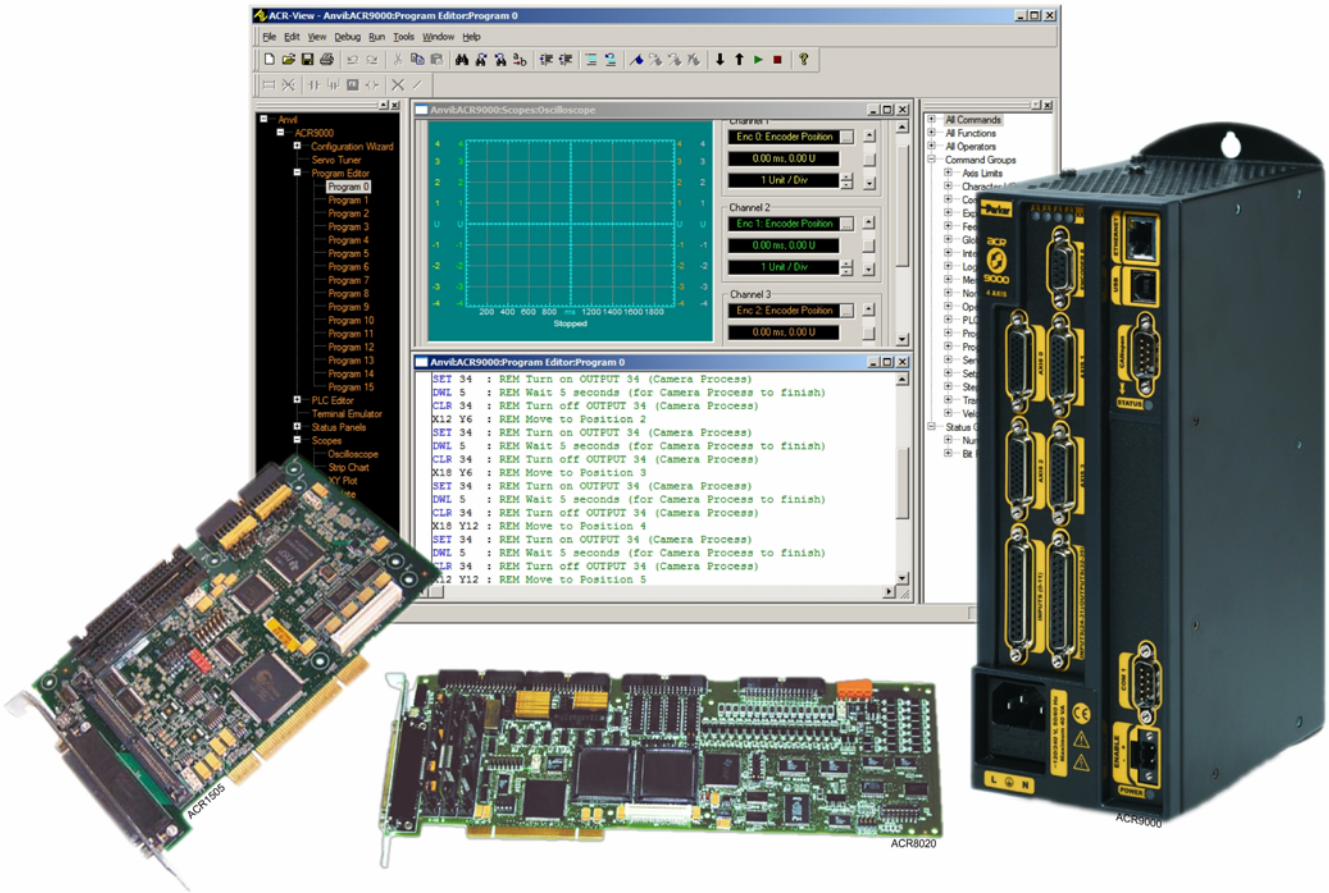
Parker Automation

p/n 88-025359-01C

ComACRsrvr User's Guide for ACR Series Products

Effective: October 2005



User Information

Warning — ACR Series products are used to control electrical and mechanical components of motion control systems. You should test your motion system for safety under all potential conditions. Failure to do so can result in damage to equipment and/or serious injury to personnel.

The ComACRsrvr Communications Server product and the information in this user guide are the proprietary property of Parker Hannifin Corporation or its licensers, and may not be copies, disclosed, or used for any purpose not expressly authorized by the owner thereof.

Since Parker Hannifin constantly strives to improve all of its products, we reserve the right to change this guide, and software and hardware mentioned therein, at any time without notice.

In no event will the provider of the equipment be liable for any incidental, consequential, or special damages of any kind or nature whatsoever, including but not limited to lost profits arising from or in any way connected with the use of the equipment or this guide.

Technical Assistance

Contact your local automation technology center (ATC) or distributor.

North America and Asia
Parker Hannifin
5500 Business Park Drive
Rohnert Park, CA 94928
Telephone: (800) 358-9070 or (707) 584-7558
Fax: (707) 584-3793
Email: emn_support@parker.com
Internet: http://www.parkermotion.com

Germany, Austria, Switzerland
Parker Hannifin
Postfach: 77607-1720
Robert-Bosch-Str. 22
D-77656 Offenburg
Telephone: +49 (0) 781 509-0
Fax: +49 (0) 781 509-176
Email: sales.hauser@parker.com
Internet: http://www.parker-emd.com

Europe (non-German speaking)
Parker Hannifin plc
Electromechanical Automation, Europe
Arena Business Centre
Holy Rood Close
Poole
Dorset, UK
BH17 7BA
Telephone: +44 (0) 1202 606300
Fax: +44 (0) 1202 606301
Email: support.digiplan@parker.com
Internet: http://www.parker-emd.com

Italy
Parker Hannifin
20092 Cinisello Balsamo
Milan, Italy via Gounod, 1
Telephone: +39 02 6601 2478
Fax: +39 02 6601 2808
Email: sales.sbc@parker.com
Internet: http://www.parker-emd.com

Parker Automation

Technical Support E-mail

emn_support@parker.com

Table of Contents

Change Summary

Change Summary

Revision C Changes

This document, 88-025359-01C, supercedes 88-025359-1B. Changes associated with ACR9000 User Guide revisions, and document clarifications and corrections are as follows:

Topic	Description
WatchdogTimout	Added an event to notify when the watchdog has timed out.
WatchdogReconnect	Added an event to reconnect the watchdog after it has timed out.
bstrUSBSerialNumber	Added a property for retrieving the USB serial number of the ACR controller.

Revision B Changes

This document, 88-025359-01B, supercedes 88-025359-1A. Changes associated with ACR9000 User Guide revisions, and document clarifications and corrections are as follows:

Topic	Description
Legacy example	Corrected example for section titled “Using the ACR legacy Support Kit”.
nCard	Expanded explanation.
AddACRCustom	Expanded explanation.
AddACRGroup	Expanded explanation.
GetACRCustom	Expanded explanation.
GetLocalAddr	Expanded example.
GetLocalArrayAddr	Corrected example
DownloadFile	Corrected example.
DownloadOS	Added warning.
fMoveACC	Expanded explanation.

Communications Server

The Communications Server (ComACRsrvr.dll) is a 32-bit OLE automation server that provides communications between ACR controllers and PC software applications. It is compatible with any 32-bit software application or programming environment that uses an OLE automation component, including:

- Microsoft .NET
- Visual Basic
- Visual C++
- Delphi
- Software packages that support Microsoft's Component Object Model (COM):
  - Wonderware's Factory Suite 2000
  - National Instruments LabVIEW

The ACR-View installation program installs the ComACRsrvr.dll file in the Windows\System (Windows 95/98/XP) or WinNt\System32 (Windows NT/2000) directory.

Communications Overview

To begin communications, an application requests a connection to the ACR controller through the Communications Server. The Communications Server manages the actual connection to each controller, and can feed information from a particular controller to all client applications that require the information.

The Communications Server makes one connection for each ACR controller per communication media (Ethernet, Bus, USB or Serial, depending on the communication options available for your particular controller). This connection is then shared across all client users of that connection/controller pair. For example, a terminal application created in Visual Basic and a terminal in ACR–View can maintain connections to the same ACR controller's USB connection. Both applications receive the responses coming from the controller and do not compete for data.

Alternatively, a status application created in C++ can use the communication server to get information from one ACR controller while ACR-View uses the communication server to talk to a different ACR controller.

Note: For more information, see the Hardware Installation Guide for your controller.

RS-232

You must specify for the Communications Server the COM port and speed on the PC (personal computer) to use for the connection. The Communications Server supports up to 32 simultaneous serial connections (any ports between COM1 to COM256).

Ethernet

You will need to specify the controller's IP address to the Communications Server, and the IP address must be reachable on the network. Each ACR controller is setup with a default IP address (192.168.10.40) and network mask (255.255.0.0) at the factory. You can change the IP address and network mask with the `IP` and `IP MASK` commands. The Communications Server supports up to 32 simultaneous Ethernet connections (each to different IP addresses).

USB

You will need to specify the controller's unique serial number to the communication server. The serial number is printed on the side of the ACR controller. By default, if the Communications Server is given a serial number of zero, it will connect to the first ACR controller it finds attached using USB. The client application can then query the communication server and save the serial number for later use. The Communications Server supports up to 32 simultaneous USB connections.

Bus

You will need to specify the bus controller's card number and bus type (ISA or PCI) to the communication server. When installed into the PC, each bus controller is assigned a unique index; this index is the card number. The Communications Server supports up to 32 simultaneous Bus connections.

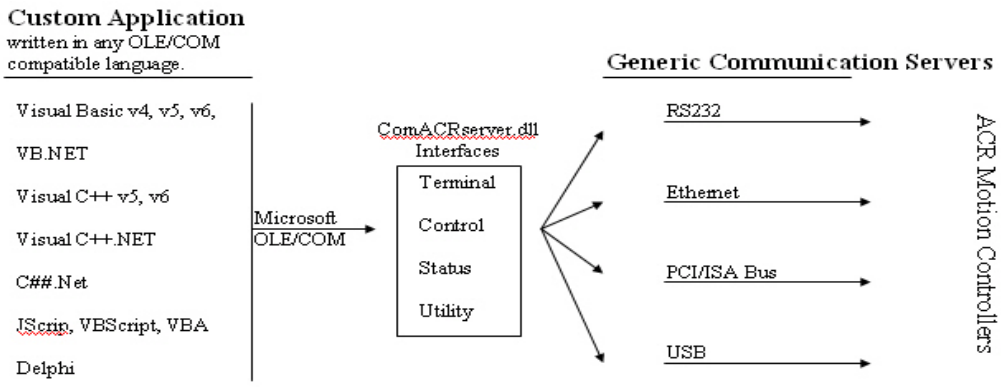
ComACRsrvr.dll Overview

The ACR motion controllers provide a common collection of services and functions across the product series. For a typical application, motion control programs are written in the AcroBasic language using the ACR-View development environment, and run on the controller.

In some cases, a custom program running on a PC might need to communicate and interact directly with an ACR controller. To aid PC communications with the ACR Series controllers, you can use the set of OLE (Object Linking and Embedding) automation interfaces that are installed along with the ACR-View application. The four interfaces are part of the Communication Server (ComACRsrvr.dll). The interfaces (Status, Control, Terminal, and Utility) provide access to the ACR motion controller independent of the physical communications layer.

The Communications Server provides two layers of abstraction: The first aggregates the physical communication layers into a functional API (Application Program Interface) that contains the combined properties and methods of the four interfaces. The second removes the dependency of using a specific programming language, by providing a COM component through which many Windows based programming environments can access the API.

For example, using any OLE/COM compatible language, an application can use the Status Interface to request the value of parameter P6415. This is performed without regard to the underlying communication method (Serial, Ethernet, USB, PCI or ISA Bus). The following diagram illustrates this concept:



Communications Server API Overview

Before using the Communication Server API, it helps to understand the underlying design. The features available in the ACR Communication Server are sorted into four Interfaces. Each Interface covers a specific area of functionality.

- The **Terminal** Interface provides a freeform, ASCII character based, input and output stream.
- The **Status** Interface allows the retrieval of specific parameters, flags and other data items.
- The **Control** Interface lets a client application change parameters, flags, and other data, as well as perform moves.
- The **Utility** Interface contains miscellaneous functions, including Uploading and Downloading files and firmware.

The functionally of each Interface is specific, but all four share a common set of properties and methods for dealing with communications. This replication of communication properties and methods provides you with the flexibility to use any Interface on a connected ACR device, independent of the other interfaces. While flexible, each instance of an Interface must first **Connect()** to an ACR series product before using the interface.

Internal to the Communication Server, there is only one connection to an ACR device. After the first Interface connects (which ever interface that may be), each call to **Connect()** just shares the single connection. For this reason, there is no extra overhead to having to call **Connect()**. For example, if an application will use three of the four interfaces, you must create and then connect each interface using the same connection information.

Communications

All of the Communications Server's interfaces share properties and methods related to communication. These properties and methods provide enough information for each interface to independently establish a communications link with the ACR controller.

To connect to a controller, a program sets the properties for the communication layer (i.e. set the COM port and speed for serial communication; set the IP address for Ethernet; etc.) and then calls the **Connect()** method.

The above connection procedure is required for each instantiation of an Interface. Each instantiation can connect to a different ACR device or a different communication transport, up to the limits of the underlying generic communication server.

Connecting to an ACR Controller

For an application to communicate with an ACR Controller, the application must first call the **Connect()** method on an interface. When calling the **Connect()** method there are two steps that the Communications Server performs.

1. The Communications Server establishes a connection with the physical medium (serial, Ethernet, etc.) using the device drivers on the system to secure the communication resource (ports, sockets, etc.) needed to talk to the connected ACR Controller. This is only performed by the very first Interface to call the **Connect()** method. Subsequent calls to the **Connect()** method do not perform this step. Instead, subsequent interfaces register themselves as users. When the last user closes down, the resources are freed.
2. By default, the Communications Server tries to verify that an ACR Controller is connected to the PC. To do this, the Communications Server sends information to the ACR Controller and inspects the reply.

   You can turn off this behavior, and not perform the check. Prior to calling the **Connect()** method, set the **bOnConnectTest** to FALSE. For example, you can do this when you know the ACR Controller is present.

   Alternatively, you can call the **TestConnect()** method to test at any time whether a connection is present. See examples three and four below.

After successfully executing the **Connect()** method, all other methods on that interface are available. In the examples below, the Control Interface is connected and then the “cntl” object is available to call any method on the Control Interface. For example, through the Control Interface you can change p-Parameters.

For expanded examples of how to code using the Communications Server, see the sample programs on the ACR CD.

Example 1

Following is an example in VB6 using early binding to connect using Ethernet, Serial and Bus, and then using the Interface to change the values of the Jog Limits for Axis0:

```
‘// Initialize the Interface
Public cntl As Control
Set cntl = New Control

‘// Connect to the Interface in one of the three(3) protocols
‘// Ethernet
cntl.bstrIP = “192.168.10.40”     ‘// Use IP the ACR is setup with
cntl.Connect 3, 0                 ‘// The 3 = Transport type Ethernet
‘// Or Serial
cntl.nBPS = 38400      ‘// Set the baud rate
cntl.nPort = 1         ‘// Using Com Port 1
cntl.Connect 2, 0      ‘// The 2 = Transport type Serial
‘// Or Bus
cntl.nBus = 0          ‘// Using a PCI type bus card
cntl.Connect 1, 0      ‘// The 1 = Transport type Bus
```

```
‘// Change the Jog Limits for Axis0 for any Connection Type
cntl.SetParmFloat 12334, 50, True  '// Set Pos Jog Limit
cntl.SetParmFloat 12335, -50, True '// Set Neg Jog Limit
```

Example 2

Following is an example in C#.NET using Ethernet, Serial and Bus, and then using the Interface to change the values of the Jog Limits for Axis0:

```
‘// Initialize the Interface
BOXBRIDGELib.Control cntl;

‘// Connect to the Interface in one of the three(3) protocols
// Ethernet
cntl.bstrIP = “192.168.10.40”;   // Use IP the ACR is setup with
cntl.Connect(3,      // The 3 = Transport type Ethernet
0);
‘// Or Serial
cntl.nBPS = 38400    ‘// Set the baud rate
cntl.nPort = 1       ‘// Using Com Port 1
cntl.Connect 2, 0    ‘// The 2 = Transport type Serial
‘// Or Bus
cntl.nBus = 0        ‘// Using a PCI type bus card
cntl.Connect 1, 0    ‘// The 1 = Transport type Bus

‘// Change the Jog Limits for Axis0 for any Connection Type
cntl.SetParmFloat 12334, 50, True '// Set Pos Jog Limit
cntl.SetParmFloat 12335, -50, True '// Set Neg Jog Limit
```

Example 3

In this VB6 Example using early binding, the ACR verification test is turned off and done manually by the client application.

```
‘// Initialize the Interface
Public WithEvents term As Terminal
Set term = New Terminal

‘// Connect to the Interface but do not verify the ACR
‘// Ethernet
term.bOnConnectTest = False
term.bstrIP = “192.168.10.40” ‘// Use IP the ACR is setup with
term.Connect 3, 0 ‘// The 3 = Transport type Ethernet

‘// Manually verify the connection
if term.TestConnect() then
    MsgBox “Found ACR!”
end if
```

Example 4

In this C# Example, the ACR verification test is turned off and done manually by the client application.

```
‘// Initialize the Interface
BOXBRIDGELib.Terminal term;

‘// Connect to the Interface but do not verify the ACR
// Ethernet
term.bOnConnectTest = false
term.bstrIP = “192.168.10.40”; // Use IP the ACR is setup with
term.Connect(3, 0); / The 3 = Transport type Ethernet

‘// Manually verify the connection
if (term.TestConnect()){
    MessageBox.Show (“Found ACR!”,””, MessageBoxButtons.OKCancel,
    MessageBoxIcon.Asterisk)
}
```

Syntax

The syntax for requesting a connection to the Communications Server varies, depending on the programming environment used.

The following connection examples are provided for Visual Basic, C++, Visual Basic .NET, C++ .NET, and C# .NET. To disconnect, see the **Disconnect** method.

After having created the object variable and established a connection, you can use the set of standard methods and properties for use by client applications.

ACR Legacy Support

The ACR Legacy Support kit allows many existing user-created applications to run over the new ACR OLE Server without modification. You can transparently redirect device communications to any of the OLE Server's available devices.

The kit provides the following:

- Acrownt.dll (updated file)
- ACRStart.exe launcher
- An optional ALS.ini configuration file.

Using the ACR Legacy Support Kit

For more information about determining the correct settings, see the ACR OLE Server documentation.

Note: You can automatically start your legacy application, bypassing the acrstart.exe application. To do this, add `-a` or `/a` in the path of a shortcut to the acrstart.exe file. When you double-click the shortcut, the last legacy application run by the acrstart.exe file is automatically run.

1. Copy the acrownt.dll to your system directory
   - For Windows 2000— \winnt\system32
   - For Windows XP— \windows\system32
2. Copy ACRStart.exe to the directory where your existing application is installed.
3. Run ACRStart.
4. To locate your application, click **Browse**. Having found the application, click **Ok**. The ACRStart.exe program remembers the location of your legacy application.
5. **Optional**: You can redirect communications to another communications device by creating an ALS.ini file in the same folder as your legacy application.

Example
The contents of the file look like this:

```
[Card0]
bstrIP=192.168.1.25
nPort=1
nBPS=115200
nBus=1
nDataWaitRate=100
nTransport=3
```

Shared Properties and Methods

The following properties and methods are shared by all interfaces.

Properties

- `BSTR` **bstrVersion**
- `Long` **nPort**
- `Long` **nBPS**
- `Long` **nBus**
- `Long` **nCard**
- `BSTR` **bstrIP**
- `BSTR` **bstrUSBSerialNumber**
- `Bool` **bOnConnectTest**
- `Bool` **isOffline**

Methods

- **Connect(**`long nTransport, long nIndex`**)**
- **bool TestConnect()**
- **SetWatchdog(**`long interval, long retries`**)**
- **Disconnect()**

Properties

bOnConnectTest

Description	This allows (or disallows) automatic ACR verification as part of **Connect()**
Property	**bOnConnectTest**
Return Type	`bool`
Range	N/A
Default	TRUE
Value	If TRUE, the Communications server verifies there is an ACR device present during the call to the **Connect()** method. If FALSE, the Communications server provides no extra verification when calling **Connect()** method.
Remarks	When **bOnConnectTest** is set to TRUE, the **Connect()** method verifies that an ACR device is physically connected and responding after successfully connecting to a communications transport. This extra verification is done using an implicit call to **TestConnect()**. See **TestConnect()** for more information. If no controller is found, the **Connect()** method throws and exception. If **bOnConnectTest** is set to false, some communication transports will not fail when using **Connect()**, even when no ACR device is physically connected. For example, a serial port only needs to be present on the PC at the **nPort** communication port address for **Connect()** to return success. Some applications may not need to perform this extra check, in which case set this value to FALSE. Setting this value to FALSE will slightly speed up **Connect()**.

bstrIP

Description	The Ethernet IP address of the ACR device.
Property	**bstrIP**
Return Type	BSTR
Range	N/A
Default	192.168.10.40
Value	Ethernet address string in dot notation—*xxx.xxx.xxx.xxx*
Remarks	To communicate over a network using TCP/IP, you must configure the network settings for the personal computer and ACR device. For more information, see the `IP` and `IP MASK` commands for the ACR Series controller.

bstrUSBSerialNumber

Description	The USB serial number of the ACR controller.
Property	**bstrUSBSerialNumber**
Return Type	BSTR
Range	N/A
Default	0
Value	USB serial number string
Remarks	The USB serial number is the same as the manufacturing serial number of the ACR controller. The USB serial number can also be auto-detected by ACR-View before connecting. This property must be set before calling **Connect()** on an ACR controller over USB.

bstrVersion

Description	Holds the version number of the ComACRsrvr.dll file
Property	**bstrVersion**
Return Type	BSTR
Range	N/A
Default	N/A
Value	String is in the format `X.X.X.X`, where `X` represents some number.
Remarks	The value comes dynamically from the version resource in the .dll file.

isOffline

Description	Indicates whether the Transport type is set to Offline.
Property	**isOffline**
Return Type	`bool`
Range	N/A
Default	TRUE
Value	If TRUE, the transport type is set to Offline, which indicates that the interface is not connected to an ACR device. See **Connect()** for the transport types.
Remarks	The default for each Interface is Offline mode. After a successful connection, it can return to an Offline mode by either explicitly calling **Connect()** with the Offline transport type, or calling **Disconnect()**. Many methods throw an exception if called in the Offline mode; You can check this property first to avoid throwing an exception.

nBPS

Description	The speed of the serial port in Bits Per Second, to set the Communications Port (COM1, etc.) for Serial communications.
Property	**nBPS**
Return Type	Long
Range	9600, 19200, 38400
Default	38400
Value	Communications speed.
Remarks	The Communication Server can use any BPS rate that is supported by the PC, but the ACR Controller can only auto-detect at rates 9600, 19200, and 38400 BPS.

nBus

Description	Indicates the type of PC Bus Card (ISA or PCI) being used
Property	**nBus**
Return Type	long
Range	0-1
Default	0 (PCI)
Value	To indicate an ACR Bus card:

Value	Description
0	ACR PCI Bus Card
1	ACR ISA Bus Card

Remarks	The ACR8020, ACR1505 and any newer bus based controllers are PCI bus based. All earlier ACR bus based controllers use the ISA bus.

nCard

Description	The index number of the ACR Controller
Property	**nCard**
Return Type	long
Range	N/A
Default	0
Value	This is the value of the index parameter in the **Connect()** method. It is updated after calling **Connect().**
Remarks	This is a read only property. It is only valid after the **Connect()** method is called, and stores the value used as the index parameter. This is useful when connecting to USB if the card index is not known. After calling **Connect(4, 0)**, the Communication Server will find the first USB attached ACR device and then place the device's card number in this property. That card number can then be stored and used later to directly connect to the USB attached ACR device. Calling **Connect()** with a specific card number is useful if more than one ACR device is connected to the PC using USB.

nPort

Description	Communications Port (COM1, etc.) on the personal computer
Property	**nPort**
Return Type	long
Range	1-256
Default	1
Value	
Remarks	Sets the communications port of the personal computer to which the serial ACR device is connected.

Methods

Connect

Description Establish a connection of type transport to an ACR Controller.

Signature **Connect(**`long nTransport, long nIndex`**)**

Return Type N/A

Parameters

`nTransport:` Indicates the physical communication layer being used, or no layer when Offline.

Transport Types	
Value	**Description**
0	Offline
1	Bus
2	Serial
3	Ethernet
4	USB

`nIndex:` Transport Type dependent data. For more information about this value, see Remarks below. The default `nIndex` (sometimes called the card number) is zero. When `nTransport`=Offline, the `nIndex` can be anything.

Return After **Connect()** is successfully called (except for `nTransport`=Offline), the ACR device is connected and ready. To ensure an ACR device is physically present, you can use set the **bOnConnectTest** property to TRUE (the default) or call the **TestConnect()** method once connected.

Remarks All Interfaces initially come up with transport = Offline. Each transport type has its own data requirements for connecting.

Transport Connection Requirements	
Parameter	**Description**
Offline	The `nIndex` value can be any value.
Bus	The `nIndex` value must be the card index assigned during installation. To find this card number see parameter P7041, or DIP switch setting on some cards. The **nBus** property must be set to ISA or PCI depending on the card type.
Serial	The `nIndex` value is the index of the card, which is typically zero. In a daisy chain configuration, this number identifies the specific controller. The **nPort** must be set to the PC communications port that will be used, and the **nBPS** must be set to the desired bits per second rate.
Ethernet	The **nIndex** value can be any value. The **nIP** property must be set to the IP address of the ACR controller.
USB	The **nIndex** value is the unique Serial ID of the ACR device. If this is set to zero, the first ACR USB device found will be connected. After connection, the Serial ID connected can be found in the **nCard** property.

Any transport specific properties (i.e. **bstrIP** for Ethernet, etc.) should be set prior to calling **Connect()**.

All Interfaces initially (prior to the **Connect()** call) come up with transport equal to Offline. In addition to establishing a connection to the physical layer, the ACR card is sent specific characters on the first connection, describe as follows:

Ethernet: USB and Bus Cards an "echo1<cr>" is sent to wake up the card and turn echo on.

Serial: Two <cr> are sent to establish a BPS rate (Auto-Baud Detect). Then <ctrl-B><ctrl-A> `nIndex`<cr> to activate the specified device, and finally an "echo1<cr>"

is sent to turn echo on.

The `nIndex` value is used to distinguish multiple cards using the same transport. For example, two ISA Bus cards in a single computer, more than one ACR connected using the USB, or a multi-drop Serial configuration. Outside of these configurations, the Card number should be left zero in the **Connect()** call.

Disconnect

Description Disconnect from the current communication transport

Signature **Disconnect()**

Return Type N/A

Parameters N/A

Return Implicitly calls **Connect(**`0,0`**)** to switch to Offline mode.

Remarks When using the Status Interface, it is recommended that the **DelStatus(**`-1`**)** method be called prior to **Disconnect()**. This stops pending status requests, and provides a more graceful shutdown of the component.

SetWatchdog

Description Modifies the Watchdog values.

Signature **SetWatchdog(**`long nInterval, long nRetries`**)**

Return Type N/A

Parameters

`nInterval:` The time, in milliseconds, between sending test keep-alive strings to the ACR device.

`nRetries:` The number of times the keep-alive test string message is sent to the ACR device, with no valid reply, before attempting to re-connect to the ACR device.

Return The Ethernet transport currently has Watchdog functionality. The ACR controller uses a separate port to receive a coded command string (keep-alive message), and echoes it back to the sender. If the Communications Server fails to get a response to a keep-alive message in `nInterval*nRetries` milliseconds, the Communications Server attempts to reconnect (until the **Disconnect()** method is called.)

The ACR controller also expects the keep-alive packets in the specified time window. If the ACR controller does not receive the keep-alive, a successful command string in `nInterval*nRetries` milliseconds, the ACR controller disconnects the regular ACR Ethernet connection it is watching and stops responding to the PC watchdog. If the Communications server fails to get a successful command string in `nInterval*nRetries` milliseconds, the Communications Server attempts to reconnect both connections (until the **Disconnect ()** method is called.)

Remarks This method has no effect on any transport except Ethernet.

The initial settings of the Ethernet Watchdog are as follows:

- `nInterval`=2000 ms(2 seconds)
- `nRetries=4`

To change these defaults settings used by the Communications Server, add the following registry values:

HKEY_CURRENT_USER\Software\Parker Hannifin\ACR-View\Settings

HBInterval=x :: where x is the value to default nInterval

HBRetries=y :: where y is the value to default nRetries

TestConnect

Description	Verifies that an ACR Controller is connected.
Signature	**TestConnect()**
Return Type	`bool`
Parameters	N/A
Return	A command is sent and the return value verified. If this process succeeds, an ACR's presence is presumed and TRUE is returned. Otherwise FALSE is returned. When the transport type = Offline, this method always returns FALSE.
Remarks	The **TestConnect()** method sends a binary command to the controller and verifies the returned data. This is the same test as done by **Connect()** when **bOnConnectTest** is set to TRUE.

STATUS Properties and Methods

The ACR Controllers provide access to large amount of status information, most of which can be acquired only using p-Parameters. The p-Parameter values can be read using the Terminal Interface by simply issuing an AcroBasic print command followed by the p-Parameter (`?P12291<cr>`) or they can be read using the ACR Binary command syntax.

Using ACR Binary syntax, a single value or a related group of 8 values can be read at one time (the Groups of specific p-Parameters are documented in ACR-View online help). The Status methods in this interface know how to convert p-Parameters into their binary equivalent commands. The methods in the Status Interface use the ACR Binary syntax to get parameter information.

To aid in the quick retrieval and notification of changes in specific p-Parameters, the Status Interface is implemented as a request queue that polls the ACR Device for status in short increments. This allows a program to be alerted when a status item changes, so it can be read, and makes reading status very quick since it does not have to block for the entire request/response time period. Conversely, the more status that a client program requests to be in the queue, the longer it will take to refresh the data.

The Status Interface can be used in two ways, in a simple request/reply form or as an event driven alerting request queue. The request/reply method is straightforward. Call a method that begins with **GetACR***, and receive the information requested. The event driver method requires more steps. First use the **AddACR*** methods to put a request into the queue. Second, wait for the COM event **StatusWaiting()** to fire. When **StatusWaiting()** fires, use **GetStatus()** to get the information. When done watching the status, use **DelStatus()** to remove the status from the request queue.

Note: Retrieving status from an ACR controller using the PC can take tens to hundreds of milliseconds. For many statuses, this is not an issue but for high-speed information changes, say the position of an axis during a move, reading data using the PC in real time may not be effective. Instead, the ACR products support an internal logging feature that can capture data as fast as the controller can change it. These internal logs can be saved in local arrays and then read, after the fact, into the PC using the methods in this interface (see `GetLocalArrayAddr()` and `GetACRMemory()` for examples.)

For examples on reading status, see the sample programs.

Warning — Before sending a `REBOOT` command, resetting, or cycling power to an ACR series controller, close the Status interface by sending `DelStatus(-1)` in your software application. Otherwise, the Status interface continues sending data, which can interfere with the boot sequence of the controller.

Properties

- `long` **nStatusWaitRate**

Methods

- **SAFEARRAY GetACRCustom (**`BSTR bstrRequest`**)**
- **SAFEARRAY GetACRGroup(**`BSTR bstrRequest`**)**
- **SAFEARRAY GetACRGroupRaw(**`long nType, long nCode, long nIndex`**)**
- **SAFEARRAY GetACRMemory (**`long nType, long nAddress, long nCount`**)**
- **long GetLocalAddr(**`long nProg, long nType, long nSize`**)**
- **long GetLocalArrayAddr(**`long nProg, long nType, long nArray, long nSize`**)**
- **long AddACRGroup(**`BSTR bstrRequest`**)**
- **long AddACRGroupRaw(**`long nType, long nCode, long nIndex`**)**
- **long AddACRCustom(**`BSTR bstrRequest`**)**
- **long AddACRMemory (**`long nType, long nAddress, long nCount`**)**
- **SAFEARRAY GetStatus(**`long nMsgid`**)**
- **DelStatus(**`long nMsgid`**)**
- **bool GetParmInfo(**`long nParameter, long nType, long nCode, long nIndex, BSTR bstrCatagory, BSTR bstrDesc`**)**
- **long GetParmType(**`long nParameter`**)**
- **long GetParmAddr(**`long nParameter`**)**
- **bool IsFlagSet(**`long nFlagGrp, long nFlagNdx`**)**
- **StatusWaiting(**`long msgID, long error`**)**
- **WatchdogReconnect()**
- **WatchdogTimeout()**

Note: For the methods in this Interface that return a SAFEARRAY of Variants, the client software should clean up the program memory (some environments, like Visual Basic, do this automatically.) In addition, for this return type it is possible that a single, empty Variant (VT_EMPTY) will be returned instead of a SAFEARRAY. The calling program should verify the Variant type returned is a SAFEARRAY and not an empty Variant before processing.

Properties

nStatusWaitRate

Description The minimum time between status alerts in milliseconds.

Property **nStatusWaitRate**

Return Type `long`

Range N/A

Default 10 ms

Value If a status request has new data available, a COM event is generated in the time between when a status changes and **nStatusWaitRate** milliseconds past the time a status changes. Setting this value to zero will disable alerts. Use this property to set the minimum time in milliseconds between alert events.

Remarks Changes to this value only take affect if you have set the **nStatusWaitRate** property before calling the **Connect()** method.

Methods

AddACRCustom

Description Add a custom p-Parameter request into the status queue.

Signature **AddACRCustom(**`BSTR bstrRrequest`**)**

Return Type `long`

Parameters

`bstrRrequest:` String of up to 32 p-Parameters, comma delimited. These parameters are used to look up individual or custom p-Parameter values (for example P6144,P6160 would return the encoder positions for Axis0 and Axis1).

Return A key identifying the request in the queue. The key can be used to retrieve data using **GetStatus()** (for example, when the alert is signaled).

Remarks Calling the Add routines places the specific status request into a constantly updated queue of requests. As data is retrieved from the device for each request in the queue, that data is compared to existing data. If the retrieved data is different, an event (callback) is generated with a key.

AddACRGroup

Description Add a group request into the status queue.

Signature **AddACRGroup(**`BSTR bstrRrequest`**)**

Return Type `long`

Parameters

`bstrRrequest:` String of up to 4 p-Parameters, comma delimited. These parameters are used to look up the group, which is then used to return the 8 p-Parameter values for each group. Any p-Parameter in a group can be used to identify a group. Up to 4 groups can be requested and any undocumented/reserved items in a group are returned as zero (for example P6144 would return 8 values starting with the encoder position for Axis0).

Return A key identifying the request in the queue. The key can be used to retrieve data using **GetStatus()** (for example, when the alert is signaled).

Remarks Calling the Add routines places the specific status request into a constantly updated queue of requests. As the data from each request in the queue is retrieved from the device, it is compared against existing data, and if different, an event (callback) is generated with a key.

AddACRGroupRaw

Description Add a group request into the status queue.

Signature **AddACRGroupRaw(**`long nType, long nCode, long nIndex`**)**

Return Type `long`

Parameters

`nType:` The data type of the values being read:

Transport Types	
Value	**Description**
0	Long
1	Float(64)
2	Float(32)

`nCode:` The ACR Group Code as documented in the ACR-View online help.

`nIndex:` The ACR Group Index as documented in the ACR-View online help.

Return A key identifying the request in the queue. The key can be used to retrieve data using **GetStatus()** (for example, when the alert is signaled).

Remarks Calling the Add routines places the specific status request into a constantly updated queue of requests. As the data from each request in the queue is retrieved from the device, it is compared against existing data, and if different, an event (callback) is generated with a key.

This method is provided for backward compatibility with existing programs. It is recommended that **AddACRGroup()** be used to access group status.

AddACRMemory

Description Add a memory value request into the status queue.

Signature **AddACRMemory(**`long nType, long nAddress, long nCount`**)**

Return Type `long`

Parameters

`nType:` The data type of the values being read:

Transport Types	
Value	**Description**
0	Long
1	Float(64)
2	Float(32)

`nAddress:` The starting physical memory address on the ACR product.

`nCount:` The number of values to read (starting at the memory location.) The values of each memory location will be placed in a corresponding position in the returned array.

Return A key identifying the request in the queue. The key can be used to retrieve data using **GetStatus()** (for example, when the alert is signaled).

Remarks Calling the Add routines places the specific status request into a constantly updated queue of requests. As data is retrieved from the device for each request in the queue, that data is compared to existing data. If the retrieved data is different, an event (callback) is generated with a key.

Use a value retrieved with **GetLocalAddr()** or **GetLocalArrayAddr()** or **GetParmAddr()** in the **nAddress** parameter.

DelStatus

Description	Delete a status request from the status queue.
Signature	**DelStatus(**`long nMsgId`**)**
Return Type	N/A
Parameters	`nMsgID:` The key to a specific status request as returned by one of the Add routines.
Return	N/A
Remarks	Removing unused status requests will speed up the update of the other requests in the queue. To clear all status requests in a queue, use `DelStatus(-1)`.

GetACRCustom

Description	Return the p-Parameter values requested.
Signature	**GetACRCustom (**`BSTR bstrRequest`**)**
Return Type	SAFEARRAY
Parameters	`bstrRrequest:` String of up to 32 p-Parameters, comma delimited. These parameters are used to look up the individual, or custom, p-Parameter values (for example P6144,P6160 would return the encoder positions for Axis0 and Axis1).
Return	The **GetACRCustom** method returns a SAFEARRAY of up to 32 Variants (return type: `long` or `float`). Each p-Parameter in the request returns the values of the type as defined in the Parameters Reference section of the ACR User's Guide-View online help.
Remarks	Example If the `bstrRequest` parameter is set equal to "P4096,P8960" then it will return an array of 1 long and 1 float, stored in Variants, which will be in a SAFEARRAY structure.

GetACRGroup

Description	Return the p-Parameter group values requested.
Signature	**GetACRGroup(**`BSTR bstrRrequest`**)**
Return Type	SAFEARRAY
Parameters	`bstrRrequest:` String of up to 4 p-Parameters, comma delimited. These parameters are used to look up the group, which is then used to return the 8 p-Parameter values for each group. Any p-Parameter in a group can be used to identify a group. Up to 4 groups can be requested and any undocumented/reserved items in a group are returned as zero (for example P6144 would return 8 values starting with the encoder position for Axis0).
Return	Returns a SAFEARRAY containing up to 32 Variants, each of which are of type `long` or `float`. Each p-Parameter in the request results in a group of 8 values of the same type.
Remarks	If the `bstrRequest` parameter is set to "P4096,P8960" then an array of 8 longs and 8 floats, stored in Variants, will be returned in the SAFEARRAY structure. Example: for request = "P4096,P8960", an array of 8 longs (p-parameters 4096-4103) and 8 floats (p-parameters 8192, 8448, 8704, 8960, 9216, 9472, 9728, 9984), stored in Variants, are returned, because the individual p-parameters all share the same group/index coding.

GetACRGroupRaw

Description Return the p-Parameter group values requested.

Signature **GetACRGroupRaw(**`long nType, long nCode, long nIndex`**)**

Return Type SAFEARRAY

Parameters

`nType:` The data type of the values being read:

Transport Types	
Value	**Description**
0	Long
1	Float(64)
2	Float(32)

`nCode:` The ACR Group Code as documented in the ACR-View online help.

`nIndex:` The ACR Group Index as documented in the ACR-View online help.

Return Returns a SAFEARRAY containing up to 8 Variants, all of which are of type `long` or `float`.

Remarks This method is for backward compatibility. It is recommended that **GetACRGroup()** method should used for getting group data.

GetACRMemory

Description Return the values requested at the specified memory location.

Signature **GetACRMemory (**`long nType, long nAddress, long nCount`**)**

Return Type SAFEARRAY

Parameters

`nType:` The data type of the values being read:

Transport Types	
Value	**Description**
0	Long
1	Float(64)
2	Float(32)

`nAddress:` The starting physical memory address on the ACR product.

`nCount:` The number of values to read (starting at the memory location.) The values of each memory location will be placed in a corresponding position in the returned array.

Return The returned array can be of any size but is limited to a single data type.

Remarks The floating-point values returned in an array are always 32-bit. However, as this reads memory locations, the data type on the ACR must be specified exactly, either 32-bit or 64-bit floating point.

This method uses the binary PEEK command. While that command only supports up to 256 returned values per call, this method breaks up requests larger than 256 items into a series of requests containing 255 items (or fewer).

Use a value retrieved with **GetLocalAddr()** or **GetLocalArrayAddr()** or **GetParmAddr()** in the **nAddress** parameter.

GetLocalAddr

Description Get the address of local variables in a specific program.

Signature **GetLocalAddr(**`long nProg, long nType, long nSize`**)**

Return Type `long`

Parameters

`nProg:` Provide the program number—local variables are dimensioned in a program space.

`nType:` The data type of the values being read:

Transport Types	
Value	**Description**
0	Long
1	Float(64)
2	Float(32)

`nSize:` After the call, this parameter holds the number of dimensioned variables available.

Return The return value is a valid ACR memory address (or zero if no memory is dimensioned for the requested variable type.)

Remarks The returned address can be used in either the **GetACRMemory()** or **AddACRMemory()** method to access local variables. **GetACRMemory()** and **AddACRMemory()** allow the communication server to query "count" number of values following the address. Be sure to make the value for "count" less than or equal to the `nSize` parameter.

After using **GetACRMemory()** and **AddACRMemory()**, you can access the status SAFEARRAY that is returned to get the corresponding local values.

Following is an example of getting a local variable of type long from program 0.

```
'\\ In the ACR Prog0, dimension three local longs
PROG0
Dim LV(3)
LV0 = 33
LV1 = 456
LV2 = 44

\\ In your program, access them (Note: SAFEARRAY is pseudo-code to make the example easier to read)
const long PROG0 = 0;
const long TYPE_LONG = 0;
long totalElements = 0;
long localAddress = GetLocalAddr(PROG0, TYPE_LONG, &totalElements);

\\ Get all the values
SAFEARRAY localLongs = GetACRMemory(TYPE_LONG, localAddress, totalElements);
\\ Instead of GetACRMemory(), the AddACRMemory() method
\\ could also have been used.
```

In the array `localLongs`, the three values in the array are 33, 456, and 44. Just as they were assigned in the ACR `Prog0`. This example requests the total number of dimensioned variables. Fewer can be requested safely, but if more are requested than are dimensioned, the behavior is undefined.

GetLocalArrayAddr

Description Get the address of a local variable array in a specific program.

Signature **GetLocalArrayAddr(**`long nProg, long nType, long nArray, long nSize`**)**

Return Type `long`

Parameters

`nProg:` Provide the program number—local variables are dimensioned in a program space.

`nType:` The data type of the values being read:

Transport Types	
Value	**Description**
0	Long
1	Float(64)
2	Float(32)

`nArray:` The specific array being looked for.

`nSize:` After the call, this parameter holds the number of dimensioned variables available.

Return The return value is a valid ACR memory address (or zero if no memory is dimensioned for the requested variable type.)

Remarks The returned address can be used in either the **GetACRMemory()** or **AddACRMemory()** method to access the local variable arrays. **GetACRMemory()** and **AddACRMemory()** allow the Communications Server.dll to query "count" number of values following the address. Be sure to make the value for "count" less than or equal to the `nSize` parameter.

After using **GetACRMemory()** and **AddACRMemory()**, you can access the status SAFEARRAY that is returned to get the corresponding local value.

Following is an example of getting a local variable of type long from program 0.

```
'\\ In the ACR Prog0, dimension a local array of longs
PROG0
Dim LA(0)
Dim LA0(3)
LA0(0) = 33
LA0(1) = 456
LA0(2) = 44

\\ In your program, access them (Note: SAFEARRAY is fictionalized here to make it easier to read)
const long PROG0 = 0;
const long TYPE_LONG = 0;
const long ARRAY_NBR = 0;
long totalElements = 0;
long localAddress = GetLocalArrayAddr(PROG0, TYPE_LONG, ARRAY_NBR, &totalElements);

\\ Get all the values
SAFEARRAY localLongs = GetACRMemory(TYPE_LONG, localAddress, totalElements);
\\ Instead of GetACRMemory(), the AddACRMemory() method
\\ could also have been used.
```

In the array `localLongs`, the three values in the array are 33, 456, and 44. Just as they were assigned in the ACR `Prog0`. This example requests the total number of dimensioned variables. Fewer can be requested safely, but if more are requested than are dimensioned, the behavior is undefined.

GetStatus

Description	Retrieve the specified status information.
Signature	**GetStatus(**`long nMsgId`**)**
Return Type	SAFEARRAY
Parameters	`nMsgID:` The key to a specific status request as returned by one of the Add routines.
Return	The returned array can be any size. It holds the values in Variants, either type long or float.
Remarks	The order of the data returned is dependent on the original request. For example, if the original request asked for status on P4099 and P4096 in that order, then the returned values are also in that order.

IsFlagSet

Description	Utility for identifying a bit in a 32-bit long.
Signature	**isFlagSet(**`long nFlagGrp, long nFlagNdx`**)**
Return Type	`bool`
Parameters	`nFlagGrp:` A value of type Long containing flags (as bits.) `nFlagNdx:` Index of the flag.
Return	Returns TRUE if bit at `nFlagNdx` is 1, and returns FALSE when the bit is 0.
Remarks	This method is helpful in finding the value of a flag when using the Communications Server interface with languages that do not normally use bits. ACR Flag values are stored in 32-bit longs, which is the lowest level of granularity provided by the Status methods. The `nFlagGrp` is the long value, the `nFlagNdx` is the position of the flag in the long. Example The following Visual Basic code determines if ACR BIT128 is set or clear in parameter 4100 (which contains ACR Flags BIT128 through BIT159). Dim rtnStat as Variant Dim bit128 as Boolean rtnStat = GetACRCustom ("P4100") bit128 = isFlagSet (rtnStat(0), 0)

GetParmAddr

Description	Get the address of a p-Parameter.
Signature	**GetParmAddr(**`long nParameter`**)**
Return Type	`long`
Parameters	`nParameter:` A numeric p-Parameter.
Return	Returns the address of the p-Parameter.
Remarks	The returned address can be used in either the **GetACRMemory()** or **AddACRMemory()** method to access p-Parameter values. **GetACRMemory()** and **AddACRMemory()** allow the Communications Server to query "count" number of values following the address. Be sure to make the value for "count" one (1), since you are requesting a single value.

GetP armType

Description Utility for identifying data type of a p-Parameter.

Signature **GetParmType(**`long nParameter`**)**

Return Type `long`

Parameters

`nParameter:` A numeric p-Parameter.

Return Returns the data type of the p-Parameter:

Transport Types	
Value	**Description**
0	Long
1	Float(64)
2	Float(32)

Remarks On the ACR Controller, data elements are stored in one of three types: Long Integer, 64-bit Floating point, and 32-bit Floating point. When specifying the data type that will be in use on the controller, the communications Server expects one of the three codes: 0 for long, 1 for 64-bit float, and 2 for 32-bit float.

GetParmInfo

Description Utility for getting information on specific p-Parameters.

Signature **GetParmInfo(**`long nParameter, long nType, long nCode, long nIndex, BSTR bstrCatagory, BSTR bstrDesc`**)**

Return Type `bool`

Parameters

`nParameter:` A numeric p-Parameter.

`nType:` The data type of the values being read:

Transport Types	
Value	**Description**
0	Long
1	Float(64)
2	Float(32)

`nCode:` The ACR Group Code as documented in the ACR-View online help.

`nIndex:` The ACR Group Index as documented in the ACR-View online help.

`bstrCatagory:` A textual description of the category a p-Parameter is in.

`bstrDesc:` A textual description of the p-Parameter.

Return Returns TRUE if p-Parameter found.

Remarks The Communications Server keeps a database of information on all the p-Parameters of the ACR product. This method gets the information for a specific p-Parameter from that database.

StatusWaiting

Description	Callback method acts as an event signaling that a status changed	
Signature	**StatusWaiting(**`long msgID, long error`**)**	
Return Type	N/A	
Parameters	`msgID:`	The key to a specific status request as returned by one of the Add routines.
	`error:`	If an error occurred getting a status update, it is reported in the `error` parameter. When a request encounters an error, the request is deleted from the queue.
Return	N/A	
Remarks	The COM event model (also know as Connection Points) uses a callback mechanism to generate events. The Client program implements and registers this method. The Status Interface calls the method when a status request (key=`msgID`) has been updated and is ready to read using the **GetStatus()** method.	

WatchdogReconnect

Description	Callback method acts as an event signaling that Ethernet communications has been re-established after watchdog timer previously timed out.
Signature	**WatchdogReconnect()**
Return Type	N/A
Parameters	`N/A`
Return	N/A
Remarks	The COM event model (also know as Connection Points) uses a callback mechanism to generate events. The Client program implements and registers this method. The Status Interface calls the method after the watchdog timer times out due to a loss of Ethernet communications.

WatchdogTimeout

Description	Callback method acts as an event signaling that the watchdog timer has timed out on an Ethernet connection
Signature	**WatchdogTimeout()**
Return Type	N/A
Parameters	N/A
Return	N/A
Remarks	The COM event model (also know as Connection Points) uses a callback mechanism to generate events. The Client program implements and registers this method. The Status Interface calls this method when the watchdog timer has timed out on an Ethernet socket connection.

UTILITY Properties and Methods

The Utility Interface provides functionality for transferring files between the PC and the ACR product. The File transfer to the ACR Controller (both OS and Program Files) is non-blocking. This allows the transfers to be monitored for status and canceled by the user. The File transfer from the ACR Controller to the PC, file upload, is a blocking transfer.

The non-blocking transfers allow the client software to be responsive during a download and inform the user of progress, but they also imply that code running after a transfer has started should not assume the transfer has completed. The code should check the status before attempting to talk to the controller. For example, a function that downloaded a program cannot expect to run that program after returning from the download call, but must wait until the status information indicates the transfer is complete.

Properties

None (other than the common properties)

Methods

- **FindACR(**`long nTransport`**)**
- **DownloadOS(**`long nDevice, BSTR bstrFile`**)**
- **DownloadFile(**`BSTR bstrPrg, BSTR bstrFile`**)**
- **UploadFile(**`BSTR bstrPrg, BSTR bstrFile`**)**
- **GetStatusDL(**`long nTotal, long nBytes`**)**
- **StopDownload()**

Methods

FindACR

Description Finds all ACR controllers attached to a PC.

Signature **FindACR(**`long nTransport`**)**

Return Type SAFEARRAY

Parameters

`nTransport:` Indicates the physical communication layer being used, or no layer when Offline.

Transport Types	
Value	**Description**
0	Offline
1	Bus
2	Serial
3	Ethernet
4	USB

* Only the USB Transport Type is supported. Other types return an empty list.

Return Returns the `nIndex` value used in the **Connect()** method for all ACR controllers attached to the PC.

Remarks For transports that require an nIndex value in **Connect(),** this method can be used to find all the possible values of nIndex. For example, USB requires the Serial number be used as nIndex to directly connect to a controller. To discover what `nIndex` is, call this method.

DownloadFile

Description Transfers a text file to the ACR Controller.

Signature **DownloadFile(**`BSTR bstrPrg, BSTR bstrFile`**)**

Return Type N/A

Parameters

`bstrPrg:` Specifies the location to which files are downloaded (for example: `SYS` or `PROG01` or `PLC01`). This parameter is not required and can be an empty string.

`bsrFile:` Specifies the fully qualified name of the file to download.

Return For information about download progress, see the **GetStatusDL()** method.

Remarks If there is any value in the `bstrPrg` parameter, the following sequence of events takes place. First, an attempt is made to halt all running programs (this is done by setting the flag parameter for each program, e.g. to stop PROG0, set flag 1033.) After trying to halt any running programs, the `bstrPrg` parameter is inspected for the string PROG or PLC. If either string is present, a `NEW` command is first sent to the controller. Finally, the `bstrPrg` value itself is sent to the controller.

After any initial processing, **DownloadFile()** reads in the text file specified in `bstrFile` and passes it, line by line, to the controller. This method is non-blocking, returning as soon as the file has started to transfer. Check the **GetStatusDL()** for completion updates during the use of this method.

Note: Once downloaded, a program/plc is in memory but has not been saved to flash. For information on permanently saving a downloaded program or plc, see the `FLASH SAVE / FLASH IMAGE` commands. Remember that these flash commands only work if no programs are currently running.

DownloadOS

Description Download a new OS to the ACR (field upgrade.)

Signature **DownloadOS(**`long nDevice, BSTR bstrFile`**)**

Return Type N/A

Parameters

`nDevice:` Specifies the ACR model. Not all ACR Controllers are field upgradeable, and those that are have different mechanisms in place to upgrade their firmware.

Device Types	
Value	**Controller**
0	ACR1200
1	ACR1500
2	ACR2000
3	ACR8010
4	ACR8020
5	ACR8020 (16-axis)
6	ACR1505
7	Reserved
8	ACR9000

`bstrFile:` Specifies the fully qualified file name of the new operating system.

Return For information about download progress, see the **GetStatusDL()** method.

Remarks

Warning—The **DownloadOS()** method requires the full attention of the ACR Controller and the Communication Server. Any other tasks should be stopped, including any use of the Status Interface. The **GetStatusDL()** method is safe to use once the Download has started.

The ACR1505, ACR8020 and ACR8020 (16 axis) controllers allow firmware updates using an ASCII file transfer in either serial or bus communication modes. The file transfer places the new firmware in flash, but does not load it into dynamic memory. A user must then physically cycle power or reset the ACR controller once the download is complete for these cards.

> **Note:** The communications server is unable to recognize a download failure for these cards. The file transfer method is not monitored. After you cycle power or reset the ACR controller, verify the firmware has loaded correctly. To do this, open a terminal in ACR-View, send the `VER` command and verify the new firmware is loaded. If the new firmware is not present, the backup firmware has loaded; Attempt to download the new firmware again.

The ACR9000 controller allows a firmware update using serial communications, Ethernet, or USB. After the file transfer process is complete, the controller loads and runs the new firmware—no additional user input is necessary.

> **Note:** When using serial communications, if the operating system download fails, the ACR controller, after cycling power, may display the OSP> prompt in a terminal. The communications Baud BPS will be set at 9600 bps. Use the same call to this method to try and download again. To manually get to an OSP> prompt use the `FIRMWARE UPGRADE` command. In other communication modes, the controller will stop responding and require a power cycle.

When is the download complete?

All ACR Series controllers save the downloaded operating system to flash. To determine the state of the download process, use the return value of the **GetStatusDL()** method. Call the **GetStatusDL()** method continually until either the transfer is complete or the status indicates an error. This allows the **GetStatusDL()** method to reconnect to the device after determining the flash save is complete. Otherwise, the client must manually call the **Connect()** method.

GetStatusDL

Description Current status of the active Download.

Signature `long` **GetStatusDL(**`long nTotal, long nBytes`**)**

Return Type N/A

Parameters

`nTotal:` Total bytes to be transferred.

`nBytes:` Total number of bytes transferred so far.

Return Return value is the status of the active Download.

Download Status	
Value	**Description**
0	No Transfer in progress
1	Transfer in progress
2	End of transfer
3	User cancelled Transfer
4	Error reading from file
5	Too many errors during Transfer
6	Transfer has timed out waiting for response
7	The ACR OS failed to verify against the hardware description file (config image)
8	Save to flash of OS in progress.
9	Problem encountered in saving OS to flash
Negative number	Unexpected error

Remarks If an OS download is in process, the `nBytes` parameter value is continually updated with the amount of data sent to the ACR controller. The return value indication the state of the download.

StopDownload

Description Aborts the file transfer to the Controller

Signature **StopDownload()**

Return Type N/A

Parameters N/A

Return See **GetStatusDL()** for download progress.

Remarks Call this method to cancel a download. The status value of 3 will be returned from **GetStatusDL()** after this method is called.

UploadFile

Description Uploads an AcroBasic program or PLC program from the ACR to the PC.

Signature **UploadFile(**`BSTR bstrPrg, BSTR bstrFile`**)**

Return Type N/A

Parameters

`bstrPrg:` Specifies the location to which files are uploaded from (for example: `PROG01` or `PLC01`).

`bstrFile:` Specifies the fully qualified path and name of the file to download.

Return This method blocks any other instructions from running until the upload is complete. There is no checking of the code uploaded.

Remarks When uploading, the `bstrPrg` parameter is sent to change the command prompt prior to sending a `LIST` command and capturing all the data returned.

TERMINAL Properties and Methods

The Terminal Interface provides functionality for doing simple request / reply with the controller using ASCII characters. To send ASCII character strings to the ACR Controller, use Write(). To receive data from the ACR Controller, use Read(). The Interface supports a COM event, **DataWaiting()**, for alerting the client program that data is ready to be read.

Properties

- `long` **nDataWaitRate**

Methods

- BSTR **Read()**
- **Write(**BSTR `send`**)**
- **DataWaiting()**

Properties

nDataWaitRate

Description	The minimum time between status alerts in milliseconds.
Property	**nDataWaitRate**
Return Type	`long`
Range	N/A
Default	50 ms
Value	If the read buffer has new data available, a COM event is generated in the time between when it comes into the read buffer and **nDataWaitRate** milliseconds past that time it comes into the read buffer. Using this property, you can set the minimum time between events, in milliseconds. The default setting is well below the perceivable time a user would notice a delay, but well above the value that would tax PC resources. Setting this value to zero will disable alerts, and the client software will be required to poll the **Read()** method for data.
Remarks	Changes to this value only take affect if To take affect, this should be set done before the **Connect()** method is called.

Methods

DataWaiting

Description	Callback method acts as an event signaling that a there is data to read.
Signature	**DataWaiting()**
Return Type	N/A
Parameters	N/A
Return	N/A
Remarks	The COM event model (also know as Connection Points) uses a callback mechanism to generate events. The Client program implements and registers this method. The Terminal Interface calls the method when there is data in the read buffer, suggesting to the client to call **Read().**

Read

Description	Get any ASCII data from the ACR Controller
Signature	**Read()**
Return Type	BSTR
Parameters	N/A
Return	This returns the data in the ACR Controller's output buffer.
Remarks	Typically, call this method when a **DataWaiting()** event is received.

Write

Description	Sends data to the ACR Controller.
Signature	**Write(**BSTR `send`**)**
Return Type	N/A
Parameters	`send`: ASCII string to send to ACR Controller.
Return	N/A
Remarks	Most ACR commands require a carriage return <cr> (hex x0D) to execute. In certain cases the **Write()** method is locked out; subsequently, the function times out. In this instance, characters may not get to the controller. For example, during a file or OS download.

CONTROL Properties/Methods

The Control interface is meant to provide the ability to update and control the ACR's state and action. Using the methods provided here, a program can set and clear flags, assign values to parameters, and perform moves. All updates can be sent via the binary commands (immediate) and some have the option of utilizing the ASCII interface (queued.)

Moves can be sent on an individual basis or in a batch as a single command. The individual move methods (**Move()** and **Arc()**) assume most of the move settings are stable and provides a group of properties to set them before the move. All properties in this interface starting with “Move” or “Acr” affect the **Move()** and **Arc()** methods.

For moves that are unpredictable and constantly changing in their settings, the **MoveBatch()** method is provided. The **MoveBatch()** method allows all move settings, and all moves, to be sent in a single command.

Properties

- `long` **nMoveProfile**
- `float` **fMoveVEL**
- `float` **fMoveFVEL**
- `float` **fMoveACC**
- `long` **nMoveMode**
- `bool` **bMoveAbsolute**
- `long` **nMoveCounter**
- `long` **nArcMode**
- `bool` **bArcAbsolute**
- `bool` **bArcCCW**

Methods

- **SetFlag(**`long nBit, bool bValue, bool bFast`**)**
- **SetParmFloat(**`long nPparm, float fValue, bool bFast`**)**
- **SetParmLong(**`long nPparm, float fValue, bool bFast`**)**
- **SetGlobal(**`long nCard, long nGlobal, double dValue, bool bFast`**)**
- **Move(**`long nMask, SAFEARRAY targets`**)**
- **Arc(**`long nMask, SAFEARRAY targets`**)**
- **Stop(**`bool bDecel`**)**
- **SendRES(**`long nMask`**)**
- **GetMoveCounter(**`long nCounter, long nIncrement`**)**
- **SetMoveCounter(**`long nCounter, long nIncrement`**)**

- **SetACRMemory(**`long nType, long nAddress, SAFARRAY values`**)**
- **SetACRMemoryMask(**`long nAddress, long nNAND, long nOR`**)**
- **SetParmLongMask(**`long nPparm, long nNAND, long nOR`**)**
- **SetFOV(**`long nMask, float fValue`**)**
- **SetROV(**`long nMask, float fValue`**)**
- **MoveBatch(**`long nType, SAFEARRAY moves`**)**
- **InitPerformance()**
- **GetPerformance(**`SAFEARRAY data`**)**

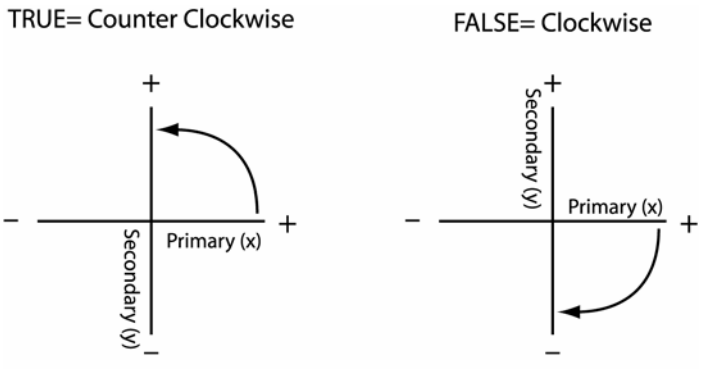
Properties

bAcrAbsolute

Description	Determines if arc centers are treated in absolute terms (TRUE) or in relative terms (FALSE.)
Property	**bArcAbsolute**
Return Type	`bool`
Range	N/A
Default	TRUE (Absolute arc centers)
Value	
Remarks	When TRUE, the `target` parameter of the **Arc** method becomes the new center of the arc. When FALSE, the `target` parameter of the **Arc** method is adjusted incrementally, relative to the current position.

bAcrCCW

Description	Determines the direction of the Arc move.
Property	**bArcCCW**
Return Type	`bool`
Range	N/A
Default	TRUE (CCW)
Value	Arc Direction is CCW if this property is TRUE, CW if FALSE.
Remarks	When TRUE, a counter-clockwise (CCW) arc is defined from the positive primary axis toward the positive secondary axis. When FALSE, a clockwise (CW) arc is defined from the positive primary axis toward the negative secondary axis.



bMoveAbsolute

Description	Determines if move targets are treated in absolute (TRUE) or relative (FALSE) terms.
Property	**bMoveAbsolute**
Return Type	`bool`
Range	N/A
Default	TRUE (Absolute moves)
Value	When TRUE, the `target` parameter of the **Move()** method is treated as the new absolute position. When FALSE, the move is relative to the current position (either backwards or forwards from the current position by the target amount).
Remarks	

fMoveACC

Description	Set a new Profile Acceleration/Deceleration for the next move.
Property	**fMoveACC**
Return Type	`float`
Range	N/A
Default	-1 (use preset velocity)
Value	Part of a motion profile is the acceleration and deceleration, which can be set prior to the move, e.g. during configuration. If you do not want to use the current acceleration setting in the controller, use this property to set a new acceleration. If this value is negative (default), it will be ignored and the existing profile velocity will be used.
Remarks	The effect of setting this value to a value other than -1 will be to permanently change the ACC for the profile, as if the ACC command had been issued. **Warning—**There is no provision in the move command structure to adjust the JRK (jerk or S-curve) or Feedrate Override (FOV or ROV). If the JRK, FOV, or ROV has been calculated and set based on an existing VEL and ACC/DEL, changing the velocity using this property can result in an unexpected motion profile. The FOV and ROV can be set independently with FOV and ROV commands.

fMoveFVEL

Description	Set a new Profile Final Velocity for the next move.
Property	**fMoveFVEL**
Return Type	`float`
Range	N/A
Default	-1 (use preset velocity)
Value	Part of a motion profile is the final velocity, which can be set prior to the move, for example during configuration. If do not want to use the current, final velocity setting in the controller, use this property to set a new final velocity. If this value is negative (default), it will be ignored and the existing profile velocity will be used.
Remarks	The effect of setting this value to a value other than -1 will be to permanently change the FVEL for the profile, as if the FVEL command had been issued. **Warning—**There is no provision in the move command structure to adjust the JRK (jerk or S-curve) or Feedrate Override (FOV or ROV). If the JRK, FOV, or ROV has been calculated and set based on an existing VEL and ACC/DEL, changing the velocity using this property can result in an unexpected motion profile. The FOV and ROV can be set independently with FOV and ROV commands.

fMoveVEL

Description	Set a new Profile Velocity for the next move.
Property	**fMoveVEL**
Return Type	`float`
Range	N/A
Default	-1 (use preset velocity)
Value	Part of a motion profile is the target velocity, which can be set prior to the move, for example during configuration. If do not want to use the current velocity setting in the controller, use this property to set a new velocity. If this value is negative (default), the default is ignored and the existing profile velocity on the controller is used.
Remarks	The effect of setting this value to a value other than -1 will be to permanently change the VEL for the profile, as if the VEL command had been issued. **Warning—**There is no provision in the move command structure to adjust the JRK (jerk or S-curve) or Feedrate Override (FOV or ROV). If the JRK, FOV, or ROV has been calculated and set based on an existing VEL and ACC/DEL, changing the velocity using this property can result in an unexpected motion profile. The FOV and ROV can be set independently with FOV and ROV commands.

nArcMode

Description Determines primary and secondary axes when performing an arc move.

Property **nArcMode**

Return Type `long`

Range 0-3

Default 0

Value The arc mode defines the primary and secondary axes for the arc as follows:

Arc Modes	
Value	**Description**
0	Primary is Axis 0, Secondary Axis 1
1	Primary is Axis 1, Secondary Axis 2
2	Primary is Axis 2, Secondary Axis 0

Remarks To define an arc, first assign the axes that will produce the compound motion to a Master. Using the **nArcMode** property, you can then set which are the primary and secondary axes: The primary axis is usually the X axis; the secondary axis is usually the Y axis.

All axes references are relative to the profile in use. For example, Axis 0 is the first axis attached to the profile specified in the **nMoveProfile** method.

nMoveCounter

Description Turns on the Move Counter.

Property **nMoveCounter**

Return Type `long`

Range N/A

Default 1 (ON, count UP)

Value The Move Counter has three (3) possible modes:

Move Counter Modes	
Value	**Description**
-1	Counter ON and counting DOWN
0	Counter OFF
1	Counter ON and countering UP

Remarks The **nMoveCounter** property sets the mode for the move counter on a device (not the Bus Device Driver Move Counter as stated in some ACR documentation. The Bus Device Driver Move Counter is manipulated using the **GetMoveCounter()** and **SetMoveCounter()** methods.

When in mode -1 or 1, the move counter parameter is updated when a move starts, and can be monitored through the (`LONG`) parameter values below.

Move Counter in Master Parameters (Profile 0 - 15):

P8208, P8464, P8720, P8976, P9232, P9488, P9744, P10000, P10256, P10512, P10768, P11024, P11280, P11536, P11792, P12048

nMoveMode

Description Selects move mode.

Property **nMoveMode**

Return Type `long`

Range N/A

Default 2 (Start/Stop)

Value There are four (4) possible move modes:

Move Modes	
Value	**Description**
0	Continuous Uses ACC to get to VEL * FOV and stays there
1	Cornering Uses ACC to get to VEL * FOV and DEC to get to FVEL
2	Start/Stop Uses ACC to get to VEL * FOV and DEC to Stop (VEL=0)
3	Rapid Start/Stop Uses ACC to get to VEL * ROV and DEC to Stop (VEL=0)

Remarks The move mode determines the type of motion the controller generates.

nMoveProfile

Description This specifies the master profile to use for the move.

Property **nMoveProfile**

Return Type `long`

Range 0-15

Default 0

Value There are up to 16 Master Profiles available for configuration and use on ACR Series controllers. This property stores the motion profile used by the next **Move()** or **Arc()** method.

Remarks For a move to succeed a Master Profile must be configured and have the physical axes attached. The profile must also include information about velocity, acceleration, deceleration, jerk and feedrate override. These settings, except for jerk and federate override, can be explicitly set using the properties in the Control Interface. If they are not set (for example set to -1), the default rates existing in the device will apply.

Note: All axes referenced in this Interface are relative to the profile specified in this property. For example, when a reference is made to Axis 0, it means the profile will "use the first axis attached to this profile."

Methods

Arc

Description Generate an arc move.

Signature **Arc(**`long nMask, SAFEARRAY targets`**)**

Return Type N/A

Parameters

`nMask:` Specifies which axes to use for the move.

`targets:` The arc centers and target position information for each axis.

As a convenience to the caller, the SAFEARRAY passed into this method will be destroyed before it returns. To keep this from happening, use **SafeArrayLock()** on the passed in data before calling this method. Then call **SafeArrayUnlock()** once the method completes.

Return N/A

Remarks The **Arc** method allows from 1 to 16 attached axes to be part of a move. This happens by setting one or more target positions for each axis.

To perform a move, the profile defined in **nMoveProfile** property must have one or more axes attached. The axes are represented by numbers 0, 1, 2, etc., based on the order they were attached to the profile. In the **Arc()** method, the target positions are stored in the SAFEARRAY `targets` parameter, while the `nMask` specifies to which axes the data is linked.

The `nMask` and `targets` parameters operate just as they do in the **Move()** method with one exception: the first two elements of the `targets` array are the Primary and Secondary Centers for the arc. At the third element, place the target information.

Note: To use the **Arc()** method, you must first set the Enable Rapid Move Modes flag*. By default, the flag is zero. Therefore, prior to sending a move to the controller, the flag is set by **Arc()** method. Once set for a motion profile, and for as long as that instance of the Control Interface is alive, subsequent calls to the **Arc()** method for the profile will not try to set the flag (they will assume it stays set).

* See the Secondary Master Flags, bit index 5 in the Parameter Reference.

Move

Description Generate a move.

Signature **Move(**`long nMask, SAFEARRAY VARIANT targets`**)**

Return Type N/A

Parameters

`nMask:` Specifies which axes to use for the move.

`targets:` The target position information for each specified axis.

As a convenience to the caller, the SAFEARRAY passed into this method will be destroyed before it returns. To keep this from happening, use **SafeArrayLock()** on the passed in data before calling this method. Then call **SafeArrayUnlock()** once the method completes.

Return N/A

Remarks The **Move()** method allows from 1 to 16 attached axes to be part of a move. This happens by setting one or more target positions for each axis.

To perform a move, the profile defined in **nMoveProfile** property must have one or more axes attached. The axes are represented by numbers 0, 1, 2, etc., based on the order they were attached to the profile. The `nMask` parameter is treated as a field of 32 bits, or flags. Each flag corresponds to an axis. Setting the `nMask` to 1 (binary 01) tells the **Move()** method that only Axis0 is moved. (As mentioned above, Axis0 is the first axis attached to the profile or master, Axis1 is the second axis attached, etc.) Setting the **nMask** to 8 (binary 1000) tells the **Move()** method that only Axis3 is moved.

The `targets` SAFEARRAY contains the data used to make the move, e.g. the target of the move. The SAFEARRAY should only hold data for axes that are flagged in the `nMask` property.

For example, setting `nMask` to 5 (binary 0101) tells the **Move()** method to move Axis0 and Axis2. Therefore, the `targets` array needs to hold two values. The targets array value at index 0 applies to the Axis0 move, the value at index 1 applies to the Axis2 move. The `targets` SAFEARRAY contains Variants because the target data can be either float or long, but not a combination of floats or longs. The first data type found is the data type used for all data in the array.

Example

// To move 3 axes - axis0, axis1, and axis4 use the following pseudo-code as a guide.

```
nMask = 19              // 10011 binary
targets(0) = 5          // move axis0 5
targets(1) = 15         // move axis1 15
targets(2) = -5         // move axis4 -5
Move(nMask, targets)    // execute all three moves
```

—or—

```
nMask = 1               // 01 binary
targets(0) = 5          // move axis0 5
Move(nMask, targets)    // execute move 1
nMask = 2               // 10 binary
targets(0) = 15         // move axis1 15
Move(nMask, targets)    // execute move 2
nMask = 16              // 10000 binary
targets(0) = -5         // move axis4 -5
Move(nMask, targets)    // execute move 3
```

Note: To use the **Move()** method , you must first set the Enable Rapid Move Modes flag*. By default, the flag is zero. Therefore, prior to sending a move to the controller, the flag is set by **Move()** method. Once set for a motion profile, and for as long as that instance of the Control Interface is alive, subsequent calls to the **Move()** method for the profile will not try to set the flag (they will assume it stays set).

*See the Secondary Master Flags, bit index 5 in the Parameter Reference.

SendRES

Description

Send a `RES` command to an axis. Typically used during initialization to set all counters to zero.

Signature

SendRES(`long nMask`**)**

Return Type

N/A

Parameters

`nMask:` Specifies which axes to apply the `RES`.

Return

N/A

Remarks

The `RES` command resets the encoder and other counters on the drive to zero on the controller for the specified axes attached to the profile defined in the **nMoveProfile** property. The `nMask` is treated as a field of 32 bits, or flags. Each flag corresponds to an axis.

The axes are represented by numbers 0, 1, 2, etc., based on the order they were attached to the profile. The `nMask` parameter is treated as a field of 32 bits, or flags. Each flag corresponds to an axis. Setting the `nMask` to 1 (binary 01) tells the **SendRES()** method that only Axis0 of profile **nMoveProfile** is moved. (As mentioned above, Axis0 is the first axis attached to the profile or master, Axis1 is the second axis attached, etc.)

This command does not use the binary syntax, so it will be queue up in the command stack behind moves, etc.

SetFlag

Description Changes the value of a specific bit flag, as defined on the ACR Device.

Signature **SetFlag(**`long nBit, bool bValue, bool bFast`**)**

Return Type N/A

Parameters

`nBit:` Bit number on card (different from p-Parameter.)

`bValue:` Value of bit:

Value	Description
TRUE	Set
FALSE	Clear

`bFast:` How to send the command:

Value	Description
TRUE	Binary
FALSE	ASCII

Return

Remarks ASCII commands queue in a command stack; they are visible in the Terminal interface. Binary commands are executed prior to any ASCII commands and are not seen by the Terminal.

When the `bFast` parameter is TRUE, a binary command is sent to the card.

When the `bFast` parameter is FALSE, an ASCII command is sent to the card.

SetGlobal

Description Changes the value of a specific, pre-dimensioned global parameter.

Signature **SetGlobal(**`long nCard, long nGlobal, double dValue, bool bFast`**)**

Return Type N/A

Parameters

`nCard:` Code value for type of card. This information is needed if using a binary command (`bFast`=TRUE) to find the memory address. Use zero if using ASCII (`bFast`=FALSE).

Device Types	
Value	**Controller**
0	ACR1200
1	ACR1500
2	ACR2000
3	ACR8010
4	ACR8020
5	ACR8020 (16-axis)
6	ACR1505
7	Reserved
8	ACR9000

`nGlobal:` Global p-Parameter number that is to be changed

`dValue:` Value to assign p-Parameter.

`bFast`: How to send the command:

Value	Description
TRUE	Binary
FALSE	ASCII

Return The range of global parameters is 0 through 4095. They are optionally allocated (using the `DIM` command) and are stored internally as 64-bit floating-point values.

The `bFast` parameter of this method determines if the parameters are to be assigned a value using a binary (`bFast`=TRUE) or an ASCII (`bFast`=FALSE)

command.

Remarks

ASCII: Does not require the `nCard` parameter value to be meaningful, and supports access to the full precision of the 64-bit number. However, the command will be displayed in the terminal and is sent without knowing if the global parameter has been dimensioned (see `DIM` command).

Binary: The command will not show up in the terminal and will throw an exception if the parameter is not valid. On the down side it can only provide a 32-bit precision number to the card, requires the user to know what card is in use, and can result in small variations in the data as it is converted from IEEE32 format to the internal 64-bit number.

SetParmFloat

Description

Changes the value of a specific p-Parameter of type Float.

Signature

SetParmFloat(`long nPparm, float fValue, bool bFast`**)**

Return Type

N/A

Parameters

`nPparm:` p-Parameter number that you want to change

`fValue:` Value to assign p-Parameter.

`bFast:` How to send the command:

Value	Description
TRUE	Binary
FALSE	ASCII

Return

On the ACR controller, parameters can have the following types: long, float, or double. Use this method to assign a value to floats and doubles. All floats and doubles are sent to the card as floats.

When the `bFast` parameter is TRUE, a binary command is sent to the card.

When the `bFast` parameter is FALSE, an ASCII command is sent to the card (for example P5456=x<cr> where x is the new value).

Remarks

Be careful to change only p-Parameters that are allowed to change. Modifying read-only parameter values has undefined behavior (and generally will not do what is intended).

Modifying p-Parameters that are 64-bit (e.g. the user defined variables) using the binary command (`bFast`=TRUE) results in a less than exact translation of the fractional part of the number. In addition, because **SetParmFloat()** only allows a 32-bit number in `fValue`, the `bFast`=FALSE setting will not be able to take advantage of the extra precision provided in the 64-bit values.

The **SetGlobal()** method can be used to set global p-Parameters and it has a double as input (which will be used if `bFast`=FALSE).

SetParmLong

Description Changes the value of a specific p-Parameter of type Long.

Signature **SetParmLong(**`long nPparm, float fValue, bool bFast`**)**

Return Type N/A

Parameters

`nPparm:` p-Parameter number that is to be changed

`fValue:` Value to assign p-Parameter.

`bFast:` How to send the command:

Value	Description
TRUE	Binary
FALSE	ASCII

Return On the ACR controller, parameters can have the following types: long, float, or double. Use this method to assign a value to longs.

When the `bFast` parameter is TRUE, a binary command is sent to the card.

When the `bFast` parameter is FALSE, an ASCII command is sent to the card (for example P5456=x<cr> where x is the new value)

Remarks Be careful to change only p-Parameters that are allowed to change. Modifying read only parameter values has undefined behavior (and generally will not do what is intended).

Do not use this method to set a parameter defined as floating point even if it is going to contain a whole number. The floating point values undergo special processing when sent using the binary syntax, and the result will not be what is expected,

Stop

Description Stops commanded motion with or without using the `DEC` and `JRK` values for the profile specified in **nMoveProfile**.

Signature **Stop(**`bool bDecel`**)**

Return Type N/A

Parameters

`bDecel:` How to stop motion:

Value	Description
TRUE	Stop all Moves
FALSE	Kill all Moves

Return When the `bDecel` parameter is TRUE, a Stop All Moves flag is set using the binary command.

When the `bDecel` parameter is FALSE, a Kill All Moves flag is set using binary.

Stop All Moves

Possible Bitsvalues for **nMoveProfile** 0-15::

Master Flags: BIT523, BIT555, BIT587, BIT619, BIT651, BIT683, BIT715, BIT747, BIT7435, BIT7467, BIT7499, BIT7531, BIT7563, BIT7595, BIT7627, BIT7659.

Kill All Moves

Possible valuesBits for **nMoveProfile** 0-15:

Master Flags: BIT522, BIT554, BIT586, BIT618, BIT650, BIT682, BIT714, BIT746, BIT7434, BIT7466, BIT7498, BIT7530, BIT7562, BIT7594, BIT7626, BIT7658.

Remarks The Stop All Moves or the Kill All Moves Request flag stops commanded motion at the Master level. It is equivalent to calling **SetFlag(**`BITx, TRUE`**)**, where x=a Stop All Moves or Kill All Moves flag number.

Stop All Moves: Uses the existing `DEC` and `JRK` values.

Kill All Moves: Ignores the existing `DEC` and `JRK` values. .

After using **Stop(),** you must manually clear the Kill All Moves flag before any new move can be made. This is the case for either value of bDecel, the Stop All Moves flag self clears and sets the Kill All Moves flag.

Note: To stop "axis" based moves like `JOG`, `CAM`, and `GEAR` requires knowledge of which axis is moving. The Communication Server does not maintain that information. Therefore, the **Stop** method is only able to stop motion for at the Master level. There are specific bit flags for stopping `JOG`, `CAM`, and `GEAR`, which can be set using `SetFlag()`.

Another option available to kill commanded motion and/or `JOG`, `CAM`, and `GEAR` on the ACR9000, is to use **SetFlag(**KAMR flag, `TRUE, TRUE`**)** with the KAMR flag associated with any axis attached to the master profile. The Kill All Motion Request (KAMR) flag (Axis0-15: BIT8467 – BIT8947) stops all motion, without regard to deceleration, on all axis connected to the common master profile of the axis this command is issued for. This flag must be cleared on all axes it is set on before any motion can be commanded (or anything else done.) The Kill All Moves flag (Axis0-7: BIT522 - BIT746, Axis8-15: BIT7434 - BIT7658) for the profile will also be set, and must be cleared prior to doing motion.

GetMoveCounter

Description

Retrieve the index of the currently executing move in a set of moves sent down using **MoveBatch()**.

Signature

GetMoveCounter(`long nCounter, long nIncrement`**)**

Return Type

N/A

Parameters

`nCounter:` The index value of the move currently active on the controller.

`nIncrement:` The step used to increment the `nCounter`.

Return

The `nCounter` and `nIncrement` are both updated after calling this method.

Remarks

The move counter this method gets is only implemented on some ACR products, using some communication mediums. If it is not supported, it will always return zero for both values.

On the bus-based products, this is an interrupt-driven mechanism to track the number of binary moves started on the controller (and not to be confused with the move counter registers present for each of the motion masters in the system parameter map and referenced in the **nMoveCounter** property). The interrupt functionality is used to take this communication "out of band". You can use this in high-performance applications to rapidly and accurately track the number of binary moves held in queue by the card.

There are two reasons to track this information. First, to enable the application to feed more moves into the controller in time to keep the internal move buffer from running dry. This is essential when running huge blended motion profiles. Second, to enable the controlling application to know the exact execution point in the series of moves, for recovery after a program interruption.

SetMoveCounter

Description Changes the intial value of the move counter.

Signature **SetMoveCounter(**`long nCounter, long nIncrement`**)**

Return Type N/A

Parameters

`nCounter:` The index value of the move currently active on the controller.

`nIncrement:` The step used to increment the `nCounter`.

Return N/A

Remarks This is generally used to initialize the move counter by populating both parameters with zero.

See **GetMoveCounter()** for information of the functionality of the move counter.

SetAcrMemory

Description Changes the value of a specific memory address on the ACR controller.

Signature **SetAcrMemory(**`long nType, long nAddress, SAFEARRAY values`**)**

Return Type N/A

Parameters

`nType:` The data type of the values being read:

Transport Types	
Value	**Description**
0	Long
1	Float(64)
2	Float(32)

`nAddress:` The starting physical memory address on the ACR product.

`values:` The data to be placed in memory starting at the address.

Return N/A

Remarks Any number of values can be placed into the ACR memory, but they must be all of the same type.

SetAcrMemoryMask

Description Changes the value of a specific memory address on the ACR controller.

Signature **SetAcrMemoryMask(**`long nAddress, long nNAND, long nOR`**)**

Return Type N/A

Parameters

`nAddress:` The starting physical memory address on the ACR product. This address must point to a variable of type long for the mask to properly work.

`nNAND:` Used to clear bits.

`nOR:` Used to set bits.

Return N/A

Remarks The two bit masks are combined with the value at the address to result in a new bit image for the data. The address must point to a long integer storage area. The `nNAND` mask is used to clear bits, and the `nOR` mask is used to set bits. The data is modified as follows:

data = (data AND NOT `nNAND`) OR `nOR`

SetParmLongMask

Description Changes the value of a specific parameter on the ACR controller.

Signature **SetParmLongMask(**`long nPparm, long nNAND, long nOR`**)**

Return Type N/A

Parameters

`nPparm:`	The parameter on the ACR product. This address must point to a variable of type long for the mask to properly work.
`nNAND:`	Used to clear bits.
`nOR:`	Used to set bits.

Return N/A

Remarks The two bit masks are combined with the value at the p-Parameter to result in a new bit image for the data. The p-Parameter must be a long integer storage area. The `nNAND` mask is used to clear bits and the `nOR` mask is used to set bits. The data is modified as follows:

data = (data AND NOT `nNAND`) OR `nOR`

SetFOV

Description Changes the value of the Feedrate Override for the specified axes.

Signature **SetFOV(**`long nMask, float fValue`**)**

Return Type N/A

Parameters

`nMask:`	Specifies which axes to use for the FOV.
`fValue:`	Set FOV of all specified axes to this value.

Return N/A

Remarks Causes an immediate setting of Feedrate Override for any or all axes specified in the `nMask`. The command is not queued and the FOV occurs when the command is first seen by the board.

SetROV

Description Changes the value of the Rapid Feedrate Override for the specified axes.

Signature **SetROV(**`long nMask, float fValue`**)**

Return Type N/A

Parameters

`nMask:`	Specifies which axes to use for the ROV.
`fValue:`	Set ROV of all specified axes to this value.

Return N/A

Remarks Causes an immediate setting of rapid Feedrate Override for any or all axes specified in the `nMask`. The command is not queued and the ROV occurs when the command is first seen by the board.

MoveBatch

Description Sends a set of fully defined moves for batch processing to the ACR Controller.

Signature **MoveBatch(**`long nType, SAFEARRAY moves`**)**

Return Type N/A

Parameters

`nType:` The data type of the values being read:

Transport Types	
Value	**Description**
0	Long
1	Float(64)
2	Float(32)

`moves:` All data required to complete any number of moves.

Return N/A

Remarks This method allows any number of moves of a specific type (long or float) to be batched and sent to the controller. This move method is completely separate from the `Move()` and `Arc()` move methods. All data used in **MoveBatch** is contained in the moves array, and multiple moves can be commanded in a single call.

InitPerformance

Description Initializes the performance counters for the ISA card to zero.

Signature **InitPerformance()**

Return Type N/A

Parameters `N/A`

Return N/A

Remarks See **GetPerformance()**

GetPerformance

Description	Get the performance counters for the ISA card.
Signature	**GetPerformance()**
Return Type	SAFEARRAY
Parameters	N/A
Return	Returned array contains 38 longs that match up to the following C structure: unsigned long pNumMovesArrived, unsigned long pNumMovesStarted, unsigned long pNumStatusFastSent, unsigned long pNumStatusSent, unsigned long pNumStatusBack, unsigned long pNumTimesWriteServiceCalled, unsigned long pNumTimesReadServiceCalled, unsigned long pNumTimesBinServiceCalled, unsigned long pNumTimesStatusServiceCalled, unsigned long pNumBytesTransfWriteService, unsigned long pNumBytesTransfReadService, unsigned long pNumBytesTransfBinService, unsigned long pNumBytesWriteStatusService, unsigned long pNumBytesReadStatusService, unsigned long pNumTimesWriteCalled, unsigned long pNumTimesReadCalled, unsigned long pNumBytesTransfWrite, unsigned long pNumBytesTransfRead, unsigned long pNumTimesBinaryWriteCalled, unsigned long pNumTimesBinaryReadCalled, unsigned long pNumBytesTransfBinaryWrite, unsigned long pNumBytesTransfBinaryRead, __int64 pTimeSpentWriteService, __int64 pTimeSpentReadService, __int64 pTimeSpentBinService, __int64 pTimeSpentStatusService, __int64 pTimeSpentRead, __int64 pTimeSpentWrite, __int64 pTimeSpentBinaryRead, __int64 pTimeSpentBinaryWrite,
Remarks	This method returns data pertinent only to the ACR ISA bus card controller. The data is returned in an array of longs that matches up to a preexisting C++ data structure of longs, shown in the Return section. Note that the final 8 values are 64 bit longs—they take up two longs in the returned array and must be recreated as 64-bit values.

Error Messages

Internally, the Communications Server uses the error codes found in the table below when it encounters problems. These error values are returned as part of a Microsoft COM exception. See the code samples for examples of properly handling such exceptions.

Note: When an error occurs, the communications server throws a COM exception with an HRESULT. The standard for an error HRESULT sets its most significant bit (S in the Bit Layout) to 1, the Facility value to FACILITY_ITF (4) and the Code value to the error number in the table. The COM exception also includes the IErrorInfo Interface. This Interface consists of a Description property that contains free form text that includes the error number in decimal form, the error message text and internal communications server reference. When coding to handle specific errors, use the HRESULT; however, to display an error, it is easier to use IErrorInfo.

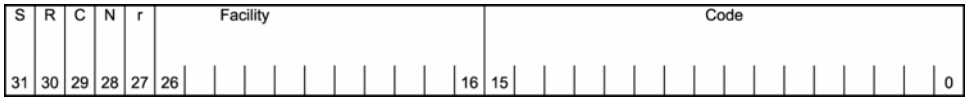


Figure 1 Bit Layout of the HRESULT

Error Code	Error Message
17000	Unknown Error Intercepted.
17001	The value requested is not present in Resource file.
17002	The PPU must be a number greater than zero.
17003	The valid range for the encoder resolution multiplier is between -4 and 4.
17004	The minimum value for a stream is 256.
17005	The value is out of range.
17006	The file prefix number must be positive.
17007	The data collected failed to pass final validation.
17008	The file name is not valid.
17009	Cannot open file, check that directory exists and you have the correct permissions access the file.
17010	The expected Map Index within the file was not found.
17011	The space reserved for storing loop information has become full.
17012	The line positioning information for the loop is not present.
17013	Problem trying to calculate value, Right Parentheses expected.
17014	Problem trying to calculate value, Function is not defined.
17015	Problem trying to calculate value, expected a primary expression.
17016	Problem trying to calculate value, trying to divide by zero.
17017	Problem trying to calculate value, an unrecognized character is being used.
17018	There is insufficient space to store the generated PLC programs. As much as possible has been stored.
17049	Some precondition for transferring the file has not been met.
17050	The Port is already connected. Only call the **Open()** method once per object or

Error Code	Error Message
	after a **Close()**.
17051	The serial port number must be between 1 and 36.
17052	The maximum number of concurrently opened ports has been reached.
17053	There is a problem allocating memory for the port connection.
17054	Problem accessing port. Check that no other application is using the port and that the port settings are valid.
17055	Unable to create the read thread for the port.
17056	To perform read or write operations on the port, it must first be open.
17057	The Settings for the port are incorrect.
17058	The change to the BPS rate encountered some problem. BPS rate is unchanged.
17059	No file name provided. To transfer a file a file name must be provided.
17060	The File Transfer method requested is not provided at this time.
17061	Only one file transfer per port at one time. Please wait until the current file transfer has completed and retry.
17062	Unable to create the file transfer thread.
17063	Unable to create the mutex for the read thread.
17064	The file name provided could not be read.
17065	Unable to create or set the file transfer event for the port.
17066	The function has terminated because it encountered too many errors.
17067	The write function has timed out. Check that communications are working, or reduce the amount of data being written.
17068	The program is unable to process incoming data fast enough and is losing it. Make sure external device is using flow control.
17069	The communications flow control could not be changed due to some error.
17070	Unable to set the specified flow control characters.
17071	Unable to resize the Transmit or Receive buffers.
17072	The data provided to populate the object state failed in some way. The data appears to be corrupt.
17073	The expected return value for the transfer was not found.
17074	Unable to create the status thread.
17075	Unable to create or set the status event.
17076	The key provided to collect status is invalid or improperly formed.
17077	There is no status information available for the specified message key.
17078	Unable to create the mutex for the status.
17079	The specified device type or index is invalid.
17080	Problem communicating with controller. Check the port settings are valid and that the controller is powered up and connected.
17081	There was a problem reported while trying to wake up the device. The device may not be responding.
17082	The transfer was canceled by the user.
17083	The communication attempt has timed out because it did not get the expected response.
17084	One of the p-Parameters provided in the status request is unknown or poorly

Error Code	Error Message
	formed.
17085	The provided hex data string representation contains characters that are not valid hex numbers.
17086	The data provided in the array did not match the expected conditions or is not there at all.
17087	The socket type provided is not currently supported by the software.
17088	The Application Window cannot be found. The application may not be running.
17089	Old version of file found and read. New items and features added since this file was created will be set at their defaults.
17090	The destination device on the Network is not reachable at this time.
17091	There was a time out trying to communicate with the device on the Network.
17092	Some failure caused the echo request to fail.
17093	This version of the Ping Request requires the icmp.dll file be present on the computer. It was not found.
17094	The Host name or IP address cannot be resolved to a know host device. Check that IP or Host name is correct.
17095	The transfer was canceled at the other end of the connection.
17096	Failed to Create the desired Thread.
17097	Unable to Start the desired Thread.
17098	Unable to Create the desired Thread Event.
17200	Foreign Error Source
17201	Standard Library Exception
17202	Windows Socket Architecture Library Exception