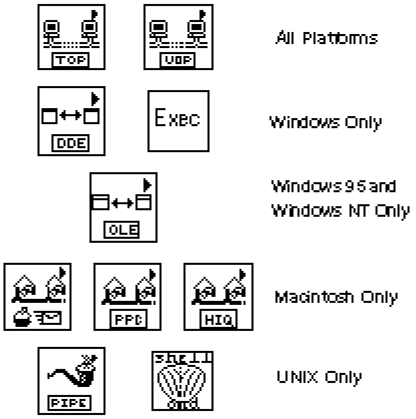
Communication Applications in LabVIEW

Communication VI Descriptions

Click here for Overview Topics.

This section is an overview of the way LabVIEW handles networking and interapplication communications in the following areas: Dynamic Data Exchange, Transmission Control Protocol, User Datagram Protocol, Object Linking and Embedding, and Named Pipes. This topic also describes the VIs that link LabVIEW to HiQ, the National Instruments numerical analysis package as well as the System Exec VI.

Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.



AppleEvents VI Descriptions
DDE VI Descriptions
HiQ Functions for Macintosh
Named Pipe VI Descriptions (UNIX)
OLE Automation VI Descriptions
PPC VI Descriptions
System Exec VI for Windows
System Exec VI for UNIX
TCP VI Descriptions
UDP VI Descriptions

Communication Overview Topics

Communications Overview
Transmission Control Protocol (TCP) VI Overview
User Datagram Protocol (UDP) VI Overview
Dynamic Data Exchange (DDE) for Windows VI Overview
OLE Automation for Windows 95/NT VI Overview
AppleEvents VI Overview (Macintosh)
Program to Program Communication (PPC) for Macintosh VI Overview

Communications Overview

For the purpose of this discussion, networking refers to communication between multiple processes. The processes can optionally run on separate computers. This communication usually occurs over a hardware network, such as ethernet or LocalTalk.

One main use for networking in software applications is to allow one or more applications to use the services of another application. For example, the application providing services (the server) could be either a data collection application running on a dedicated computer, or a database program providing information for other applications.

The purpose of this discussion is to introduce you to the terminology used in networking and communication applications, and to give you an overview of how to program networked applications.

Communication Protocols

For communication between processes to work, the processes must use a common communications language, referred to as a *protocol*.

A communication protocol lets you specify the data that you want to send or receive and the location of the destination or source, without having to worry about how the data gets there. The protocol translates your commands into data that network drivers can accept. The network drivers then take care of transferring data across the network as appropriate.

Several networking protocols have emerged as accepted standards for communications. In general, one protocol is not compatible with a different protocol. Thus, in communication applications, one of the first things you must do is decide which protocol to use. If you want to communicate with an existing, off the shelf application, then you have to work within the protocols supported by that application.

When you are writing the application, you have more flexibility in choosing a protocol. Factors that affect your protocol choice include the type of machines the processes can run on, the kind of hardware network you have available, and the complexity of the communication that your application needs.

Several protocols are built into LabVIEW, some of which are specific to a type of computer. LabVIEW uses the following protocols to communicate between computers:

- **TCP VI Overview (All Platforms)** - available on all computers.
- **UDP VI Overview (All Platforms)** - available on all computers.
- **DDE VI Overview (Windows)** - available on the PC, for communication between Windows applications.
- **OLE Automation VI Overview (Windows 95/NT)** - available on the PC, for use with Windows 95/NT.
- **AppleEvents VI Overview (Macintosh)** - available on Macintosh, for sending messages between Macintosh applications.
- **PPC VI Overview (Macintosh)** - available on Macintosh, for sending and receiving data between Macintosh applications.

Each protocol is different, especially in the way they refer to the network location of a remote application.

They are incompatible with each other, so if you want to communicate between a Macintosh and a PC, you must use a protocol compatible with both, such as TCP.

Other communication options provided by LabVIEW include:

- **HiQ Functions for Macintosh** -.available on Macintosh only
- **System Exec VI Descriptions (Windows and UNIX)** - which you can use to execute a system level command. There are actually two System Exec VIs, one for use with all versions of Windows, the other for use with UNIX.
- **Named Pipe VI Descriptions (UNIX)** - available on UNIX only.

Note: The three previous communication options do not contain overview material.

File Sharing vs Communication Protocols

Before you get too deeply involved in communication protocols, consider whether another approach is more appropriate for your application. For instance, consider an application where a dedicated system acquires data and you want the data recorded on a different computer.

You could write an application that uses networking protocols to send data from the acquisition computer to the data repository machine, where a separate application collects the data and stores it on disk.

A simpler method is to use the filesharing capabilities available on most networked computers. With filesharing, drivers that are part of the operating system let you connect to other machines. The remote machines disk storage is treated as an extension of your own disk storage. Once you connect two systems, filesharing usually makes this connection transparent, so that any application can write to the remote disk as if connected locally.

Filesharing is frequently the simplest method for transferring data between machines.

Client/Server Model

The client/server model is a common model for networked applications. In the client/server model, one set of processes (clients) request services from another set of processes (servers).

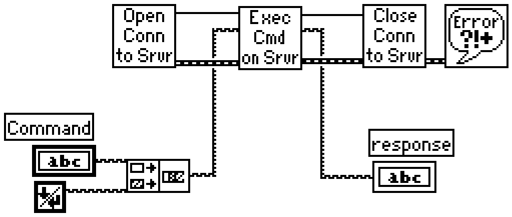
For example, in your application you could set up a dedicated computer for acquiring measurements from the real world. The computer acts as a server when it provides data to other computers on request. It acts as a client when it requests another application, such as a database program, to record the data that it acquires.

In LabVIEW, you can use client and server applications with all protocols except Macintosh AppleEvents. You can use AppleEvents to send commands to other applications. You cannot set up a command server in LabVIEW using AppleEvents. If you need server capabilities on the Macintosh, use either TCP, UDP or PPC.

General Model for a Client
General Model for a Server

General Model for a Client

The following block diagram shows what a simplified model for a client looks like in LabVIEW:



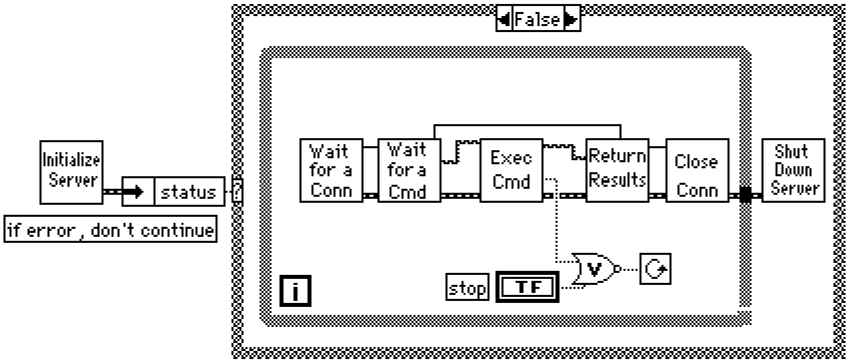
In the preceding diagram, LabVIEW first opens a connection to a server. It then sends a command to the server, gets a response back, and closes the connection to the server. Finally, it reports any errors that occurred during the communication process.

For higher performance, you can process multiple commands once the connection is open. After the commands are executed, you can close the connection.

This basic block diagram structure serves as a model and is used elsewhere in this manual to demonstrate how to implement a given protocol in LabVIEW.

General Model for a Server

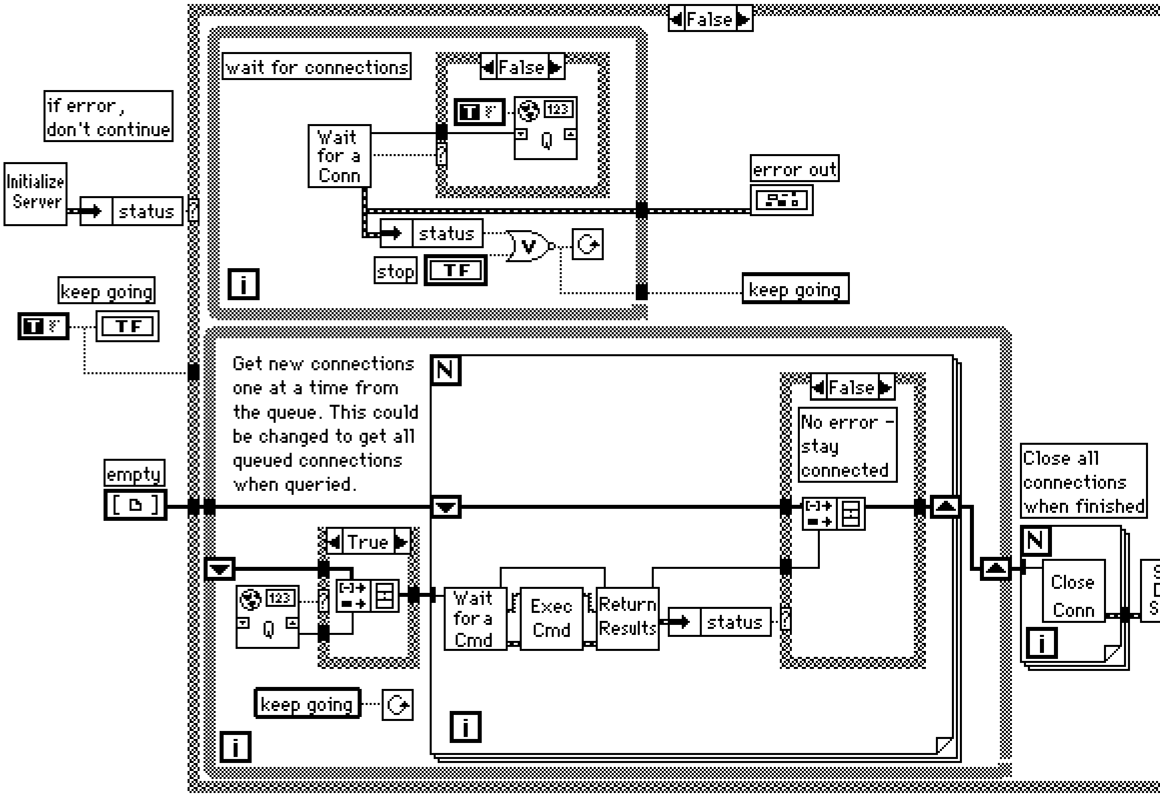
The following block diagram shows a simplified model for a server in LabVIEW:



In the preceding diagram, LabVIEW first initializes the server. If the initialization is successful, LabVIEW goes into a loop, where it waits for a connection. Once the connection is made, LabVIEW waits to receive a command. LabVIEW executes the command and returns the results. The connection is then closed. LabVIEW repeats this entire process until it is shut down locally by pressing a stop button on the front panel, or remotely by sending a command to shut the VI down.

This VI does not report errors. It may send back a response indicating that a command is invalid, but it does not display a dialog box when an error occurs. Because a server might be unattended, consider carefully how the server should handle errors. You probably do not want a dialog box to be displayed, because that requires user interaction at the server (someone would have to press the OK button). However, you might want LabVIEW to write a log of transactions and errors to a file or a string.

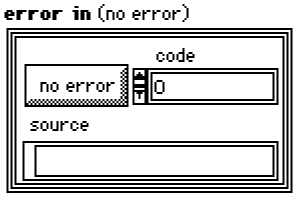
You can increase performance by allowing the connection to stay open, so that you can receive multiple commands, but this blocks others clients from connecting until the current client disconnects. If the protocol supports multiple simultaneous connections, you can restructure LabVIEW to handle multiple clients simultaneously, as shown in the following diagram.



The preceding diagram uses LabVIEWs multitasking capabilities to run two loops simultaneously. One loop continuously waits for a connection. When a connection is received, it is added to a queue. The other loop checks each of the open connections and executes any commands that have been received. If an error occurs on one of the connections, the connection is disconnected. When the user aborts the server, all open connections are closed. This basic block diagram structure is a model which is used elsewhere in this discussion to demonstrate how to implement a given protocol in LabVIEW.

Error In and Error Out Clusters

Many of the Communication VIs report errors in clusters, as the following illustration shows.

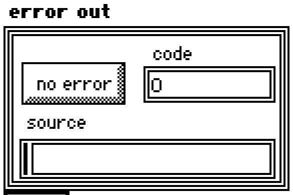


error in (no error)

code

no error 0

source



error in describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. The **error in** cluster contains the following parameters:

TF **status** is TRUE if an error occurs. If **status** is TRUE, this VI does not perform any operations, and error out contains the same information as error in.

I32 **code** is the error code. A value of 0 means no error.

abc **source** either gives the name of the TCP VI where the error occurs followed by the error message, or the name of the last TCP VI to execute followed by the `no error` message if no error occurs.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces.

If a VI receives an error cluster with a TRUE status flag, the VI passes the error cluster out without changing it or attempting any of the VIs commands. If you do not wire **error in**, it defaults to `no error`.

Named Pipe VIs

Named Pipe VI Descriptions (UNIX)

Transmission Control Protocol VIs

TCP VI Descriptions

User Datagram Protocol VIs

UDP VI Descriptions

Dynamic Data Exchange VIs

Object Linking and Embedding Automation VIs

OLE Automation VI Descriptions

Apple Event VIs

Program to Program Communication VIs

PPC VI Descriptions

HiQ VIs

HiQ Functions for Macintosh

AppleEvents VI Overview (Macintosh)

Click here to access the AppleEvents VI Descriptions topic.
Click here to access the LabVIEW Specific AppleEvent VI Descriptions topic.
Click here to access the Low Level AppleEvent VI Descriptions topic.

This topic discusses the LabVIEW VIs for interapplication communication (IAC), a feature of Apple Macintosh system software version 7 by which Macintosh applications can communicate with each other. You can use LabVIEW with two forms of IAC, AppleEvents and program to program communication (PPC).

AppleEvents

AppleEvents are a high-level method of communication in which applications use messages to request other applications to perform actions or return information. An application can send these messages to itself, other applications on the same machine, or other applications located anywhere on a network. Apple has defined a large *vocabulary* for messages to help standardize this form of interapplication communication. You can combine *words* in this vocabulary to form very complex messages. This vocabulary is described in detail in the *AppleEvent Registry,* a document available from Apple. Most applications written for System 7, including LabVIEW, respond to some subset of AppleEvents.

Note: PPC, a low-level form of IAC, provides higher performance than AppleEvents, because the overhead required to transmit information is lower. However, because PPC does not define what kinds of information you can transfer, many applications do not support it. PPC is the best way to send large amounts of information between applications that support PPC.Program to Program Communication VI Overview , for more information about PPC.

For applications to communicate with IAC, the computer must use system software version 7.0 or greater with Program Linking enabled.

LabVIEW can send and respond to AppleEvents. You can use AppleEvent VIs to send AppleEvents. LabVIEW responds to two types of AppleEvents: LabVIEW-defined events and a subset of standard AppleEvents. See the Sending AppleEvents topic for more information.

Some of the ways you can use AppleEvents in LabVIEW applications are listed below:

- You can command LabVIEW to tell another application (even an application on another computer connected by a network) to perform an action. For example, LabVIEW can tell a spreadsheet program to create a graph See the Sending AppleEvents topic for more information.
- You can use a program, such as HyperCard as a front end to instruct LabVIEW to run specific VIs.
- You can communicate with and control LabVIEW applications on other machines connected by a network by sending them instructions to perform specific operations. See the Sending AppleEventstopic for more information.

- You can command LabVIEW to send messages to itself, instructing itself to load, run, and unload specific VIs. For example, in large applications where memory is tight, you can replace subVI calls with a utility VI (the AESend Open, Run, Close VI) and dynamically load, run, and unload the VIs.See the Sending AppleEvents topic for more information.

Sending AppleEvents

You can find VIs for sending AppleEvents in **Functions**»**Communication**»**AppleEvent**. With these VIs, you can select a target application for an AppleEvent, create AppleEvents, and send the AppleEvents to the target application.

You can find VIs that send specific AppleEvent messages in **Functions**»**Communication AppleEvent**»**LabVIEW Specific Apple Event**. These VIs let you send several standard AppleEvents (Open Document, Print Document, and Close Application) and all the LabVIEW custom AppleEvents. These high-level VIs require little understanding of AppleEvent programming details. These diagrams also serve as good examples of how to create and send AppleEvents.

You can use the low-level AESend VI if you want to send an AppleEvent for which LabVIEW provides no VI. The **Functions**»**Communication**»**AppleEvent**»**Low Level Apple Events** palette also contains VIs that can help you create an AppleEvent. However, creating and sending an AppleEvent at this level requires detailed understanding of AppleEvents as described in *Inside Macintosh, Volume VI* and the *AppleEvent Registry*.

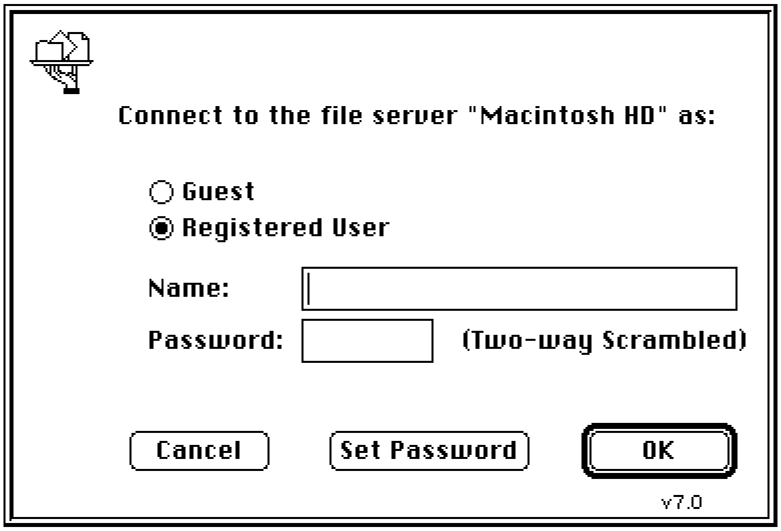
General AppleEvent VI Behavior

When sending an AppleEvent, you must specify the *target* application for the event. To receive the AppleEvent, the target application must be open. You can use the AESend Finder Open VI to open an application.

User Identity Dialog Box
Target ID
Send Options

User Identity Dialog Box

Before you send an AppleEvent to another computer, you must use the Users & Groups control panel utility on the destination computer to set up a user name and password for yourself. The first time you send an AppleEvent to an application or Finder on the destination computer, a dialog box prompts you to enter your name and password. The system compares this information to the configuration of the Users & Groups control panel utility on the destination computer.



Connect to the file server "Macintosh HD" as:

○ Guest
● Registered User

Name:

Password: (Two-way Scrambled)

The current design of the AppleEvent Manager does not include a programmatic method for bypassing this dialog box, so you should take this into account when designing VIs that use IAC. For example, you cannot command an unattended remote computer to send an AppleEvent to a third computer; someone must enter user information into the User Identity Dialog Box that appears on the remote computer. The PPC VIs allow for *unauthenticated* sessions if guest access is enabled on the computer with which you wish to communicate, so you may find the PPC VIs more useful for certain kinds of LabVIEW-to-LabVIEW communication.

Target ID

Most VIs that send AppleEvents need a description of the target application that receives the AppleEvent. The **target ID** is a complex cluster of information, defined by Apple Computer Inc., describing the target application and its location. The following VIs generate the **target ID**, so you do not need to create this cluster on the block diagram.

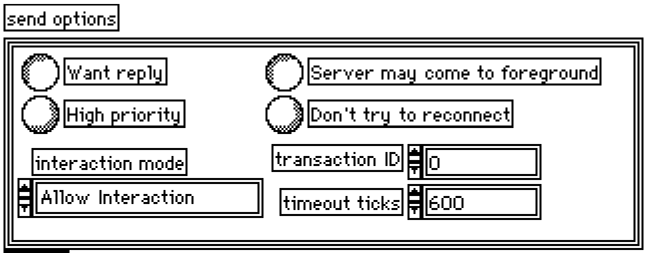
- PPC Browser creates the **target ID** by displaying a dialog box by which you interactively select AppleEvent-aware applications on the network.
- Get Target ID creates the **target ID** programmatically based on the applications name and network location.

These VIs are discussed in more detail in the AppleEvents VI Descriptions topic.

You need to look at the **target ID** cluster only if you want to pass target information from one VI to another. To create a **target ID** cluster for the front panel of a VI that passes target information to another VI or to an AppleEvent, you can copy the **target ID** cluster from the front panel of one of the AppleEvent VIs.

Send Options

Many of the VIs that send an AppleEvent have a **send options** input, which specifies whether the target application can interact with the user and the length of the AppleEvent timeout.


send options

○ Want reply
○ Server may come to foreground
○ High priority
○ Don't try to reconnect

interaction mode: Allow Interaction

transaction ID: 0

timeout ticks: 600

send options is a cluster containing the following parameters in the order listed below.

TF **want reply** specifies whether you want to receive a reply. The default is TRUE.

TRUE Asynchronously wait for a response from the application until the VI receives a response or a timeout occurs.
FALSE Send the AppleEvent and do not wait for a reply.

TF **high priority** determines whether the AppleEvent is added to the beginning or end of the target applications event queue. The default is FALSE.

TRUE Put the event at the front of the target application event queue.
FALSE Add the event to the end of the target application event queue.

I32 **interaction mode** determines the level of user interaction of the target application. The default is 1.

0: (Never Interact) Do not interact with the user.
1: (Allow Interaction) Can interact with the user if the target application needs information.
2: (Always Interact) Can interact with the user even if the target application does not need information.

TF **server may come to foreground** specifies whether the application can come to the foreground if it needs user interaction. The default is TRUE.

TRUE Can automatically switch to the foreground.
FALSE Notifies the user by flashing the application icon in the menu bar.

dont try to reconnect determines whether the system should try to reconnect if it is disconnected. The default is FALSE.

TRUE Do not attempt to reconnect if disconnected.
FALSE Attempt to reconnect if disconnected.

transaction ID is a number associated with a sequence of AppleEvents. If you will be sending multiple AppleEvents related to a single transaction, use the same number throughout the transaction. The default is 0.

U32 **timeout ticks** determines how long in ticks (1/60 of a second) LabVIEW waits for a reply before timing out if you entered TRUE for **want reply.** Use a value of 0 if you do not want a timeout. The default is 600 ticks, or 10 seconds.

AppleEvents Client/Server Model

You cannot use the AppleEvent VIs to create LabVIEW diagrams that behave as servers. The VIs are used to send messages to other applications. If you need diagram-based server capabilities, you must use TCP or PPC.

LabVIEW itself acts as an AppleEvent server, in that it understands and responds to a set of AppleEvents. Specifically, using AppleEvents, you can instruct LabVIEW to open VIs, print them, run them, and close them. You can ask LabVIEW whether a given VI is running. You can also tell LabVIEW to quit.

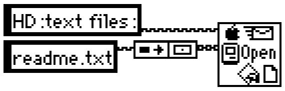
Using these server capabilities, you can instruct other LabVIEW applications to run VIs, and control

LabVIEW remotely. You can also command LabVIEW to send messages to itself, instructing the loading of specific VIs. For example, in large applications where memory is limited, you can replace subVI calls with calls to the AESend Open, Run, Close VI to load and run VIs as necessary. Notice that when you run a VI this way its front panel opens, just as if you had selected **File»Open...**.

AppleEvents Client Examples

Launching Other Applications

To send a message to an application, that application must be running. You can use the AESend Finder Open VI to launch another application. This VI sends a message to the Finder. The Finder is, in itself, an application that understands a limited number of AppleEvents. The following simple example shows how you can use AppleEvents to launch Teach Text with a specific text file:



If the application is on a remote computer, then you must specify the location of that computer. You can use inputs to the AESend Finder Open VI to specify the network zone and the server name of the computer with which you want to communicate. If the network zone and server name are not specified, as in the preceding application, they default to those of the current computer.

Notice that if you try to send messages to another computer, you are automatically prompted to log onto that computer. There is no method for avoiding this prompt, because it is built into the operating system. This can cause problems when you want your application to run on an unattended computer system.

Sending Events to Other Applications

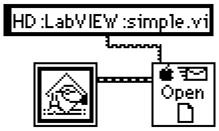
Once an application is running, you can send messages to that application using other AppleEvents. Not all applications support AppleEvents, and those that do may not support every published AppleEvent. To find out which AppleEvents an application supports, consult the documentation that comes with that application.

If the application understands AppleEvents, you call an AppleEvent VI with the Target ID for the application. A Target ID is a cluster that describes a target location on the network (zone, server, and supporting application). You do not need to worry about the exact structure of this cluster because LabVIEW provides VIs that you can use to generate a Target ID.

There are two ways to create a Target ID. You can use the Get Target ID VI to programmatically create a Target ID based upon the application name and network location. Or, you can use the PPC Browser VI, which displays a dialog box listing applications on the network that are aware of AppleEvents. You interactively select from this list to create a Target ID.

You can also use the PPC Browser VI to find out if another application uses AppleEvents. If you run the VI and select the computer that is running the application, the dialog box lists the application if it is AppleEvent aware.

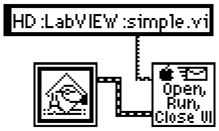
In the following diagram, LabVIEW interactively selects an AppleEvent aware application on the network and tells it to open a document. In this case, LabVIEW is telling the application to open a VI.



Dynamically Loading and Running a VI

The AESend Open, Run, Close VI sends messages asking LabVIEW to run a VI. First, it sends the Open Document Message and LabVIEW opens a VI. Then, the Open Run Close VI sends the LabVIEW Run VI message and LabVIEW runs the specified VI. Next, Open Run Close sends the VI Active? message, and LabVIEW returns the status of a specified VI, until the VI is no longer running. Finally, the VI sends the Close VI message.

Assuming the target LabVIEW is on another computer, you could use the following diagram to load and run the VI. If you are sending it to the current LabVIEW, you do not need the PPC Browser VI.



Advanced AppleEvents Topics

Constructing and Sending Other AppleEvents
Creating AppleEvent Parameters

Constructing and Sending Other AppleEvents

In addition to VIs that send common AppleEvents, you can use lower-level VIs to send any AppleEvent. Using these VIs requires more knowledge of AppleEvents than using the VIs described earlier in this chapter. If you are interested in using these VIs, you should be familiar with the discussion of AppleEvents in *Inside Macintosh, Volume VI,* and the *AppleEvent Registry*.

When sending an AppleEvent, you must include several pieces of information. The event class and event ID identify the AppleEvent you are sending. The event class is a four-letter code which identifies the AppleEvent group. For example, an event class of core identifies an AppleEvent as belonging to the set of core AppleEvents. The event ID is another four-letter code that identifies the specific AppleEvent that you wish to send. For example, odoc is the four-letter code for the Open Documents AppleEvent, one of the core AppleEvents. To send an AppleEvent using the AESend VI, concatenate the event class and event ID together as an eight-character string. For example, to send the Open Documents AppleEvent, pass the AESend VI the eight-character code coreodoc.

If you are sending the AppleEvent to another application, you have to specify **target ID** and **send options** .

You can also specify an array of parameters if the target application needs additional information to execute the specified AppleEvent. Because the data structure for AppleEvent parameters is inconvenient for use in LabVIEW diagrams, the AESend VI accepts these parameters as ASCII strings. These strings must conform to the grammar described in the Creating AppleEvent Parameters . You can use this grammar to describe any AppleEvent parameter. The AESend VI interprets this string to create the appropriate data structure for an AppleEvent, and then sends the event to the specified target.

Creating AppleEvent Parameters

In many cases, an AppleEvent parameter is a single value; however, it can be quite complex, with a hierarchical structure containing components that in turn can contain other components. In LabVIEW, a parameter is constructed as a string, which has a simple grammar with which you can describe all kinds of data that an AppleEvent parameter can be, including complex structures.

An AppleEvent parameter string begins with a keyword, a four-letter code describing the parameters meaning. For example, if the parameter is a direct parameter (one of the most common types of parameters) you must specify that the keyword is a keyDirectObject by using the four-letter code ---- (four dashes). Other examples of keywords include savo, short for save options, which is used when sending the Close VI AppleEvent to LabVIEW. Documentation detailing an applications supported AppleEvents should indicate the keywords used for each parameter. See the Sending AppleEvents to LabVIEW from Other Applications topic for a list of the AppleEvents that you can use with LabVIEW.

Following the keyword, you must specify the parameter data as a string. You can use AppleEvents with many different data types, including strings and numbers. When you specify the data string, the AESend VI converts it to a desired data type based upon the way the data is formatted and optional directives that can be embedded in the string. Each piece of data has a four-letter type code associated with it, indicating its data type. The target application uses this code to interpret the data. For example, if comma-separated items are enclosed in brackets, a list of *AE Descriptors* is created, and the list has a data type of list; each of the comma-separated items could in turn be other items, including lists.

You can use a number of VIs in the AppleEvents VI palette to create some of the more common parameter strings, including aliases, which are used when referencing files in parameters, and descriptor lists, which are used to specify a list of items as a parameter. You can concatenate or cascade these strings together to create a more complex parameter.

Table 6-1 describes the format of AppleEvent descriptor strings and indicates VIs that can create the descriptor, where appropriate.

AppleEvent Descriptor String Formats Table

To send data as	Format the string as	Parameter is of code type:	Examples	VI that can construct string:
an integer	A series of decimal digits, optionally preceded by a minus sign.	long or shor	1234 -5678	n/a
enumerated data	A four-letter code. If it is too long, it is truncated; if it is too short, it is padded with spaces. If you put single quotes () around it, it can contain any characters; otherwise, it cannot contain: @ : - , ([{ }]) and cannot begin with a digit.	enum	whos @all long >= 86it	n/a
a string	Enclose the desired sequence of characters within open and close curly quotes (Òentered with option-[andÓ entered	TEXT	Òput x into card field 5Ó ÒHi	n/a

	with option-shift-[). Notice that the string is not null-terminated.		ThereÓ	

an AE record	Enclose a comma-separated list of elements in curly braces, where each element consists of a keyword (a typecode) followed by a colon, followed by a value, which can be any of the types listed in this table.	reco	{x:100, y:-100}
			{origin: {x:100, y:-100}, extent: {x:500, y:500}, cont: [1,5,25]}
				AECreate Record

an AE descriptor list	Enclose a comma-separated list of descriptors in square brackets.	list	[123, -58, ÒtestÓ]	AECreate Descriptor List
hex data	Enclose an even number of hex digits between French quotes (Çentered with option-\ andÈ entered with option-shift-\).	?? (must be coerced Ð see next item)	Ç01 57 64 fe AB C1È	(Hex data is a component of the string produced by Make Alias)

some other data type	Embed data created in one of the types of this table in parentheses and put the desired type code before it. If the data is a numeric, LabVIEW coerces the data to the specified type if possible and returns the errAECoercionFail error code if it cannot. If the data is of a different type, LabVIEW replaces the old typecode with the specified type code.	The specified type code	sing(1234)
			alis(Ç*hex dump of an alias*È)
			type(line)
			rang{star: 5, stop: 6}
				n/a
				Make Alias creates a hex dump of a file description.
				n/a
				n/a

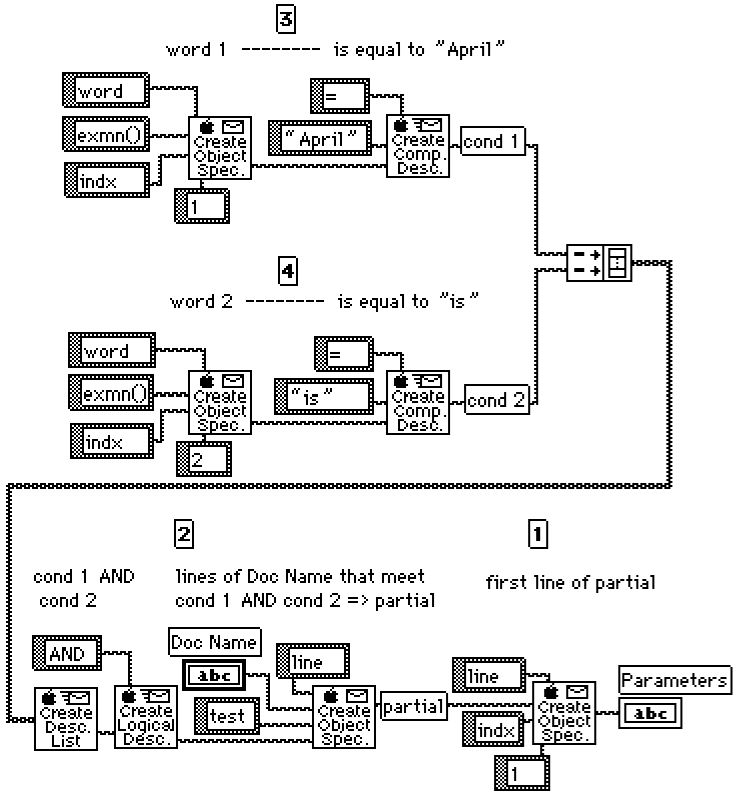
null data	Coerce an empty string to no type.	null	( )	n/a

AppleEvent Parameter Creation Using Object Specifiers

Apple has created a high-level interface for creating AppleEvents called the Object Support Library. This interface is actually layered on top of the AppleEvent parameter data structures described in the Creating AppleEvent Parameters topic. This interface helps create common types of parameters, including range specifications. LabVIEW object support VIs are located on the Low Level Apple Events palette.

Object Support VI Example

The following example creates an AppleEvent parameter using the object support VIs. This example creates an AppleEvent parameter to be sent to a word processor, asking the word processor to return the first line of a specified document whose first word is `April` and whose second word is `is`.



The following string that the previous diagram creates is quite complicated; tabs are added to make the string easier to read. For further information about the Object Support Library consult the *AppleEvent Registry*.

```
obj {
want: type(line),
from: obj {
want: type(line),
from: Doc Name,
form: test,
seld: logi {
term:  [
cmpd{
relo:=,
obj1:April,
obj2:obj {
want: type(word),
from: exmn( ),
form: indx,
seld: 1
}
},
cmpd{
relo:=,
obj1:is,
obj2:obj {
want: type(word),
from: exmn( ),
form: indx,
seld: 2
}
}
],
logc: AND
}
},
form: indx,
seld: 1
}
```

Sending AppleEvents to LabVIEW from Other Applications

LabVIEW responds to required AppleEvents, which Apple expects all System 7 applications to support, and to LabVIEW specific AppleEvents, designed specifically for LabVIEW. Both categories are described in the following topics.

Required AppleEvents
LabVIEW Specific AppleEvents
Replies to AppleEvents

LabVIEW Specific AppleEvents

LabVIEW also responds to the LabVIEW specific AppleEvents Run VI, Abort VI, VI Active?, and Close VI. With these events and the Open Documents AppleEvent, you can use other applications to programmatically tell LabVIEW to open a VI, run it, and close it when it is finished. A thorough understanding of AppleEvents, as described in Inside Macintosh, Volume VI, and the *AppleEvent Registry* is a prerequisite for sending these AppleEvents to LabVIEW from other applications. You can send these events between two or more LabVIEW applications by using the utility VIs described in the Sending AppleEvents topic.

The LabVIEW specific AppleEvents are described in later topics, in a format similar to that used in the *AppleEvent Registry*.

Replies to AppleEvents

If LabVIEW is unable to perform an AppleEvent, the reply contains an error code. If the error is not a standard AppleEvent error, the reply also contains a string describing the error. The LabVIEW Specific Error Codes for AppleEvent Messages summarizes the LabVIEW specific errors that can be returned in a reply to an AppleEvent.

Event: Run VI
Event: Abort VI
Event: VI Active?
Event: Close VI

Event: Run VI

Description

Tells LabVIEW to run the specified VI(s). Before executing this event, the LabVIEW application must be running, and the VI must be open (you can open the VI using the Open Documents AppleEvent).

Event Class

LBVW (Custom events use the Applications creator type for the event class)

Event ID

GoVI ----

Event Parameters

Description	Keyword	Default Type
VI or List of VIs	`keyDirectObject(----)`	typeChar (char) (required) or list of typeChar (list)

Reply Parameters

Description	Keyword	Default Type
none		

Possible Errors

Error	Value	Description
`kLVE_InvalidState`	1000	The VI is in a state that does not allow it to run.
`kLVE_FPNotOpen`	1001	The VI front panel is not open.
`kLVE_CtrlErr`	1002	The VI has controls on its front panel that are in an error state.
`kLVE_VIBad`	1003	The VI is broken.
`kLVE_NotInMem`	1004	The VI is not in memory.

Event: Abort VI

Description

Tells LabVIEW to abort the specified VI(s). Before executing this event, the LabVIEW application must be running, and the VI must be open (you can open the VI using the Open Documents

AppleEvent). This message can only be sent to VIs that are executed from the top-level (subVIs are aborted only if the calling VI is aborted).

Event Class

LBVW	(Custom events use the Applications creator type for the event class)

Event ID

RsVI

Event Parameters

Description	Keyword	Default Type
VI or List of VIs	`keyDirectObject (----)`	typeChar (char) (required) or list of typeChar (list)

Reply Parameters

Description	Required? Keyword	Default Type
none		

Possible Errors

Error	Value	Description
`kLVE_InvalidState`	1000	The VI is in a state that does not allow it to run.
`kLVE_FPNotOpen`	1001	The VI front panel is not open.
`kLVE_NotInMem`	1004	The VI is not in memory.

Event: VI Active?

Description

Requests information on whether a specific VI is currently running. Before executing this event, the LabVIEW application must be running, and the VI must be open (you can open the VI using the Open Documents AppleEvent). The reply indicates whether the VI is currently running.

Event Class

LBVW	(Custom events use the Applications creator type for the event class)

Event ID

VIAc

Event Parameters

Description	Keyword	Default Type
VI Name (required)	`keyDirectObject (----)`	typeChar (char)

Reply Parameters

Description	Required? Keyword	Default Type
Active?	`keyDirectObject (----)`	typeBoolean (required)

(bool)

Possible Errors

Error	Value	Description
`kAEvtErrFPNotOpen`	1001	The VIs front panel is not open.
`kLVE_NotInMem`	1004	The VI is not in memory.

Event: Close VI

Description

Tells LabVIEW to close the specified VI(s). Before executing this event, the LabVIEW application must be running, and the VI must be open (you can open the VI using the Open Documents AppleEvent).

Event Class

LBVW (Custom events use the Applications creator type for the event class)

Event ID

ClVI

Event Parameters

Description	Keyword	Default Type
VI or List of VIs	`keyDirectObject` (----)	typeChar (char) (required) or list of typeChar (list)
Save Options (not required)	`keyAESaveOptions` (savo)	typeEnum (enum) possible values: yes and no

Reply Parameters

Description	Keyword	Default Type
none		

Possible Errors

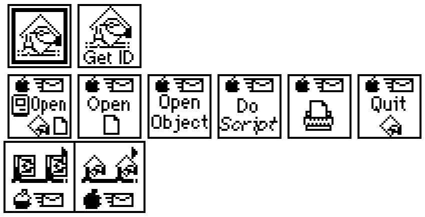
Error	Value	Description
`kAEvtErrFPNotOpen`	1001	The VIs front panel is not open.
`kLVE_NotInMem`	1004	The VI is not in memory.
`cancelError`	43	The user cancelled the close operation.

AppleEvents VI Descriptions

Click here to access the AppleEvents VI Overview (Macintosh) topic.

The following illustration shows the AppleEvents VI palette, which you access by selecting **Functions»Communication»AppleEvent**.

Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.



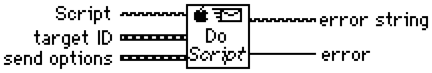
AESend Do Script
AESend Finder Open
AESend Open
AESend Open Document
AESend Print Document
AESend Quit Application
Get Target ID
PPC Browser

Subpalette Descriptions
LabVIEW Specific AppleEvent VI Descriptions
Low Level AppleEvent VI Descriptions

For examples of how to use the AppleEvent VIs, see the examples located in `examples\comm\AE Examples.llb`.

AESend Do Script

Sends the Do Script AppleEvent to a specified target application.



TF **Script** is a string containing instructions that the target application understands. It is typically in a language specific to the target application. An example of an application with a script language is Claris HyperCard.

TF **target ID** is a cluster of information describing the target application and its location. See the Target ID topic for a further description of this cluster.

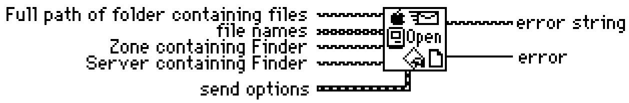
TF **send options** is a cluster that specifies whether the target application can interact with the user and the length of the AppleEvent timeout. See the Send Options topic for a discussion of the **send options** parameters.

abc **error string** describes error information.

I32 **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the AppleEvent Error Codes topic for more information.

AESend Finder Open

Sends the AppleEvent to open specified applications or documents to the System 7 Finder on the specified machine.



TF **Full path of folder containing files** describes the location of the folder that contains the files for the Finder to open. This string must end in a colon (:).

[abc] **file names** is an array containing the names of the files in the folder described by Full path of folder containing files. The Finder opens these files.

TF **Zone containing Finder** describes the AppleTalk zone where the target Finder resides. If this string is empty, this VI assumes that the zone is the same as that of the host Macintosh**.**

TF **Server containing Finder** describes the name of the Macintosh where the target Finder resides. If this string is empty, this VI assumes that the server is the host computer Finder.

TF **send options** is a cluster that specifies whether the Finder can interact with the user and the length of the AppleEvent timeout. See the Send Options topic for a discussion of the **send options** parameters.

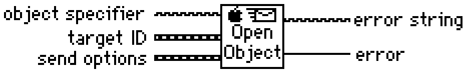
TF **error string** describes error information.

I32 **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the AppleEvent Error Codes topic for more information.

Note: **Apple may change the set of AppleEvents to which the Finder responds so that they more closely conform to the standard set of AppleEvents. As a result, the AppleEvent that AESend Finder Open sends to the Finder may not be supported in future versions of the system software.**

AESend Open

Sends the Open AppleEvent to a specified target application.



TF **object specifier** specifies the object that LabVIEW opens in the target application.

TF **target ID** is a cluster of information describing the target application and its location. See the Target ID topic for a further description of this cluster.

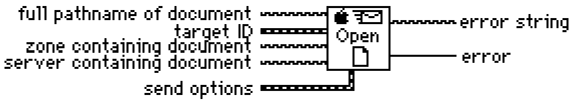
TF **send options** is a cluster that specifies whether the target application can interact with the user and the length of the AppleEvent timeout. See the Send Options topic for a discussion of the **send options** parameters.

TF **error string** describes error information.

I32 **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the AppleEvent Error Codes topic for more information.

AESend Open Document

Sends the Open Document AppleEvent to the specified target application, telling the application to open the specified document.



TF **full pathname of document** describes the location of the document that the application opens.

TF **target ID** is a cluster of information describing the target application and its location. See the Target ID topic for a further description of this cluster.

TF **zone containing document** describes the location of the document that the application opens. Notice that the application and document can reside in different locations. If **zone containing document** and **server containing document** are blank, AESend Open Document assumes the document is on the host computer.

TF **server containing document** describes the location of the document that the application opens. Notice that the application and document can reside in different locations. If **zone containing document** and **server containing document** are blank, AESend Open Document assumes the document is on the host computer.

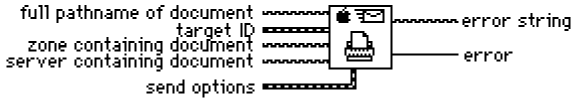
TF **send options** is a cluster that specifies whether the target application can interact with the user and the length of the AppleEvent timeout. See the Send Options topic for a discussion of the **send options** parameters.

TF **error string** describes error information.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the AppleEvent Error Codes topic for more information.

AESend Print Document

Sends the Print Document AppleEvent to the specified target application, telling the application to print the specified document.



TF **full pathname of document** describes the location of the document that the application prints.

TF **target ID** is a cluster of information describing the target application and its location. See the Target ID topic for a further description of this cluster.

TF **zone containing document** describes the location of the document that the application prints. Notice that the application and document can reside in different locations. If **zone containing document** and **server containing document** are blank, the Print Documents AppleEvent assumes the document is on the host computer.

TF **server containing document** describes the location of the document that the application prints. Notice that the application and document can reside in different locations. If **zone containing document** and **server containing document** are blank, the Print Documents AppleEvent assumes the document is on the host computer.

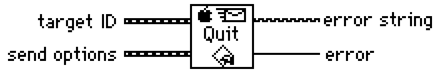
TF **send options** is a cluster that specifies whether the target application can interact with the user and the length of the AppleEvent timeout. See the Send Options topic for a discussion of the **send options** parameters.

TF **error string** describes error information.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the AppleEvent Error Codes topic for more information.

AESend Quit Application

Sends the Quit Application AppleEvent to a specified target application.



TF **target ID** is a cluster of information describing the target application and its location. See the Target ID topic for a further description of this cluster.

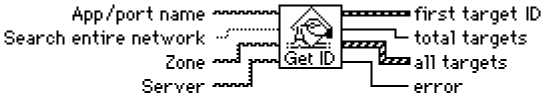
TF **send options** is a cluster that specifies whether the target application can interact with the user and the length of the AppleEvent timeout. See the Send Options topic for a discussion of the **send options** parameters.

error string describes error information.

error if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the AppleEvent Error Codes topic for more information.

Get Target ID

Returns a target ID for a specified application based on its name and location. You can either specify the applications name and location or the VI searches the entire network for the application.



App/port name is the name of the application for which you want the target ID. This parameter must exactly match the name of the application.

Search entire network determines whether the VI searches the entire network for the specified application. See the table in this topic for a further discussion of search options.

Zone specifies the target computerÕs zone that the VI searches. See the table in this topic for a further discussion of search options.

Server specifies the server that the VI searches. See the table in this topic for a further discussion of search options.

first target ID contains the target ID of the first application found whose name matches **App/port name**.

total targets is the total number of applications that the VI finds whose names match **App/port name**.

all targets contains an array of the target IDs of all matching applications.

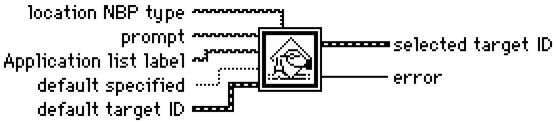
error is non-zero only if something goes wrong during the search. Finding zero targets is not necessarily an error if there are no applications with the specified name running and available on the network. See the AppleEvent Error Codes topic for more information.

The following table summarizes the operation of **Search entire network**, **Zone**, and **Server**:

To search the following locations:	**Use the following parameters:**
The current computer	Zone and Server must be unwired. Search entire network must be FALSE.
A specific computer on the network	Zone and Server must specify the target computerÕs zone and server. (If you do not wire Zone, the VI searches the current zone.) Search entire network must be FALSE.
A specific zone	**Zone** must specify the zone to be searched. **Server** must be unwired. **Search entire network** must be FALSE.
The entire network	**Search entire network** must be TRUE. The VI ignores **Zone** and **Server**.

PPC Browser

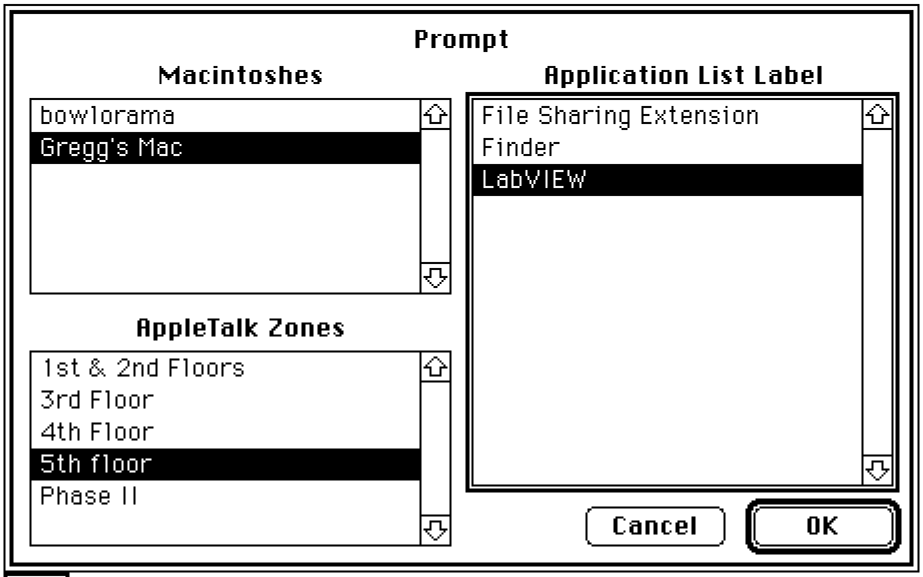
Invokes the PPC Browser dialog box for selecting an application on a network or on the same computer



TF **location NBP type** determines which computers on the network LabVIEW displays in the dialog box. If this string is empty (default), only computers with applications using the PPC Toolbox appear in the dialog box.

TF **prompt** is the message that LabVIEW displays in the dialog box. If this string is empty (default), Choose a program to link to: appears in the dialog box.

You can use this standard Macintosh dialog box to select a zone from the network, an object in that zone (in System 7, this is typically the name of a persons computer), and an application. The VI then returns the **target ID** cluster.



Prompt

Macintoshes

bowlorama
Gregg's Mac

AppleTalk Zones

1st & 2nd Floors
3rd Floor
4th Floor
5th floor
Phase II

Application List Label

File Sharing Extension
Finder
LabVIEW

TF **Application list label** is the title that appears for the list of PPC ports. If this string is empty (default), the title Programs appears.

TF **default specified** determines whether LabVIEW uses the **default target ID** parameter.

TF **default target ID** identifies and highlights the target that PPC Browser attempts to find.

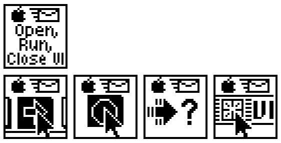
TF **target ID** is a cluster that the VI returns after you make selections from the dialog box. See the Target ID topic for a further description of this cluster.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. **error** 128 signifies that the user cancelled the dialog box. See the AppleEvent Error Codes topic for more information.

LabVIEW Specific AppleEvent VI Descriptions

LabVIEW specific AppleEvent VIs send messages that only LabVIEW applications (standard and run-time systems) recognize. You can access the LabVIEW Specific AppleEvents VIs by selecting **Functions»Communication»LabVIEW Specific AppleEvents**.

Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.

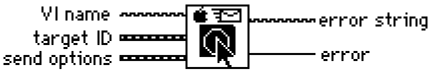


AESend Abort
AESend Close
AESend Open, Run, Close
AESend Run
AESend VI Active?

You should use these VIs only when communicating with LabVIEW applications. You can send these messages either to the current LabVIEW application or to a LabVIEW application on a network. See the AppleEvent Error Codes topic for more information.

AESend Abort

Sends the Abort VI AppleEvent to the specified target LabVIEW application.



TF **VI name** is the actual name of the VI, not its pathname. The VI must already be open. You can open the VI by using the AESend Open Document VI.

TF **target ID** is a cluster of information describing the target application and its location. See the Target ID topic for a further description of this cluster.

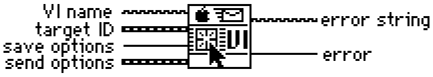
TF **send options** is a cluster that specifies whether the target application can interact with the user and the length of the AppleEvent timeout. See the Send Options topic for a discussion of the **send options** parameters.

TF **error string** describes error information.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. If **error** is 1000 the target VI is not running. See the AppleEvent Error Codes topic for more information.

AESend Close

Sends the Close VI AppleEvent to the specified target LabVIEW application.



TF **VI name** is the actual name of the VI, not its pathname. The VI must already be open. You can open the VI by using the AESend Open Document VI.

TF **target ID** is a cluster of information describing the target application and its location. See the Target ID topic for a further description of this cluster.

TF **save options** is an integer value that determines what the target LabVIEW should do if the VI has been modified. If **save options** has a value of 0, the target LabVIEW prompts for a save. If **save options** has a value of 1, the target LabVIEW saves the VI (if modified) without a prompt. If **save options** has a value of 2, the target LabVIEW does not save the VI.

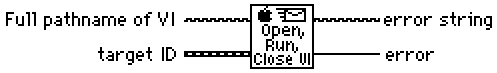
TF **send options** is a cluster that specifies whether the target application can interact with the user and the length of the AppleEvent timeout. See the Send Options topic for a discussion of the **send options** parameters.

TF **error string** describes error information.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the AppleEvent Error Codes topic for more information.

AESend Open, Run, Close

Uses the Open Document, Run VI, VI Active?, and Close VI AppleEvent VIs to make a specified LabVIEW application open, run, and close a VI.



For this VI, you must specify the complete pathname of the VI you want to run. See Path Controls and Refnums, for a description of path controls and indicators available in the Controls palette.

TF **Full pathname of VI** describes the full path of the VI that LabVIEW opens, runs, and closes.

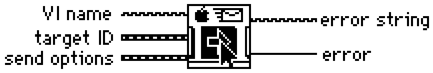
TF **target ID** is a cluster of information describing the target application and its location. See the Target ID topic for a further description of this cluster.

TF **error string** describes error information.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the AppleEvent Error Codes topic for more information.

AESend Run

Sends the Run VI AppleEvent to the target LabVIEW application.



TF **VI name** is the actual name of the VI, not its pathname. The VI must already be open. You can open it by using the AESend Open Document VI.

TF **target ID** is a cluster of information describing the target application and its location. See the Target ID topic for a further description of this cluster.

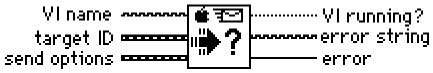
TF **send options** is a cluster that specifies whether the target application can interact with the user and the length of the AppleEvent timeout. See the Send Options topic for a discussion of the **send options** parameters.

TF **error string** describes error information.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the AppleEvent Error Codes topic for more information.

AESend VI Active?

Sends the VI Active? AppleEvent to the specified target LabVIEW application. **VI running?** is a Boolean indicating whether the VI is currently executing.



TF **VI name** is the actual name of the VI, not its pathname. The VI must already be open. You can open the VI by using the AESend Open Document VI.

TF **target ID** is a cluster of information describing the target application and its location. See the Target ID topic for a further description of this cluster.

TF **send options** is a cluster that specifies whether the target application can interact with the user and the length of the AppleEvent timeout. See the Send Options topic for a discussion of the **send options** parameters.

TF **VI running?** is TRUE if the specified VI is currently running as a top level VI.

TF **error string** describes error information.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the AppleEvent Error Codes topic for more information.

Low Level AppleEvent VI Descriptions

Click here to select the AppleEvents VI Overview (Macintosh) topic.

You can use the VIs in this topic to construct AppleEvent parameters and send the AppleEvent. The high-level VIs for sending AppleEvents, described earlier in this chapter, are based on the AESend VI, and are good examples of creating AppleEvents and their parameters.

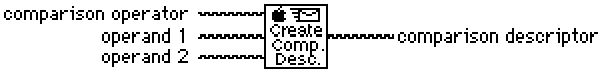
You can access the Low Level Apple Events palette, by selecting **Functions» Communication»Low Level Apple Events**.

Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.

AECreate Comp Descriptor
AECreate Descriptor List
AECreate Logical Descriptor
AECreate Object Specifier
AECreate Range Descriptor
AECreate Record
AESend
Make Alias

AECreate Comp Descriptor

Creates a string describing an AppleEvent comparison record, which specifies how to compare AppleEvent objects with another AppleEvent object or a descriptor record.



For example, you can use the output comparison descriptor string as an argument to the AESend VI, or as an argument to AECreate Object Specifier to build a more complex descriptor string. See the Object Support VI Example topic for an example of its use.

TF **comparison operator** is a descriptor type string that describes the comparison operation to perform on the operands. The standard set of comparisons are:

> `kAEGreaterThan`The value of operand1 is greater than the value of operand 2.
>= `kAEGreaterThanEquals`The value of operand1 is greater than or equal to the value of operand 2.
= `kAEEquals`The value of operand 1 is equal to the value of operand 2.
< `kAELessThan`ÐThe value of operand 1 is less than the value of operand 2.
<= `kAELessThanEquals`The value of operand1 is less than or equal to the value of operand 2.
bgwt `kAEBeginsWith`The value of operand 1 begins with the value of operand 2 (for example, the string operand begins with the string opera).
ends `kAEEndsWith`The value of operand1 ends with the value of operand 2. For example, the string operand ends with the string and.
cont `kAEContains`The value of operand1 contains the value of operand 2. For example, the

string operand contains the string era.

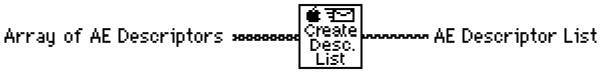
TF **operand 1** is an AppleEvent object specifier descriptor string. It specifies the first object in the comparison to perform.

TF **operand 2** is an AppleEvent record descriptor string. It can be an object specifier descriptor string or any other record descriptor string with a value to compare to the value of operand 1.

TF **comparison descriptor** is the AppleEvent descriptor string.

AECreate Descriptor List

Creates a string describing a list of AppleEvent descriptors, which you can then use with the AESend VI. You commonly use Descriptor lists when you create the operands for a logical descriptor.

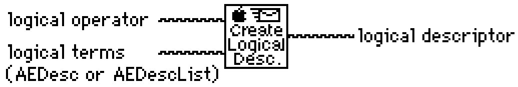


TF **Array of AE Descriptors** should contain AppleEvent descriptor strings, such as those output by the AECreateÉ series of AppleEvent VIs.

TF **AE Descriptor List** is an AppleEvent descriptor string with the correct syntax for a list of all the descriptors from the Array of AE Descriptors input array.

AECreate Logical Descriptor

Creates a string describing an AppleEvent logical descriptor, which you use with the AESend VI.



AppleEvent logical records describe logical, or Boolean expressions of multiple terms, such as the AND of two AppleEvent comparison records. For example, you can use the output logical descriptor string as an argument to the AESend VI, or as an argument to AECreate Object Specifier VI to build a more complex descriptor string. See the Object Support VI Example topic for an example of its use.

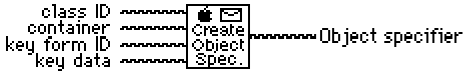
TF **logical operator** is a string describing the logical operation. The possible values are AND, OR, and NOT.

TF **logical terms** is an AppleEvent list descriptor string, such as the output by the AECreate Descriptor List VI. If the value of logical operator is AND or OR, this list can have any number of elements. If the value of logical operator is NOT, this list has only a single element.

TF **logical descriptor** is an AppleEvent descriptor string created from the inputs.

AECreate Object Specifier

Creates a string describing an AppleEvent object, which you use with the AESend VI.



An object specifier is an AppleEvent record whose type is obj and describes a specific object. It has four elements: the class of the object, the containing object, a code indicating the form of the description, and the description of the object.

TF **class ID** is a string that describes the class of the specified object. Examples of such strings are: `ccel` for class Cell, `ccol` for class Column, and `ctbl` for class Table (from the Table Suite of AppleEvents); and `cDB` for the class DataBase (from the DataBase Suite of AppleEvents).

TF **container** is an AppleEvent object specifier descriptor string that describes the containing object of the specified object. It should be another object specifier that this VI creates. If this string is left empty (the default value), the NULL object specifier is the container, and signifies the target application, which is the outermost container of any object specifier

key form ID is a string describing the form of the key data. It tells how to interpret the key data. The standard key forms are:

`prop` `FormPropertyID` means that the key data is the name of a property.

`name` `FormName` means that the key data is the name of the object.

`ID` FormUniqueID means that the key data is a unique identifier for the object.

`indx` `FormAbsolutePosition` means that the key data is a descriptor string for either a positive integer, indicating the offset of the requested element from the beginning of the container, or a negative integer, indicating its offset from the end of the container. The key data can also be a descriptor string for an absolute ordinal (type abso) with one of the following values: `firs`, `last`, `midd`, `any`, or `all`.

`rele` `FormRelativePosition` means the key data is a descriptor string for a relative position (type `enum`) with a value of `next` or `prev`.

`test` `FormTest` means the key data is a descriptor string for either a comparison record or a logical record (as created by either the AECreate Comp Descriptor VI or the AECreate Logical Descriptor VI).

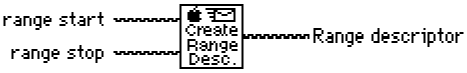
`rang` `FormRange` means the key data is a descriptor string for a range descriptor record (as created by the AECreate Range Descriptor VI.)

key data is a string describing the object. The value of the **key form ID** parameter determines its value.

Object specifier is the output descriptor string that the given inputs create. You can use it as the input for calls to the AECreate Object Specifier VI or anywhere that requires an object specifier.

AECreate Range Descriptor

Creates a string describing an AppleEvent range descriptor record, which you use with the AESend VI.



Range descriptor records are used in object specifiers whose key form is formRange (`rang`). They describe a range of objects with two object specifiers: the start and the end of the range

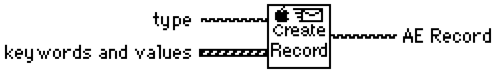
range start is an object specifier descriptor string that describes the beginning of the range.

range stop is a object specifier descriptor string that describes the end of the range.

range descriptor is the output descriptor string that the given inputs create.

AECreate Record

Creates a string describing an AppleEvent descriptor record, which can then be used with the AESend VI. You can use a record descriptor to bundle descriptors of different types. Each descriptor has its own keyword, or name, and value



type is a descriptor type string that describes the type of the AppleEvent record descriptor. Only the first four characters are significant, however it is acceptable to have more or less than four characters.

keywords and values is an array of clusters containing the strings that describe the elements of the record being created.

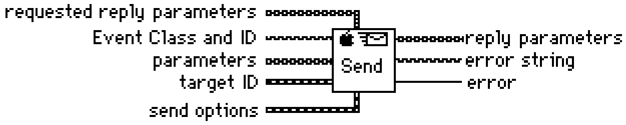
TF **keywords** is a descriptor type string that is the name of the record element. Only the first four characters are significant.

TF **values** is an AppleEvent descriptor string that is the value of the record element. It can be any descriptor string, such as those output by the AECreateÉ series of AppleEvent VIs.

TF **AE Record** is an AppleEvent record descriptor string created from the inputs. Other VIs requiring an AppleEvent descriptor string input can use this string.

AESend

Sends an AppleEvent specified in parameters to the specified target application.



TF r**equested reply parameters** is an array of strings containing a description of the reply parameters you want. Each element should be an eight character string. The first four characters constitute the keyword for the reply parameter. For example, ---- as a keyword specifies the direct object, or default parameter. The second four characters are the type of the parameter. For example, bool means the parameter is Boolean (TRUE or FALSE).

TF **Event Class and ID** is an eight character string containing two four character substrings. The first four characters specify the event class. For example, you can use aevt for the Required Suite, or core for the Core Suite. The second four characters specify the event ID. For example, you can use odoc for Open Document, or quit for Quit.

TF **parameters** is an array of AppleEvent descriptor strings for the arguments sent in the AppleEvent. The first four characters of each string are the keyword for the parameter. For example, the primary argument is called the direct object and always has the four character keyword ----. The descriptor string for that parameter follows the first four characters.

TF **target ID** describes the application to which the AppleEvent is being sent. See the Target ID topic for a further description of this cluster.

TF **send options** is a cluster describing options available for sending the AppleEvent. The most important option is the Want reply option. If this Boolean is TRUE, then the AESend VI waits to receive a reply from the target application. See the Send Options topic for a further description of **send options** cluster.

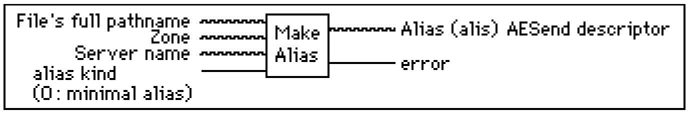
[abc] **reply parameters** is an output array of strings that corresponds to the input requested reply parameters. Each string in this array is a descriptor string for the reply parameter sent back by the target application.

TF **error string** usually gives a more meaningful description of errors that occur when you send the AppleEvent.

TF **error** is the error number of any error that occurs when you send the AppleEvent. Errors that occur when a parameter is incorrectly specified, have corresponding error codes in the 12346 to 12364 range. In this case, the error string describes in more detail what was wrong with the parameter. Errors in the range -1700 to -1732 indicate that something went wrong in the creation, sending or receiving of the AppleEvent. This can indicate a problem either in LabVIEW or in the target application. Errors in the range -900 to -932 indicate that something went wrong at the AppleEvent transport layer, the PPC Toolbox. This means the connection could not be established between LabVIEW and the target application.

Make Alias

Creates a unique description of a file from its pathname and location on the network. You can use this description with the AESend VI when sending an AppleEvent that refers to a file.



An alias is a data structure used by the Macintosh toolbox to describe file system objects (files, directories and volumes). Do not confuse this with a Finder™ alias file. A minimal alias contains a full path name to the file and possibly the zone and server that the file resides on. A full alias contains more information, such as creation date, file type, and creator. (The complete description of the structure of an alias is confidential to Apple Computer.) Aliases are the most common way to specify a file system object as a parameter to an AppleEvent.

TF **Files full pathname** describes the file or folder. It includes any information about where the file or folder resides on the network.

TF **Zone** is the AppleTalk zone where the server machine resides. If Server name is empty this string is unused.

TF **Server name** is the name of the machine where the file or folder resides

TF **alias kind** describes the alias. The possible values are:

0: Minimal alias. Uses Zone and Server. You cannot use it in an AppleEvent sent to the Finder™. The VI creates the alias from scratch, and does not check to see whether the file actually exists or is accessible from the desktop. It must have a volume name with a colon following it.

1: Full alias. Ignores Zone and Server. You can use it in AppleEvents sent to any application, including the Finder™. The VI creates this alias from scratch. If the file does not exist or is not accessible from the desktop, it returns an error.

2: From Finder™ alias file. Ignores Zone and Server. You can use it in AppleEvents sent to any application, including the Finder™. Files full pathname specifies a Finder™ alias file that points to the specific file. Finder™ alias files contain full aliases, and does not check to see whether the file actually exists or is accessible from the desktop. The VI copies the contents of the alias file to create the output alias.

TF **Alias** is the AppleEvent descriptor string.

TF **error** describes any errors that occur.

Get Target ID VI

Get Target ID

PPC Browser VI

PPC Browser

AESend Do Script VI

AESend Do Script

AESend   Finder Open VI

AESend Finder Open

AESend Open VI

AESend Open

AESend Open Document VI

AESend Open Document

AESend Print Document VI

AESend Print Document

AESend Quit Application VI

AESend Quit Application

LabVIEW Specific AppleEvent Subpalette

LabVIEW Specific AppleEvent VI Descriptions

Low Level Apple Events Subpalette

Low Level AppleEvent VI Descriptions

AESend Abort VI

AESend Abort

AESend Close VI

AESend Close

AESend Open, Run, Close VI

AESend Open, Run, Close

AESend Run VI

AESend Run

AESend VI Active? VI

AESend VI Active?

Advanced Topics

AESend VI

AESend

Make Alias VI

Make Alias

AECreate Comp Descriptor VI

AECreate Comp Descriptor

AECreate Logical Descriptor VI

AECreate Logical Descriptor

AECreate Object Specifier VI

AECreate Object Specifier

AECreate Range Descriptor VI

AECreate Range Descriptor

AECreate Descriptor List VI

AECreate Descriptor List

AECreate Record VI

AECreate Record

Low Level AppleEvent Subpalette

Low Level AppleEvent VI Descriptions

Required AppleEvents

LabVIEW responds to the required AppleEvents, which are Open Application, Open Documents, Print Documents, and Quit Application. These events are described *in Inside Macintosh*, Volume VI.

Program to Program Communication VI Overview

This topic describes the LabVIEW VIs for Program to Program Communication (PPC), a low-level form of Apple IAC by which Macintosh applications send and receive blocks of data.

Click here to access the PPC VI Descriptions topic.

Introduction to PPC

Program to Program Communication (PPC) is a high performance protocol for transferring blocks of data between applications. You can use it to create VIs that act as clients or servers. Although supported by all Macintoshes running System 7.x, it is not commonly used by most Macintosh applications. Instead, most Macintosh applications use AppleEvents, for sending commands between applications, to communicate.

LabVIEW VIs can use PPC to send and receive large amounts of information between applications on the same computer or different computers on a network. For two applications to communicate with PPC, they must both be running and prepared to send or receive information. To launch an application remotely, you can use the AESend Finder Open VI.

Although PPC is not as commonly supported as AppleEvents, it does provide some advantages. PPC is a higher performance protocol than AppleEvents because PPC requires less overhead to transmit information. Also, in LabVIEW you can create VIs that use PPC to act as clients or servers. You cannot create diagrams that act as AppleEvent servers. However, because PPC does not define the form or meaning of information that it transfers, it is more complicated to use.

PPC is similar in structure to TCP, in terms of both server and client applications. The PPC method for specifying a remote application is different from the TCP method. Other than that, the two protocols provide similar performance and features. Both protocols handle queueing and reliable transmission of data. You can use both protocols with multiple open connections.

In deciding between TCP and PPC, the main point to consider is which platforms you plan to run your VIs on, and with which platforms you can communicate. If your application is Macintosh only, PPC is a good choice, because it is built into the operating system. TCP is built into Macintosh operating system version 7.5. To use TCP with an earlier system you must buy a separate TCP/IP driver from Apple. If buying the separate driver is not an issue, then you may want to use TCP, because the TCP interface is simpler than PPC. PPC uses some fairly complicated data structures to describe addresses.

If your application must communicate with other platforms or run on other platforms, then you should use TCP/IP.

General PPC Behavior

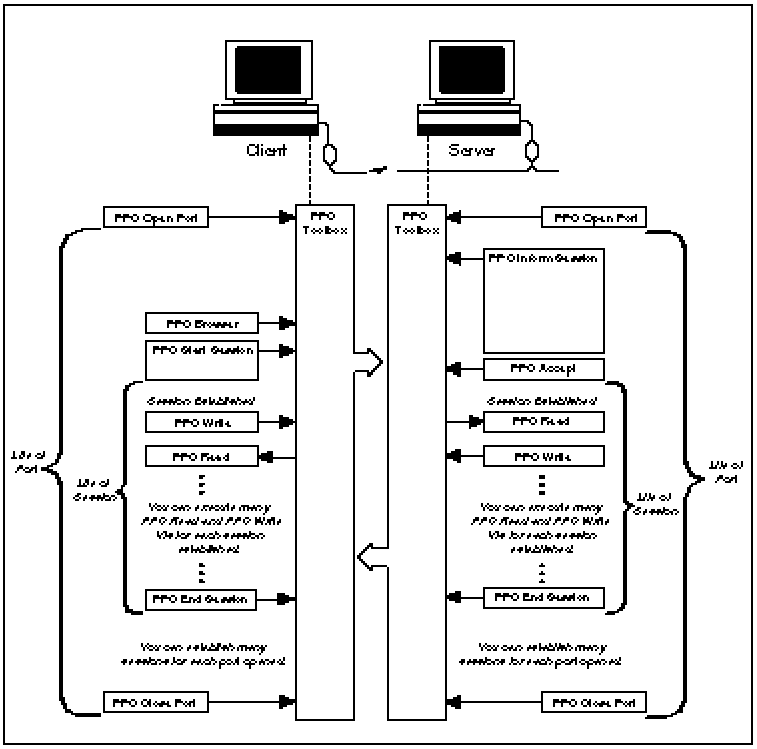
To communicate using PPC, each application must open a named *port*, over which communication sessions are established, as shown in the following figure. The application that requests communication is the *client*; and the application with which the client communicates is the *server*. The server application makes its availability known by issuing a PPC Inform Session operation. The client requests a session with the server application, which can either accept or reject the request. If the server application accepts the request, then the system establishes a session and the two applications can send and receive blocks of information between them. When the applications finish communicating, you should end the session. You may also want to close the port if you do not want to establish more sessions with that port.

You use the PPC Open Port VI to open a port for communication. PPC Open Port returns a port reference number, which you use in subsequent operations relating to that port. You can have multiple ports open simultaneously, as long as they each have a different name. Each port can support multiple sessions.

You can initiate a session using the PPC Start Session VI. You pass PPC Start Session a target ID and the port reference number through which you want to communicate. If the target application accepts the session, PPC Start Session returns a session reference number, which you use in subsequent communication for that session. PPC Start Session also incorporates an authentication (password) mechanism.

To receive session requests, use the PPC Inform Session VI. You can configure this VI to accept all requests automatically, or you can decide whether to accept or reject the request based on the information about the requesting application that this VI returns. You should accept or reject the request using the PPC Accept Session VI immediately, because the other computer waits (hangs) until you accept or reject its attempt to initiate a session, or until an error occurs.

When a session is established, you can use the PPC Write and PPC Read VIs to communicate with the other application. When you are finished with a session, you should execute the PPC End Session VI and close the port using the PPC Close Port VI.



Ports, Target IDs, and Sessions

To communicate using PPC, both clients and servers must open ports that they use for subsequent communication. The Open Port VI opens the port using a cluster that contains, among other things, the name that you want to use for the port.

Ports are used to distinguish between different services that an application provides. Each application can have multiple ports open simultaneously.

Each port can support several simultaneous sessions or conversations. To open a session, a client uses a Target ID indicating the location of the server. PPC uses the same type of Target ID that the AppleEvent VIs use. You can use the PPC Browser or the Get Target ID VIs to generate the Target ID for the remote application.

A server waits for clients to attempt to open a session by using the PPC Inform Session VI. The server can accept or reject the session by using the PPC Accept Session VI.

A client can attempt to open a session with a server by using the PPC Start Session VI.

After the session is started, you can use the PPC Read and PPC Write VIs to transfer data. You can close a session using PPC End Session, and you can close a port using the PPC Close Port VI.

PPC Client Example
PPC Server Example

PPC Client Example

The following discussion explains how you can use PPC to fulfill each component of the general Client model.

Open Conn to Srvr Use the PPC Open Connection and PPC Open Session VIs to open a connection to a server. This requires that you specify the Target ID of the server, which you can get by using either the PPC Browser VI or the Get Target ID VI. The end result is a port refnum and a session refnum, which are used to communicate with the server.

Exec Cmd on Srvr To execute a command on the server, use the PPC Write VI to send the command to the server. Next, use the PPC Read VI to read the results from the server. With the PPC Read VI, you must specify the number of characters you want to read. As with TCP, this can be awkward, because the length of the response can vary. The server can have a similar problem, because the length of a command may vary. Following are several methods for addressing the problem of varying sized commands. These methods can also be used with TCP.

- Precede the command and the result with a fixed size parameter that specifies the size of the command or result. In this case, read the size parameter, and then read the number of characters specified by the size. This option is efficient and flexible.
- Make each command and result a fixed size. When a command is smaller than the size, you can pad it out to the fixed size.
- Follow each command and result with a specific terminating character. To read the data, you then need to read data in small chunks until you get the terminating character.

Close Conn to Srvr Use the PPC Close Session and PPC Close Connection VIs to close the connection to the server.

PPC Server Example

The following discussion explains how you can use PPC to fulfill each component of the general Server:

Initialize Server Use PPC Open Port in the initialization phase to open a communication port.

Wait for a Conn Use the PPC Inform Session VI to wait for a connection. With PPC, you can either automatically accept incoming connections, or you can choose to accept or reject the session by using the PPC Accept Session VI. This process of waiting for a session and then approving the session allows you to screen connections.

Wait for a Cmd When a connection is established, you can read from that session to retrieve a command. As was discussed in the PPC Client Example topic, you must decide the format for commands. If commands are preceded by a length field, then you need to first read the length field, and then read that amount of data.

Exec Cmd

Execution of a command should be protocol independent, because it is something done on the local computer. When finished, you pass the results to the next stage, where they are transmitted to the client.

Return Results

Use the PPC Write VI to return the result. As discussed in the PPC Client Example topic, the data must be formatted in a form that the client can accept.

Close Conn

Use the PPC Close Session VI to close the connection.

Shut Down Server

Finally, when the server is finished, Use the PPC Close Port VI to close the port that you opened in the initialization phase.

PPC Server with Multiple Connections

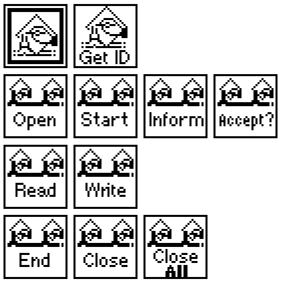
PPC handles multiple sessions and multiple ports easily. The methods for implementing each component of a server, as described in the preceding topic, also work for a server with multiple connections.

PPC VI Descriptions

Click here to access the Program to Program Communication VI Overview topic.

The following illustration shows the PPC VI palette, which you access by selecting **Functions»Communication»PPC**.

Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.

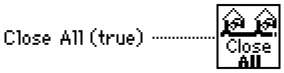


Close All PPC Ports
Get Target ID
PPC Accept Session
PPC Browser
PPC Close Port
PPC End Session
PPC Inform Session
PPC Open Port
PPC Read
PPC Start Session
PPC Write

For examples of how to use the PPC VIs, see the examples located in `examples\comm\PPC Examples.llb`.

Close All PPC Ports

Closes all the PPC ports that the PPC Open Port VI opened.



Closing a port terminates all outstanding calls associated with the port with a portClosedErr (error -916).

You can use the Close All PPC Ports to handle abnormal conditions that leave ports open. An example of an abnormal condition is when a VI is aborted before it can terminate normally and close the PPC port. You can use the Close All PPC Ports VI during VI development, when such mistakes are more likely to be made, or as a precaution at the beginning of any program that opens ports.

TF **Close All (true)** if TRUE closes all the ports that the PPC Open Port VI opened.

Get Target ID

Returns a target ID for a specified application based on its name and location. You can either specify the application's name and location or the VI searches the entire network for the application.

TF

TF **App/port name** is the name of the application for which you want the target ID. This parameter must exactly match the name of the application.

TF **Search entire network** determines whether the VI searches the entire network for the specified application. See the table in this topic for a further discussion of search options.

TF **Zone** specifies the target computerÕs zone that the VI searches. See the table in this topic for a further discussion of search options.

TF **Server** specifies the server that the VI searches. See the table in this topic for a further discussion of search options.

TF **first target ID** contains the target ID of the first application found whose name matches **App/port name**.

TF **total targets** is the total number of applications that the VI finds whose names match **App/port name**.

TF **all targets** contains an array of the target IDs of all matching applications.

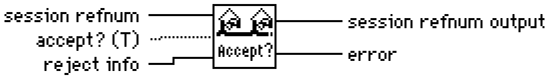
TF **error** is non-zero only if something goes wrong during the search. Finding zero targets is not necessarily an error if there are no applications with the specified name running and available on the network. See the AppleEvent Error Codes topic for more information.

The following table summarizes the operation of **Search entire network**, **Zone**, and **Server**:

To search the following locations:	Use the following parameters:
The current computer	Zone and Server must be unwired. Search entire network must be FALSE.
A specific computer on the network	Zone and Server must specify the target computerÕs zone and server. (If you do not wire Zone, the VI searches the current zone.) Search entire network must be FALSE.
A specific zone	**Zone** must specify the zone to be searched. **Server** must be unwired. **Search entire network** must be FALSE.
The entire network	**Search entire network** must be TRUE. The VI ignores **Zone** and **Server**.

PPC Accept Session

Accepts or rejects a PPC session request based on the Boolean **accept?**.



You should accept or reject the request using the PPC Accept Session VI immediately, because the other computer waits (hangs) until the VI accepts or rejects its attempt to initiate a session or an error occurs.

TF **session refnum** is a session reference number, which you use in subsequent communication for this session.

TF **accept? (T)** determines whether the VI accepts or rejects a PPC session.

TF **reject info** contains an application-defined value you return if you reject a session.

TF **session refnum output** is the same value as **session refnum** if **accept?** is TRUE. Otherwise, the value of **session refnum output** is 0.

error if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the PPC Error Codes topic for a list of PPC error codes and their descriptions.

PPC Browser

Invokes the PPC Browser dialog box for selecting an application on a network or on the same computer

location NBP type determines which computers on the network LabVIEW displays in the dialog box. If this string is empty (default), only computers with applications using the PPC Toolbox appear in the dialog box.

prompt is the message that LabVIEW displays in the dialog box. If this string is empty (default), Choose a program to link to: appears in the dialog box.

You can use this standard Macintosh dialog box to select a zone from the network, an object in that zone (in System 7, this is typically the name of a person's computer), and an application. The VI then returns the **target ID** cluster.

Application list label is the title that appears for the list of PPC ports. If this string is empty (default), the title Programs appears.

default specified determines whether LabVIEW uses the **default target ID** parameter.

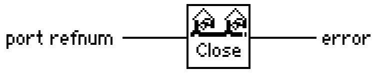
default target ID identifies and highlights the target that PPC Browser attempts to find.

target ID is a cluster that the VI returns after you make selections from the dialog box. See the Target ID topic for a further description of this cluster.

error if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. **error** 128 signifies that the user cancelled the dialog box. See the AppleEvent Error Codes topic for more information.

PPC Close Port

Closes the specified PPC port.



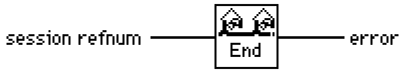
Closing a port terminates all outstanding calls associated with the port with a portClosedErr (error -916).

port refnum is a unique port reference number**.**

error if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See PPC Error Codes topic for a list of PPC error codes and their descriptions.

PPC End Session

Ends the specified PPC session.



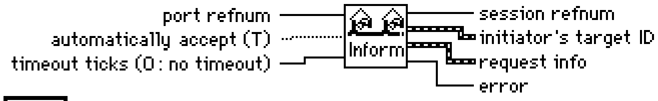
Ending a session causes all outstanding calls associated with the session (PPC Read and PPC Write calls) to finish with a sessClosedErr (error-917).

session refnum is a session reference number, which you use in subsequent communication for this session.

error if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the PPC Error Codes topic for a list of PPC error codes and their descriptions.

PPC Inform Session

Waits for a PPC session request.



I16 **port refnum** contains the unique port reference number identifying the port the VI closes if a timeout occurs or if the VI aborts before completing execution.

TF **automatically accept** if TRUE causes the VI to automatically accept any session request. Otherwise, accept or reject the request using the PPC Accept Session VI immediately, because the other computer waits (hangs) until the VI accepts or rejects its attempt to initiate a session or an error occurs.

TF **timeout ticks** if non-zero specifies the number of ticks PPC Inform Session waits for LabVIEW to establish a session before returning the errTimedOut error. One tick equals 1/60 of a second.

TF **session refnum** is a session reference number, which you use in subsequent communication for this session.

TF **initiators target ID** describes the application attempting to start a session.

TF **request info** is a cluster containing information about the user attempting to start a session. **request info** contains the following parameters in the order listed.

TF **user name**, a string, is the name of the user that is attempting to start a session.

U32 **user data** is an application-defined value that is the same as the **user data** value passed to PPC Start Session.

TF **request origin** indicates the origin of the application requesting a session.

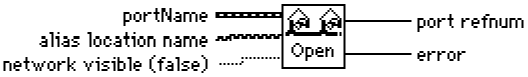
1: (Local Origin) The requesting application is on the same computer.
2: (Remote Origin) The requesting application is remote.

TF **service type** is a ring indicator. The **service type** is always 1 (Real Time) for the current version of Apple PPC protocol.

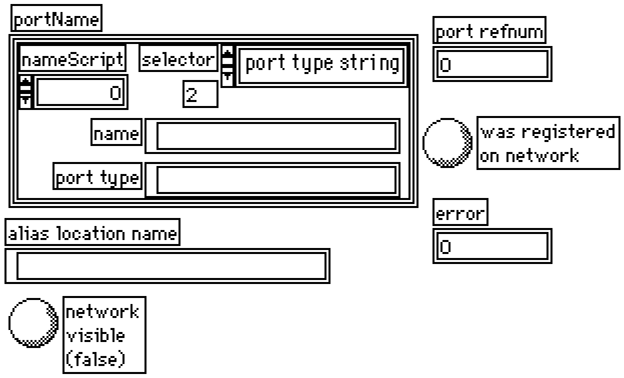
TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See PPC Error Codes topic for a list of PPC error codes and their descriptions.

PPC Open Port

Opens a port for PPC communication and returns a unique port reference number in **port refnum**. You can use a single port for multiple sessions.



When opening a port using PPC Open Port, you must specify a **portName** cluster.



portName is a cluster containing the following parameters in the order listed below.

nameScript is a 32-bit integer used in international localization that specifies the language system you are using. Use a **nameScript** value of 0 for Roman language systems (for example, English); consult *Inside Macintosh*, *Volume VI* for a list of available script codes.

selector describes the format of the **type string** parameter.

1: (creator/type) Signifies that **type string** is an 8-character string; the first four characters are the creator (for example, LBVW), and the last four characters define the port type.

2: (port type string) Signifies that **type string** is a 32-character (or less) description of the service provided by the port .

port type string is an 8-character string; the first four characters are the creator (for example, LBVW), and the last four characters define the port type, when **selector** has a value of 1. The **type string** is a 32-character (or less) description of the service that the port provides when **selector** has a value of 2. (In almost all cases, you should specify a value of 2 for **selector**, and use a description of the service provided by the port for **type string**. Consult *Inside Macintosh, Volume VI*, for more information about other cases.)

name is the name you give to the port. The value of **name**, which can be no more than 32 characters, is displayed in the PPC Browser dialog box list of port names. The Get Target ID VI uses **name** to identify the port.

alias location name establishes an alias name for the port. The PPC Browser uses this alias to determine which machines to display in its dialog box. If you leave this string empty the VI uses the default alias PPCToolBox**.**

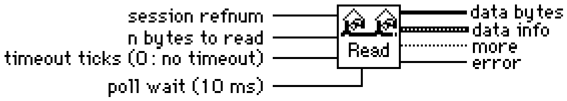
network visible determines whether the port is accessible to other machines on the network.

port refnum is a port reference number, which you use in subsequent operations relating to that port**.**

error if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the PPC Error Codes topic for a list of PPC error codes and their descriptions.

PPC Read

Reads a block of information from a specified session. If a timeout occurs or the VI aborts before completing execution, the port that **port refnum** represents closes.



PPC Read executes asynchronously by starting to read the specified data and then polling until the read is finished.

session refnum is a session reference number, which you use in subsequent communication for this session.

n bytes to read specifies the number of bytes the VI reads.

timeout ticks value, if non-zero, specifies the number of ticks the PPC Inform Session VI waits for a session to be established before returning the errTimedOut error. One tick equals 1/60 of a second.

poll wait (10 ms) determines how frequently PPC Read checks to see whether LabVIEW has read the data successfully.

data bytes is an array of unsigned 8-bit integers that is written by the sender.

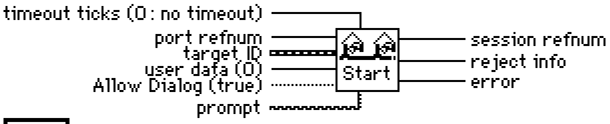
data info is a cluster of application-specific information that LabVIEW uses when reading and writing blocks of data in a PPC session. This cluster contains three 32-bit integers: **block creator**, **block type**, and **user data**. You can use these values to send information about the block of data to the receiving application.

TF **more** is a Boolean indicating whether more data exists for the given block that the VI reads. The application that writes the data can send the data in multiple pieces.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the PPC Error Codes topic for a list of PPC error codes and their descriptions.

PPC Start Session

Attempts to start a session with the application specified by **target ID** through the specified port. If a timeout occurs or the VI aborts before completing execution, the port represented by **port refnum** closes.



TF **timeout ticks (0: no timeout)** if non-zero specifies the number of ticks PPC Inform Session waits for a session to be established before returning the errTimedOut error. One tick equals 1/60 of a second.

TF **port refnum** is a unique reference number that specifies the port through which LabVIEW attempts to start a session with the application**.**

TF **target ID** is a cluster of information describing the target application and its location.

TF **user data** is an application-defined value that the VI sends with the request for a session.

TF **Allow Dialog (true)** if TRUE displays the User Identity Dialog Box if the target application requires authorization.

TF **prompt** appears in the dialog box.

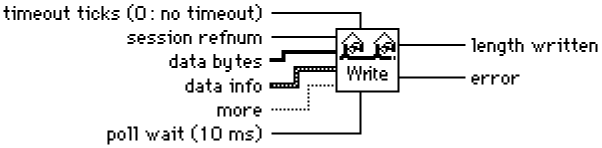
TF **session refnum** is a session reference number, which you use in subsequent communication for this session.

TF **reject info** contains an application-defined number if the target application rejects the request.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the PPC Error Codes topic for a list of PPC error codes and their descriptions.

PPC Write

Writes a block of information to the specified session. If a timeout occurs or the VI aborts before completing execution, the port represented by **port refnum** is closed. PPC Write executes asynchronously by starting to write the specified data and then polling until the write is finished.



TF **timeout ticks** if non-zero specifies the number of ticks the PPC Inform Session VI waits for LabVIEW to establish a session before returning the errTimedOut error. One tick equals 1/60 of a second.

TF **session refnum** is a session reference number, which you use in subsequent communication for this session.

[U8] **data bytes** is an array of unsigned 8-bit integers to send to the target application.

TF **data info** is a cluster of application-specific information you use when reading and writing blocks of data in a PPC session. **data info** contains three 32-bit integers: **block creator**, **block type**, and **user data**. You can use these values to give information about the block of data to the receiving application.

TF **more** should be TRUE if you want to write more data for a given block. For example, if you want to write a block of data in several calls to PPC Write, set **more** to TRUE on all but the last write of the sequence.

TF **poll wait (10 ms)** determines how frequently PPC Write checks to see whether LabVIEW has written the data successfully; for higher throughput, a value of zero is best.

TF **length written** is the actual number of bytes written. Except when the VI returns an error, **length written** should always be the length of the byte array input.

TF **error** if negative, indicates a Macintosh error. If positive, **error** indicates an error internal to the CIN that generated it. See the PPC Error Codes topic for a list of PPC error codes and their descriptions.

PPC Accept Session VI

PPC Accept Session

PPC Browser VI

PPC Browser VI

Close All PPC Ports VI

Close All PPC Ports

PPC Close Port VI

PPC Close Port

PPC End Session VI

PPC End Session

Get Target ID VI

Get Target ID

PPC Inform Session VI

PPC Inform Session

PPC Open Port VI

PPC Open Port

PPC Read VI

PPC Read

PPC Start Session VI

PPC Start Session

PPC Write VI

PPC Write

UDP VI Overview

This section describes a set of VIs that you can use with User Datagram Protocol (UDP), a protocol in the TCP/IP suite for communicating across a single network or interconnected set of networks.

Click here to access the UDP VI Descriptions topic.

User Datagram Protocol (UDP)
Using UDP

Note: If you are writing both the client and server, and your system can use TCP/IP, then TCP is probably the best protocol to use because it is a reliable, connection-based protocol. UDP is a connectionless protocol with higher performance, but it does not ensure reliable transmission of data.

User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) transmits data across networks. UDP can communicate to specific processes on a computer. When a process opens a network connection to a particular port it only receives datagrams that are addressed to that port on that computer. When a process sends a datagram, it must specify the computer and port as the destination.

There are several reasons why UDP is rarely used directly. UDP does not guarantee data delivery. Each datagram is routed separately, so datagrams may arrive out of order, be delivered more than once or not delivered at all.

Typically, UDP is used in applications where reliability is not critical. For example, an application might transmit informative data to a destination frequently enough that a few lost segments of data are not problematic.

Using UDP

UDP is not a connection-based protocol like TCP. This means that a connection does not need to be established with a destination before sending or receiving data. Instead, the destination for the data is specified when each datagram is sent. The system does not report transmission errors.

You can use the UDP Open VI to create a connection. A port must be associated with a connection when it is created so that incoming data can be sent to the appropriate application. The number of simultaneously open UDP connections depends on the system. UDP Open returns a Network Connection refnum, an opaque token used in all subsequent operations pertaining to that connection.

You can use the UDP Write VI to send data to a destination and the UDP Read VI to read it. Each write requires a destination address and port. Each read contains the source address and port. Packet boundaries are preserved. That is, a read never contains data sent in two separate write operations.

In theory, you should be able to send data packets of any size. If necessary, a packet is disassembled into smaller pieces and sent on its way. At their destination, the pieces are reassembled and the packet is presented to the requesting process. In practice, systems only allocate a certain amount of memory to reassemble packets. A packet that cannot be reassembled is thrown away. The largest size packet that can be sent without dissassembly depends on the network hardware.

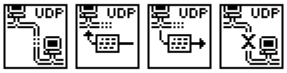
When LabVIEW finishes all communications, calling the UDP Close VI frees system resources.

UDP VI Descriptions

Click here to access the UDP VI Overview topic.

The following illustration shows the UDP VI palette, which you access by selecting **Functions»Communication»UDP**:

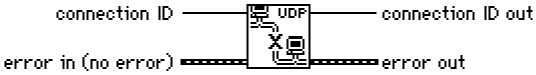
Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.



UDP Close
UDP Open
UDP Read
UDP Write

UDP Close

Closes the UDP connection specified by **connection ID**.



□ **connection ID** is a network connection refnum that identifies the UDP connection that you want to close.

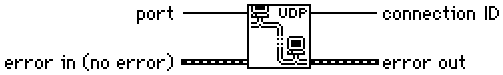
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

□ **connection ID out** has the same value as **connection ID**. If the connection is not aborted, the connection is still valid, and the remote machine can continue to send data.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

UDP Open

Attempts to open a UDP connection on the given **port**. **Connection ID** is an opaque token used in all subsequent operations relating to the connection.



U16 **port** is the local port with which you want to establish a UDP connection.

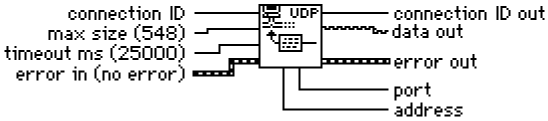
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **connection ID** is a network connection refnum that uniquely identifies the UDP connection. You use this **connection ID** value to refer to this connection in subsequent VI calls.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

UDP Read

Returns a datagram in the string **data out** that has been received on the UDP connection specified by **connection ID**.



TF **connection ID** is a network connection refnum that identifies the UDP connection. You use this **connection ID** value to refer to this connection in subsequent VI calls.

TF **max size (548)** is the maximum number of bytes to read.

TF **timeout** is in milliseconds. If the operation does not complete in the specified time, the VI completes and returns an error. The default value is 25,000. A timeout value of -1 means wait indefinitely.

TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **connection ID out** has the same value as **connection ID**. If the connection is not aborted, the connection is still valid, and the remote machine can continue to send data.

TF **data out** is a string that contains the data read from the UDP connection.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

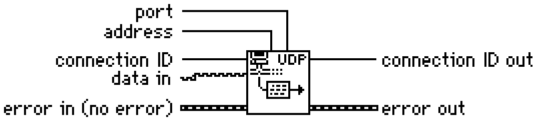
U16 **port** is the port of the UDP connection that sent the datagram.

TF **address** refers to the computer where a datagram originates.

address and **port** indicate the source of the datagram. If no data is received in the specified **timeout** period, a timeout error is reported in **error out**. Due to limitations of the MacTCP driver, the Macintosh has a time out resolution of 1 second and a minimum timeout of 2 seconds. **max size** is the maximum size to expect for the incoming datagram. If the incoming datagram is larger than **max size**, the datagram is truncated.

UDP Write

Writes the string **data in** to the remote UDP connection specified by **address** and **port**.



TF **port** is the port of the specified address where you want to send a datagram.

TF **address** refers to the computer where you want to send a packet.

TF **connection ID** is a refnum identifying the UDP connection.

TF **data in** is a string that contains the data to write to the UDP connection.

TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **connection ID out** has the same value as **connection ID**.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

In an Ethernet environment, restrict data to 8192 bytes. In a LocalTalk environment, restrict data to 1458 bytes because of gateway performance considerations.

UDP Close VI

UDP Close

UDP Open VI

UDP Open

UDP Read VI

UDP Read

UDP Write VI

UDP Write

DDE VI Overview (Windows)

This topic describes the LabVIEW VIs for Dynamic Data Exchange (DDE) for Windows 3.1, Windows 95, and Windows NT. These VIs execute DDE functions for sharing data with other applications that accept DDE connections.

Click here to access the DDE VI Descriptions topic.
Click here to access the DDE Server VI Descriptions topic.

Dynamic Data Exchange

Dynamic Data Exchange is a *client-controlled* data passing protocol. One application, the *client*, passes data to another application, the *server*.

Both applications must be running, and both must give Windows their callback function address before DDE communication can begin. The callback function accepts any DDE messages that Windows sends to the application.

A DDE client initiates a conversation with another application (a DDE server) by sending a connect message. After establishing a connection, the client can send commands or data to the server, or request data from the server.

A client can request data from a server by a *request* or an *advise*. A request is a single transfer of data. If the client wants to montor a value over a period of time, the client must use an advise. An advise establishes an active link between the two applications. The server then informs the client every time the advise value changes. When the client no longer needs the changed values, it sends an advise stop message to the server.

When all the DDE communication for the conversation is complete, the client sends a close conversation message to the server.

DDE is most appropriate for communication with standard, off-the-shelf applications, such as Microsoft Excel.

With LabVIEW, you can create VIs that act as clients to other applications (meaning they request or send data to other applications). You can also create VIs that act as servers that provide named information for access by other applications. As a server, LabVIEW does not use connection-based communication. Instead, you provide named information to other applications, which can then read or set the values of that information by name.

Using DDE as a Client

The Dynamic Data Exchange VIs give LabVIEW full DDE client capability.

To use DDE, you must first establish a conversation using the DDE Open Conversation VI. The VI must specify the *service* and the *topic*. The service usually corresponds to the name of the server application and the topic to the active file. DDE messages then carry data to or from specific locations in the active file. For more information on how a specific application handles topic names and data item locations, consult the documentation for that application.

When you have established a conversation, you can send data using the DDE Poke VI, send commands using the DDE Execute VI, obtain data with the DDE Request VI, or initiate an advise protocol with the DDE Advise Start VI.

The DDE Request VI sends a DDE message to the server every time you call it. The server must then check the data requested and return it in another DDE message. If your VI checks the value frequently, an advise protocol might be more efficient than a request.

The DDE Advise Start VI creates a local copy of the data value you are interested in. When you call the DDE Advise Check VI, the VI returns this value without sending any DDE messages. At the same time, the server application sends DDE messages every time the value changes, so that the local value is always current. If the value seldom changes but is often needed, an advise can significantly reduce the required number of DDE messages.

Caution: **During a conversation, you must pass the conversation refnum to all other DDE VIs involved in that conversation. Windows uses these refnums to identify the conversation. If you alter the conversation refnum, or do not specify or wire the conversation refnum, the VI fails. The same is true for the advise refnum. If you alter advise refnum, or do not specify or wire advise refnum for the DDE Advise Check VI or the DDE Advise Stop VI, the VIs fail and may cause a system failure.**

The DDE protocol used by LabVIEW is ASCII based, and the transmission is terminated when a null byte is reached. If the binary data has a null byte (00) in it, the transmission ends.

To send a number to another application, you must convert that number to a string. In the same way, you must convert numbers received through a request or advise from the string format. Use the conversion VIs from **Functions»String**. See String Functions for further information on how to use string conversion VIs.

Stop all advises and closes all conversations using DDE Advise Stop and DDE Close Conversation after all DDE commands have executed. This releases the system resources associated with these VIs.

Services, Topics, and Data Items

With TCP/IP, you identify the process you want to talk to by its computer address and a port number. With DDE, you identify the application you want to talk to by referencing the name of a service and a topic. The server decides on arbitrary service and topic names. A given server generally uses its application name for the service, but not necessarily. That server can offer several topics that it is willing to communicate. With Excel, for example, the topic might be the name of a spreadsheet.

To communicate with a server, first find the names of the service and topic that you want to discuss. Then open a conversation using these two names to identify the server.

Unless you are going to send a command to the server, you usually work with data items that the server is willing to talk about. You can treat these as a list of variables that the server lets you manipulate. You can change variables by name, supplying a new value for the variable. Or, you can request the values of variables by name.

Using DDE as a Server

The first step to becoming a DDE server is to use the DDE Srv Register Service VI to tell Windows what your service name and topic are going to be. At this point other applications can open DDE conversations with your service.

You can call the DDE Srv Register Service VI multiple times with different service names to establish multiple services or multiple times with the same service name but different topic names to establish multiple topics for one service.

After specifying your service and topic names, you can define items for that service using the DDE Srv Register Item VI. After this call, other applications can request or poke the item, as well as initiate advises on that item. LabVIEW fully manages all these transactions.

To change the value of an item, call the DDE Srv Set Item VI. This VI changes the value and informs all clients that have advises on them.

To monitor whether a client has changed an item with a poke, call the DDE Srv Check Item VI. This VI either returns the current value immediately or waits until a client changes the value. If a client pokes the value before DDE Srv Check Item is called with wait for poke true, DDE Srv Check Item returns immediately and reports that the value was poked.

You call the DDE Srv Unregister Item VI and the DDE Srv Unregister Service VI to close down your DDE server when you are finished. LabVIEW automatically disconnects any client conversations connected to your server when DDE Srv Unregister Service is called.

Using NetDDE

NetDDE is built into Windows for WorkGroups 3.11, Windows 95 and Windows NT. It is also available for Windows 3.1 with an add-on package from WonderWare. If you are using Windows 3.1 with the WonderWare package, consult the WonderWare documentation on how to use netDDE.

When you communicate over the network, the meaning of the service and topic strings change. The service name changes to indicate that you want to use networked DDE, and includes the name of the computer you want to communicate with. The service name is of the following form.

```
\\computer-name\ndde$
```

You can supply any arbitrary name for the topic. You then edit the `SYSTEM.INI` file to associate this topic name with the actual service and topic that you can use on the remote computer. This configuration also includes parameters that configure the network connection.

If you are using Windows for WorkGroups, Windows 95, or Windows NT, use the following instructions:

SERVER MACHINE
CLIENT MACHINE

SERVER MACHINE

Windows for Workgroups
Windows 95
Windows NT

Windows for Workgroups

Add the following line to the `[DDE Shares]` section of the file `system.ini` on the server (application receiving DDE commands):

```
lvdemo = service_name,topic_name,,31,,0,,0,0,0
```

where:

`lvdemo` can be any name
`service_name` is typically the name of the application, such as `excel`
`topic_name` is typically the specific file name, such as `sheet1`

enter other commas and numbers as shown.

Windows 95

Note: ***NetDDE is not automatically started by Windows 95. You need to run the program*** `\WINDOWS\NETDDE.EXE`***. (This can be added to the startup folder so that it is always started.)***

To set up a netDDE server on Windows 95:

1. Run`\WINDOWS\REGEDIT.EXE`
2. In the tree display, open the folder My `Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NetDDE\DD Shares`
3. Create a new DDE Share by selecting **Edit»New»Key** and give it the name `lvdemo`.
4. With the `lvdemo` key selected, add the required values to the share as follows. (For future reference, these keys are just being copied from the `CHAT$` share but you cannot cut, copy, or paste keys or values with `REDEGIT`.) Use **Edit»New** to add new values. When you create the key, the default value, named (`Default`) and a value of (`value not set`) appears. Leave these values alone and add the following:

Value Type	Name	Value
Binary	Additional item count	00 00 00 00
String	Application	service_name
String	Item	service_name
String	Password1	service_name
String	Password2	service_name
Binary	Permissions1	1f 00 00 00
Binary	Permissions2	00 00 00 00
String	Topic	topic_name

5. Close `REGEDIT`.
6. Restart the machine. (NetDDE must be restarted for changes to take affect.)

CLIENT MACHINE

On the client machine (application initiating DDE conversation) no configuration changes are necessary.

Use the following inputs to `DDE Open Conversation.vi`:

Service: `\\machine_name\ndde$`

Topic: `lvdemo`

where:

`machine_name` specifies the name of the server machine

`lvdemo` matches the name specified in the `[DDE Shares]` section on the server.

Consider the examples `Chart Client.vi` and `Chart Server.vi` found in `examples\network\ddeexamp.llb`. To use those VIs to pass information between two computers using netDDE, you should do the following:

Server Machine:

1. Do not modify any front panel values.
2. In the `system.ini` file of the Server machine, add the following line in the `[DDEShares]` section:
   `lvdemo = TestServer,Chart,,31,,0,,0,0,0`

   Client Machine:

   On the front panel, set the controls to the following:
   Service = `\\machine_name\ndde$`
   Topic = `lvdemo`
   Item = `Random`

   SERVER MACHINE

Examples of Client Communication with Excel

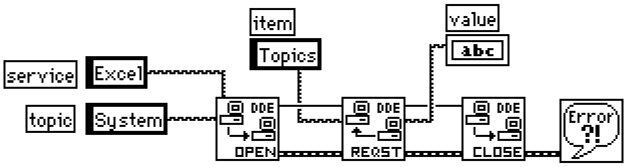
Each application that supports DDE has a different set of services, topics, and data items that it can talk about. Select Services, Topics, and Data Items for more information on this topic. For example, two different spreadsheet programs can take very different approaches to how they specify spreadsheet cells. To find out what a given application supports, consult the documentation that came with that application.

Microsoft Excel, a popular spreadsheet program for Windows, has DDE support. You can use DDE to send commands to Excel. You can also manipulate and read spreadsheet data by name. For more information on how to use DDE with Excel, refer to the Microsoft Excel Users Guide 2.

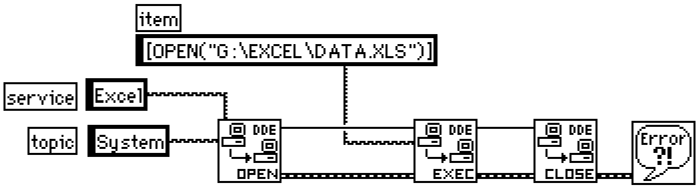
With Excel, the service name is Excel. For the topic, you use the name of an open document, such as spreadsheet document, or the word System.

If you use the name System, you can request information about the status of Excel, or send general commands to Excel (commands that are not directed to a specific spreadsheet). For instance, for the topic System, Excel talks about items such as Status, which has a value of Busy if Excel is busy, or Ready if Excel is ready to execute commands). Another, more useful data item you can use when the topic is Status is Topics, which returns a list of topics Excel can talk about, including all open spreadsheet documents and the System topic.

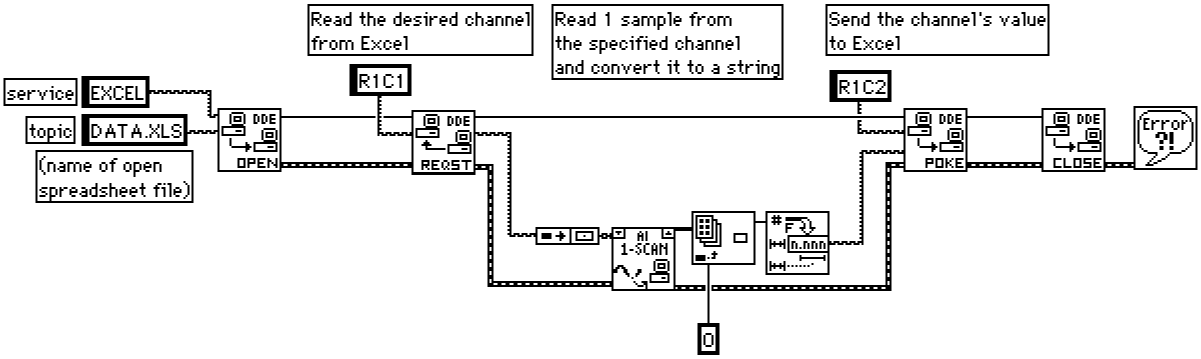
The following VI shows how you can use the Topics command in LabVIEW. The value returned is a string containing the names of the open spreadsheets and the work Excel.



Another way you can use the System topic with Excel is to instruct Excel to open a specific document. To do this, you use the DDE Execute.vi to send an Excel Macro to Excel that instructs Excel to open the document, as shown in the following LabVIEW diagram:



After you open a spreadsheet file, you can send commands to the spreadsheet to read cell values. In this case, your topic is the spreadsheet document name. The item is the name of a cell, a range of cells, or a named section of a spreadsheet. For example, in the following diagram LabVIEW can retrieve the value in the cell at row one column one. It then acquires a sample from the specified channel, and sends the resulting sample back to Excel.


Read the desired channel from Excel
Read 1 sample from the specified channel and convert it to a string
Send the channel's value to Excel
service
EXCEL
topic
DATA.XLS
(name of open spreadsheet file)
R1C1
R1C2
DDE
OPEN
DDE
REQST
DDE
POKE
DDE
CLOSE
AI
1-SCAN
0
Error
?!

LabVIEW VIs as DDE Servers

You can create LabVIEW VIs that act as servers for data items. The general concept is that a LabVIEW VI indicates that it is willing to provide information regarding a specific service in topic. LabVIEW can use any name for the service and topic name. It might specify the service name to be the name of the application (LabVIEW), and the topic name to be either the name of the Server VI, or a general classification for the data it provides, such as Lab Data.

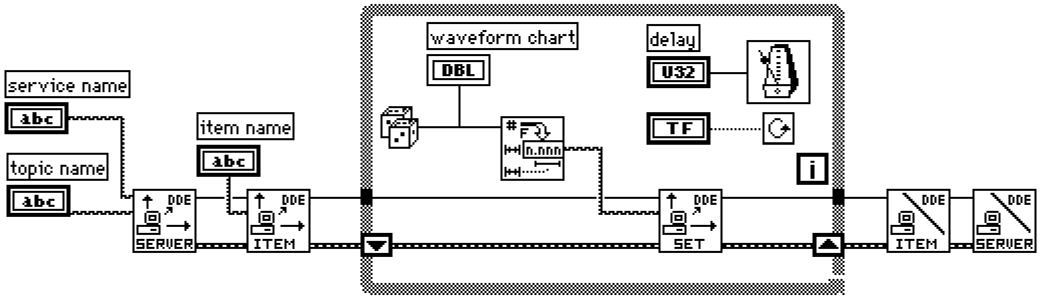
The Server VI then registers data items for a given service that it talks about. LabVIEW remembers the data names and their values, and handles communication with other applications regarding the data. When the server VI changes the value of data that is registered for DDE communication, LabVIEW notifies any client applications that have requested notification concerning that data. In the same way, if another application sends a Poke message to change the value of a data item, LabVIEW changes this value.

You cannot use the DDE Execute Command with a LabVIEW VI acting as a server. If you want to send a command to a VI, you must send the command using data items.

Also, notice that LabVIEW does not currently have anything like the System topic that Excel provides. The LabVIEW application is not itself a server to which you can send commands or request status information.

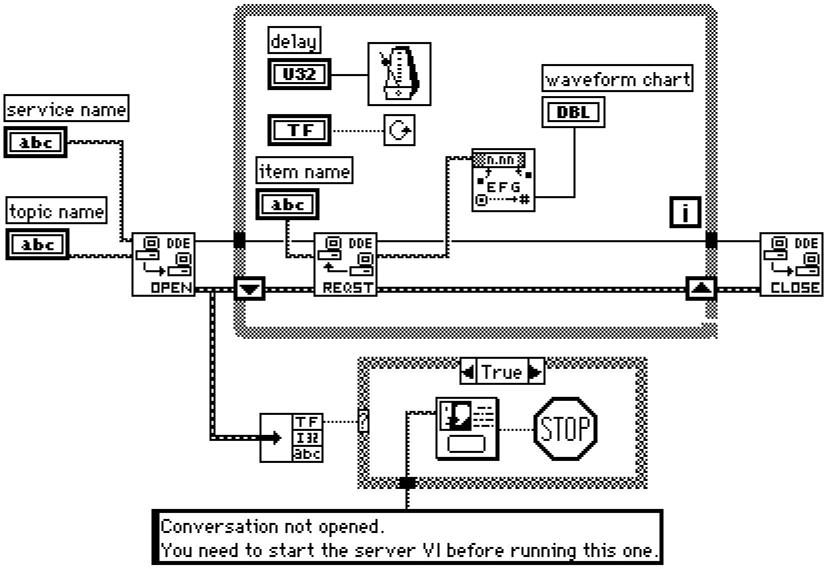
It is important to understand that LabVIEW VIs act as servers and that at this time LabVIEW does not itself provide any services to other applications.

The following example shows how to create a DDE Server VI that provides data to other client applications. In this case, the data is a random number. You can easily replace the random number with real world data from data acquisition boards or devices connected to the computer by GPIB, VXI, or serial connections.



The VI in the preceding diagram registers a server with LabVIEW. The VI registers an item that it is willing to provide to clients. In the loop, the VI periodically sets the value of the item. As mentioned earlier, LabVIEW notifies other applications that data is available. When the loop is complete, the VI finishes by unregistering the item and unregistering the server.

The clients for this VI can be any applications that understand DDE, including other LabVIEW VIs. The following diagram illustrates a client to the VI shown in the previous diagram. It is important that the service, topic, and item names are the same as the ones used by the server.

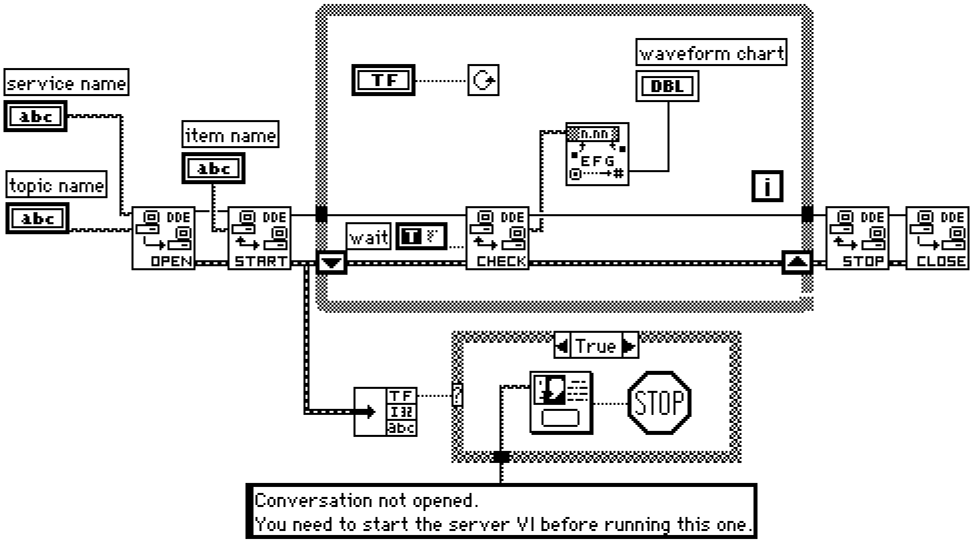


Conversation not opened.
You need to start the server VI before running this one.

Requesting Data versus Advising Data

The previous client example used the DDE Request VI in a loop to retrieve data. With DDE Request, the data is retrieved immediately, regardless of whether you have seen the data before. If the server and the client do not loop at exactly the same rate, you can duplicate or miss data.

One way to avoid duplicating data is to use the DDE Advise VIs to request notification of changes in the value of a data item. The following diagram shows how you can implement this scheme:



In the preceding diagram, LabVIEW opens a conversation. It then uses the DDE Advise Start VI to request notification of changes in the value of a data item. Every time through the loop, LabVIEW calls the DDE Advise Check VI, which waits for a data item to change its value. When the loop is finished, LabVIEW ends the advise loop by calling the DDE Advise Stop VI, and closing the conversation.

Synchronization of Data

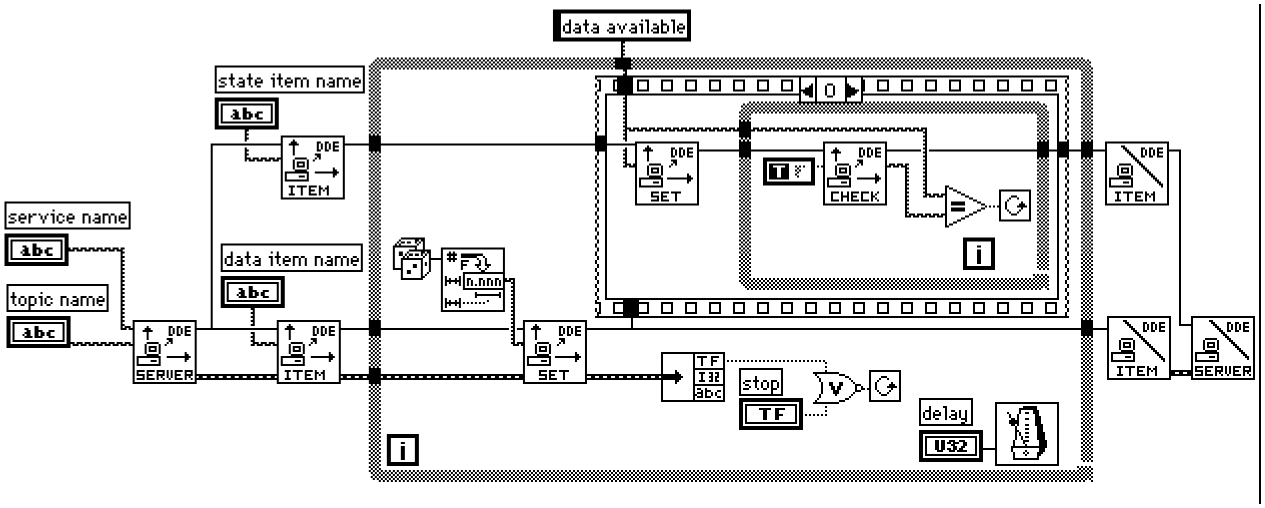
The client server examples in the preceding section work well for monitoring data. However, in these examples there is no assurance that the client receives all the data that the server sends. Even with the DDE Advise loop, if the client does not check for a data change frequently enough, the client can miss a data value that the server provided.

In some applications, missed data is not a problem. For example, if you are monitoring a data acquisition system, missed data may not cause problems when you are observing general trends. In other applications, you may want to ensure that no data is missed.

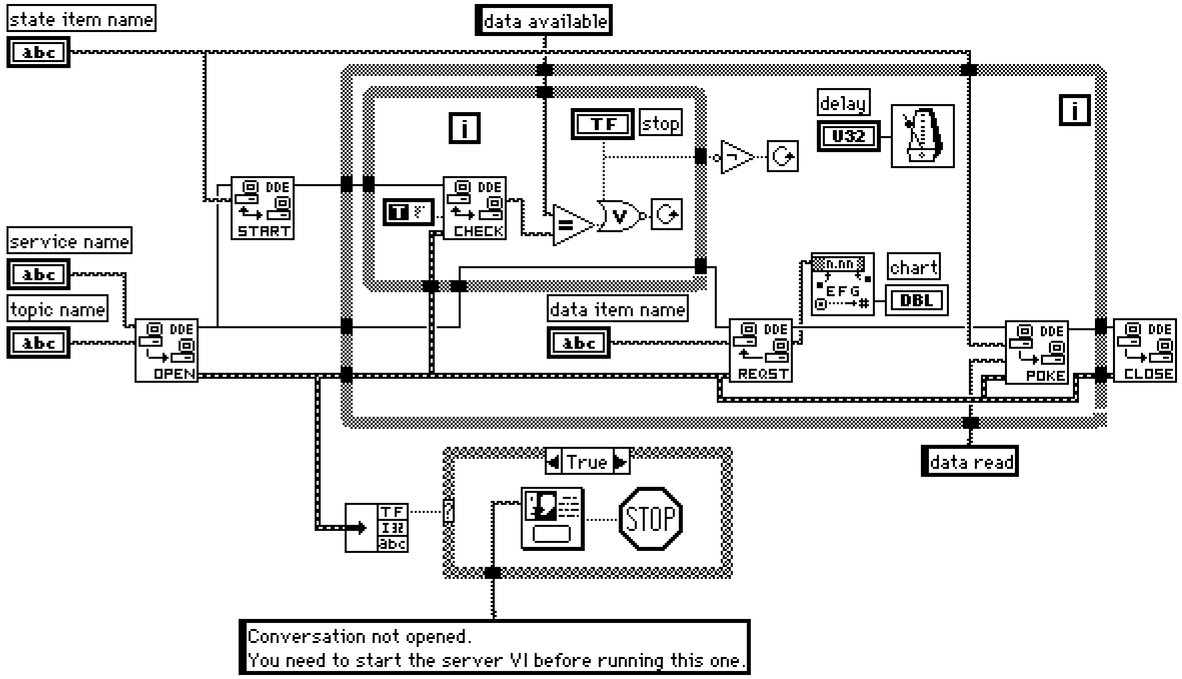
One major difference between TCP and DDE is that TCP queues data so that you do not miss it and you get it in the correct order. DDE does not provide this service.

In DDE, you can set up a separate item, which the client uses to acknowledge that it has received the latest data. You then update the acquired data item to contain a new point only when the client acknowledges receipt of the previous data.

For example, you can modify the server example shown in the Requesting Data versus Advising Data topic to set a state item to a specific value after it has updated the acquired data item. The server then monitors the state item until the client acknowledges receipt of data. This modification is shown in the following block diagram:



A client for this server, as shown in the following diagram, monitors the state item until it changes to data available. At that point, the client reads the data from the acquired data item provided by the server, and then updates the state item to data read value.



This technique makes it possible to synchronize data transfer between a server and a single client. However, it has some shortcomings. First, you can have only one client. Multiple clients can conflict with one another. For example, one client might receive the data and acknowledge it before the other client notices that new data is available.You can build more complicated DDE diagrams to deal with this problem, but they quickly become awkward. For applications that involve only a single client, this is not a problem.

Another problem with this technique of synchronizing communication is that the speed of your acquisition becomes controlled by the rate at which you transfer data. You can address this issue by breaking the acquisition and the transmission into separate loops. The acquisition can queue data which the transmission loop would send. This is similar to the TCP Server example in which the server handles multiple connections.

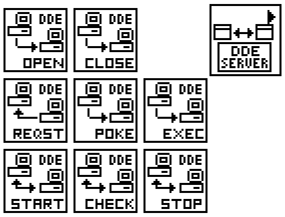
If your application needs reliable synchronization of data transfer, you may want to use TCP/IP instead, because it provides queueing, acknowledgment of data transfer, and support for multiple connections at the driver level.

DDE VI Descriptions

Click here to access the DDE VI Overview (Windows) topic.

The top-level DDE VIs are used as clients. The DDE Server VI subpalette contains the DDE Server VI Descriptions. The following illustration shows the **DDE** palette, which you access by selecting **Function» Communication»DDE**.

Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.



VI Descriptions

DDE Advise Check
DDE Advise Start
DDE Advise Stop
DDE Close Conversation
DDE Execute
DDE Open Conversation
DDE Poke
DDE Request

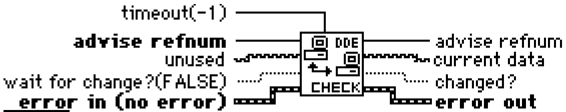
Subpalette Descriptions

DDE Server VI Descriptions

For examples of how to use the DDE VIs, see the examples in the `examples\comm\DDEexamp.llb` library.

DDE Advise Check

Checks an advise value previously established by DDE Advise Start.



TF **advise refnum** is the unique number that identifies this DDE advise link.

TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **timeout** specifies how long to wait for the function to complete. The default value of -1 specifies no timeout. If the specified amount of time expires before completion, the VI returns an error.

TF **unused** was previously the old data input. The DDE VIs now can track changes to the data internally so this input is no longer needed. It remains so that VIs that used it do not break.

TF **wait for change?** specifies whether the VI should get the current value and return immediately or wait until the value changes before returning.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

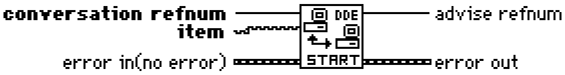
TF **advise refnum** is the unique number that identifies this DDE advise link. It passes through this VI to assist in execution timing.

TF **current data** always returns the most recently received value for the advise item.

TF **changed?** specifies whether the old data is the same as **current data**.

DDE Advise Start

Initiates an advise link.



TF **conversation refnum** is the unique number that identifies this DDE conversation.

TF **item** is the location of the data from the server application that the VI communicates to the client application.

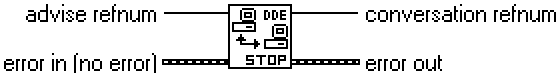
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **advise refnum** is the unique number that identifies this DDE advise link.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

DDE Advise Stop

Cancels an advise link, previously established by DDE Advise Start.



TF **advise refnum** is the unique number that identifies this DDE advise link.

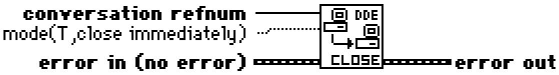
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **conversation refnum** is the unique number that identifies this DDE conversation. Pass it through this VI to assist in execution timing.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

DDE Close Conversation

Closes a DDE conversation.



TF **conversation refnum** is the unique number that identifies this DDE conversation.

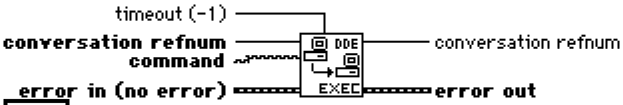
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **mode** controls when the conversation closes. If mode is TRUE, the DDE conversation is always closed. If mode is FALSE, the conversation closes only when an error passes in. Thus, if you wire the **error out** of another DDE VI to the **error in** of the Close Conversation VI with a FALSE **mode**, the conversation terminates only if an error occurs in the first VI.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

DDE Execute

Tells the DDE server to execute **command**.



TF **conversation refnum** is the unique number that identifies this DDE conversation.

TF **command** contains the command to be sent.

TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

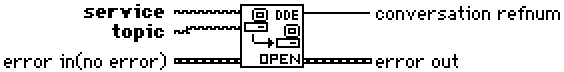
TF **timeout** specifies how long to wait for the function to complete. The default value of -1 specifies no timeout. If the specified amount of time expires before completion, the VI returns an error.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **conversation refnum** is the unique number that identifies this DDE conversation. It passes through this VI to assist in execution timing.

DDE Open Conversation

Establishes a connection between LabVIEW and another application. You must call this VI before you use any other DDE VIs (except Server VIs).



TF **service** is the name of the DDE server.

TF **topic** is the name of the DDE topic.

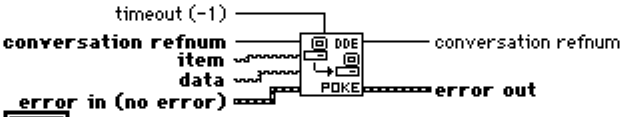
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **conversation refnum** is the unique number that identifies this DDE conversation. Returns 0 if an error occurs.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

DDE Poke

Tells the DDE server to put the value **data** at **item.**



TF **conversation refnum** is the unique number that identifies this DDE conversation.

item is the location where the VI pokes the **data**.

data contains the data the VI sends.

error in describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

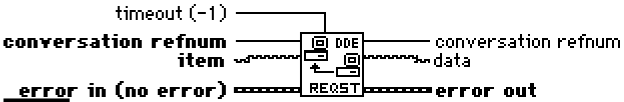
timeout specifies how long to wait for the function to complete. The default value of -1 specifies no timeout. If the specified amount of time expires before completion, the VI returns an error.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

conversation refnum is the unique number that identifies this DDE conversation. It passes through this VI to assist in execution timing.

DDE Request

Initiates a DDE message exchange to obtain the current value of **item**.



conversation refnum is the unique number that identifies this DDE conversation.

item is the location of the requested data.

error in describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

timeout specifies how long to wait for the function to complete. The default value of -1 specifies no timeout. If the specified amount of time expires before completion, the VI returns an error.

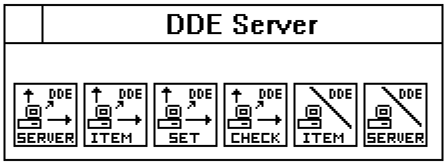
error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

conversation refnum is the unique number that identifies this DDE conversation. It passes through this VI to assist in execution timing.

data contains the data that the DDE Request returns.

DDE Server VI Descriptions

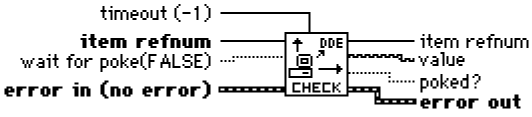
This topic discusses the DDE Server VIs. To access these VIs, pop up on the DDE Server icon located on the **DDE** palette.



DDE Srv Check Item
DDE Srv Register Item
DDE Srv Register Service
DDE Srv Set Item
DDE Srv Unregister Item
DDE Srv Unregister Service

DDE Srv Check Item

Sets the value of a previously defined DDE Item.



DDE Srv Check Item.vi

TF **item refnum** is the unique number that identifies this DDE item.

TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **timeout** specifies how long to wait for the function to complete. The default value of -1 specifies no timeout. If the specified amount of time expires before completion, the VI returns an error.

TF **wait for poke** specifies whether the VI should get the current value and return immediately or wait until a client pokes the value before returning.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

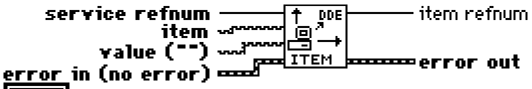
TF **item refnum** is the unique number that identifies this DDE item.

TF **value** is the new value for the item.

TF **poked?** specifies whether the item has been poked by a DDE client since the last DDE Srv Check Item.

DDE Srv Register Item

Establishes a DDE item for the service specified by service refnum.



TF **service refnum** is the unique number that identifies this DDE service.

TF **item** is the name of the DDE item

TF **value** is the initial value for the item.

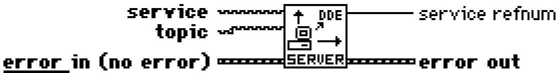
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **item refnum** is the unique number that identifies this DDE item. Returns 0 if an error occurs.

DDE Srv Register Service

Establishes a DDE service to which clients can connect.



TF **service** is the name of the DDE server.

TF **topic** is the name of the DDE topic.

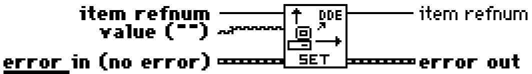
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **service refnum** is the unique number that identifies this DDE service. Returns 0 if an error occurs.

DDE Srv Set Item

Sets the value of a previously defined DDE Item.



TF **item refnum** is the unique number that identifies this DDE item.

TF **value** is the new value for the item.

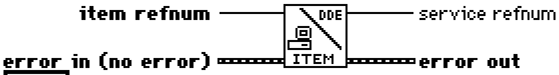
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **item refnum** is the unique number that identifies this DDE item.

DDE Srv Unregister Item

Removes the specified item from its service. DDE clients can no longer access the item after this VI completes.



TF **item refnum** is the unique number that identifies this DDE item.

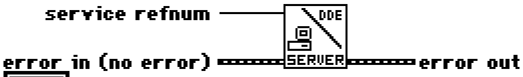
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

service refnum is the unique number that identifies this DDE service.

DDE Srv Unregister Service

Removes the specified service. DDE clients can no longer connect to this service and all current conversations are closed.



service refnum is the unique number that identifies this DDE service.

error in describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

DDE Server Subpalette

DDE Server VI Descriptions

DDE Advise Check VI

DDE Advise Check

DDE Advise Start VI

DDE Advise Start

DDE Advise Stop VI

DDE Advise Stop

DDE Close Conversation VI

DDE Close Conversation

DDE Execute VI

DDE Execute

DDE Open Conversation VI

DDE Open Conversation

DDE Poke VI

DDE Poke

DDE Request VI

DDE Request

DDE Srv Check Item VI

DDE Srv Check Item

DDE Srv Register Item VI

DDE Srv Register Item

DDE Srv Register Service VI

DDE Srv Register Service

DDE Srv Set Item VI

DDE Srv Set Item

DDE Srv Unregister Item VI

DDE Srv Unregister Item

DDE Srv Unregister Service VI

DDE Srv Unregister Service

Windows NT

Launch `DDEShare.exe`, found in the `winnt/system32` directory. Select from the **Shares»DDE Shares»Add a Share...** to register the service name and topic name on the server.

OLE Automation VI Overview (Windows 95/NT)

This topic discusses the LabVIEW VIs for OLE (Object Linking and Embedding) Automation, a feature that you can use with LabVIEW to access objects exposed by automation servers in the system.

Click here to access the OLE Automation VI Descriptions topic.

OLE Automation Concepts
Using LabVIEW to Implement OLE Automation

The OLE Automation VI Library contains two levels of VIs. VIs that are available on the Communication palette represent the higher-level of functionality. These VIs use lower-level subVIs which are hidden from the user, providing for a higher-level of encapsulation. Helper VIs are provided.

OLE Automation Concepts

In the context of Object Linking and Embedding, objects are defined as data abstractions exported by an application. You manipulate these objects by using another Windows application. Linking and Embedding are two of the methods used to access OLE objects.

You use OLE Automation to make the functions and methods of one application available for use by other applications. You then access these functions or methods, which are usually grouped into objects.

An application supports automation as either a server or a client. Applications that expose objects and provide methods for operating on those objects are called *OLE automation servers*. Applications that use the methods exposed by another application are called *OLE automation clients/controllers*. The OLE VIs enable LabVIEW to become an automation client.

Using LabVIEW to Implement OLE Automation

An OLE object exposes both methods and properties. Methods have the ability to modify a wide range of values, whereas properties can set or get the value of a specific characteristic of the object. Some servers provide a type library listing all exposed objects and the methods and properties of each object.

The typical steps in creating a client application using C are as follows:

- Get the IDispatch interface of the Object whose methods you want to access.
- Get the DispatchID of the method of that object.
- Invoke the method using the Invoke functions of the IDispatch interface, packing all parameters into the parameter list.

In LabVIEW, do as follows:

- Use the Create Automation VI to get an Automation refnum, which uniquely defines the IDispatch interface.
- Use the Execute Method VI to execute a method belonging to that object. If there is just one parameter, it can be flattened. The type descriptors and the flattened string are then passed in as input parameters. If there are multiple outputs, they are bundled in a cluster. The resultant cluster is then flattened and wired to the correct input of the VI.

The implementation uses DLLs to perform the actual OLE calls. Parameters are passed to these DLLs as flattened data.

OLE Automation VI Descriptions

Click here to access the OLE Automation VI Overview (Windows 95/NT) topic.

The following illustration shows the OLE Automation VI palette, which you access by selecting **Function»Communication»OLE**.

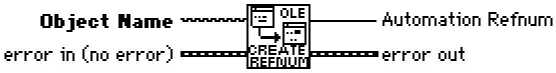
Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.

Create Automation Refnum
Execute Method
Get Property
List Methods or Properties
List Objects in Type Library
Release Refnum
Set Property

For examples of how to use the OLE Automation VIs, see the examples in `examples\comm\OLE-xxx.llb`.

Create Automation Refnum

Given the object name (registered class name) of an OLE object, returns an Automation Refnum uniquely identifying the instantiation.



TF **Object Name**. The class name of an OLE object.

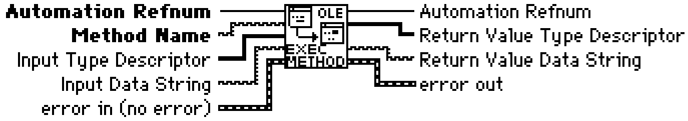
TF **error in** describes error conditions prior to the execution of this VI. The default input is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **Automation Refnum**. The Automation Refnum passed to a VI.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains that same information. Otherwise, it describes the error status produced by this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Execute Method

Executes a method.



TF **Automation Refnum**. Value uniquely defining an instantiation of an OLE class.

TF **Method Name**. Name of the method in that class to be invoked.

[I16] **Input Type Descriptor**. See Type Descriptors for more information.

TF **Input Data String.** The flattened string, passed as an input parameter. For more information, see the *Using LabVIEW to Implement OLE Automation* section.

TF **error in** describes error conditions prior to the execution of this VI. The default input is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **Automation Refnum**. The Automation Refnum passed to a VI. The dup is provided to simplify dataflow programming in a manner similar to the dup file refnums in file I/O functions.

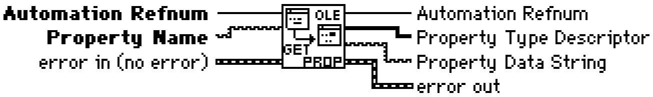
[I16] **Return Value Type Descriptor**. See Type Descriptors for more information.

TF **Return Value Data String**. The flattened string, passed as an output parameter. For more information, see the *Using LabVIEW to Implement OLE Automation* topic.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains that same information. Otherwise, it describes the error status produced by this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Get Property

Gets the value of a property.



TF **Automation Refnum** Value uniquely defining an instantiation of an OLE class.

TF **Property Name**. Name of property in that class.

TF **error in** describes error conditions prior to the execution of this VI. The default input is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **Automation Refnum**. The Automation Refnum passed to a VI. The dup is provided to simplify dataflow programming in a manner similar to the dup file refnums in file I/O functions.

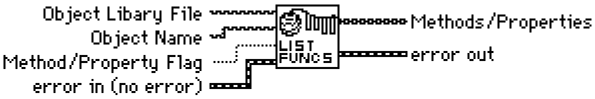
TF **Property Type Descriptor**. See Type Descriptors for more information.

TF **Property Data String**. The flattened string, passed as an output parameter. For more information, see the *Using LabVIEW to Implement OLE Automation* topic.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains that same information. Otherwise, it describes the error status produced by this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

List Methods or Properties

Lists all the methods or properties of an object.



Object Library File. Full path name of the object library file (*.olb, *.tlb).

TF **Object Name**. String containing the name of the object.

TF **Method/Property Flag**. If set, lists all methods. Otherwise, lists all properties.

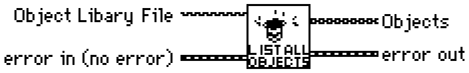
TF **error in** describes error conditions prior to the execution of this VI. The default input is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **Methods/Properties.** An array of strings containing all the methods and/or properties in that object.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains that same information. Otherwise, it describes the error status produced by this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

List Objects in Type Library

Lists all the objects in a type library.



TF **Object Library File**. Full path name of the object library file (*.olb, *.tlb).

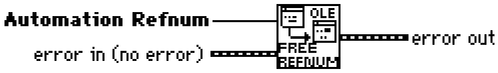
TF **error in** describes error conditions prior to the execution of this VI. The default input is no error. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **Objects**. An array of strings containing all objects defined in the object library file.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains that same information. Otherwise, it describes the error status produced by this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Release Refnum

Releases the refnum passed in as input.



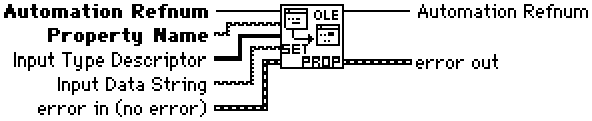
TF **Automation Refnum**. Value uniquely defining an instantiation of an OLE class.

TF **error in** describes error conditions prior to the execution of this VI. The default input is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains that same information. Otherwise, it describes the error status produced by this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Set Property

Sets the value of a property.



TF **Automation Refnum**. Value uniquely defining an instantiation of an OLE class.

TF **Property Name**. Name of the property in that class.

TF **Input Type Descriptor**. See Type Descriptors for more information.

TF **Input Data String**.The flattened string, passed as an input parameter. For more information, see the Using LabVIEW to Implement OLE Automation section.

TF **error in** describes error conditions prior to the execution of this VI. The default input is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **Automation Refnum**. The Automation Refnum passed to a VI. The dup is provided to simplify dataflow programming in a manner similar to the dup file refnums in file I/O functions.

error out contains error information. If **error in** indicates an error, then **error out** contains that same information. Otherwise, it describes the error status produced by this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Create Automation Refnum VI

Create Automation Refnum

Execute Method VI

Execute Method

Get Property VI

Get Property

List Methods or Properties VI

List Methods or Properties

List Objects in Type Library VI

List Objects in Type Library

Release Refnum VI

Release Refnum

Set Property VI

Set Property

Glossary

Prefix	Meaning	Value
n-	nano-	10-9
m-	micro-	10-6
m-	milli-	10-3
k-	kilo-	103
M-	mega-	106

Numbers/Symbols

A

abort The procedure that terminates a program when a mistake, malfunction, or error occurs.

ANSI American National Standards Institute.

APDA Apple Programmer Developer Association.

array Ordered, indexed set of data elements of the same type.

ASCII American Standard Code for Information Interchange.

asynchronous Mode in which multiple processes share processor time. For example, execution one executes while the others wait for interrupts, as while performing device I/O or waiting for a clock tick.

C

CIN Code Interface Node. Special block diagram node through which you can link conventional, text-based code to a VI.

client The application that sends or calls messages from the server application in a dynamic data exchange.

cluster A set of ordered, unindexed data elements of any data type including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.

connection ID A unique identification of a connection that you use for reference in subsequent VI calls.

control Front panel object for entering data into a VI interactively or into a subVI programmatically.

D

DARPA
Defense Advanced Research Projects Agency.

datagram
IP-packaged data components that contain, among other things, the data and a header that indicates the source and destination addresses.

DDE
Dynamic Data Exchange. A client-controlled Windows protocol for communication between applications.

dialog box
An interactive screen with prompts in which the user specifies additional information needed to complete a command.

dotted decimal
A method of describing a 32-bit internet address in which the address is notation divided into four 8-bit binary numbers and written as four integers separated by decimal points.

driver
Software used to manipulate a device or interface board.

E

ethernet
A network system that carries audio and video information as well as computer data.

H

handler
A device driver installed as part of the operating system of the computer.

I

IAC
Interapplication Communication. A feature of Apple Macintosh system software version 7 by which applications can communicate with each other.

icon
Graphical representation of a node on a block diagram.

IEEE
Institute of Electrical and Electronic Engineers.

indicator
Front panel object that displays output.

internet
See internetwork.

internetwork
Single or interconnected networks.

IP
Internet Protocol. Protocol that performs the low-level service of packaging data into components (datagrams). *See* TCP/IP.

L

LabVIEW
Laboratory Virtual Instrument Engineering Workbench.

LF
Line feed.

M

MB	Megabytes of memory.

N

NaN	Digital display value for a floating-point representation of *not a number*, typically the result of an undefined operation, such as log(-1).

O

OLE	Object Linking and Embedding.
OLE Automation	A feature which allows LabVIEW to access objects by automation servers in the system.

P

palette	A collection of function or control icons from which you can select the control or function you need.
PC	Personal Computer. Used to refer to IBM-compatible computers.
poke	An instruction that places a value into a specific location in memory.
PPC	Program-to-Program Communication. A low-level form of IAC by which applications send and receive blocks of data.
protocol	Set of rules or conventions that cover the exchange of information between computer systems.

R

refnum	An identifier of a DDE conversation or open files that can be referenced by related VIs.
remote address	Address of the remote machine associated with a connection.

S

SCSI	Small Computer System Interface (bus).
sec	Seconds.
server	The application that receives messages from the client application in a dynamic data exchange.
spreadsheet	Any of a number of programs that arrange data and formulas in a matrix of cells.
string	A connected sequence of characters or bits treated as a single data item.

T

TCP

Transmission Control Protocol. *See* TCP/IP.

TCP/IP

Transmission Control Protocol/Internet Protocol. A suite of communications protocols that you use to transfer blocks of data between applications.

timeout

The time (in milliseconds) that a VI waits for an operation to complete. Generally, a timeout of -1 causes a VI to wait indefinitely.

U

UDP

User Datagram Protocol. *See* TCP/IP.

utility

A program that helps the user run, enhance, create, or analyze other programs and systems.

V

VI

Virtual instrument. LabVIEW program; so called because it models the appearance and function of a physical instrument.

W

wire

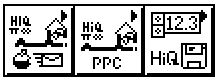
Data path between nodes.

HiQ Functions for Macintosh

You can use these functions to share between LabVIEW and HIQ as well as to control HiQ from LabVIEW. See the HiQ documentation on LabSuite for more detailed information on how to use the functions.

The following illustration shows the HiQ palette, which you can access by selecting **Functions»Communication»HiQ**.

Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.



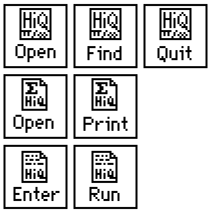
HiQ Apple Event Command VI Descriptions
HiQ PPC Data Transfer VI Descriptions
HiQ File Transfer VI Descriptions

HiQ Apple Event Command VI Descriptions

HiQ supports seven Apple Event commands: the four required events--Run, Open, Print, and Quit; a DoScript event for executing a script in a specified HiQ Worksheet; and an event for finding open HiQ VIs.

The following illustration shows the HiQ Apple Event Commands palette, which you can access by selecting **Functions»Communication»HiQ»HiQ Apple Event Commands**.

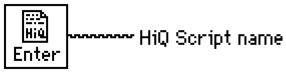
Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.



Enter HiQ Script
Execute Script
Find an Open HiQ
Open HiQ
Open Worksheet
Print Worksheet
Quit HiQ

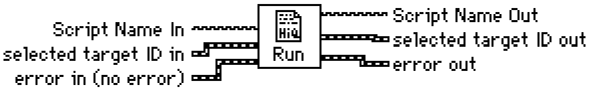
Enter HiQ Script

Prompts you to enter a HiQ-Script name.



Execute Script

Communicates to the selected HiQ to execute a specific HiQ-Script. If Script Name In is empty, this VI prompts for a HiQ-Script name using the Enter HiQ Script VI.



TF **Script Name In** is the name of the HiQ-Script to execute.

TF **selected target ID in** is a cluster of information describing the location of the target HiQ.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

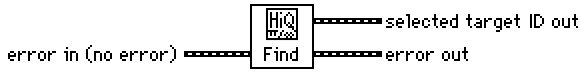
TF **Script Name Out** is the name of the HiQ-Script that executed.

TF **selected target ID out** is a cluster of information describing the location of the target HiQ.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Find an Open HiQ

Displays the PPC dialog box you can use to choose an open HiQ. HiQ must already be executing.



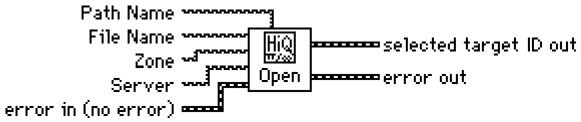
TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **selected target ID out** is a cluster of information describing the location of the target HiQ.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Open HiQ

Displays the file dialog box you can use to choose the HiQ executable you want to open. If you want to open a HiQ on the network, you must first mount the network drive using Chooser under the Apple menu.



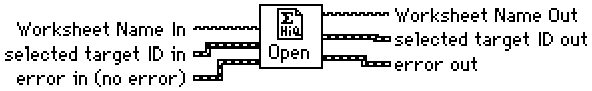
TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **selected target ID out** is a cluster of information describing the location of the target HiQ.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Open Worksheet

Opens a worksheet from the selected HiQ. If Worksheet Name In is empty, this VI prompts you for a Worksheet name using a File dialog box.



TF **Worksheet Name In** is the name of the HiQ Worksheet to print. You can specify the full pathname, if required.

TF **selected target ID in** is a cluster of information describing the location of the target HiQ.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

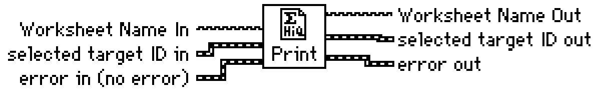
TF **Worksheet Name Out** is the name of the HiQ Worksheet that was printed.

TF **selected target ID out** is a cluster of information describing the location of the target HiQ.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Print Worksheet

Communicates with the selected HiQ to print a specific Worksheet. If Worksheet Name In is empty, this VI prompts you for a Worksheet name using a File dialog box.



Worksheet Name In is the name of the HiQ Worksheet to open. You can specify the full pathname, if required.

selected target ID in is a cluster of information describing the location of the target HiQ.

error in (no error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

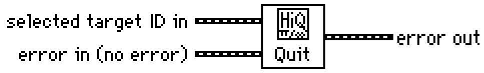
Worksheet Name Out is the name of the HiQ Worksheet that was printed.

selected target ID out is a cluster of information describing the location of the target HiQ.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Quit HiQ

Closes the selected HiQ.



selected target **ID in** is a cluster of information describing the location of the target HiQ.

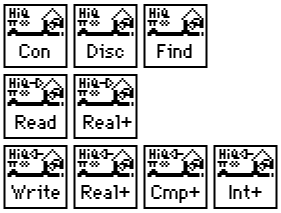
error in (no error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

HiQ PPC Data Transfer VI Descriptions

The following illustration shows the HiQ PPC Data Transfer palette, which you can access by selecting **Functions»Communication»HiQ»HiQ PPC Data Transfer**.

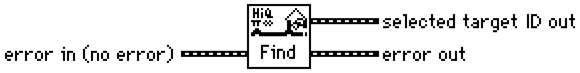
Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.



Find an Open HiQ PPC Port
HiQ PPC Connect
HiQ PPC Disconnect
HiQ PPC Read
HiQ PPC Read Real+
HiQ PPC Write
HiQ PPC Write Integer+
HiQ PPC Write Real+
HiQ PPC Write Complex+

Find an Open HiQ PPC Port

Displays the PPC Browser dialog box for selecting an open HiQ PPC port on a network or on the same computer.



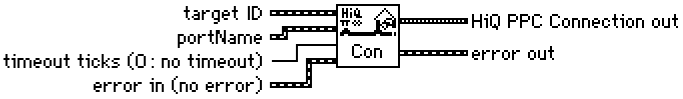
TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **selected target ID out** is a cluster of information describing the location of the HiQ PPC server.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

HiQ PPC Connect

Establishes a connection with a HiQ PPC server.



TF **target ID** is a cluster of information describing the location of the HiQ PPC server and contains a complex cluster of information defined by Apple Computer, Inc. The Open HiQ and the Find an Open HiQ VIs create this cluster.

TF **portName** is a cluster containing the following parameters in the order listed below. Wiring for this input is optional.

TF **nameScript** is a 32-bit integer used in international localization that specifies the language system you are using. Use a **nameScript** value of 0 for Roman language systems (for example, English); consult *Inside Macintosh*, *Volume VI* for a list of available script codes.

TF **selector** describes the format of the type string parameter.

1: (creator/type) Signifies that **type string** is an 8-character string; the first four characters are the creator (for example, LBVW), and the last four characters define the port type.

2: (port type string) Signifies that **type string** is a 32-character (or less) description of the service provided by the port

TF **name** is the name you give to the port. The value of name, which can be no more than 32 characters, is displayed in the PPC Browser dialog box list of port names. The Get Target ID VI uses **name** to identify the port.

TF **port type string** is an 8-character string; the first four characters are the creator (for example, LBVW), and the last four characters define the port type, when **selector** has a value of 1. The **type string** is a 32-character (or less) description of the service that the port provides when selector has a value of 2. (In almost all cases, you should specify a value of 2 for **selector**, and use a description of the service provided by the port for **type string**. Consult *Inside Macintosh, Volume VI*, for more information about other cases.)

TF **timeout ticks (0: no timeout).** If non-zero, **timeout ticks** specifies the number of ticks PPC Inform Session waits for LabVIEW to establish a session before returning an error. One tick equals 1/60 of a second.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **HiQ PPC Connection out** is a cluster of information describing the active PPC session. You use this cluster as an input to other HiQ PPC VIs to identify the active PPC session. The HiQ PPC Connection out cluster contains the following parameters.

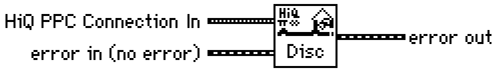
I16 **port refnum** is a port reference number describing the local port associated with the current PPC session.

TF **session refnum** is a session reference number describing the current HiQ PPC session.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

HiQ PPC Disconnect

Closes the active PPC port and ends the active PPC session.



TF **HiQ PPC Connection** is a cluster of information describing the active PPC session.

TF **port refnum** is a port reference number describing the local port associated with the current PPC session.

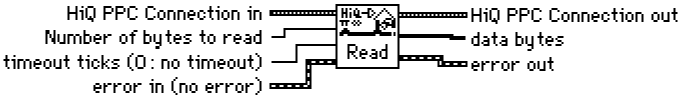
TF **session refnum** is a session reference number describing the current HiQ PPC session.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

HiQ PPC Read

Reads the specified number of bytes from the server HiQ. A HiQ-Script must be executing on the server that writes the specified number bytes.



TF **HiQ PPC Connection in** is a cluster of information describing the active PPC session.

TF **Number of bytes to read** specifies how many bytes to read from HiQ.

TF **timeout ticks (0: no timeout)** If non-zero, **timeout ticks** specifies the number of ticks PPC Inform Session waits for LabVIEW to establish a session before returning an error. One tick equals 1/60 of a second.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

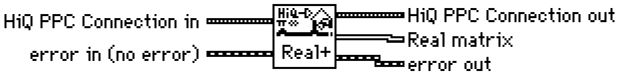
TF **HiQ PPC Connection out** is a cluster of information describing the active PPC session.

TF **data bytes** is a 1D array containing the data bytes received from HiQ.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

HiQ PPC Read Real+

Reads real data from the HiQ server. This VI works with the HiQ PPC_HiQ_LV_WriteData function.



TF **HiQ PPC Connection in** is a cluster of information describing the active PPC session.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

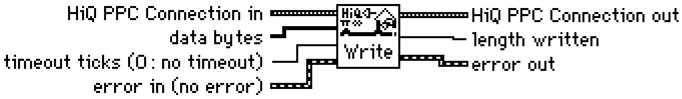
TF **HiQ PPC Connection out** is a cluster of information describing the active PPC session.

[DBL] **Real matrix** is a 2D array containing the data received from HiQ.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

HiQ PPC Write

Writes a one-dimensional array of 1-byte data to the HiQ server. A HiQ-Script must be executing on the server that reads the data.



TF **HiQ PPC Connection in** is a cluster of information describing the active PPC session.

TF **data bytes** is a 1D array containing the data bytes to write to HiQ.

TF **timeout ticks (0: no timeout)** If non-zero, **timeout ticks** specifies the number of ticks PPC Inform Session waits for LabVIEW to establish a session before returning an error. One tick equals 1/60 of a second.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

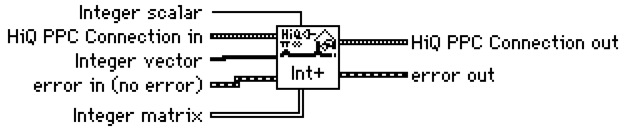
TF **HiQ PPC Connection out** is a cluster of information describing the active PPC session.

TF **length written** contains the actual number of bytes written to HiQ.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

HiQ PPC Write Integer+

Writes either scalar, vector, or matrix integer data to the HiQ server. This VI works with the PPC_HiQ_LV_ReadData function.



TF **HiQ PPC Connection in** is a cluster of information describing the active PPC session.

I32 **Integer vector** is a 1D array containing the data to be written to HiQ.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **Integer scalar** contains the scalar data to be written to HiQ.

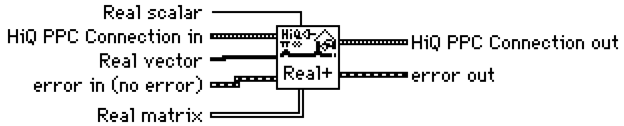
TF **Integer matrix** is a 2D array containing the data to be written to HiQ.

TF **HiQ PPC Connection out** is a cluster of information describing the active PPC session.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

HiQ PPC Write Real+

Writes either scalar, vector, or matrix real data to the HiQ server. This VI works with the PPC_HiQ_LV_ReadData function.



TF **HiQ PPC Connection in** is a cluster of information describing the active PPC session.

TF **Real vector** is a 1D array containing the real data to be written to HiQ.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

DBL **Real scalar** contains the scalar real data to be written to HiQ.

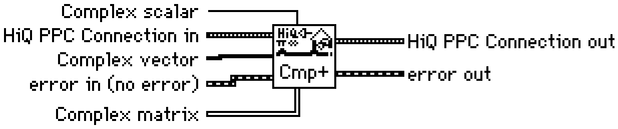
[DBL] **Real matrix** is a 2D array containing the real data to be written to HiQ.

TF **HiQ PPC Connection out** is a cluster of information describing the active PPC session.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

HiQ PPC Write Complex+

Writes either scalar, vector, or matrix complex data to the HiQ server. This VI works with the PPC_HiQ_LV_ReadData function.



TF **HiQ PPC Connection in** is a cluster of information describing the active PPC session.

[CDB] **Complex vector** is a 1D array containing the complex data to be written to HiQ.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

CDB **Complex** scalar contains the scalar, complex data to be written to HiQ.

TF **Complex Matrix** is a 2D array containing the complex data to be written to HiQ.

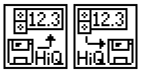
TF **HiQ PPC Connection out** is a cluster of information describing the active PPC session.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

HiQ File Transfer VI Descriptions

The following illustration shows the HiQ File Transfer palette, which you can access by selecting **Functions»Communication»HiQ»HiQ File Transfer**.

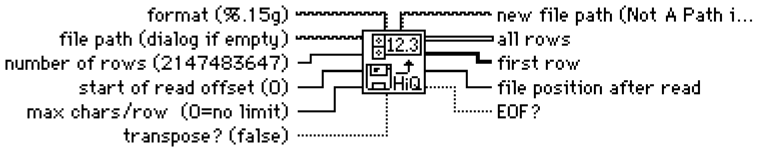
Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.



Read from HiQ Text File
Write to HiQ Text File

Read from HiQ Text File

Reads a specified number of rows from a HiQ, numeric text file beginning at a specified character offset and converts the data to a two-dimensional, double-precision array of numbers. You can optionally transpose the array. This VI opens the HiQ file before reading from it and closes the HiQ file after reading from it.


format (%.15g)
file path (dialog if empty)
number of rows (2147483647)
start of read offset (0)
max chars/row (0=no limit)
transpose? (false)
new file path (Not A Path i...
all rows
first row
file position after read
EOF?

file path (dialog if empty) consists of the path name of the file. If file path is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select Cancel in the dialog box.

number of rows (2147483647) is the maximum number of rows or lines the VI reads. For this VI, a row consists of a character string ending with a carriage return, line feed, or a carriage return followed by a line feed; a string ending with end of file; or a string that has the maximum line length specified by the **max characters per row** input. If **number of rows** <0, the VI reads the entire file. The default value is -1.

start of read offset (0) is the position in the file, measured in characters, at which the VI begins reading.

max chars/row (0=no limit) is the maximum number of characters the VI reads before ending the search for the end of a row or line. The default is 0, which means that there is no limit to the number of characters the VI reads.

transpose? (false) Set to TRUE to transpose the data after converting it from a string. The default value is FALSE.

format (%.15g) specifies how to convert the characters to numbers; the default is %.15g. Refer to the discussion of format strings and the Spreadsheet String To Array function.

new file path (Not A Path) is the path of the file from which the VI reads data. Not A Path is returned if you select **Cancel** from the dialog box.

all rows is the data read from the file in the form of a 2D array of double-precision numbers.

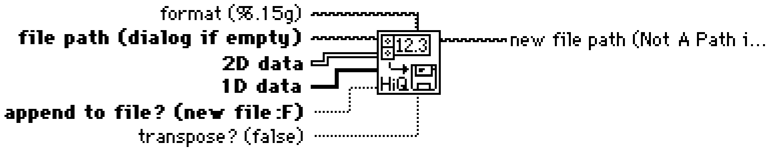
first row is the first row of the **all rows** array in the form of a 1D array of double-precision numbers. You can use this output when you want to read one row into a 1D array.

file position after read is the location of the file mark after the read; it points to the character in the file following the last character read.

EOF? is TRUE if you attempt to read past the end of the file.

Write to HiQ Text File

Converts a 1D or 2D array of double-precision numbers to a text string and writes the string to a new file or appends the string to an existing file. You can optionally transpose the data. This VI opens or creates the file beforehand and closes it afterwards. You can use this VI to create a text file that HiQ can read.


format (%.15g)
file path (dialog if empty)
2D data
1D data
append to file? (new file:F)
transpose? (false)
12.3
HiQ
new file path (Not A Path i...

TF **file path (dialog if empty)** consists of the path name of the file. If file path is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select Cancel in the dialog box.

TF **2D data** contains the double-precision numbers the VI writes to the file if 1D data is not wired or is empty.

TF **1D data** contains the double-precision numbers the VI writes to the file if this input is not empty. The VI converts the 1D array into a 2D array before optionally transposing it, converting it to a string and writing it to the file. If **transpose?** is FALSE, each call to this VI creates a new line or row in the file.

TF **append to file? (new file:F)** Set to TRUE if you want to append the data to an existing file. Set to FALSE (default value) if you want to write the data to a new file or to replace an existing file.

TF **transpose? (false)** Set to TRUE to transpose the data before converting it to a string. The default value is FALSE.

TF **format (%.15g)** specifies how to convert the numbers to characters. If the format string is %.15g (default), the VI creates a string long enough to contain the number, with fifteen digits to the right of the decimal point. If the format is %d, the VI converts the data to integer form using as many characters as necessary to contain the entire number. Refer to the discussion of format strings and the Array To Spreadsheet Stringfunction.

TF **new file path (Not A Path if...)** is the path of the file to which the VI wrote data. Not A Path is returned if you select **Cancel** from the dialog box.

HiQ Apple Event Command VIs

HiQ Apple Event Command VI Descriptions

HiQ PPC Data Transfer VIs

HiQ PPC Data Transfer VI Descriptions

HiQ File Transfer VIs

HiQ File Transfer VI Descriptions

Enter HiQ Script VI

Enter HiQ Script

Execute Script VI

Execute Script

Find an Open HiQ VI

Find an Open HiQ

Open HiQ VI

Open HiQ

Open Worksheet VI

Open Worksheet

Print Worksheet VI

Print Worksheet

Quit HiQ VI

Quit HiQ

Find an Open HiQ PPC Port VI

Find an Open HiQ PPC Port

HiQ PPC Connect VI

HiQ PPC Connect

HiQ PPC Disconnect VI

HiQ PPC Disconnect

HiQ PPC Read VI

HiQ PPC Read

HiQ PPC Read Real+ VI

HiQ PPC Read Real+

HiQ PPC Write VI

HiQ PPC Write

HiQ PPC Write Integer+ VI

HiQ PPC Write Integer+

HiQ PPC Write Real+ VI

HiQ PPC Write Real+

HiQ PPC Write Complex+ VI

HiQ PPC Write Complex+

Read from HiQ Text File VI

Read from HiQ Text File

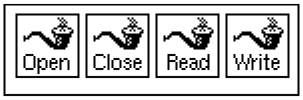
Write to HiQ Text File VI

Write to HiQ Text File

Named Pipe VI Descriptions (UNIX)

You can use these VIs to pass data between applications. Named Pipes make process synchronization simpler. The following illustration displays the Named Pipe VIs, which you access by selecting **Functions»Communication»PIPES**.

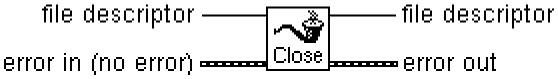
Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.



Close Pipe
Open Pipe
Read From Pipe
Write to Pipe

Close Pipe

Closes the named pipe specified by a **file descriptor**.



TF **file descriptor** is the **file descriptor** that you want to use when closing the pipe.

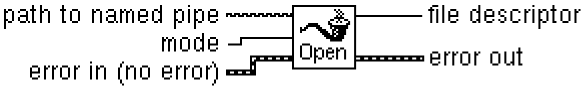
TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **file descriptor** is the **file descriptor** that you want to use when closing the pipe.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Open Pipe

Returns a **file descriptor**, which you pass to subsequent named pipe VIs. You can choose a path for the named pipe and whether you want to use the named pipe for writing or reading data.



TF **path to named pipe** is the path to the named pipe.

◂▸ **mode** is either read or write.

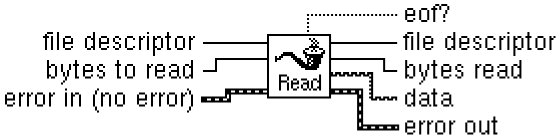
TF **error in** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **file descriptor** is the **file descriptor** that you want to use when reading and writing to the opened pipe.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Read From Pipe

Reads up to **bytes to read** data from the named pipe specified by **file descriptor**, returning the results in the **data** string output. For this VI to function, you must have opened the pipe as a read pipe. ReadPipe VI does not wait for data, so if the specified amount of data is not available, the VI returns whatever data is available. **EOF?** is TRUE if the other end of the pipe has been closed.



TF **file descriptor** is the **file descriptor** that you want to use when reading from the opened pipe.

TF **bytes to read** is the number of bytes to be read.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **file descriptor** is the **file descriptor** that you want to use when reading from the opened pipe.

TF **bytes read** is the number of bytes read, which may be less than the number of bytes in **bytes to read**.

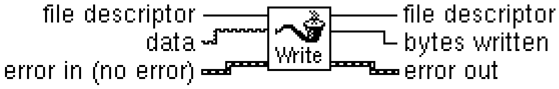
TF **data** is the data read from the pipe.

TF **EOF?** is the end of file.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Write to Pipe

Writes a **data** string to the named pipe specified by a **file descriptor**. For this VI to function, you must have opened the pipe as a write pipe.



TF **file descriptor** is the **file descriptor** that you want to use when writing to the opened pipe.

TF **data** is the data to write to the pipe.

TF **error in (no error)** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **file descriptor** is the **file descriptor** that you want to use when writing to the opened pipe.

TF **bytes written** is the number of bytes written, which may be less than the number of bytes in **data**.

TF **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

Close Pipe VI

Close Pipe

Open Pipe VI

Open Pipe

Read from Pipe VI

Read From Pipe

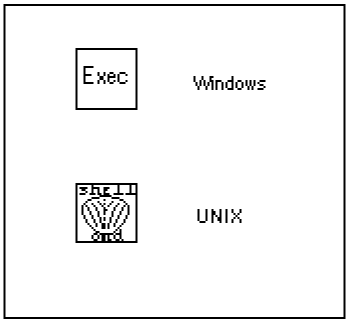
Write to Pipe VI

Write to Pipe

System Exec VI Descriptions (Windows and UNIX)

The following illustration displays the System Exec VIs, which you access by selecting **Functions »Communication**.

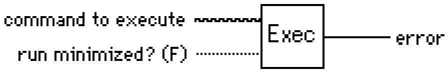
Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.



System Exec VI for Windows
System Exec VI for UNIX

System Exec (Windows)

Runs an executable program by name.



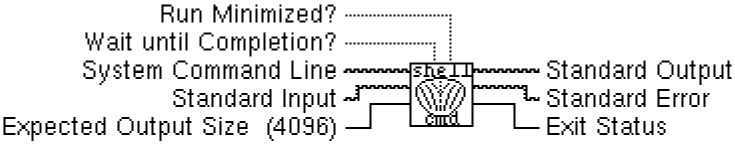
TF **command to execute**. Tells LabVIEW what command to call to execute your program.

TF **run minimized ? (F)**. If set to TRUE, minimizes the run of your executable program. The default is set to FALSE.

TF **error**. Returns an error.

System Exec (UNIX)

Runs an executable program by name. The System Exec VI also provides access to the standard input, output, and error I/O streams for the application you execute. With this VI, you can also choose whether you want the System Exec VI to wait for the application you execute to complete.



TF **System Command Line**.

TF **Run Minimized ?** If set to TRUE, minimizes the run of your executable program. The default is set to FALSE. In UNIX, LabVIEW does not use the **Run Minimized ?** parameter.

TF **Wait until Completion?** If TRUE, the string wired to **Standard Input** is available as input to the command, and the System Exec VI returns the **Standard Output** and **Standard Error** when the command completes. If FALSE, the System Exec VI runs the command in the background, and disables the input and output streams.

TF **Standard Input** is the input to the command.

TF **Expected Output Size (4096)** is used for efficiency reasons and should be a number slightly larger than the output size expected. By default, it has a value of 4,096 characters. The command still runs correctly if you exceed the output size, but the System Exec

VI is less efficient in LabVIEW memory usage.

TF **Standard Output** is the output from the command.

TF **Standard Error** is the error output from the command.

TF **Exit Status** is the integer status returned by the command.

System Exec for Windows

System Exec (Windows)

System Exec for Unix

System Exec (Unix)

TCP VI Overview

This topic discusses Internet Protocol (IP), Transmission Control Protocol (TCP), and internet addresses, and describes the LabVIEW TCP VIs.

Click here to access the TCP VI Descriptions topic.

TCP/IP (all platforms)

TCP/IP is a suite of communication protocols, originally developed for the Defense Advanced Research Projects Agency (DARPA). Since its development, it has become widely accepted, and is available on a number of computer systems.

The name TCP/IP comes from two of the best known protocols of the suite, the Transmission Control Protocol (TCP) and the Internet Protocol (IP). TCP, IP, and the User Datagram Protocol (UDP) are the basic tools for network communication.

TCP/IP enables communication over single networks or multiple, interconnected networks, which are known as an internetwork or internet. The individual networks can be separated by great geographical distances. TCP/IP routes data from one network or internet computer to another. Because TCP/IP is available on most computers, it can transfer information between diverse systems.

Transmission Control Protocol (TCP) ensures reliable transmission across networks, delivering data in sequence without errors, loss, or duplication. When you pass data to TCP, it attaches additional information and gives the data to IP, which puts the data into datagrams and transmits it. This process reverses at the receiving end, with TCP checking the data for errors, ordering the data correctly, and acknowledging successful transmissions. If the sending TCP does not receive an acknowledgment, it retransmits the data segment. For these reasons, TCP is usually the best choice for network applications.

Internet Protocol

Internet Protocol (IP) transmits data across the network. This low-level protocol takes data of a limited size and sends it as a *datagram* across the network. A datagram contains, among other things, the data and a header indicating the source and destination addresses. IP determines the correct path for the datagram to take across the network or internet, and sends the data to the specified destination. IP makes a best-effort attempt to deliver data, but cannot guarantee delivery. Also, because IP routes each datagram separately, it may arrive out of sequence. In fact, IP may deliver a single packet more than once if it is duplicated in transmission. IP does not determine the order of packets. Instead, higher-level protocols layered above IP order the packets and ensure reliable delivery. For this reason, IP is rarely used directly; instead, TCP and UDP, which are built on top of IP, are most often used to transfer information.

Using TCP

TCP is a connection-based protocol, which means that sites must establish a connection before transferring data. TCP permits multiple simultaneous connections.

You initiate a connection either by waiting for an incoming connection or by actively seeking a connection with a specified address. In establishing TCP connections, you have to specify both the address and a port at that address. A port is represented by a number between 0 and 65535. With UNIX, port numbers less than 1024 are reserved for privileged applications. Different ports at a given address identify different services at that address, and make it easier to manage multiple simultaneous connections.

You can actively establish a connection with a specific address and port using the TCP Open Connection VI. Using this VI, you specify the address and port with which you want to communicate. If the connection is successful, the VI returns a connection ID that uniquely identifies that connection. Use this connection ID to refer to the connection in subsequent VI calls.

You can use two methods to wait for an incoming connection:

- With the first method, you use the TCP Listen VI to create a listener and wait for an accepted TCP connection at a specified port. If the connection is successful, the VI returns a connection ID and the address and port of the remote TCP.
- With the second method, you use the TCP Create Listener VI to create a listener, and then use the Wait on Listener VI to listen for and accept new connections. Wait on Listener returns the same listener ID that was passed to the VI, as well as the connection ID for a connection. When you are finished waiting for new connections, you can use TCP Close to close a listener. You cannot read from or write to a listener.

The advantage of using the second method is that you can cancel a listen operation by calling TCP Close. This is useful in the case where you want to listen for a connection without using a timeout, but you want to cancel the listen when some other condition becomes true (for example, when the user presses a button).

When a connection is established, you can read and write data to the remote application using the TCP Read and TCP Write VIs.

Finally, use the TCP Close Connection VI to close the connection to the remote application. Notice that if there is unread data and the connection closes, that data may be lost. This behavior is dependent upon your operating system. For example, the Sun operating system implementation keeps unread data even after the remote application closes the connection, while Windows NT immediately deletes any unread data when a close connection is received. Connected parties should use a higher-level protocol to determine when to close the connection. Once a connection is closed, you may not read or write from it again.

Internet Addresses

Each host on an IP network has a unique, 32-bit internet address. This address identifies the network on the internet to which the host is attached, and the specific computer on that network. You use this address to identify the sender or receiver of data. IP places the address in the datagram headers, so that each datagram is routed correctly.

One way of describing this 32-bit address is the IP dotted decimal notation. This divides the 32-bit address into four 8-bit numbers. The address is written as the four integers, separated by decimal points. For example, the 32-bit address

10000100   00001101   00000010   00011110

is written in dotted decimal notation as

132.13.2.30

Another way of using the 32-bit address is by names that are mapped to the IP address. Network drivers usually perform this mapping by consulting a local hosts file that contains name to address mappings, or consulting a larger database using the Domain Name System to query other computer systems for the address for a given name. Your network configuration dictates the exact mechanism for this process, which is known as hostname.

TCP Client Example

The following discussion is a generalized description of how to use the components of the Client block diagram model with the TCP protocol.

Open Conn to Srvr Use the TCP Open Connection VI to open a connection to a server. You must specify the internet address of the server, as well as the port for the server. The address identifies a computer on the network. The port is an additional number that identifies a communication channel on the computer that the server uses to listen for communication requests. When you create a TCP server, you specify the port that you want the server to use for communication.

Exec Cmd on Srvr To execute a command on the server, use the TCP Write VI to send the command to the server. You then use the TCP Read VI to read back results from the server. With the TCP Read VI, you must specify the number of characters you want to read. This can be awkward, because the length of the response may vary. The server can have the same problem with the command, because the length of a command can vary.
Click here to access the Timeouts and Errors topic.

The following are several methods you can use to address varying sized commands:

- Precede the command and the result with a fixed size parameter that specifies the size of the command or result. In this case, read the size parameter, and then read the number of characters specified by the size. This option is efficient and flexible.
- Make each command and result a fixed size. When a command is smaller than the size, you can pad it out to the fixed size.
- Follow each command and result with a specific terminating character. To read the data, you then need to read data in small chunks until you get the terminating character.

Close Conn to Srvr Use the TCP Close Connection VI to close the connection to the server.

Timeouts and Errors

The TCP Client Example topic discussed communication protocol for the server. When you design a network application consider carefully what should happen if something fails. For example, if the server crashes, how would each of the client VIs handle it?

One solution is to make sure that each VI has a timeout. This way, if something fails to produce results, after a certain amount of time, the client continues to execute. In continuing, the client can try to reestablish execution, or it can report the error, and if necessary, shut the client application down. Select Error In and Error Out Clusters for more information about errors.

TCP Server Example

The following discussion explains how you can use TCP to fulfill each component of the general server model.

Initialize Server No initialization is necessary with TCP, so this step can be left out.

Wait for a Conn Use the TCP Listen VI to wait for a connection. You must specify the port that is used for communication. This port must be the same port that the client attempts to connect. The TCP Client Example topic provides more information about this VI..

Wait for a Cmd If a connection is established, read from that port to retrieve a command. As discussed in the TCP Client example, you must decide the format for commands. If commands are preceded by a length field, first read the length field, and then read the amount of data indicated by the length field.

Exec Cmd Execution of a command should be protocol independent, because it is performed on the local computer. When finished, pass the results to the next stage, where they are transmitted to the client.

TF Use the TCP Write VI to return results. As discussed in the TCP Client example, the data must be in a form that the client can accept.

TF Use the TCP Close Connection VI to close the connection.

TF This step can be left out with TCP, because everything is finished after you close the connection.

TCP Server with Multiple Connections

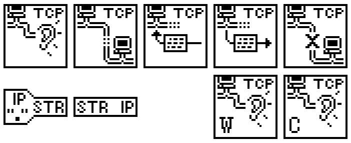
TCP handles multiple connections easily. You can use the methods described in the TCP Server Example to implement the components of a server with multiple connections.

TCP VI Descriptions

Click here to access the TCP VI Overview topic.

The following illustration shows the TCP VI palette, which you access by selecting **Functions»Communication»TCP**.

Click on one of the icons below for VI description information. You can also click on the text jumps below the icons to access VI descriptions.

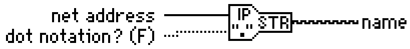


IP To String
String To IP
TCP Close Connection
TCP Create Listener
TCP Listen
TCP Open Connection
TCP Read
TCP Wait on Listener
TCP Write

For examples of how to use the TCP VIs, see the examples in the `examples\comm\tcpex.llb` library.

IP To String

Converts an IP network address to a string.



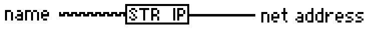
TF **net address** contains the network address.

TF **dot notation** determines whether **name** is in dot notation format.

TF **name** is the string equivalent of **net address**.

String To IP

Converts a string to an IP network address.

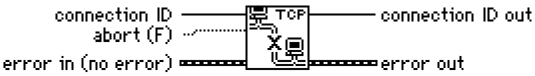


TF **name** contains the string you want to convert. If you do not specify a string as an input, the output is the current machines IP address.

TF **net address** is the IP network address equivalent to name.

TCP Close Connection

Closes the connection associated with **connection ID**.



TF **connection ID** is a network connection refnum that identifies the connection that you want to close.

TF **abort** determines whether LabVIEW closes the connection normally (the default value) or aborts the connection. Currently, this parameter is ignored.

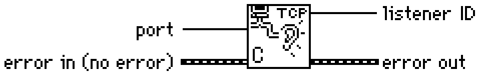
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **connection ID out** has the same value as **connection ID**.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TCP Create Listener

Creates a listener for a TCP connection.



TF **port** is the port that the VI uses to listen for a connection.

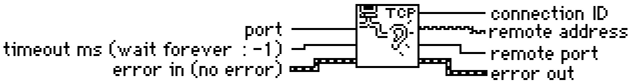
TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **listener ID** is a network connection refnum that uniquely identifies the Listener.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TCP Listen

Creates a listener and waits for an accepted TCP connection at the specified port.



When a listen on a given port begins, you may not use another TCP Listen VI to listen on the same port. For example, suppose a VI has two TCP Listen VIs on its block diagram. If you start a listen on port 2222 with the first TCP Listen VI, any attempts to listen on port 2222 with the second TCP Listen VI fails.

TF **port** is the port that the VI uses to listen for a connection.

TF **timeout** is in milliseconds. If the connection is not established in the specified time, the VI completes and returns an error. The default value for timeout is -1, which means wait indefinitely.

TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error` See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TF **connection ID** is a network connection refnum that uniquely identifies the TCP connection. You use this **connection ID** value to refer to this connection in subsequent VI calls.

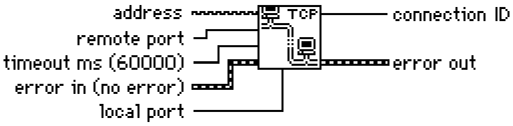
TF **remote address** is the address of the remote machine associated with the TCP connection. This address is in IP dot notation format. See the Internet Protocol (IP) topic for a description of IP dot notation.

TF **remote port** is the port the remote system uses for the TCP connection.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TCP Open Connection

Attempts to open a TCP connection with the specified address and port.



TF **address** is the address with which you want to establish a TCP connection. This address can be in IP dot notation or it can be a hostname. See the Internet Addresses topic for a description of valid specifications for **address**.

TF **remote port** is the port of the specified address with which you want to establish a TCP connection.

TF **timeout** is in milliseconds. If the connection is not established in the specified time, the VI completes and returns an error. The default value for timeout is 60,000 ms (1 minute). A timeout value of -1 means wait indefinitely.

TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

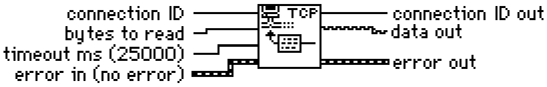
TF **local port** is the port where you specify the local TCP connection port. Some servers only allow connections to clients that use port numbers within a specified range that is dependent on the server. If the value is 0, TCP chooses an unused port.

TF **connection ID** is a network connection refnum that uniquely identifies the TCP connection. You use this **connection ID** value to refer to this connection in subsequent VI calls.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TCP Read

Receives up to **bytes to read** bytes from the specified TCP connection, returning the results in **data out**.



TF **connection ID** is a refnum identifying the TCP connection.

TF **bytes to read** is the number of bytes to read from the specified connection.

TF **timeout** is in milliseconds. If the operation does not complete in the specified time, the VI completes and returns an error. The default value
is 25,000. A timeout value of -1 means wait indefinitely.

TF **error in** describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error` See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

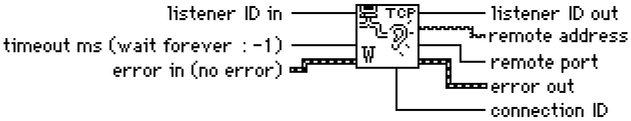
TF **connection ID out** has the same value as **connection ID**.

TF **data out** is a string that contains the data read from the TCP connection.

TF **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

TCP Wait on Listener

Waits for an accepted TCP connection at the specified port.



listener ID in is a network connection refnum identifying the Listener.

timeout is in milliseconds. If the connection is not established in the specified time, the VI completes and returns an error. The default value is 25,000. A timeout value of -1 means wait indefinitely.

error in describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error` See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

listener ID out has the same value as **listener ID in**.

remote address is the address of the remote machine associated with the TCP connection. This address is in IP dot notation format. See the Internet Addresses topic for a description of IP dot notation.

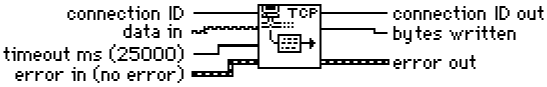
remote port is the port the remote system uses for the TCP connection.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

connection ID is a network connection refnum that uniquely identifies the TCP connection. You use this **connection ID** value to refer to this connection in subsequent VI calls.

TCP Write

Writes the string **data in** to the specified TCP connection.



connection ID is a refnum identifying the TCP connection.

data in is a string that contains the data to write to the TCP connection.

timeout is in milliseconds. If the operation does not complete in the specified time, the VI completes and returns an error. The default value is 25,000. A timeout value of -1 means wait indefinitely.

error in describes error conditions that occur prior to the execution of this VI. The default input of this cluster is `no error`. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

connection ID out has the same value as **connection ID**.

bytes written is the number of bytes the VI writes to the specified connection.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error In and Error Out Clusters topic for a further description of the **error in** and **error out** clusters.

LabVIEW and TCP/IP

You can use the TCP/IP suite of protocols with LabVIEW on all platforms. LabVIEW has a set of TCP and UDP VIs that you can use to create client or server VIs.

Setup

Before you can use TCP/IP, you need to make sure that you have the right setup. This setup varies, depending on the computer you use.

Windows 3.x
Windows 95/NT
Macintosh
UNIX

Windows 3.x

To use TCP/IP, you must install an ethernet board along with its low-level driver. In addition, you must purchase and install TCP/IP software that includes a Windows Sockets (WinSock) DLL conforming to standard 1.1. WinSock is a standard interface that enables application communication with a variety of network drivers. Several vendors provide network software that includes the WinSock DLL. Install the ethernet board, the board drivers, and the WinSock DLL according to the software vendor instructions.

Several vendors supply WinSock drivers that work with a number of boards. You can contact the vendor of your board to inquire if they offer a WinSock DLL you can use with the board. Install the WinSock DLL according to vendor instructions.

National Instruments has tested a number of WinSock DLLs to verify which work correctly. These tests showed that many DLLs do not fully comply with the standard, so you may want to try a demo version of a DLL before you buy the real version. You can usually obtain a demo version from the manufacturer. Most demo versions are fully functional, but they expire after a certain amount of time.

If you have access to the internet, several of these demos are available by anonymous ftp from `sunsite.unc.edu`. in the `directory/pub/micro/pc-stuff/ms-windows/ winsock/packages`. Refer to your LabVIEW Release Notes for a detailed list of WinSock DLLs tested by National Instruments.

Windows 95/NT

TCP support is built into Windows NT. You do not need to use a third-party DLL to communicate using TCP.

Macintosh

TCP/IP is built in to Macintosh operating system version 7.5. To use TCP/IP with an earlier system, you need to install the MacTCP driver software, available from the Apple Programmer Developer Association (APDA). You can contact APDA at (800) 282-2732 for information on licensing the MacTCP driver.

UNIX

TCP/IP support is built-in. Assuming your network is configured properly, no additional setup for LabVIEW is necessary.

TCP Close Connection VI

TCP Close Connection

TCP Create Listener VI

TCP Create Listener

TCP Listen VI

TCP Listen

TCP Open Connection VI

TCP Open Connection

TCP Read VI

TCP Read

TCP Wait on Listener VI

TCP Wait On Listener

TCP Write VI

TCP Write

IP To String VI

IP To String

String To IP VI

String To IP

Communications Common Questions

This section answers common questions about LabVIEW and networking communications. Questions are divided into sections according to the relevant platform: Questions for All Platforms, Windows Only, and Macintosh Only. Please contact National Instruments if you have further questions or suggestions regarding LabVIEW.

Questions for All Users
Questions for Windows Users
Questions for Macintosh Users

Questions for All Users

How do I use LabVIEW to communicate with other applications?

Communicating with other applications, often called interprocess or interapplication communication, can be done with the standard networking protocols on each platform. LabVIEW has support for TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) on all platforms.

Windows

In addition, LabVIEW for Windows supports DDE (Dynamic Data Exchange).

Macintosh

In addition, LabVIEW for Macintosh supports IAC (Interapplication Communication). IAC includes Apple Events and PPC (Program to Program Communication).

UNIX

LabVIEW for UNIX only supports TCP and UDP.

In addition, for many instrumentation applications, file I/O provides a simple, adequate method of sending information between applications.

How do I launch another application with LabVIEW?

On Windows and UNIX, use the System Exec VI (**Functions**»**Communication**). On Macintosh, use AESend Finder Open (**Functions** »**Communication**»**AppleEvent**).

When would I want to use UDP instead of TCP?

Typically, UDP is used in applications where reliability is not critical. For example, an application might transmit informative data to a destination frequently enough that a few lost segments of data are not problematic. Also, UDP can be used to broadcast to any machine(s) wanting to listen to the server.

What port numbers can I use with TCP and UDP?

A port is represented by a number between 0 and 65535. With UNIX, port numbers less than 1024 are reserved for privileged applications (e.g. ftp). When you specify a local port, you can use the value of 0 which would cause TCP and UDP to choose an unused port.

Why cant I broadcast using UDP?

Because the broadcast address varies among domains, you need to find out from your system administrator what broadcast address to use. For example, the broadcast address 0xFFFFFFFF is not correct for your domain. Additionally, your machine may default to not allow broadcasting unless the process is run by the root user.

Questions for Windows Users

What winsock.dll can I use with LabVIEW?

This question pertains to Windows 3.x only, as Windows 95 and Windows NT include this file in their operating systems.

Any WinSock driver that conforms to standard 1.1 should work with LabVIEW. You can find Information regarding National InstrumentsÕ in-house testing of the winsock.dll in your online Release Notes.

Recommended:

- TCPOpen version 1.2.2 from Lanera Corporation
  (408) 956-8344.
- Trumpet (version 1.0 tested). Available via anonymous ftp to ftp.utas.edu.au in the directory /pc/trumpet/winsock/*. For information send electronic mail to trumpet-info@petros.psychol.utas.eduau.
- Super-TCP version 3.0 R1 from Frontier Technologies Corporation (414) 241-4555.
- NEWT/Chameleon version 3.11 from NetManage, Inc.
  (408) 973-7171.
- Windows for Workgroups winsock.dll from Microsoft.

Not Recommended:

National InstrumentsÕ limited testing of these products yielded various problems and crashes while attempting TCP/IP communication. At this time, National Instruments can neither recommend these products nor support customers attempting TCP/IP communication with these winsock.dlls.

- Distinct TCP/IP version 3.1 from Distinct Corporation
  (408) 741-0781.
- PCTCP version 2.x from FTP Software, Inc. (508) 685-4000.

How do I call an Excel macro using DDE?

Use the DDE Execute VI. This VI tells the DDE server to execute a command string in which you specify the action for Excel to perform and the name of the macro. Make sure to include the correct parentheses and brackets around the command. Refer to the *Excel UserÕs Guide* for more information. Some common examples are shown below:

Command String	**Action**
[RUN("MACRO1")]	Runs MACRO1
[RUN("MACRO1!R1C1")]	Runs MACRO1 starting at Row 1, Column 1
[OPEN("C:\EXCEL\SURVEY.XLS")]	Opens SURVEY.XLS

Why doesn't DDE Poke work with Microsoft Access?

Microsoft Access cannot accept data directly from DDE clients. To get data into an Access database you must create a macro in that database to import the data from a file. In the simple case these macros need only be two actions long. First do a SetWarnings to suppress Access dialogs, then do a TransferSpreadsheet or TransferText to get the data. After this macro is defined, you can call the macro by sending an execute to that database with the macro name as the data. Refer to the example VI `Sending Data to Access.vi` located in `examples\network\access.llb` to see how this is done.

What commands do I use to communicate with a non-LabVIEW application using DDE?

The DDE commands are specific to the application with which you are interfacing. Consult the *LabVIEW User Manual* or online reference for the specific application to see which commands are available.

How do I install LabVIEW as a shared application on a file server?

Provided the user has a license for each client, the process is as follows:

- Install the LabVIEW Full Development System on the server. (Unless there is NI hardware on the server, it is not necessary to install NI-DAQ or GPIB.DLL).
- Each local machine should use its own `labview.ini` file for LabVIEW preferences. If a `labview.ini` file does not already exist on the local machine, you can create this (empty) text document using a text editor, such as Microsoft Notepad. The first line of `labview.ini` must be [labview]. To have a local setting for `labview.ini`, LabVIEW requires a command line argument containing the path to the preferences. For example, if the `labview.exe` file is on drive `W:\LABVIEW` and the `labview.ini` file is on `C:\LVWORK` (the hard drive on the local machine), modify the command line option of the LabVIEW icon in Program Manager to be:

```
W:\LABVIEW\LABVIEW.EXE-pref
```

Note: pref must be lower case. Additionally, each local machine must have its own LabVIEW temporary directory. This is done in LabVIEW by choosing EditÈPreferences....

- You do not need GPIB.DLL on the server machine, unless you are using a GPIB board on this machine. You then need the `gpibdrv` file in the LabVIEW directory. Then, on each machine that has a GPIB board, you need to install the driver for that board. You can do this by either using the drivers that came with the board, or by doing a custom LabVIEW installation, in which only the desired GPIB driver is installed on the local machine.
- The same procedure for GPIB.DLL applies to NI-DAQ.

Why does the Synch DDE Client / Server hang on NT after many transfers?

There are some problems with DDE in LabVIEW for NT that result in VIs hanging during DDE Poke and DDE Request operations. This limitation is specific to Windows NT.

Are there plans for LabVIEW to support OLE?

OLE (Object Linking and Embedding) is a way of embedding objects from one application into another application. For example, a spreadsheet might be included on a word processing document. When the text document is loaded, the current values that are found in the spreadsheet are automatically included into the document. National Instruments is currently investigating support for OLE for a future version of LabVIEW; however, no dates have been set on when a version including OLE support will be available.

OLE Automation is a technique by which Automation servers can expose methods and properties to other applications and Automation controllers can access the methods and properties of other applications. LabVIEW 4.x is an OLE Automation controller. There is a library of VIs, which you can use to execute properties and methods exposed by Automation servers.

Questions for Macintosh Users

What is a target ID?

Target ID is used in the Apple Events and PPC VIs on the Macintosh; it serves as a reference to the application that you are trying to launch, run, or abort. The target ID to an application can be accessed by one of the following commands:

Get Target ID - takes the name and location of the application as input, searches the network for it, and returns the target ID;

PPC Browser - pops up a dialog box that you can use to select an application, which may be across the network or on your computer.

The target ID you generated at the beginning of your VI should be used as an input to all subsequent Apple Event functions to open, print, close, or run the application.

Why can't I see my application in the dialog box generated by PPC Browser?

If the application you want to connect cannot be used with Apple Events, it does not show up in the PPC Browser dialog box. If you are certain that the desired application supports Apple Events, make sure that you have turned on File Sharing on your Macintosh. Select **Control » Sharing Setup** to turn File Sharing on.

How can I close the Finder using Apple Events?

Use the VI AESend Quit Application to quit the Finder or any other application.

PPC Error Codes

Code	Name	Description
-900	notInitErr	PPC Toolbox has not been initialized.
-902	nameTypeErr	Invalid or inappropriate locationKindSelector in locationName.
-903	noPortErr	Invalid port name. Unable to open port or bad portRefNum.
-904	noGlobalsErr	The system is unable to allocate memory. This is a critical error, and you should restart.
-905	localOnlyErr	Network activity is currently disabled.
-906	destPortErr	Port does not exist at destination.
-907	sessTableErr	PPC Toolbox is unable to create a session.
-908	noSessionErr	Invalid session reference number.
-909	badReqErr	Bad parameter or invalid state for this operation.
-910	portNameExistsErr	Another port is already open with this name (perhaps in another application).
-911	noUserNameErr	User name unknown on destination machine.
-912	userRejectErr	Destination rejected the session request.
-913	noMachineNameErr	User has not named his Macintosh in the Network Setup Control Panel.
-914	noToolboxNameErr	A system resource is missing.
-915	noResponseErr	Unable to contact destination application.
-916	portClosedErr	The port was closed.
-917	sessClosedErr	The session has closed.
-919	badPortNameErr	PPCPortRec is invalid.
-922	noDefaultUserErr	User has not specified owner name in Sharing Setup Control Panel.
-923	notLoggedInErr	The default userRefNum does not yet exist.
-924	noUserRefErr	Unable to create a new userRefNum.
-925	networkErr	An error has occurred in the network.

-926	noInformErr	PPCStart failed because destination did not have an inform pending.
-927	authFailErr	UserÕs password is wrong.
-928	noUserRecErr	Invalid user reference number.
-930	badServiceMethodErr	Service method is other than ppcServiceRealTime.
-931	badLocNameErr	Location name is invalid.
-932	guestNotAllowedErr	Destination port requires authentication.

AppleEvent Error Codes

Code	Name	Description
-1700	errAECoercionFail	Data could not be coerced to the requested descriptor type.
-1701	errAEDescNotFound	Descriptor record was not found.
-1702	errAECorruptData	Data in an Apple event could not be read.
-1703	errAEWrongDataType	Wrong descriptor type.
-1704	errAENotAEDesc	Not a valid descriptor record.
-1705	errAEBadListItem	Operation involving a list item failed.
-1706	errAENewerVersion	Need a newer version of Apple Event Manager.
-1707	errAENotAppleEvent	The event is not an Apple event.
-1708	errAEReplyNotValid	AEResetTimer was passed an invalid reply parameter.
-1709	errAEERReplyNotValid	AEResetTimer was passed an invalid reply parameter.
-1710	errAEUnknownSendMode	Invalid sending mode was passed.
-1711	errAEWaitcanceled	User canceled out of wait loop for reply or receipt.
-1712	errAETimeout	Apple event timed out.
-1713	errAENoUserInteraction	No user interaction allowed.
-1714	errAENotASpecialFunction	Wrong keyword for a special function.
-1715	errAEParamMissed	Handler did not get all required parameters.

-1716	errAEUnknownAddressType	Unknown Apple event address type.
-1717	errAEHandlerNotFound	No handler in the dispatch tables fits the parameters to AEGetEventHandler or AEGetCoercionHandler.
-1718	errAEReplyNotArrived	The contents of the reply you are accessing have not arrived yet.
-1719	errAEIllegalIndex	Index is out of range in a put operation.

LabVIEW Specific PPC Error Codes

Code	Name	Description
1	errNoPPCToolBox	The PPC ToolBox either does not exist (it requires System 7.0 or later) or it could not be initialized.
2	errNoGlobals	The CIN in the PPC VI could not get its globals.
3	errTimedOut	The PPC operation exceeded its timeout limit.
4	errAuthRequired	The target specified in the PPC Start Session VI required authentication, but the authentication dialog was not allowed.
5	errbadState	The PPC Start Session VI found itself in an unexpected state.

TCP and UDP Error Codes

Code	Name	Description
53	mgNotSupported	LabVIEW: Manager call not supported
54	ncBadAddressErr	The net address was ill formed
55	ncInProgressErr	Operation is in progress.
56	ncTimeOutErr	Operation exceeded the user-specified time limit.
57	ncBusyErr	The connection was busy.
58	ncNotSupportedErr	Function not supported.
59	ncNetErr	The network is down, unreachable, or has been reset.
60	ncAddrInUseErr	The specified address is currently in use.
61	ncSysOutOfMemErr	System could not allocate necessary memory.
62	ncSysConnAbortedErr	System caused connection to be aborted.
63	ncConnRefusedErr	Connection is not established.
65	ncAlreadyConnectedErr	Connection is already established.
66	ncConnClosedErr	Connection was closed by peer.

LabVIEW Specific Error Codes for AppleEvent Messages

Code	Name	Description
1000	kLVE_InvalidState	The VI is in a state that does not allow it to run.
1001	kLVE_FPNotOpen	The VI front panel is not open.
1002	kLVE_CtrlErr	The VI has controls on its front panel that are in an error state.
1003	kLVE_VIBad	The VI is broken.

1004	kLVE_NotInMem	The VI is not in memory.

DDE Error Codes

Code	Name	Description
00000		No error.
14001	DDE_INVALID_REFNUM	Invalid refnum.
14002	DDE_INVALID_STRING	Invalid string.
14003	DDEML_ADVACKTIMEOUT	Request for a synchronous advise transaction has timed out.
14004	DDEML_BUSY	Response set the `DDE_FBUSY` bit.
14005	DDEML_DATAACKTIMEOUT	Request for a synchronous data transaction has timed out.
14006	DDEML_DDL_NOT_INITIALIZED	DDEML called without first calling `DdeInitialize`, or was passed an invalid instance identifier.
14007	DDEML_DLL_USAGE	A monitor or client-only application has attempted a DDE transaction.
14008	DDEML_EXECACKTIMEOUT	Request for a synchronous execute transaction has timed out.
14009	DDEML_INVALIDPARAMETER	Parameter not validated by the DDML.
14010	DDEML_LOW_MEMORY	Server application has outrun client, consuming large amounts of memory.
14011	DDEML_MEMORY_ERROR	A memory allocation failed.
14012	DDEML_NOTPROCESSED	Request or poke is for an invalid item
14013	DDEML_NO_CONV_ESTABLISHED	Client conversation attempt failed.
14014	DDEML_POKEACTIMEOUT	Transaction failed.
14015	DDEML_POSTMSG_FAILED	Request for a synchronous poke transaction has timed out.
14016	DDEML_REENTRANCY	An application with a synchronous transaction in progress attempted to initiate another transaction, or a DDEML callback function called `DdeEnableCallback`.
14017	DDEML_SERVER_DIED	Server-side transaction attempted on conversation terminated by client, or service terminated before completing a transaction.

14018	DDEML_SYS_ERROR	Internal error in the DDMEML.
14019	DDEML_UNADVACKTIMEOUT	Request to end advise has timed out.
14020	DDEML_UNFOUND_QUEUE_ID	Invalid transaction identifier passed to DDEML function.