog		runsub run_analog
9	endif			
10	if	[selection2] = "True"		If True = True = True
11	runsub	run_digital		runsub run_digital
12	endif			
13				

To make use of the response to the display_choice_message box, the answers are applied to conditional statements. The responses can be used in an IF or ELSEIF conditional statement, and the responses can also be used as conditions for exiting a DO Loop by using it in the exitdo conditional statement.

In the example above, the responses of the display_choice_message box selections are assigned by the choice1_val and choice2_val keywords to the variables ‘[selection1]’ and ‘[selection2]’ on rows 5 and 6. Rows 7 through 9 form a conditional IF block that will run the subroutine ‘run_analog’ if ‘[selection1]’ is assigned the value of ‘True’. Rows 10 through 12 form a conditional IF block that will run the subroutine ‘run_digital’ if ‘[selection2]’ is assigned the value of ‘True’.

Test Info Entry Form

Display_Test_Info /Uut_SN/Operator_Info

Command (Argument) Syntax:

Display_test_info (*no argument*) Displays a message box that contains a field labeled ‘SN’, a field labeled ‘Operator’, a button labeled ‘OK’ and a button labeled ‘Cancel’. Pauses script execution until the ‘OK’ or ‘Cancel’ button is pressed. Stores alphanumeric data entered in the ‘SN’ and ‘Operator’ fields if the ‘OK’ button is pressed. Does not store data entered in the fields if the ‘Cancel’ button is pressed.

UUT_sn (*variable, tag or cell assignment*) Assigns value entered in ‘SN’ field of the display_test_info message box to a variable, tag or cell assignment.

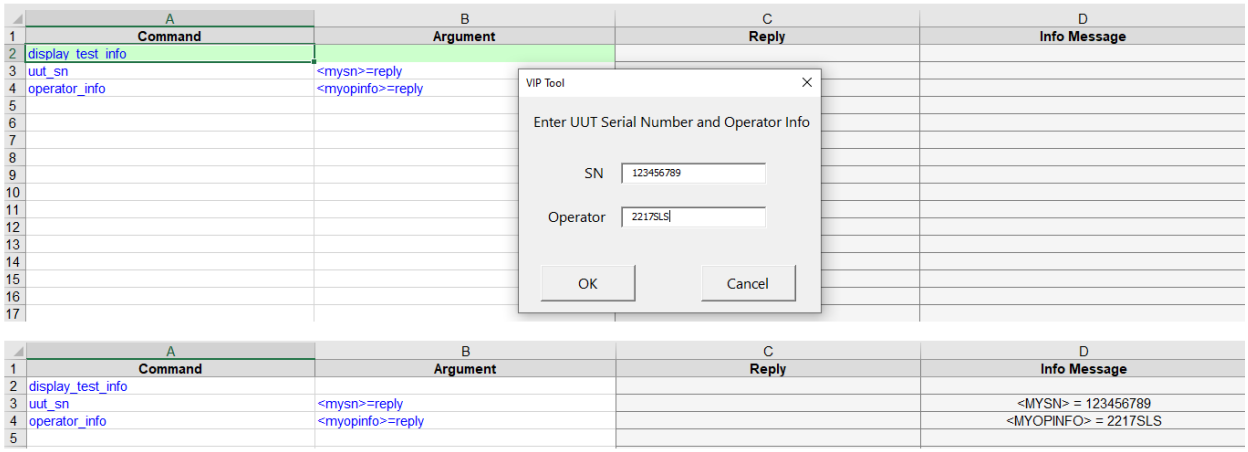
Operator_Info (*variable, tag or cell assignment*) Assigns value entered in ‘Operator’ field of the display_test_info message box to a variable, tag or cell assignment.

Display_test_info is a method for pausing execution of the script and allowing the operator to enter Unit Under Test (UUT) serial number and operator identification info into a script. When the message is displayed, the operator can select the ‘SN’ and ‘Operator’ fields and enter alphanumeric data into either or both fields. Pressing the ‘OK’ button on the message box causes the message box to close and store the serial number and operator information. Pressing the ‘Cancel’ button causes the message box to close and discard any information that either field may contain.

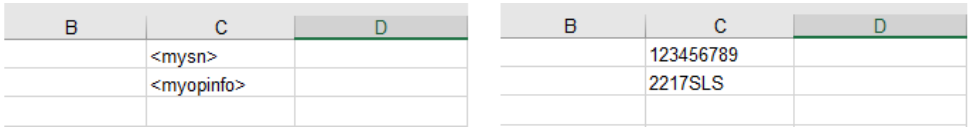
The display_test_info message box is a modal message box. This means that script execution halts and does not continue until either the ‘OK’ button or ‘Cancel’ button is pressed on the display_info message box. When this message box is open, the Abort key cannot be used to stop execution of the script.

UUT_sn and operator_info are keywords that are used to pass the data to the variable, tag or cell assignment that is used as the argument for the keyword. This data is useful for providing serial number and operator ID info on a report sheet.

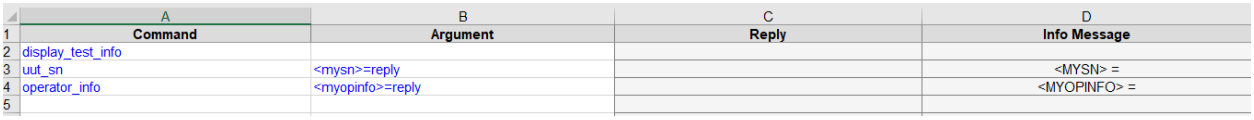
The display_test_info, UUT_sn, and operator_info keywords are not case sensitive.



In the above example, '123456789' is entered in the 'SN' field and '2217SLS' is entered in the 'Operator' field. When the 'OK' button is pressed, the message box closes. On row 3, the uut_sn keyword transfers the information from the 'SN' field to the tag '<mysn>'. On row 4, the operator_info keyword transfers the information entered in the 'Operator' field to the tag '<myopinfo>.



On a separate embedded report sheet, the values transferred to the '<mysn>' and '<myopinfo>' appear in cells that have those tags assigned to them.



In the above example, instead of pressing the 'OK' button, the 'Cancel' button was pressed. In that case, the values entered in the 'SN' and 'Operator' fields is discarded, and there is no data to transfer to the '<mysn>' and '<myopinfo>' tags.

Numerical Entry Form

Display_Numform /Numform_Val

Command (*Argument*) Syntax:

Display_numform (*number form message as text, variable, tag or cell assignment*) Displays a message box that contains a numeric field, a button labeled 'OK' and a button labeled 'Cancel'. Pauses script execution until the 'OK' or 'Cancel' button is pressed. Stores numeric data entered in the numeric field if the 'OK' button is pressed. Does not store data entered in the numeric field if the 'Cancel' button is pressed.

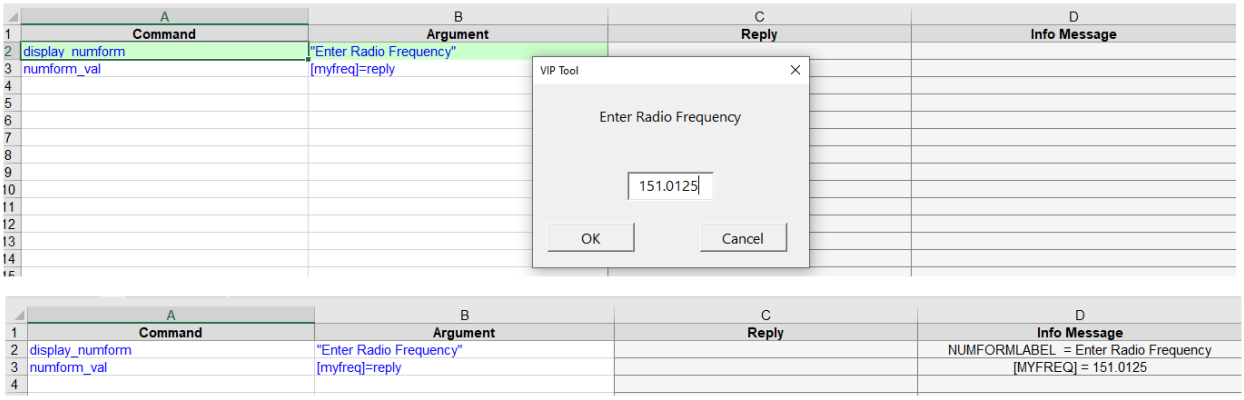
Numform_val (*variable, tag or cell assignment*) Assigns value entered in the numeric field of the display_numform message box to a variable, tag or cell assignment.

Display_numform is a method for pausing execution of the script and allowing the operator to enter a numeric value into a script. When the message is displayed, the operator can select the numeric field and enter numeric data into the field. Pressing the 'OK' button on the message box causes the message box to close and store the numeric value. Pressing the 'Cancel' button causes the message box to close and discard any information the numeric field may contain. If alphabetic data is entered in the numeric field, a message indicating only numeric data is allowed will be displayed.

The display_numform message box is a modal message box. This means that script execution halts and does not continue until the 'OK' button or 'Cancel' button is pressed on the display_numform message box. When this message box is open, the Abort key cannot be used to stop execution of the script.

Numform_val is a keyword that is used to pass the data to the variable, tag or cell assignment that is used as the argument for the keyword. This data is useful for entering numeric data into a script.

The display_numform and numform_val keywords are not case sensitive.

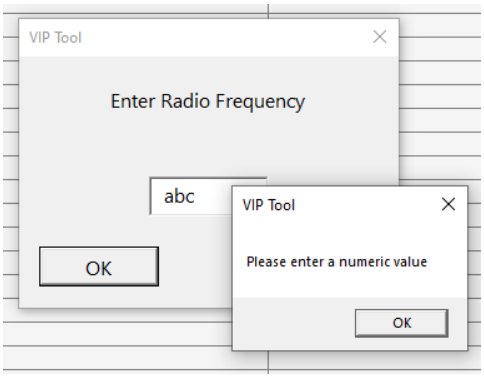


In the above example, the argument "Enter Radio Frequency" is applied to the display_numform keyword. This results in the argument appearing on the display_numform message box.

The value "151.0125" is entered in the numeric field of the display_numform message box, and the "OK" button is pressed. On row 3, the value entered in the numeric field of the display_numform message box is passed to the variable '[myfreq]' using the numform_val keyword.

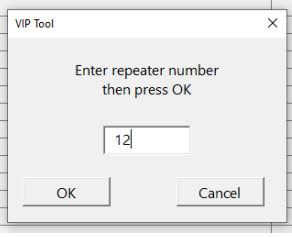
	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	display_numform	"Enter Radio Frequency"		NUMFORMLABEL = Enter Radio Frequency
3	numform_val	[myfreq]=reply		[MYFREQ] =
4				

The above illustration depicts the result when the 'Cancel' button is pressed. Any value entered in the numeric field of the display_numform message box is discarded, and no value is assigned to the variable '[myfreq]'.



In the above example, alphabetic data has been entered in the numeric field of the display_numform message box. In this case, the error message "Please enter a numeric value" is displayed. The operator must press the 'OK' button on the error message to proceed with running the script.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[mycr]=	" "		[MYCR] =
3	[repeater_message]=	"Enter repeater number" & [mycr]		[REPEATER_MESSAGE] = Enter repeater number
4	[repeater_message]=	[repeater_message] & "then press OK".		[REPEATER_MESSAGE] = Enter repeater number then press OK
5	display_numform	[repeater_message]		
6	numform_val	[my_repeater_number]=reply		
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				



The argument syntax for the display_numform keyword can take the form of entering the text directly, or a variable, tag or cell assignment holding the text value can be used. In the above example, a message is concatenated with a carriage return to form a multi-line message.

The rules for entering the argument to the display_numform keyword are identical to the rules used for entering the argument for the pause keyword; refer to the section of this manual describing the pause keyword and the section entitled 'Concatenation' for additional details.

Text Entry Form

Display_Textform /Textform_Val

Command (*Argument*) Syntax:

Display_textform (*text form message as text, variable, tag or cell assignment*) Displays a message box that contains an alphanumeric field, a button labeled 'OK' and a button labeled 'Cancel'. Pauses script execution until the 'OK' or 'Cancel' button is pressed. Stores alphanumeric data entered in the numeric field if the 'OK' button is pressed. Does not store data entered in the alphanumeric field if the 'Cancel' button is pressed.

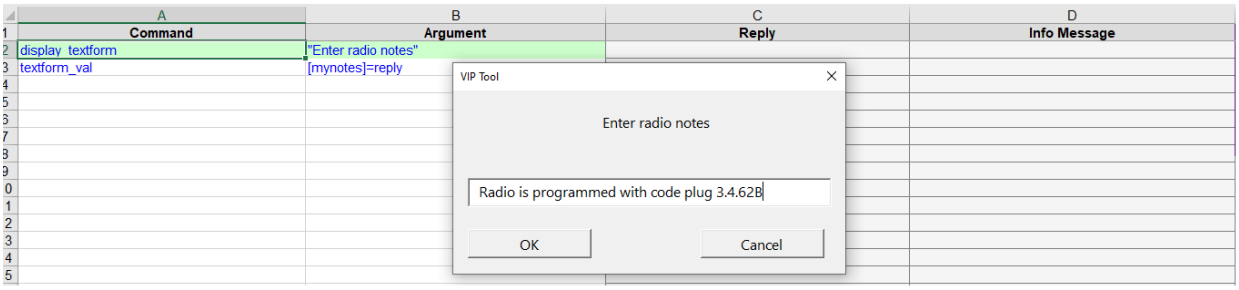
Textform_val (*variable, tag or cell assignment*) Assigns value entered in the alphanumeric field of the display_numform message box to a variable, tag or cell assignment.

Display_textform is a method for pausing execution of the script and allowing the operator to enter an alphanumeric value into a script. When the message is displayed, the operator can select the alphanumeric field and enter alphanumeric data into the field. Pressing the 'OK' button on the message box causes the message box to close and store the alphanumeric value. Pressing the 'Cancel' button causes the message box to close and discard any information the alphanumeric field may contain.

The display_textform message box is a modal message box. This means that script execution halts and does not continue until the 'OK' button or 'Cancel' button is pressed on the display_textform message box. When this message box is open, the Abort key cannot be used to stop execution of the script.

Textform_val is a keyword that is used to pass the data to the variable, tag or cell assignment that is used as the argument for the keyword. This data is useful for entering alphanumeric data into a script, particularly for entering data into a report when prompted.

The display_textform and textform_val keywords are not case sensitive.



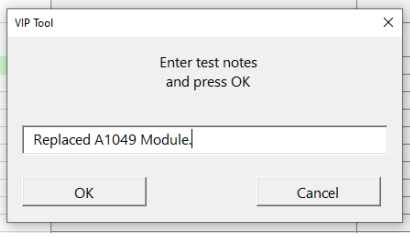
In the above example, the argument "Enter radio notes" is applied to the display_textform keyword. This results in the argument appearing on the display_textform message box.

The value "Radio is programmed with code plug 3.4.62B" is entered in the alphanumeric field of the display_textform message box, and the "OK" button is pressed. On row 3, the value entered in the alphanumeric field of the display_numform message box is passed to the variable '[mynotes]' using the textform_val keyword.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	display_textform	"Enter radio notes"		TEXTFORMLABEL = Enter radio notes
3	textform_val	[mynotes]=reply		[MYNOTES] =
4				

The above illustration depicts the result when the 'Cancel' button is pressed. Any value entered in the alphanumeric field of the display_textform message box is discarded, and no value is assigned to the variable '[mynotes]'.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[myprompt]=	"Enter test notes"		[MYPROMPT] = Enter test notes
3	[myprompt]=	[myprompt] & " "		[MYPROMPT] = Enter test notes
4	[myprompt]=	[myprompt] & "and press OK"		[MYPROMPT] = Enter test notes and press OK
5	display_textform	[myprompt]		
6	textform_val	<mynotes>=reply		
7				
8				
9				
10				
11				
12				
13				
14				



The argument syntax for the display_textform keyword can take the form of entering the text directly, or a variable, tag or cell assignment holding the text value can be used. In the above example, a message is concatenated with a carriage return to form a multi-line message.

The rules for entering the argument to the display_textform keyword are identical to the rules used for entering the argument for the pause keyword; refer to the section of this manual describing the pause keyword and the section entitled 'Concatenation' for additional details.

Message Positioning

Message_Horizontal /Message_Vertical/Center_Messages

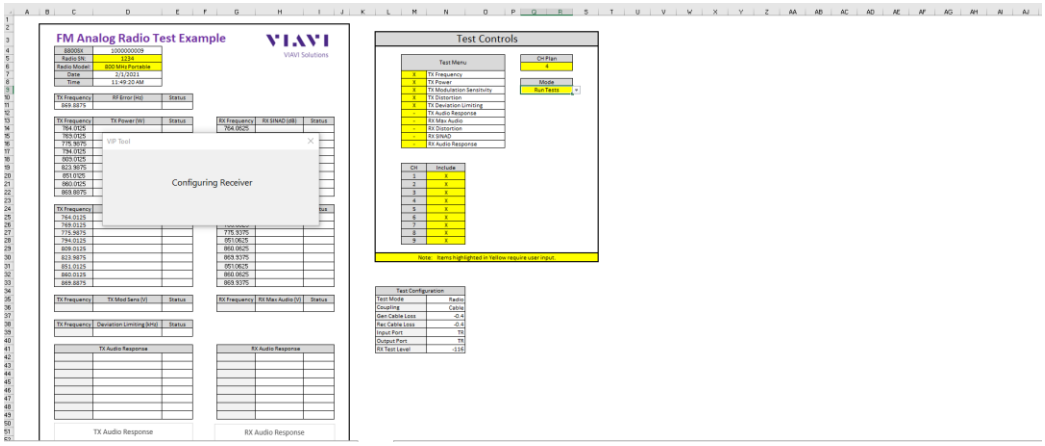
Command (*Argument*) Syntax

Message_horizontal (*Direct numerical entry 0.1 to 0.9*) Positions all messages boxes and forms (except for error messages) horizontally. The argument for message_horizontal only allows a direct entry of a numerical value between 0.1 and 0.9. The value of 0.5 will center the message boxes horizontally. Values less than 0.5 will position the message boxes progressively further to the left; values greater than 0.5 will position the message boxes progressively further to the right. The settings entered by this command will remain valid until reset by another message_horizontal message, or the keyword center_messages is executed. When a script begins running, the default value of message_horizontal is 0.5.

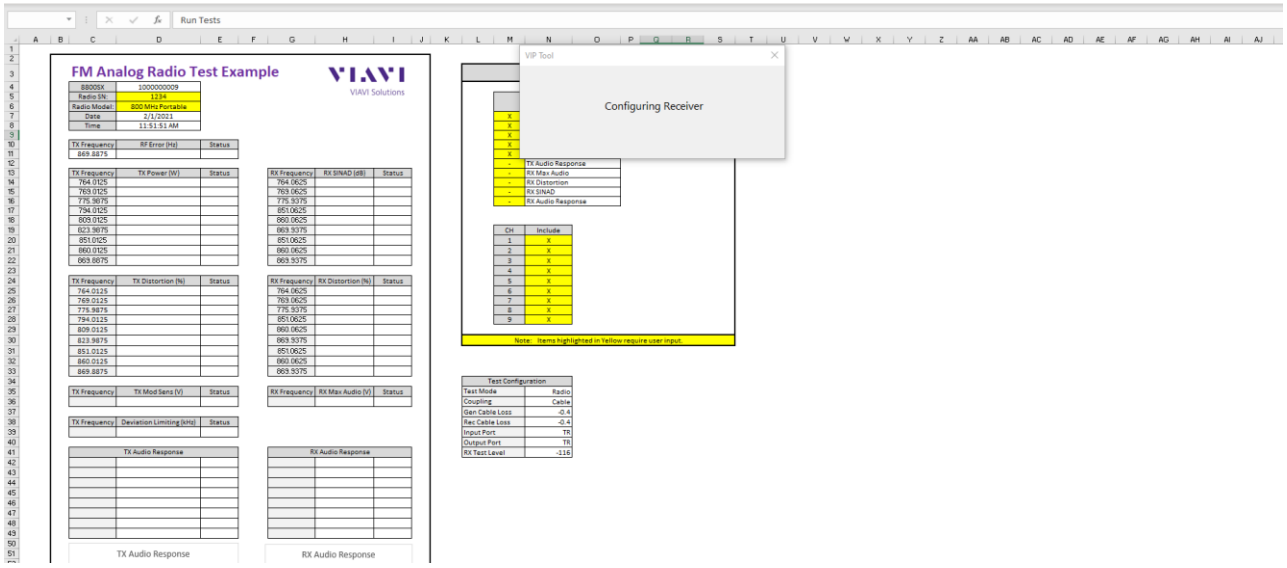
Message_vertical (*Direct numerical entry 0.1 to 0.9*) Positions all text messages and forms (except for error messages) vertically. The argument for message_vertical only allows a direct entry of a numerical value between 0.1 and 0.9. The value of 0.5 will center the message boxes vertically. Values less than 0.5 will position the message boxes progressively toward the to the top of the display; values greater than 0.5 will position the message boxes progressively toward the bottom of the display. The settings entered by this command will remain valid until reset by another message_vertical message, or the keyword center_messages is executed. When a script begins running, the default value of message_vertical is 0.5.

Center_messages (*No argument*) Resets the positions all messages boxes and forms to the center of the display.

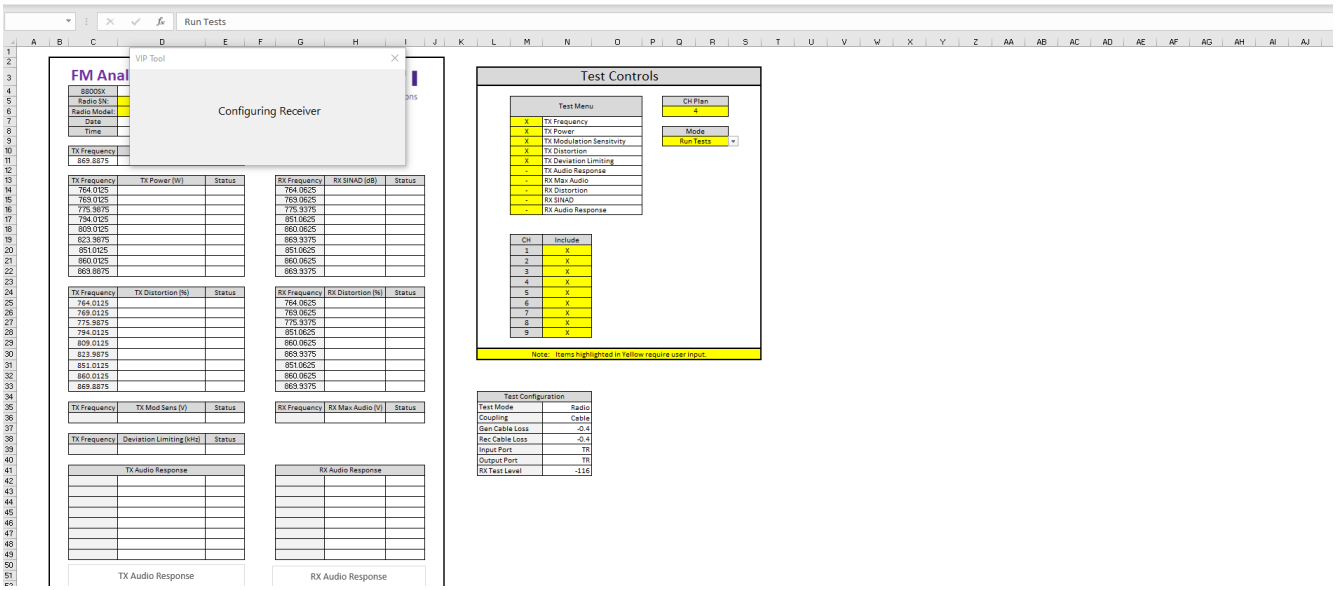
The Message_horizontal, message_vertical, and center_messages keywords are not case sensitive.



The above illustration depicts a message that is displayed after using message_horizontal with an argument value of 0.2.



The preceding illustration depicts a message that is displayed after using message_vertical with an argument value of 0.3.



The above illustration depicts a message that is displayed after using message_horizontal with an argument value of 0.2 and message_vertical with an argument value of 0.3.

Special Functions

Automated Find SINAD Function

Find_{TargSINADdB}_SINAD_{Lower Limit}_{Upper Limit}

Command_{targSINADdb}_sinad_{Lower Limit} _{Upper Limit} (*Argument*) Syntax:

Find_{targSINADdb}_sinad_{lower limit}_{upper limit} (*variable, tag or cell assignment*) Finds receiver SINAD value. Pauses execution of the script at its current step and displays the find_SINAD message. After displaying the message, find_sinad performs a find SINAD algorithm by changing the RF generator of the instrument and monitoring the instrument's SINAD meter. When the target SINAD value, as defined by value placed in the {targSINADdb} placeholder, falls between the lower limit and upper limit value, as defined by the {lower limit} and {upper limit} placeholders, the message is removed and the RF generator level is returned in dBm and microvolt values, separated by a comma. The return value can then be assigned to a variable, tag, or cell assignment placed in the argument column for the command. When the message box closes, the script moves to the next step.

The command includes three placeholders which are defined by {targSINADdb}, {lower limit} and {upper limit} in the description. The {targSINADdb} placeholder holds the target value for SINAD measurement, the {lower limit} placeholder holds the value for lower limit of the desired SINAD measurement, and the {upper limit} placeholder holds the value for upper limit of the desired SINAD measurement. The placeholder values can be entered as direct entry or assigned from a variable or tag. The argument specifies the destination of the value calculated by the keyword and can be a variable, tag, or cell assignment.

NOTE:	Any variable or tag assigned to a placeholder cannot have a space or an underline in its name.

Find_{targSINADdb}_sinad_{lower limit}_{upper limit} is not case sensitive.

Receiver SINAD, which is a measurement of receiver sensitivity, is specified as a value of SINAD in dB at a specified RF input level. For example, a receiver may be specified to have 12 dB SINAD of at -118 dBm, meaning that the receiver will measure 12 dB SINAD or better when the RF input level to the receiver is -118 dBm. Sometimes a receiver may be specified at 10 dB SINAD. And, sometimes, the SINAD level is defined in microVolts RF input instead of dBm.

The find_sinad function provides the flexibility of defining what the SINAD target is (for example 12 dB SINAD or 10 dB SINAD). The output of the find_sinad function allows the programmer to output the SINAD measurement in dBm or microVolts by defining the output of the function in CSV format as follows: ***dbm,microvolt***. If the SINAD specification is in dBm, the programmer selects the first field of the response as the reply; if the SINAD specification is in microVolts, the programmer selects the second field of the response as the reply.

The nature of a SINAD measurement is that it is taken at a very low RF input level, so the demodulated audio, which the instrument uses to evaluate the receiver sensitivity, is a constantly varying signal. Therefore, a SINAD measurement will 'move around' considerably over a limited range. The placeholder arguments {lower limit} and {upper limit} provide a window of measurement that accommodates this movement. If the instrument detects a SINAD measurement that falls between the lower limit and upper limit, the measurement is accepted, whereupon the find_sinad function reports the reading, then terminates. The find_sinad function uses averaging to prevent a momentary measurement from providing an inaccurate measurement. It is important to consider that the window provided by the lower limit and upper limit definitions not be made too narrow – the measurement may move around and never fall precisely within the window of measurement if the window is too narrow. The find_sinad function will terminate its algorithm and fail the measurement if the SINAD measurement cannot be accomplished in twenty steps of the algorithm or less. Typically, with proper setup of the window and assuming a good receiver being measured, the find_sinad will accomplish the measurement in six or less steps.

The following syntax example is used to find 12 dB SINAD within a measurement window of 2 dB:

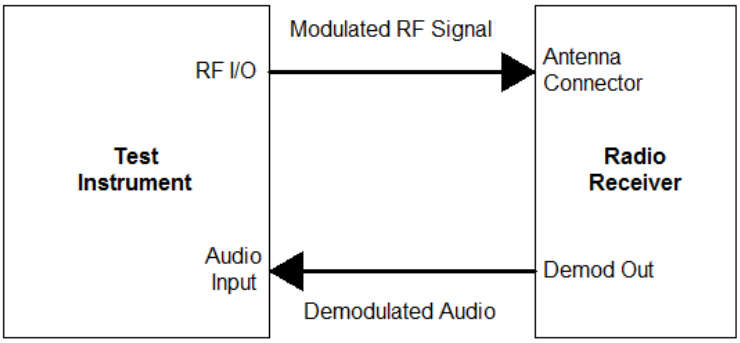
```
Find_12dB_SINAD_11_13
```

This syntax will cause the find_sinad algorithm to find a 12 dB SINAD reading that is between 11 and 12 dB, which is a window of 2 dB.

The next example is used to find 10 dB SINAD within a measurement window of 1 dB:

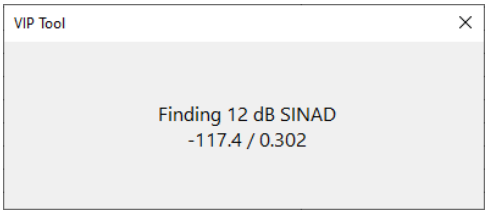
```
Find_10dB_SINAD_9.5_10.5
```

This syntax will cause the find_sinad algorithm to find a 10 dB SINAD reading that is between 9.5 and 10.5 dB, which is a window of 1 dB.



To run find_sinad, the RF I/O connector of the test instrument must be connected to the antenna connector of the radio receiver.

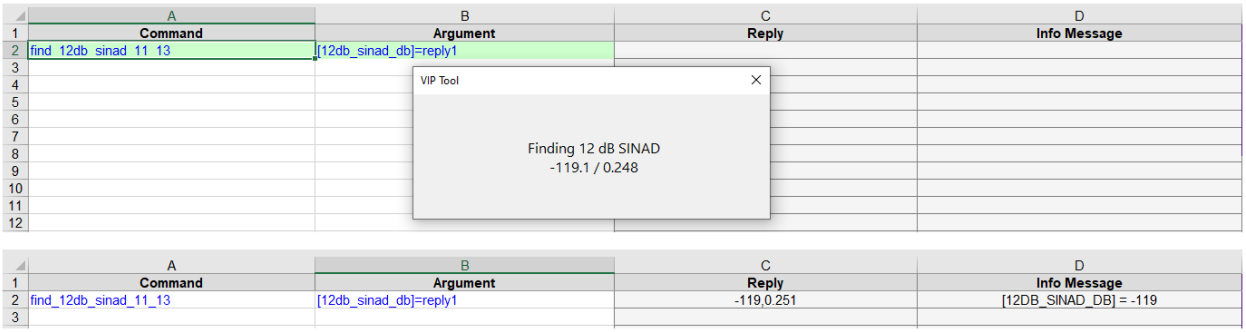
The demodulated audio output of the radio must be connected to the audio input connector of the instrument. Generally, this connection is from the speaker output of the radio, and the use of a breakout box may be required to gain access to this signal.



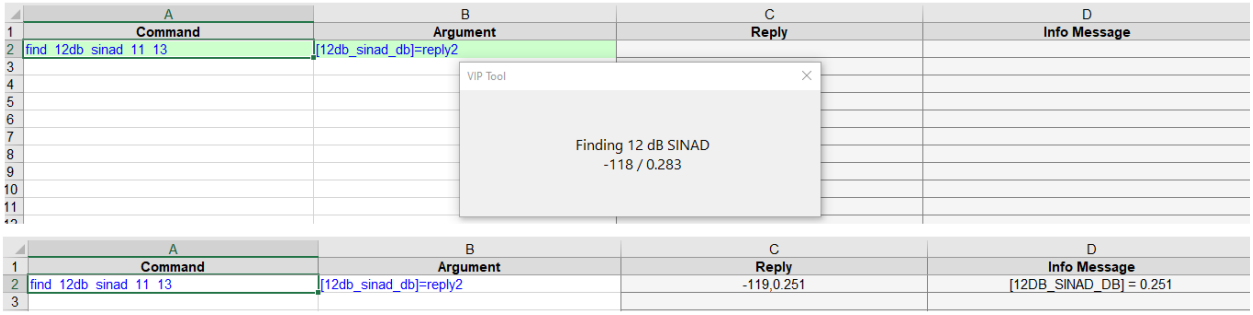
The find_sinad message box opens automatically when the keyword is executed. The first line will indicate the SINAD level the algorithm is to find, and the second line will update the SINAD reading of the current step of the algorithm in the format ***dbm/microvolt***.

Before find_sinad is used, the script must perform the following steps:

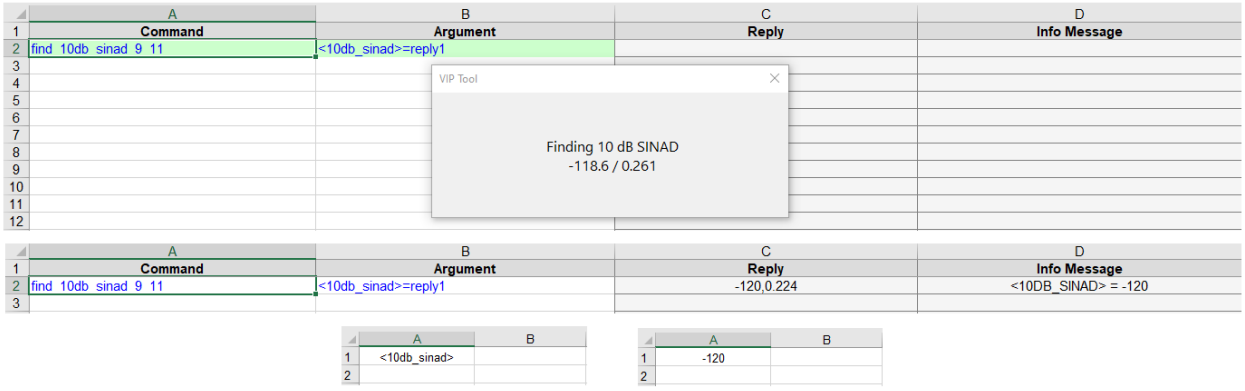
1. Set the instrument's RF generator frequency to the radio's receiver frequency.
2. Modulate the instrument's RF generator with the type of modulation required by the radio.
3. Set the instrument's modulation generator to the specified frequency (typically 1 kHz).
4. Set the instrument's SINAD meter to measure the frequency of the modulation.
5. Modulate the RF generator with any squelch tones required to un-squelch the radio receiver.
6. Modulate the instrument's RF generator at the modulation level required by the test.
7. Route the instrument's audio input to the instrument's audio level meter.
8. Enable the instrument's RF generator.



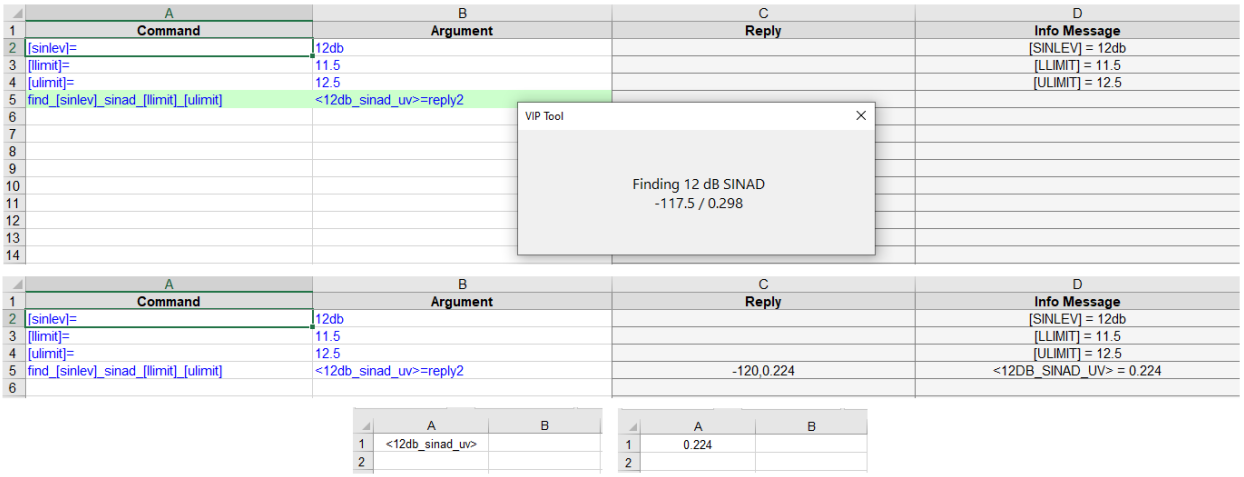
In the above illustration, find_12db_sinad_11_13 is used to find 12 dB SINAD between 11 dB and 13 dB SINAD. The argument assigns the dBm value of the RF generator to the variable '[12db_sinad_db]' by defining reply1 as the value to be assigned to the variable.



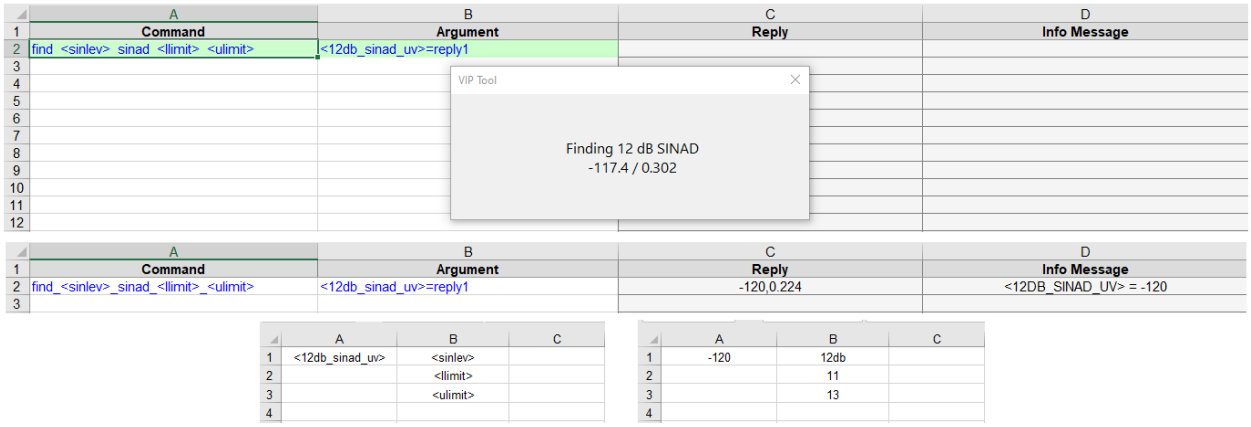
The example above uses the same command syntax as the previous example, but the argument differs in that it assigns the microvolt reading to the variable by defining reply2 to be assigned to the variable.



The example above uses the syntax find_10dB_sinad_9_11 to find the 10 dB SINAD value between 9 dB and 11 dB SINAD. This example assigns the dBm reading to the tag '<10dB_sinad>'. This tag's value appears on a report sheet cell to which the tag is assigned.



In the above example, variables are used fill the placeholders of the find_sinad keyword. The variable '[sinlev]' is assigned the value of "12dB" and is placed in the {targSINADdb} placeholder. The variable '[llimit]' is assigned the value of '11.5' and placed in the {lower limit} placeholder. The variable '[ulimit]' is assigned the value of '12.5' and placed in the {upper limit} placeholder. This syntax causes find_sinad to find the 12 dB SINAD level between 11.5 and 12.5 dB. This example assigns the microvolt reading to the tag '<12dB_sinad_uv>'. This tag's value appears on a report sheet cell to which the tag is assigned.



In the above example, the value of cells that have tags assigned to them are used to plug in the placeholder values in the find_sinad keyword. The tag '<sinlev>' is assigned to a cell that is holding the value '12db', and and is placed in the {targSINADdb} placeholder. The tag '<llimit>' is assigned to a cell that is holding the value '11', and is placed in the {lower limit} placeholder. The tag '<ulimit>' is assigned to a cell that is holding the value '13', and is placed in the {upper limit} placeholder. This syntax causes find_sinad to find the 12 dB SINAD level between 11 and 13 dB. This example assigns the dbm reading to the tag '<12dB_sinad_uv>'. This tag's value appears on a report sheet cell to which the tag is assigned.

Watts to dBm Calculator

Watt_To_dBm_{Watts}

Command_{Watts} (*Argument*) Syntax:

Watt_to_dbm_{Watts} (*variable, tag, or cell assignment*) Converts a value in units of watts to units of dBm. The command includes a placeholder, which is defined by {Watts} in the description, for the value to be converted. The placeholder value can be entered as direct entry, or assigned from a variable or tag. The argument specifies the destination of the value calculated by the keyword and can be a variable, tag, or cell assignment.

The Watt_to_dbm_{watts} keyword is provided to convert a value in Watts to a value in dBm without resorting to using a calculation on a separate sheet. This serves to promote portability of a script because the script contains the calculation and does not require linking to a sheet that may not exist in the current workbook.

The watt_to_dbm_{watts} keyword applies the following computation:

10 * Log10 ({watts} * 1000)

If the command is executed as watt_to_dbm_10, the following computation is resolved:

Watt_to_dbm_10 = 10 * Log10 (10 * 1000) = 40

This converts the value of 10 Watts, as entered in the {Watts} command placeholder, to 40 dBm.

The {Watts} place holder uses a variable or a tag to provide the Watts value to be converted.

NOTE:	The name of the variable or tag cannot be an array, nor can it have the underline '_' symbol as part of the name of the variable or tag. If the name of the variable or tag contains an underline, then the script will generate an error.

The Watt_to_dbm_{watts} keyword is not case sensitive.

	Command	Argument	Reply	Info Message
2	[powWatts]=	10		[POWWATTS] = 10
3	WATT_TO_DBM_[powWatts]	[mypow_dbm]=reply	40	[MYPOW_DBM] = 40
4	watt_to_dbm_<watts>	[mypow_dbm]=reply	36.98970004	[MYPOW_DBM] = 36.9897000433602
5	Watt_to_dBm_.001	[mypow_dbm]=reply	0	[MYPOW_DBM] = 0
6	Watt_to_dBm_1		30	
7				

	A	B
1		<watts>
2		

	A	B
1		5
2		

In the above example, on row 2 the variable '[powWatts]' is assigned the value of 10. On row 2, the value held by '[powWatts]' is converted to 40 dBm and assigned to the variable '[mypow_dbm]. On row 4, the cell with the tag '<watts>' assigned to it holds a value of '5', and that value is converted to 36.99 dBm. On row 5, the value of '.001' Watts, or 1 milliWatt, is included as part of the command in the form of watt_to_dbm_.001. 1 milliWatt is converted to 0 dBm. On row 6, 1 watt is converted to dBm the same way with the syntax watt_to_dbm_1. 1 Watt is converted to 30 dBm.

NOTE:	Note that conversion is still possible, and the converted value appears in the Reply column if no argument is supplied to the keyword.

dBm to Watts Calculator

dBm_To_Watts_{dBm}

Command_{dBm} (*Argument*) Syntax:

dBm_to_watts_{dBm} (*variable, tag, or cell assignment*) Converts a value in units of dBm to units of Watts. The command includes a placeholder, which is defined by {dBm} in the description, for the value to be converted. The placeholder value can be entered as direct entry or assigned from a variable or tag. The argument specifies the destination of the value calculated by the keyword and can be a variable, tag, or cell assignment.

The dbm_to_watt_{dbm} keyword is provided to convert a value in dBm to a value in Watts without resorting to using a calculation on a separate sheet. This serves to promote portability of a script because the script contains the calculation and does not require linking to a sheet that may not exist in the current workbook.

The dbm_to_watt_{dbm} keyword applies the following computation:

$10^{(\{dBm\}/10)} / 1000$

If the command is executed as dbm_to_watt_40, the following computation is resolved:

dBm_to_watt_40 = 10({40}/10) / 1000 = 10

This converts the value of 40 dBm, as entered in the {dbm} command placeholder, to 10 Watts.

The {dbm} place holder uses a variable or a tag to provide the dBm value to be converted.

NOTE:	The name of the variable or tag cannot be an array, nor can it have the underline '_' symbol as part of the name of the variable or tag. If the name of the variable or tag contains an underline, then the script will generate an error.

The dBm_to_watt_{dbm} keyword is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[powdBm]=	40		[POWDBM] = 40
3	DBM_TO_WATT_[powdBm]	[mypow_Watts]=reply	10	[MYPOW_WATTS] = 10
4	dbm_to_watt_<dbm>	[mypow_Watts]=reply	5.011872336	[MYPOW_WATTS] = 5.01187233627273
5	dBm_to_Watt_0	[mypow_Watts]=reply	0.001	[MYPOW_WATTS] = 0.001
6	dBm_to_Watt_30		1	
7				

	A	B
1		<dBm>
2		

	A	B
1		37
2		

In the above example, on row 2 the variable '[powdBm]' is assigned the value of 40. On row 2, the value held by '[powdBm]' is converted to 10 Watts and assigned to the variable '[mypow_Watts]. On row 4, the cell with the tag '<dbm>' assigned to it holds a value of '37', and that value is converted to 5.01 Watts. On row 5, the value of '0' dBm is included as part of the command in the form of dBm_to_Watts_0. 0 dBm is converted to .001 Watts, or one milliWatt. On row 6, 30 dB, is converted to Watts the same way with the syntax dBm_to_Watts_30. 30 dBm is converted to 1 Watt.

NOTE:	Note that conversion is still possible, and the converted value appears in the Reply column if no argument is supplied to the keyword. It should be noted that negative numbers can be used in the placeholder, either directly or using a variable or tag. For example, dbm_to_watts_-3 will convert -3 dBm to 0.5 milliwatts.

Audio Gain/Loss In dB Calculator

Db_Calc_{V1}_{V2}

Command_{V1} _{V2} (*Argument*) Syntax:

dB_calc_{v1}_{v2} (*variable, tag, or cell assignment*) Calculates gain or loss of voltage in dB. The command includes two placeholders, which are defined by {v1} and {v2} in the description. The {V1} placeholder holds the value for the measured voltage and the {V2} placeholder holds the value for the reference voltage. The placeholder values can be entered as direct entry or assigned from a variable or tag. The argument specifies the destination of the value calculated by the keyword and can be a variable, tag, or cell assignment.

The db_calc_{v1}_{v2} keyword is provided to convert the gain or loss of a voltage measurement in dB without resorting to using a calculation on a separate sheet. This serves to promote portability of a script because the script contains the calculation and does not require linking to a sheet that may not exist in the current workbook.

The db_calc_{v1}_{v2} keyword applies the following computation:

20 log (V1/V2)

Where V2 is the reference voltage and V1 is the measured voltage.

If the command is executed as db_calc_1_2, the following computation is resolved:

db_calc_1_2 = 20 log (1/2) = -6 dB

V2, the reference voltage, is greater than V1, the measured voltage, which results in a loss, expressed as -6 dB.

The {V1} and {V2} placeholders use a variable or a tag to provide the voltage values.

NOTE:	The name of the variable or tag cannot be an array, nor can it have the underline '_' symbol as part of the name of the variable or tag. If the name of the variable or tag contains an underline, then the script will generate an error.

The db_calc_{v1}_{v2} keyword is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[V1]=	6		[V1] = 6
3	[V2]=	3		[V2] = 3
4	db_calc_[V1]_[V2]	[myafdb]=reply	6.020599913	[MYAFDB] = 6.02059991327962
5	db_calc_3_6	[myafdb]=reply	-6.020599913	[MYAFDB] = -6.02059991327962
6	db_calc_<measured>_<reference>	[myafdb]=reply	6.020599913	[MYAFDB] = 6.02059991327962
7	db_calc_5.5_6		-0.755771218	
8				

	A	B	C
1		<measured>	
2		<reference>	
3			

	A	B	C
1		2	
2		1	
3			

In the above example, on row 2 the variable '[V1]' is assigned the value of '6'. On row 3 the variable '[V2]' is assigned the value of 3. Because the measured voltage is greater than the reference voltage, the calculated dB value will be a positive number, which indicates that the dB value represents gain. On row 4, the dB value is calculated using db_calc. The result is a gain of 6 dB, which represents a doubling of voltage. This value is assigned to the variable '[myafdb]'.

On row 5, the values for the two voltages are entered directly. V1, the measured voltage, is '3' and V2, the reference voltage, is '6'. 3V is half the value of 6V, so the result is -6 6dB, which indicates a loss. B

On row 6, tags are used to represent the two voltages in the db_calc command. The tag '<measured>' holds the value of '2' and is applied as V1 in the db_calc syntax. The tag '<reference>' holds the value the value of '1' dBm is applied as V2 in the db_calc syntax. "2" is twice the value of "1" so the calculated dB value is '6'.

On row 7, the measured value is '5.5' and the reference value is '6'. '5.5' is less than 6, so the output is a negative value, indicating a loss. In this case, the calculated value is -.076 dB.

NOTE:	Conversion is still possible, and the converted value appears in the Reply column if no argument is supplied to the keyword.

Select A Cell for Viewing Function

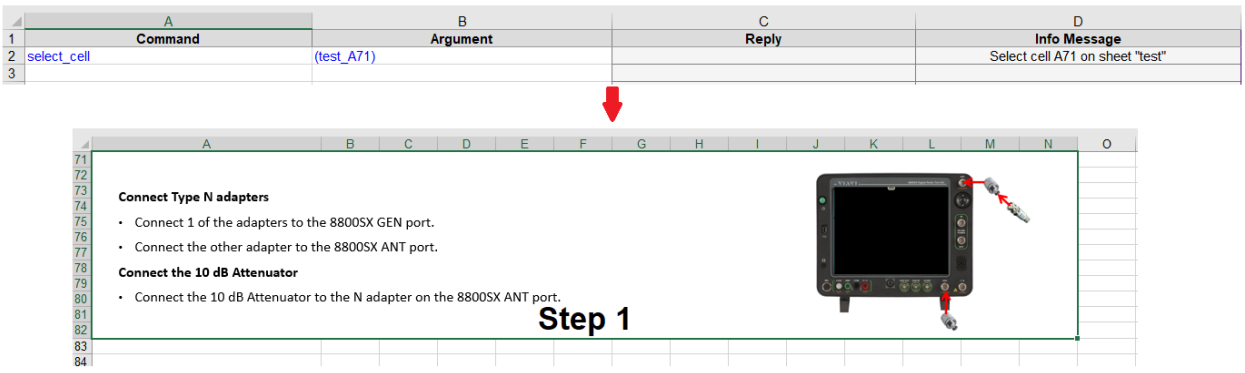
Select_Cell

Command (*Argument*) Syntax:

Select_Cell (*cell assignment*) Selects the sheet and cell defined in its cell assignment argument and places the row of the cell at the top of its worksheet. On the "Script" sheet, this command has no effect.

Select_cell is used as a tool for bringing a cell on a specific sheet into view. The sheet and cell to be viewed are defined in its argument in the form of a cell assignment. If select_cell is executed while viewing the "Script" sheet, the command will have no effect. However, if any sheet other than the "Script" sheet is being viewed when select_cell is executed, the sheet defined in the argument for select_cell will be accessed, and the row of the cell defined in the argument will be placed at the top of the sheet.

The select_cell keyword is not case sensitive.



In the above example, the argument for select_cell is the cell assignment "(test_A71)". If the "Script" sheet is not being viewed at the time select_cell is executed, the embedded sheet "Test" is brought into view and row 71 is

placed at the top of the display. In this example, cell A71 of the “Test” sheet, containing hook-up instructions, is brought into view. Select_cell is useful when combined with a message box that will pause execution of the script. The selected cell can be viewed for as long as necessary, then a message box button can be used to resume operation of the script.

Transfer Trace Data to Another Sheet

Transfer_Trace

Command (*Argument*) Syntax:

Transfer_trace (*row number of trace x data on trace results* **to** *simple cell assignment*) Transfers trace data from the Trace Data sheet to another sheet within the VIP Tool workbook. The argument specifies the row number of the X data to be transferred from the Trace Data sheet, the keyword to and a simple cell assignment of the starting cell the data will be transferred to. The function will transfer the X data to the sheet defined in the cell assignment, starting at the designated cell, and filling in the data in each cell to the right. It then will copy the Y data starting at the cell below the target cell, filling in the data in each cell to the right of that cell. Note that the argument cannot substitute values with variables or tags; the row number of the X data must be entered explicitly, and the cell assignment cannot be an array.

An example of transfer_trace syntax is as follows: transfer_trace 1 to (report_L1). This syntax will transfer the trace data stored on rows 1 and rows 2 of the trace data sheet to rows 1 and 2 of the report sheet, starting in the ‘L’ column. Always designate the row of the trace data sheet containing X data, which is always the first row of the two row XY data.

Transfer_trace is a utility that allows trace data to be copied to another sheet, such as a report sheet. This allows data to be contained on one sheet, and there are no broken links if the sheet is exported and imported into another VIP Tool workbook. Transfer_trace is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	:ca:trace:live?		560 Data Pairs Received	
3	transfer_trace	1 to (report_L1)		Trace data on Trace Sheet rows 1 and 2 transferred to REPORT sheet starting at L1.
4				

	A	B	C	D	E	F	G	H	I	J
1	2/7/2021	15:49:40	560	X Data (Frequency)		1.49E+08	149003577.8	149007155.6	149010733.5	1490143
2	2/7/2021	15:49:40	:ca:trace:live?	Y Data (Amplitude)		-57	-53.6	-55.4	-57	-5
3										

Trace Data Sheet

	K	L	M	N	O	P	Q	R	S	T	U	V
		149000000	149003577.8	149007155.6	149010733.5	149014311.3	149017889.1	149021466.9	149025044.7	149028622.5	149032200.4	149035778
		-57	-53.6	-55.4	-57	-56.2	-55.7	-56.3	-53.2	-56.7	-56.3	-54

Report Sheet

In the above example, transfer_trace 1 to (report_L1) copies the XY trace data to a sheet within the VIP Tool workbook named Report. The copied X data begins on row 1, column L of the Report sheet. The copied Y data begins on row 2, column L of the report sheet.

Clear Range of Cells in a Row

Clear_Row

Command (*Argument*) Syntax:

Clear_row (*number of cells to clear* **from** *simple cell assignment*) Clear data from a range of cells on a single on a sheet within the VIP Tool workbook. The argument specifies the number of cells to clear from the designated row on the target sheet, the keyword from and a simple cell assignment of the starting cell the data erased from. The function will clear data from the sheet defined in the cell assignment, starting at the designated cell, and clearing the data in each cell to the right for the number of cells specified. Note that the argument cannot substitute values with variables or tags; the row number of the X data must be entered explicitly and the cell assignment cannot be an array.

An example of clear_row syntax is as follows: clear_row 560 from (report_L1). This syntax will clear the the data stored in 560 cells on row 1 the report sheet, starting in the 'L' column.

Clear_row is a utility that allows data to be cleared from cells on another sheet, such as a report sheet. It can be used in conjunction with Transfer_trace to clear previous trace data stored on a sheet when a new script is executed. Clear_row is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	clear_row	560 from (report_L2)		560 cells cleared from REPORT sheet starting at L2.
3				

	K	L	M	N	O	P	Q	R	S	T	U	V
		149000000	149003577.8	149007155.6	149010733.5	149014311.3	149017889.1	149021466.9	149025044.7	149028622.5	149032200.4	149035778
		-57	-53.6	-55.4	-57	-56.2	-55.7	-56.3	-53.2	-56.7	-56.3	-54.

	K	L	M	N	O	P	Q	R	S	T	U	V
1		149000000	149003577.8	149007155.6	149010733.5	149014311.3	149017889.1	149021466.9	149025044.7	149028622.5	149032200.4	14903577
2												
3												

Report Sheet

In the above example, clear_row 560 from (report_L2) removes the data from 560 cells in row 2 of the sheet named Report, starting at cell L2 of the Report sheet.

Beep Function

Beep

Command (*Argument*) Syntax:

beep (no argument) Causes Windows tone to play through computer speakers.

When encountered by the script, beep will cause a 'Windows' tone to be played through the computer's speaker. The keyword beep is not case sensitive.

Time Function

Time

Command (*Argument*) Syntax:

Time (*optional variable, tag, or cell assignment*) If the argument is empty, prints the current time to the Reply column. If a variable, tag, or cell assignment is put into the argument to receive data, assigns the current time to the variable, tag, or cell assignment.

Time is a utility keyword that is used if the current time is required to be recorded in a report.

Time is placed in the Command column and a variable, tag, or cell assignment is placed in the Argument column so that the time keyword can pass the current time value to the variable, tag, or cell assignment in 24 hour hours:minutes:seconds format. The keyword time is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	time		16:03:54	
3	time	[mytime]=reply		[MYTIME] = 16:3:54
4	time	<mytime>=reply		<MYTIME> = 16:3:54
5	time	(examples_c1)=reply		16:3:54 copied to sheet EXAMPLES cell C1
6				

	A	B	C	D
1		<mytime>		
2				

	A	B	C	D
1		16:03:54	16:03:54	
2				

Date Function

Date

Command (*Argument*) Syntax:

Date (*optional variable, tag, or cell assignment*) If the argument is empty, prints the current date to the Reply column. If a variable, tag, or cell assignment is put into the argument to receive data, assigns the current date to the variable, tag, or cell assignment.

Date is a utility keyword that is used if the current date is required to be recorded in a report.

Date is placed in the Command column and a variable, tag, or cell assignment is placed in the Argument column so that the time keyword can pass the current time value to the variable, tag, or cell assignment in mm/dd/yyyy format. The keyword date is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	date		1/18/2021	
3	date	[mydate]=reply		[MYDATE] = 1/18/2021
4	date	<mydate>=reply		<MYDATE> = 1/18/2021
5	date	(examples_c1)=reply		1/18/2021 copied to sheet EXAMPLES cell C1
6				

	A	B	C	D
1		<mydate>		
2				

	A	B	C	D
1		1/18/2021	1/18/2021	
2				

Close and Open Socket Functions

Close_Socket/Open_Socket

Command (*Argument*) Syntax:

Close_socket (*No argument*) Closes the socket connection to the instrument.

Open_socket (*No argument*) Opens the socket connection to the instrument.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	Sub	Begin		
3	"idn?	[TestUnit]=Reply3	AEROFLEX,3920,1000595557,3.7.8,2	[TESTUNIT] = 1000595557
4	close_socket			
5	# Store Current Test Selections and Radio SN			
6	For	1 to 7		For 1 to 7
7	[Temp_NextCount]=	<TS_NextCount>		[TEMP_7] = -
8	Next			
9	[RadioSN]=	<RadioSN>		[RADIOSN] = 12345
10	For	1 to 9		For 1 to 9
11	[CHSel_NextCount]=	<CHSEL_NextCount>		[CHSEL_9] = -
12	Next			
13	[TempCode]=	<Code>		[TEMPCODE] = ---
14	# Clear all Tag Values to clear the Report Sheet			
15	Clear_tags			
16	# Restore the saved Test Selections and Radio SN			
17	For	1 to 7		For 1 to 7
18	<TS_NextCount>=	[Temp_NextCount]		<TS_7> = -
19	Next			
20	Start_Timer_1	<StartTime>=Reply	9:57:02 AM	<STARTTIME> = 9:57:02 AM
21	<RadioSN>=	[RadioSN]		<RADIOSN> = 12345
22	<TestUnit>=	[TestUnit]		<TESTUNIT> = 1000595557
23	<Code>=	[TempCode]		<CODE> = ---
24	For	1 to 9		For 1 to 9
25	<CHSEL_NextCount>=	[CHSel_NextCount]		<CHSEL_9> = -
26	Next			
27	# Find the selected Band and Read the Frequencies and specs from the Test Specs Sheet			
28	RunSub	Read_Band_Frequencies		runsub Read_Band_Frequencies
29	open_socket			
30	# If Run Tests is selected, Execute the selected tests			
31	If	(Report_Q9) = "Run Tests"		If Run Tests = Run Tests = True
32	Runsub	Perform_Digital_Tests		runsub Perform_Digital_Tests
33	EndIf			
34	END			
35	EndSub			
36				

The VIP Tool constantly monitors the communication with the instrument while a socket is open. There may be instances when the script is processing intensive tasks that do not require communication with the instrument, such as long loops that gather information from other sheets or assign multiple values to variables and tags. In these cases, closing the socket may speed up execution of these sections of the script. Close_socket and open_socket are commands that can be used within a script to close then re-open the socket to an instrument. These commands should be used sparingly and with care. <u>These commands are only useful if extensive computation that does not involve communication with the instrument is being executed by the script.</u>

When a script is executed, the socket to the instrument is opened, allowing communication over Ethernet with the instrument. Therefore, when the script is running, the socket is already open. If the script contains an extensive block of computation that does not require communication with the instrument to send and receive

data to and from the instrument, the socket can be closed with the close_socket keyword. At the end of the section of code containing the computation, the socket must be opened again using the open_socket keyword so that communication can be re-established with the instrument.

The close_socket and open_socket keywords are not case sensitive.

Report Tools

Define a Name for a Saved Report Command

Test_Name

Command (*Argument*) Syntax:

Test_Name (*text_entry*) Defines the base name for PDF and CSV reports. Populates the 'Test Name' field of the Setup sheet.

Test_name provides a means to define the base name that PDF and CSV report sheets are saved under. When a VIP Tool report is saved, in either format, the report is given the name defined in the argument for test_name, with a date and time stamp in the following format:

Test_name_date_time

The argument for test_name must be directly entered; variables, tags or cell assignments are not allowed. The argument can have double quotation marks, but they are not necessary.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	test_name	My Test		Test Name = My Test
3				

Test Name	**My Test**
Report Sheet	**Report**
Open PDF Report On Save	**On**

In the above example "My Test" was the argument used for test_name. When the keyword was executed, it placed the test name in the "Test Name" field of the Setup sheet.

The keyword test_name is not case sensitive.

Select a Report Sheet to Save Command

Report_Sheet

Command (*Argument*) Syntax:

Report_Sheet (*text_entry*) Defines which embedded report sheet will be saved when a programmatic save report command is encountered. Populates the 'Report Sheet' field of the Setup sheet.

Report_sheet provides a means to define the default sheet that is saved whenever a save report command is encountered. When a save_report command does not have an overriding argument, the report sheet named in this field is saved.

The argument for report_sheet must be directly entered; variables, tags or cell assignments are not allowed. The argument can have double quotation marks, but they are not necessary. The keyword report_sheet is not case sensitive.

	A	B	C	D
1	Command	Argument	Reply	Info Message
2	report_sheet	Examples		Report Sheet = Examples
3				

Test Name	My Test
Report Sheet	Examples
Open PDF Report On Save	On

In the above example “Examples” was the argument used for report_sheet. When the keyword was executed, it placed the report sheet name in the “Report Sheet” field of the Setup sheet.

The keyword report_sheet is not case_sensitive.

Save a Report as PDF Command

Save_Report_PDF/Save_Report_Now_PDF

Command (*Argument*) Syntax:

Save_report_pdf (*optional text_entry*) When executed, the command will wait until the script has finished executing, and will then save the defined report sheet at that time in the PDF format. If no argument is present, the report sheet named in the setup sheet “Report Sheet” field will be saved. If a report sheet name is included in the argument column, then the command will save the sheet named in the argument instead.

Save_report_now_pdf (*optional text_entry*) When executed, the command will immediately save the defined report sheet in PDF format. If no argument is present, the report sheet named in the setup sheet “Report Sheet” field will be saved. If a report sheet name is included in the argument column, then the command will save the sheet named in the argument instead.

Save_report_pdf is used when the programmer intends for an entire script to run before a report is saved. It can appear anywhere in the script, and the VIP Tool will wait until the script has completed before saving the report. Save_report_now_pdf is used if the programmer wishes to save a PDF report in its current state or to save multiple reports from a script.

If the Setup sheet “Open PDF Report on Save”, the report will be opened in the PDF viewer of the operator’s computer, otherwise, the save function will happen in the background.

The argument for save_report_pdf or save_report_now_pdf must be directly entered; variables, tags or cell assignments are not allowed. The argument can have double quotation marks, but they are not necessary.

The save_report_pdf and save_report_now_pdf keywords are not case sensitive.

Save a Report as CSV Command

Save_Report_CSV/Save_Report_Now_CSV

Command (*Argument*) Syntax:

Save_report_csv (*optional text_entry*) When executed, the command will wait until the script has finished executing, and will then save the defined report sheet at that time in the csv format. If no argument is present, the report sheet named in the setup sheet "Report Sheet" field will be saved. If a report sheet name is included in the argument column, then the command will save the sheet named in the argument instead.

Save_report_now_csv (*optional text_entry)* When executed, the command will immediately save the defined report sheet in CSV format. If no argument is present, the report sheet named in the setup sheet "Report Sheet" field will be saved. If a report sheet name is included in the argument column, then the command will save the sheet named in the argument instead.

Save_report_CSV is used when the programmer intends for an entire script to run before a report is saved. It can appear anywhere in the script, and the VIP Tool will wait until the script has completed before saving the report. Save_report_now_CSV is used if the programmer wishes to save a PDF report in its current state or to save multiple reports from a script.

The argument for save_report_csv or save_report_now_csv must be directly entered; variables, tags or cell assignments are not allowed. The argument can have double quotation marks, but they are not necessary.

The save_report_csv and save_report_now_csv keywords are not case sensitive.

Remove Previous Values From Tags Command

Clear Tags

Command (*Argument*) Syntax:

Clear_tags (*No argument*) Programmatically deletes the values, but not formulas, assigned to tags.

The clear_tags keyword is used to delete only the values assigned to tags; it does not delete the tag itself. This keyword is generally intended for clearing the results of a previous test from a report sheet that uses tags to assign values. When the clear_tags keyword is placed at the beginning of a script, all previous tag values are removed so that no results from a previous test are present on the report when the new script runs. Running clear_tags essentially 'clears the slate'.

The clear_tags keyword will not remove a formula from a cell that is assigned a tag, so the result of any formula will be visible in a cell that still contains the formula. This functionality is implemented so that formulas a script relies on are not deleted when clear_tags is executed.

The clear_tags keyword is not case sensitive.

Debug and Notation Tools

Remark Symbol

Symbol (Remark)

Command (*Argument*) Syntax:

[precedes a command column remark] (*optional additional remark*) Creates a remark in the script.

When encountered by the script, the '#' symbol causes the program to recognize the contents of the command and argument columns to be recognized as remarks, and all text in those columns will be displayed in a green font. The program will step to a row containing a remark, then step past it – it does not step over it as it would an inactivated command.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	#This row is marked as a remark.	If the command column is marked as a remark by the "#" symbol, remarks can also appear in the argument column.		
3	*idn?		Aeroflex, 88XX, 1000000009, 2.4.4-0, Jan 12 2021, 08:55:28	
4	*idn?		Aeroflex, 88XX, 1000000009, 2.4.4-0, Jan 12 2021, 08:55:28	
5	#This row is also marked as a remark. Note that, though it is a remark, the script will actually step to the remark, as shown by the green background of this row.			
6	*idn?			
7				

/ Inactivate Command Symbol

/ Symbol (Inactivate Command)

Command (*Argument*) Syntax:

/ [included in command column cell] (*no argument*) Inactivates a command in the script.

When the '/' symbol is detected in a command column cell, the command in the cell is inactivated. The contents of the command and argument columns will appear in a gray font when the symbol is detected in the command column. The program will not step through the row but will step over or 'skip' the row. The '/' symbol can be entered manually into the command column cell. Pressing the 'Toggle Selection Active/Inactive' button will place or remove the symbol in any selected cell.

Breakpoint Function

Break

Command (*Argument*) Syntax:

break (*no argument*) Pauses execution of script when invoked (debugging tool).

Break is used to insert a breakpoint while a script is running. When a script encounters the break command, program operation pauses (but execution does not end) on that row. When the script is paused at a breakpoint, the Reply column will display the message “Holding at Breakpoint”. Pressing the ‘Run From Here’ or ‘Single Step’ buttons on the VIP Tool ribbon will allow the script to resume, either in full automatic mode, or single step mode, respectively.

When the script is not running, selecting a row and pressing the ‘Insert Breakpoint Here’ button will insert a row and place the break command in the new row. Pressing the ‘Remove All Breakpoints’ button will remove all instances of break on the Script sheet and will remove the row(s) formerly occupied by the break command(s). Break can also be typed into any command column row, which will fulfill the same purpose.

The break command is not case sensitive.

The break command is always displayed in a red font for easy identification.

21				
22	sub	test1		
23	print	[sinad]		[SINAD] = 21.88
24	print	[my_message]		[MY_MESSAGE] = Test Complete
25	BREAK		Holding at breakpoint	Press Run Script, Run From Here, or Single Step to Continue
26	if	[sinad] < 13		
27	pause	[my_message]		
28	endif			
29	endsub			
30				

In the above example, break has been inserted into a subroutine on row 25. The program is paused at that point.

NOTE:	Note that rows 23 and 24 contain the print keyword. The print keyword will display in the Reply column the contents of a variable, tag or cell assignment that is placed in its Argument. It is useful to use the print keyword in conjunction with the break keyword in to determine what value may be held by a variable, tag, or cell assignment at the point the break has been inserted.

Debug Print Function

Print

Command (*Argument*) Syntax:

Print (*variable, tag or cell assignment*) Prints contents of variable, tag or cell assignment to the Info Message column.

Print is a debugging tool that is used to display to the user the current contents of a variable, tag or cell assignment.

Print is placed in the Command column and the variable, tag or cell assignment is placed in the Argument. When executed, the contents of the variable, tag, or cell assignment is displayed in the Info Message column. The keyword print is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[myvar]=	220.625		[MYVAR] = 220.625
3	print	[myvar]		[MYVAR] = 220.625
4	print	<mytag>		<mytag> = 451.03
5	print	(examples_A1)		Sheet EXAMPLES cell A1 = 151.015
6				

Utility Commands

Copy Function

Copy

Command (*Argument*) Syntax:

Copy (*text, variable, tag, or cell assignment To cell assignment*) Copies text entry or the contents of a variable, tag or cell assignment to a cell assignment.

Copy is a utility keyword that is used to copy data from any sheet in the VIP Tool workbook to a cell on any report sheet embedded in the VIP Tool workbook. The data to be copied can be directly entered in the argument, or it can be used to transfer the *value* of a cell assignment, or data stored in a variable or tag. Copy will not copy a *formula* stored in a cell assignment, only the *value* held by a cell assignment. However, a formula can be written out and enclosed in quotations and copied to a cell on an embedded worksheet.

The keyword copy is not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	copy	(calc_a1) to (Examples_L1)		Copied "123.456" to Sheet Examples Cell L1
3	copy	(calc_a2) to (Examples_L2)		Copied "This is a test" to Sheet Examples Cell L2
4	copy	151.255 to (Examples_L3)		Copied "151.255" to Sheet Examples Cell L3
5	copy	"My text" to (Examples_L4)		Copied "My text" to Sheet Examples Cell L4
6				

	A	B
1	123.456	
2	This is a test	
3		

Calc Worksheet

	K	L	M
		123.456	
		This is a test	
		151.255	
		My text	

Examples Worksheet

In the above example, the copy command on row 1 copies the value of cell A1 on an embedded sheet called “Calc” to cell L1 on an embedded worksheet called “Examples. Row 2 copies the value of the “Calc” worksheet

cell A2 to the “Examples” sheet cell L2. Row 3 directly copies the value of ‘151.255’ to the “Examples” sheet cell L3 and row 4 directly copies the value “My Text” to cell L4 of the “Examples” sheet.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[myvar]=	"Text or Number"		[MYVAR] = Text or Number
3	copy	[myvar] to (examples_b2)		Copied "Text or Number" to Sheet examples Cell B2
4	copy	<mytag> to (examples_b1)		Copied "150" to Sheet examples Cell B1
5				

Tag on any Report Sheet
Show Tags Mode

	A	B	C
1			
2			
3		<mytag>	
4			

Tag on any Report Sheet
Hide Tags Mode

	A	B	C
1			
2			
3		150.0000	
4			

	A	B	C
1		150.0000	
2		Text or Number	
3			

Examples Worksheet

The above illustration demonstrates copying values held by tags to a report sheet. Row 2 assigns the text value “Text or Number” to the variable ‘[myvar]’. Row 3 copies the value of ‘[myvar]’ to cell B2 on the “Examples” sheet. Row 4 copies the value of the tag ‘<mytag>’, which can exist on any worksheet, to cell B1 of the Examples worksheet.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	for	1 to 3		For 1 to 3
3	copy	(calc_a:nextcount) to (examples_a:nextcount)		Copied "989.727" to Sheet examples Cell A3
4	next			
5				

	A	B
1	123.456	
2	This is a test	
3	989.727	
4		

Calc Worksheet

	A	B
1	123.456	
2	This is a test	
3	989.727	
4		

Examples Worksheet

As with any variable, tag, or cell assignment, array indexes can be used to address values. In the example above example, row 3 of the worksheet addresses column A of the “Calc” sheet with the index held by the nextcount counter. As the destination, column A of the “Examples” worksheet is addressed. The For Next loop counts from 1 to 3, so the value of nextcount steps from 1 to 3. Therefore, cells A1, A2 and A3 of the “Calc” worksheet are sequentially addressed in the For Next loop, and their values are sequentially copied to cells A1, A2 and A3 of the “Examples” worksheet.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	copy	"=D1 / 2" to (calc_b1)		Copied "=D1 / 2" to Sheet calc Cell B1
3				

B1 | =D1 / 2

	A	B	C	D
1		1.5		3
2				

Calc Worksheet

Copy will only directly copy the value of a cell assignment to another cell. However, a formula can be copied to another cell by writing out the formula in the standard Excel format. In the above example, “=D1 / 2 “ is copied to cell B1 of the “Calc” worksheet. This process copied the formula into cell B1 of the “Calc” worksheet. Cell B1 of the “Calc” worksheet displays the value of ‘1.5’, which is the result of the formula “=D1/2”, because the value displayed is 3 divided by 2 (‘3’ being the value held in cell D1).

Create Tag Function

Show_Tags/Create_Tag/Hide_Tags

Command (*Argument*) Syntax:

Show_tags (*No argument*) Programmatically places the VIP Tool in the Show Tags mode. Tags become visible when the show_tags function is executed. The effect of this function is identical to pressing the Show Tags button.

Create_Tag (*text or variable to cell assignment*) Programmatically creates tags. The argument consists of text or a variable that holds text, followed by the keyword to, and puts the tag in the cell defined by the cell assignment.

Hide_tags (*No argument*) Programmatically places the VIP Tool in the Hide Tags mode. Tags become invisible and the current value of the tag, if any, becomes visible when the hide_tags function is executed. Identical function to pressing the Hide Tags button when a script is not running.

Show_tags and hide_tags are commands that will place the VIP Tool in the Show Tags or Hide Tags modes, respectively. These commands are of little use when running a script used to control the instrument in a test routine, but instead are provided as utility commands for use in conjunction with using the create_tag keyword.

The create_tag keyword is a utility that is used to create a tag and assign it to a cell. The purpose of create_tag is to automatically create tag arrays programmatically rather than manually entering each tag when creating a report sheet. Create_tag allows the programmer to define a base tag name and assign it an index number using a counter within a loop. The cell assignment in the argument can be defined as a range of cells, using either a 'count' or 'range' variation of the selected counter's syntax.

The show_tags, create_tag, and hide_tags keywords are not case sensitive.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	show_tags			
3	create_tag	"Mytag" to (examples_a1)		Created and copied "<Mytag>" to Sheet examples Cell A1
4	hide_tags			
5				

	A	B
1	<Mytag>	
2		

The illustration above demonstrates creating a tag using text entry. Row 2 contains the keyword command show_tags so that the created tag can be recorded; show_tags is required for any create_tag application because the VIP Tool will always hide tags if any row is executed by pressing the Run Script, Run Selection, Run From Here, or Single Step From Here buttons.

Row 3 contains the create_tag keyword. The argument creates a tag using the text "Mytag" and assigns it to cell A1 of an embedded sheet called "Examples" in the VIP Tool workbook.

Row 4 contains the hide_tags keyword, which hides the tags and registers the tag locations in the VIP Tool workbook. After this script is executed, pressing the Show Tags button will reveal the newly created tag '<Mytag>' in cell A1 of the "Examples" sheet.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	[tagname]=	"Mytag"		[TAGNAME] = Mytag
3	show_tags			
4	create_tag	[tagname] to (examples_a1)		Created and copied "<Mytag>" to Sheet examples Cell A1
5	hide_tags			
6				

	A	B
1	<Mytag>	
2		

The example above accomplishes the same task as the previous example by using the contents of a variable to name the created tag. On row 2, the variable '[tagname]' is assigned the text value "Mytag". On row 4, create_tag uses the contents of the variable '[tagname]' as the name to assign to the newly created tag.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	show_tags			
3	[tagname]=	"Freq"		[TAGNAME] = Freq
4	for	1 to 3		For 1 to 3
5	create_tag	[tagname]_nextcount to (examples_A:nextcount)		Created and copied "<Freq_3>" to Sheet examples Cell A3
6	next			
7	hide_tags			
8				

	A	B
1	<Freq_1>	
2	<Freq_2>	
3	<Freq_3>	
4		

In the above example, create_tag is placed inside a For Next loop. The embedded For Next loop, nextcount, is used as an index to create an array of tags. On row 3, the variable '[tagname]' is assigned the text value "Freq". This variable is used to create the base name of the tag array. The For Next loop argument on row 4 sets the loop to step from 1 to 3, so this loop will iterate 3 times, creating a tag on each iteration.

On row 5, the create_tag argument assigns the current value of nextcount to the base name with the syntax '[tagname]_nextcount'. So, on the first iteration of the For Next loop, the tag name 'Freq_1" is created. On the second iteration of the For Next loop, the tag name 'Freq_2' is created, and so on.

The cell assignment in the argument works the same way; on the first iteration of the For Next loop, the tag name is assigned to cell A1 of a sheet named 'Examples', on the second iteration of the For Next loop, the tag name is assigned to cell A2 of the same sheet, and so on.

After running the script, pressing the Show Tags button reveals that the script has created three tags in column A of the "Examples" sheet.

	A	B	C	D
1	**Command**	**Argument**	**Reply**	**Info Message**
2	show_tags			
3	[tagname]=	"RX_Freq"		[TAGNAME] = RX_Freq
4	for	1 to 10		For 1 to 10
5	[group]=	"A"		[GROUP] = A
6	create_tag	[tagname]_[group]_nextcount to (examples A:nextcount)		Created and copied "<RX_Freq_A_10>" to Sheet examples Cell A10
7	[group]=	"B"		[GROUP] = B
8	create_tag	[tagname]_[group]_nextcount to (examples B:nextcount)		Created and copied "<RX_Freq_B_10>" to Sheet examples Cell B10
9	next			
10	hide_tags			
11				

	A	B	C
1	<RX_Freq_A_1>	<RX_Freq_B_1>	
2	<RX_Freq_A_2>	<RX_Freq_B_2>	
3	<RX_Freq_A_3>	<RX_Freq_B_3>	
4	<RX_Freq_A_4>	<RX_Freq_B_4>	
5	<RX_Freq_A_5>	<RX_Freq_B_5>	
6	<RX_Freq_A_6>	<RX_Freq_B_6>	
7	<RX_Freq_A_7>	<RX_Freq_B_7>	
8	<RX_Freq_A_8>	<RX_Freq_B_8>	
9	<RX_Freq_A_9>	<RX_Freq_B_9>	
10	<RX_Freq_A_10>	<RX_Freq_B_10>	
11			

The above illustration depicts a method for introducing multiple elements to a tag array name, and for assigning different tag names to different columns of a report sheet using a For Next loop.

The For Next loop argument on row 4 sets the loop to step from 1 to 10, so this loop will iterate 10 times, creating two unique tags on each iteration.

On row 3, the variable '[tagname] is assigned the value of "RX_Freq".

On row 5, the variable '[group]' is assigned the text value of "A".

On row 6, the argument for create_tag is:

[tagname]_[group]_nextcount to (examples_A:nextcount)

This creates the base tag name "RX_Freq_A_" and appends the value of nextcount on the end of the tag name. The argument assigns the tag destination to the A column of a sheet named "Examples", with the row number defined by the value of nextcount. The first iteration of the loop will create the tag '<RX_Freq_A_1>' and is sent to cell A1 of the "Examples" sheet. The second iteration of the loop will create the tag '<RX_Freq_A_2>' and is sent to cell A2 of the "Examples" sheet and so on.

On row 7, the '[group]' variable is given the text value "B".

On row 8, the argument for create_tag is:

[tagname]_[group]_nextcount to (examples_B:nextcount)

This creates the base tag name "RX_Freq_B_" and appends the value of nextcount on the end of the tag name. The argument assigns the tag destination to the B column of a sheet named "Examples", with the row number defined by the value of nextcount. The first iteration of the loop will create the tag '<RX_Freq_B_1>' and is sent to cell B1 of the "Examples" sheet. The second iteration of the loop will create the tag '<RX_Freq_B_2>' and is sent to cell B2 of the "Examples" sheet and so on.

After running the script, pressing the Show Tags button reveals that the script has created twenty tags, 10 tags in column A of the "Examples" sheet, and 10 tags in column B of the "Examples" sheet

This page intentionally left blank.

.

Appendix A: VIP Tool and Instrument RCI Resources

VIP Tool Resources

The VIP Tool web page provides several resources for users of the tool. At this link you will find application notes, videos and more.

VIP Tool Resources

3900 Series RCI Manuals

A number of different RCI manuals are available for the 3900 series of test instruments. The manuals consist of the RCI commands for the platform/ Analog Duplex System, P25 System, DMR System, NXDN system and more. The 3900 series RCI documentation, in PDF format, is available for download at the following links:

3900 Series Digital Radio Test Set Remote Programming Manual

Remote Programming Manual: 3900 Series Digital Radio Test Set P25

Remote Programming Manual: 3900 Series Digital Radio Test Set DMR

Remote Programming Manual: 3900 Series Digital Radio Test Set dPMR

Remote Programming Manual: 3900 Series Digital Radio Test Set NXDN

Remote Programming Manual: 3900 Series Digital Radio Test Set TETRA

Remote Programming Manual: 3900 Series Digital Radio Test Set HPD®

Remote Programming Manual: 3900 Series Digital Radio Test Set ARIB STD-T98

8800 Series RCI Manual

The 8800 Series RCI Manual can be downloaded at the following link:

8800 Digital Radio Test Set RCI Programming Manual

3550 Series RCI Manual

The 3550 Series RCI Manual can be downloaded at the following link:

3550/3550R RCI Manual

This page intentionally left blank.

Document Number:Release Date: 14 Feb 2021

Revision: 1.0

VIAVI Solutions

North America	(Toll Free) 1-844-GO-VIAVI / 1-844-468-4284
Latin America	+52 55 5543 6644
EMEA	+49 7121 862273
APAC	+1 512 201 6534