TECAN.

Technical Reference

Software Communication Drivers

12/6/06

740373-A

TECAN.

Technical Reference

Software Communication Drivers

Tecan Systems, Inc.
2450 Zanker Road
San Jose, CA 95131 USA

T 1 408 953 3100, Toll Free 1 800 231 0711
F 1 408 953 3101

E-mail: tecansystemsinfo@tecan.com
Web site: www.tecansystems.com

12/6/06

740373-A A

Part Number 740373-A

Copyright and Trademark Information

Product Documentation Warranty Information

The information contained in this document is subject to change without notice. Tecan Systems makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Tecan Systems shall not be liable for errors contained in this document or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Contents

1 RSP Communication Server

1.1 Overview

All messages travelling between the robotic hardware and the control PC pass through the RSP Communication Server. The server eliminates the need to create low-level communication code. The RSP Communication Server, an in-process server, conforms to the Component Object Model. The functional features are:

- **Bi-directional communication** between the robotic hardware and the control PC
- **Multi-threading communication** between the robotic hardware and the control PC
- **Logging of messages** to a screen window, a text file, or both
- **Error handling** that conforms to the standard COM interface

1.1.1 Server Name and Registry

The location of the RSP Communication Server is not important, but it must be registered before it can provide service to the client.

The line command for registering is:

```
    C:\> Regsvr32 RSPCommServer.dll
```

1.1.2 Simulation Mode

The RSP Communication Server can simulate communication without having attached hardware. However, it is the client's responsibility to build the proper virtual hardware configuration. The communication server provides two methods to configure the virtual hardware, `SimuAddArm` and `SimuAddDevice`.

When in simulation mode, the communication server calls `SimuAddArm(1,2876,2108,1700)` by default. As a consequence of this default call, only a left arm with machine range `2876,2108,1700` (in steps) exists for simulation. The client can overwrite the setting by calling `SimuAddArm` again.

Use `SimuAddDevice` to add pumps in simulation mode. For simulation, for any call to `SimuAddArm` or `SimuAddDevice`, the RSP Communication Server assumes that the arm or device is powered up. Therefore, you must explicitly initialize the robotic instrument.

At any time, the client can switch back and forth between simulation mode and real communication mode by setting the property `RSPSimulation` as `true` or `false`.

In the simulation mode, all other methods work in the same way as in the real communication, except that log entries are different.

Table 1-1 shows the differences in logging between the simulation mode and real communication mode.

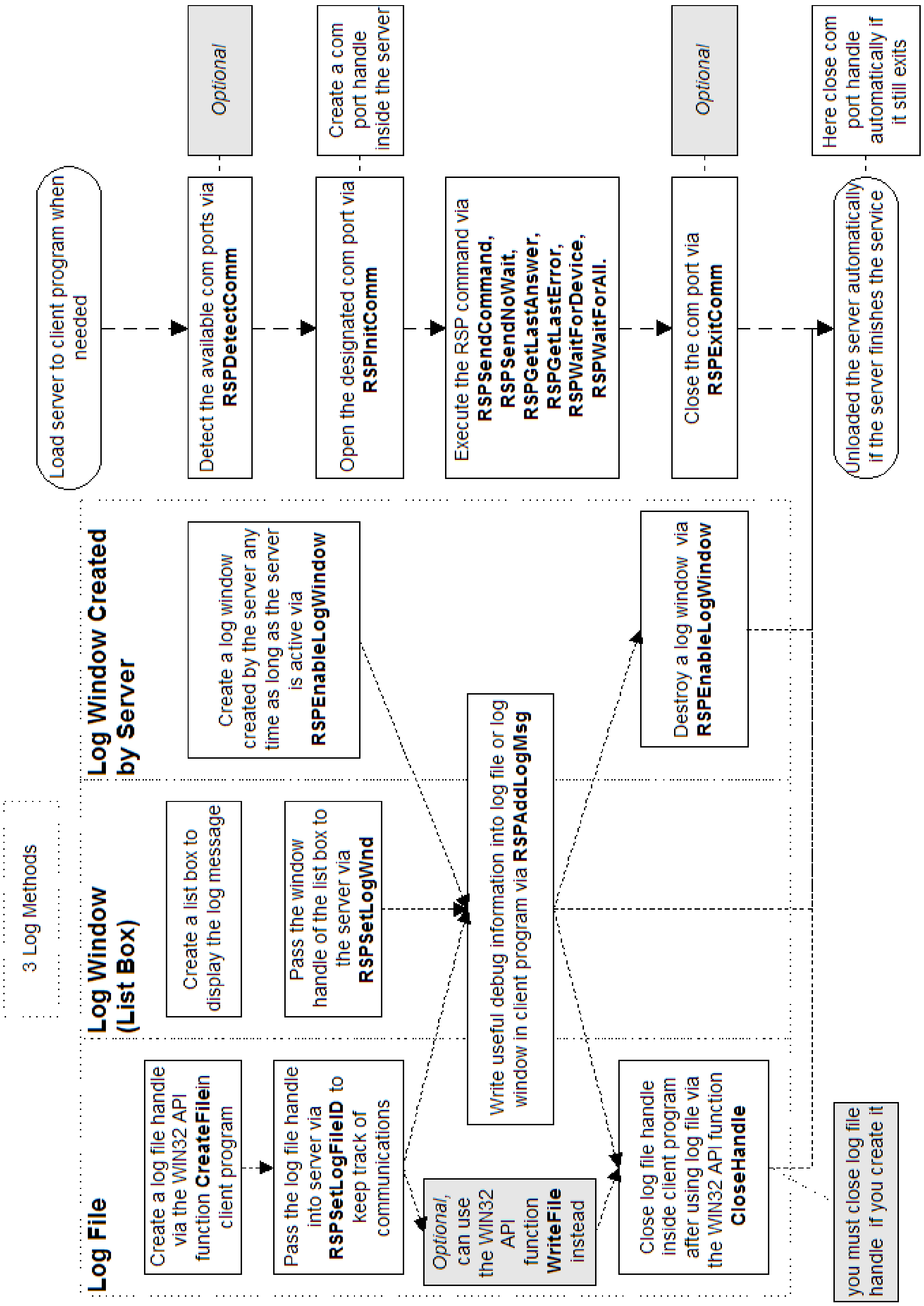
Table 1-1 *Log Entry Differences for Simulation Mode*

Simulation Mode	Real Communication Mode	Log Entry Message Type Indicators
>>	>	Indicates the entry is a comment sent to the machine from the control PC
— (em dash)	- (hyphen)	Indicates the entry is a response from the machine to the control PC. — or - indicates no error.
**	*	Indicates the entry is a response from the machine to the control PC. ** or * indicates an error.

1.1.3 Notes for the Client

Fig. 1-1 shows the procedure for using the RSP Communication Server in a client program.

Fig. 1-1 Procedure for Using the Communication Server in a Client Program


Load server to client program when needed
Detect the available com ports via RSPDetectComm
Optional
Open the designated com port via RSPInitComm
Create a com port handle inside the server
Execute the RSP command via RSPSendCommand, RSPSendNoWait, RSPGetLastAnswer, RSPGetLastError, RSPWaitForDevice, RSPWaitForAll.
Close the com port via RSPExitComm
Optional
Unloaded the server automatically if the server finishes the service
Here close com port handle automatically if it still exits
3 Log Methods
Log Window Created by Server
Create a log window created by the server any time as long as the server is active via RSPEnableLogWindow
Destroy a log window via RSPEnableLogWindow
Log Window (List Box)
Create a list box to display the log message
Pass the window handle of the list box to the server via RSPSetLogWnd
Write useful debug information into log file or log window in client program via RSPAddLogMsg
Log File
Create a log file handle via the WIN32 API function CreateFilein client program
Pass the log file handle into server via RSPSetLogFileID to keep track of communications
Optional, can use the WIN32 API function WriteFile instead
Close log file handle inside client program after using log file via the WIN32 API function CloseHandle
you must close log file handle if you create it

The typical steps to use the RSP Communication Server are:

1 Select the log method to record communication exchanges.

There are two ways to log the commands sent to the instrument and the replies received from the instrument:

– Create a log file to saved logged information to a text file.

The client program can create a log file handle with the WIN32 API function `CreateFile`. The log file is in text format, so any text editor (such as WordPad or NotePad) can be used to access it. Call `RSPSetLogFileID` to pass the log file handle to the server.

– Create a log window to display the logged message.

The client program can create a log window to display the logged message if you desire. Use the log window created by the communication server to display the logged information.

As long as the server is active, you can create or destroy the log window created inside the communication server using `RSPEnableLogWindow`.

The client program can share the same log file or log window. You can use `RSPAddLogMsg` to write useful information into the log file or log window (apart from the data written to the file by the server).

2 Open the COM port.

To find the available COM ports of the computer, call `RSPDetectComm`. This is optional; it limits the choice of COM ports offered to you.

Call `RSPInitComm` to open the designated COM port.

The COM port is initialized 9600 baud, 8 data bits, 1 stop bit, and no parity.

3 After the port is open, use any of the communication functions (`RSPSendCommand`, `RSPSendNoWait`, `RSPGetLastAnswer`, `RSPGetLastError`, `RSPWaitForDevice`, or `RSPWaitForALL`).

4 Close the log file.

Before exiting, the client program must close the log file by calling the WIN32 API function `CloseHandle` if it is created as listed in Step 1 above. To close the COM port, the client program can either call `RSPExitComm`, or you can simply rely upon the server to close the port automatically on exit.

1.2 Warnings

WARNING! *Read the safety notes and communication instructions in your device manuals before operating your devices.*

WARNING! *Operate your devices with caution. Risks to users, property and the environment can arise when devices are used carelessly or improperly.*

1.3 RSP Communication Server Interface

This section contains descriptions, syntax, and parameter information for properties, methods, and events used by the RSP Communication Server.

1.3.1 RSP Communication Server Properties

Use the following list to locate individual properties:

`RSPSimulation`

Switches between simulation and real communication mode.

VB	`Property RSPSimulation As Boolean`

C++	`get_RSPSimulation(VARIANT_BOOL *pVal);`
	`put_RSPSimulation(VARIANT_BOOL newVal);`

C#	`bool RSPSimulation`
Parameter Type	Boolean
Input Parameters	None
Returned Parameters	None

Notes

You can change the mode at design time or at run time.

Use the methods `SimuAddArm` and `SimuAddDevice` to add more device commands to the simulation mode. If the client does not call `SimuAddArm` or `SimuAddDevice`, the communication server assumes by default that the robotic instrument only has a left arm with the machine range (2876,2108,1700) steps.

`BaudRate`

Represents the baud rate the communication server uses.

VB	`Property BaudRate As RspBaudRate`

C++	`get_BaudRate(RspBaudRate *pVal);`
	`put_BaudRate(RspBaudRate newVal);`

C#	`RSPCOMMSERVERLib.RspBaudRate BaudRate`
Parameter Type	RspBaudRate enum
Input Parameters	None
Returned Parameters	None

Notes

Only two baud rates can be selected (namely, 9600 and 38400). These baud rates are defined in an enum type named RspBaudRate. RspBaudRate.RSP9600 represents baud rate 9600 and RspBaudRate.RSP38400 represents baud rate 38400.

CommandAckTimeOut

Represents the number of the acknowledge timeout increment (in 50 ms increments). The value must be between 1 and 40, with a default value of 18.

VB	`Property CommandAckTimeOut As Long`

C++	`get_CommandAckTimeOut(long *pVal);`
	`put_CommandAckTimeOut(long newVal);`

C#	`int CommandAckTimeout`
Parameter Type	Long
Input Parameters	None
Returned Parameters	None

Notes

Since each timeout increment is 50 ms, a `CommandAckTimeOut` value of 10 means a total of 500 ms (10 * 50=500) timeout period.

CommandRetryCount

Represents how many times to resend the command in case of communication failure. The value must be between 0 and 10, with a default value of 7.

VB	`Property CommandRetryCount As Integer`

C++	`get_CommandRetryCount(short *pVal);`
	`put_CommandRetryCount(short newVal);`

C#	`short CommandRetryCount`
Parameter Type	Integer
Input Parameters	None
Returned Parameters	None

Notes

A `CommandRetryCount` value of 0 means the command is sent only once and there is no retry in case of failure.

`EnableLog`

Allows the client application to turn on or off the communication logging.

VB	`Property EnableLog As Boolean`

C++	`get_EnableLog(VARIANT_BOOL *pVal);`
	`put_EnableLog(VARIANT_BOOL newVal);`

C#	`bool EnableLog`
Parameter Type	Boolean
Input Parameters	None
Returned Parameters	None

Notes

None.

`LogComPort`

Represent whether Com Port information will be added to each log entry. If the parameter is set to True, then the COM port number will appear in the log.

VB	`Property LogComPort As Boolean`

C++	`get_LogComPort(VARIANT_BOOL *pVal);`
	`put_LogComPort(VARIANT_BOOL newVal);`

C#	`bool LogComPort`
Parameter Type	Boolean
Input Parameters	None
Returned Parameters	None

Notes

None.

1.3.2 RSP Communication Server Methods

Use the following list to locate individual methods.

RSPInitComm

Opens the designated COM port.

VB	`Sub RSPInitComm(byPortNUmber As Byte)`
C++	`HRESULT RSPInitComm(BYTE byPortNumber)`
C#	`void RSPInitComm(byte byPortNumber)`
Input Parameters	• `byPortNumber`: COM port number (1,2,3... 255)
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate the open status. One of: • S_OK: the port open successfully • Anything other than S_OK: see [create cross-reference]Appendix A, Error Handling.

Notes

Data bits, stop bits, and parity settings are fixed and non-selectable.

A handle of the designated COM port is produced inside the server if the open function is successful. RSPGetCommID can retrieve the handle; the handle is closed by RSPExitComm or automatically closed when the server is unloaded.

A VB client can access the COM errors using the `Err` object.

RSPGetCommID

Gets the handle of the opened COM port.

VB	`Function RSPGetCommID() As Long`
C++	`HRESULT RSPGetCommID(long *pLCommHandle)`
C#	`int RSPGetCommID()`
Input Parameters	None
Returned Parameters	Always S_OK (0)
Return Value	The handle of the COM port. One of: • Handle of the COM port is the COM port is opened successfully • “0” if the COM port fails to open

Notes

The communication server gets the COM port handle with a long integer. For C++, cast it as `HANDLE` before using it in the client.

Inside the server, the communication server initializes the handle of the COM port as “0”. It is assigned a new value after `RSPInitComm` if the COM port is successfully opened.

RSPExitComm

Closes the COM port if the COM port has been opened.

VB	`Sub RSPExitComm()`
C++	`HRESULT RSPExitComm()`
C#	`void RSPExitComm()`
Input Parameters	None
Returned Parameters	Always S_OK (0)
Return Value	None

Notes

The COM port is closed automatically when the server is unloaded; it is not necessary to call this function from the client program.

RSPDetectComm

Determines the status of the designated COM port. This is a quick, easy way to determine whether or not a COM port is available.

VB	`SubRSPDetectComm(byPortNumber As Byte)`
C++	`HRESULT RSPDetectComm(BYTE byPortNumber)`
C#	`void RSPDetectComm(byte byPortNumber)`
Input Parameters	• `byPortNumber`: COM port number (1,2,3... 255)
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate the detecting status. One of: • S_OK: the port open successfully • Anything other than S_OK: see [create cross-reference]Appendix A, Error Handling.

Notes

If access is denied, then the designated COM port has been opened before.

A VB client can access the COM errors using the `Err` object.

RSPSendNoWait

Sends a command to the designated device and returns without waiting for the reply.

VB	`Sub RSPSendNoWait(pCommand As String, byArmNo As Byte, byDevNo As Byte)`
C++	`HRESULT RSPSendNoWait(BSTR bstrCommand, BYTE byArmNo, BYTE byDevNo)`
C#	`void RSPSendNoWait(string bstrCommand, byte byArmNo, byte byDevNo)`
Input Parameters	• `bstrCommand`: Command string • `byArmNo`: - 1—Left arm - 2—Right arm • `byDevNo`: - 1, 2, 3, 4—Diluter - 6—Fast Wash - 7—LICOS - 8—Arm - 9—Interactive driver control or digital I/O
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate the sending status. One of: • `S_OK`: the command sent successfully • Anything other than `S_OK`: see [create cross-reference]Appendix A, Error Handling.

Notes

The device number is defined according to the robotic instrument command protocol.

The maximum length of the command string is 255.

This function only sends commands to device, and does not wait for the reply. Normally, using `RSPGetLastAnswer` following this function returns the answer message from the device. `RSPSendCommand` can be used instead of this command if the automatic reply from the device is desired.

A VB client can access the error using the `Err` object.

RSPGetLastAnswer

Gets the answer string from the designated arm and device. The answer string is an in/out parameter.

VB	`Sub RSPGetLastAnswer(byArmNo As Byte, byDevNo As Byte, pAnswer As String)`
C++	`HRESULT RSPGetLastAnswer(BYTE byArmNo, BYTE byDevNo, BSTR *bstrAnswer)`
C#	`void RSPGetLastAnswer(byte byArmNo, byte byDevNo, ref string bstrAnswer)`
Input Parameters	• `byArmNo`: - 1—left arm - 2—right arm • `byDevNo`: - 1, 2, 3, 4—Diluter - 6—Fast Wash - 7—LICOS - 8—Arm - 9—Interactive driver control or digital I/O • `bstrAnswer`: Pointer to answer BSTR string
Returned Parameters	Answer string from the designated arm and device
Return Value	Always S_`OK` (0)

Notes

The maximum length of the answer string is 255.

RSPGetLastError

Gets the device error information.

VB	`Sub RSPGetLastError(byArmNo As Byte, byDevNo As Byte)`
C++	`HRESULT RSPGetLastError(BYTE byArmNo, BYTE byDevNo)`

C#	`void RSPGetLastError(byte byArmNo, byte byDevNo)`
Input Parameters	• `byArmNo`: - 1—left arm - 2—right arm • `byDevNo`: - 1, 2, 3, 4—Diluter - 6—Fast Wash - 7—LICOS - 8—Arm - 9—Interactive driver control or digital I/O
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate the status. One of: • S_OK: the port opened successfully • Anything other than S_OK: see [create cross-reference]Appendix A, Error Handling.

Notes

Non-C++ users should note that this function does not return an error number. If there is an error, the function will raise an Err object that has to be handled using the standard VB “On Error GoTo...” syntax. See the VB documentation for more information.

RSPSendCommand

Sends a command to the designated device to get the reply message from the device and the device error detail. The answer string is an in/out parameter.

VB	`Sub RSPSendCommand(pCommand As String, byArmNo As Byte, byDevNo As Byte, pAnswer As String)`
C++	`HRESULT RSPSendCommand(BSTR bstrCommand, BYTE byArmNo, BYTE byDevNo, BSTR *bstrAnswer)`

C#	`void RSPSendCommand(string pCommand, byte byArmNo, byte byDevNo, ref string bstrAnswer)`
Input Parameters	• `pCommand`: Command BSTR string • `byArmNo`: - 1—left arm - 2—right arm • `byDevNo`: - 1, 2, 3, 4—Diluter - 6—Fast Wash - 7—LICOS - 8—Arm - 9—Interactive driver control or digital I/O • `bstrAnswer`: Pointer to answer BSTR string
Returned Parameters	Device answer string
Return Value	An HRESULT type of long integer to indicate the status. One of: • `S_OK`: the command was sent successfully • Anything other than `S_OK`: see [create cross-reference]Appendix A, Error Handling.

Notes

The maximum length of both the command and answer strings is 255.

This function combines the actions of `RSPSendNoWait`, `RSPGetLastAnswer`, and `RSPGetLastError`.

A VB client can access the error using the `Err` object.

RSPWaitForDevice

Waits for the designated device to become ready before leaving the procedure.

VB	`Sub RSPWaitForDevice(byArmNo As Byte, byDevNo As Byte)`
C++	`HRESULT RSPWaitForDevice(BYTE byArmNo, BYTE byDevNo)`

C#	`void RSPWaitForDevice(byte byArmNo, byte byDevNo)`
Input Parameters	• `byArmNo`: - 1—left arm - 2—right arm • `byDevNo`: - 1, 2, 3, 4—Diluter - 6—Fast Wash - 7—LICOS - 8—Arm - 9—Interactive driver control or digital I/O
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

None

RSPWaitForAll

Waits for all devices to be inactive before leaving the procedure.

VB	`Sub RSPWaitForAll()`
C++	`void RSPWaitForAll`
C#	`void RSPWaitForAll()`
Input Parameters	None
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

None

RSPCheckDevStatus

Checks to see whether the device is still executing the previous command, or if it has already finished execution. The client can use this method to check whether or not the command sent to the machine has completed.

VB	`Function RSPCheckDevStatus (byArmNo As Byte, byDevNo As Byte) As Boolean`
C++	`RSPCheckDevStatus (BYTE byArmNo, BYTE byDevNo, BOOL *bBusy)`
C#	`bool RSPCheckDevStatus(byte byArmNo, byte byDevNo)`
Input Parameters	• `byArmNo`: - 1—left arm - 2—right arm • `byDevNo`: - 1, 2, 3, 4—Diluter - 6—Fast Wash - 7—LICOS - 8—Arm - 9—Interactive driver control or digital I/O
Returned Parameters	Status flag to check if the device is busy or not: • False—Ready • True—Busy
Return Value	Always S_`OK` (0)

Notes

None

RSPEnableLogWindow

Creates or destroys the log window. You can create a log window inside the server, which monitors the communication between the computer and the machine. The client calls this function to create/destroy the log window. The log window is a modeless window.

VB	`Sub RSPEnableLogWindow(byEnable As Byte, [pwndParent As Long])`
C++	`HRESULT RSPEnableLogWindow(BYTE byEnable, int pwndParent)`

C#	`void RSPEnableLogWindow(byte byEnable, int pWndParent)`
Input Parameters	• `byEnable`: A byte type of integer to control whether the log window displays or not: - 1—Display the log window - 0 or other—Destroy the log window, if it exists • `pWndParent`: Parent window handle
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate if the log window has been created. One of: • S_OK: the log window opened successfully • Anything other than S_OK: see [create cross-reference]Appendix A, Error Handling.

Notes

The formats of a message sent to a machine or a reply received from a machine are:

- Command sent to the machine: `> [Arm number][Device number], [Command string]`

  For example, `> 18, RV` means that the arm number is 1, the device number is 8, and the command is RV.

- Answer from the machine: `- | * [Arm number][Device number], [Error number], [Answer string from the machine, if any]`, where `-` means there was no error and `*` means there was an error.

  For example, `- 18, 0, RSP900-V4.20-07/90` means that the arm number is 1, the device number is 8, the error number is 0 (that is, there was no error), and the answer string is `RSP900-V4.20-07/90`.

  For example, `* 18, 10` means that the arm number is 1, the device number is 8, and the error number is 10.

RSPGetLogWindowID

Gets the log window handle.

VB	`Function RSPGetLogWindowID() As Long`
C++	`HRESULT RSPGetLogWindowID(long *hWnd)`
C#	`int RSPGetLogWindowID()`
Input Parameters	None
Returned Parameters	The log window handle as a long integer
Return Value	Always S_OK (0)

Notes

The communication server outputs the log window handle as a long integer; for C++, if you want the client to use the log window handle, change the long integer to `HWND`.

`RSPSetLogFileID`

Passes the log file handle from the client program to the server. Use the log file to debug the program or procedure.

VB	`Sub RSPSetLogFileID(lLogFileID As Long)`
C++	`HRESULT RSPSetLogFileID(long lLogFileID)`
C#	`void RSPSetLogFileID(int lLogFileID)`
Input Parameters	• `lLogFileID`: Handle of log file
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

Inside the server, if the client program uses this function to create the log file handle and pass it into the server, the communication server writes the send command, reply message, and some other actions and results into a log file automatically.

For example, you can use the WIN32 function CreateFile to create a handle to the log file: `hLogFile = CreateFile (“Log.txt”, GENERIC_WRITE, FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL).`

The log file must be opened inthe client program. You can use the same log file for the client program and for the server. RSPAddLogMsg can be used to write additional information into the log file.

The log file must be closed by the client program after using it. You can use the WIN32 function CloseHandle to close it.

The log file is in ASCII text format so you can use standard software to access it.

The log format of commands and answers is documented in ”RSPEnableLogWindow” on page 1-16.

`RSPGetLogFileID`

Gets the log file handle used in the server.

VB	`Function RSPGetLogFileID() As Long`
C++	`HRESULT RSPGetLogFileID(long *pLLogFileID)`
C#	`int RSPGetLogFileID()`
Input Parameters	None
Returned Parameters	Handle of log file
Return Value	Always S_OK (0)

Notes

Normally, this function is not needed because the log file handle is created in the client program and passed into the server.

RSPSetLogWnd

Passes a user-created log window handle to the server. The user should create a log window with a control of type `ListBox` and pass the `ListBox` control window handle to the RSP Communication Server using this method.

VB	`Sub RSPSetLogWnd(lLogHWND As Long)`
C++	`HRESULT RSPSetLogWnd(long lLogHWND)`
C#	`void RSPSetLogWnd(int lLogHWND)`
Input Parameters	• `lLogHWND`: The log window handle as a long integer
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

In the server, the send command, reply message and some other actions and results are written into the log window automatically. The client can use `RSPAddLogMsg` to log additional information into the log window.

The log format of commands and answers is documented in ”RSPEnableLogWindow” on page 1-16.

RSPAddLogMsg

Writes one line of text into the log file or log window if either one exists. The client application can use this function to write a line of text into the log file or log window.

VB	`Sub RSPAddLogMsg(bstrLogMsg As String, byLevel As Byte, [varWhich])`
C++	`HRESULT RSPAddLogMsg(BSTR bstrLogMsg, BYTE byLevel, VARIANT varWhich)`
C#	`void RSPAddLogMsg(string bstrLogMsg, byte byLevel, object varWhich)`
Input Parameters	• `bstrLogMsg`: The string that the communication server writes to the log file or log window • `byLevel`: The integer that determines the indent level: - 0 or other integer—the communication server will not indent the text - 1—indent six spaces - 2—indent twelve spaces • `varWhich`: An optional parameter to guide the writing: - 0 or other integer—write the string to all the log output (log file and log window(s), if any) - 1—write the string only to the log window (including the window created by the server) - 2—write the string only to the log file, if there is a log file
Returned Parameters	None
Return Value	An HRESULT type of log integer to indicate the writing result: • S_OK: the command has been executed successfully • Anything other than S_OK: see [create cross-reference]Appendix A, Error Handling.

Notes

The maximum string length of one line written into the log file is 512 characters.

A VB client can access the error using the `Err` object.

SimuAddArm

Adds therobotic instrument arm command set and proper reply message for the simulation mode.

VB	`Sub SimuAddArm(byArm As Byte, lRangeX as Long, lRangeY as Long, lRangeZ As Long)`
C++	`HRESULT SimuAddArm(BYTE byArm, long lRangeX, long lRangeY, long lRange 2)`
C#	`void SimuAddArm(byte byArm, int lRangeX, int lRangeY, int lRangeZ)`
Input Parameters	• `byArm`: - 1—left arm - 2—right arm • `lRangeX`: Machine range in X direction, in motor steps • `lRangeY`: Machine range in Y direction, in motor steps • `lRangeZ`: Machine range in Z direction, in motor steps
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

If SimuAddArm is called more than once, the most recent call overwrites the previous one. For example, if the client first calld `SimuAddArm(1, 2500, 1500, 27)`, and subsequently calls `SimuAddArm(1, 1200, 1400, 1000)`, the communications server will think that the left arm has the machine range of `(1200, 1400, 1000)`

In simulation mode, the default call is `SimuAddArm(1, 2876, 2108, 1700)`. The client can overwrite the machine by calling this function again.

SimuAddDevice

Adds a new device to the simulation mode.

VB	`SubSimuAddDevice(byArm As Byte, byDev As Byte, ustType As RspDeviceType, useRes As RspDeviceRes)`
C++	`HRESULT SimuAddDevice(BYTE byArm, BYTE byDev, RspDeviceType ustType, RspDeviceRes ustRes)`
C#	`void SimuAddDevice(byte byArm, byte byDev, RSPCOMMSERVERLib.RspDeviceType ustType, RSPCOMMSERVERLib.RspDeviceRes ustRes)`
Input Parameters	• `byArm`: - 1—left arm - 2—right arm • `byDev`: - 1, 2, 3, 4—Diluter • `ustType`: RSPDevice type • `ustRes`: RSPDevice resolution
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

If the device does not have the property of resultion, such as an IO board or linear option board, the last parameter has no effect on simulation the device.

If `SimuAddDevice` is called more than once, the most recent call overwrites the prrevious one. For example, the client first calls `SimuAddDevice(1, 1, RspXp3000, RspHighRes)` and subsequently calls `SimuAddArm(1, 1, RspX13000, RspStandard Res)`. After the second call, the communication server thinks the device `l` is `XL3000` pump with standard resolution.

The input parameter `ustType` is a user-defined type, defined as follows:

```
typedef enum RspDeviceType
   {
      RspXp3000 = 1     // XP3000 pump
      RspXl3000 = 2     // XL3000 pump
      RspXl3000M = 3    // XL3000 multi-channel pump
      RspXe1000 = 4     // XE1000 pump
      RspSv    = 5      // Smart Valve
      RspPeri  = 6      // Peristaltic pump
      RspIo    = 7      // IO board
      RspLo    = 8      // Linear option board
   } RspDeviceType;
```

The input parameter `ustRes` is a user-defined type, defined as follows:

```
typedef enum RspDeviceRes
   {
      RspStandardRes = 0    // Standard resolution
      RspHighRes = 1        // High resolution
   } RspDeviceRes;
```

RSPGetServerVersion

Gets the version of the communication server. This function allows the client program to determine the revision number, which may imply whether a given feature or function is available.

VB	`Function RSPGetServerVersion() As Byte`
C++	`HRESULT RSPGetServerVersion(BYTE *pByVersion)`
C#	`byte RSPGetServerVersion()`
Input Parameters	None
Returned Parameters	A three-digit integer, representing the server version
Return Value	Always S_OK (0)

Notes

This function is obsolete, and is only included for backward compatibility. We discourage the use of this function. Please use other standard ways to obtain a DLL version.

In future releases, this function might be removed.

1.3.3 RSP Communication Server Events

RSPCommandComplete

This event is fired by the RSP Communication Server when a command sent by the server is completed by the external device and a command reply has been received successfully from the device.

VB	`Event RSPCommandComplete(byArmNo As Byte, byDevNo As Byte, lError As Long)`
C++	`HRESULT RSPCommandComplete(BYTE byArmNo, BYTE byDevNo, long lError)`
C#	`RSPCOMMSERVERLib.IRSPCommEvents.RSPCommandComplete (byte byArmNo, byte byDevNo, int errNo)`
Input Parameters	• `byArmNo`: - 1—left arm - 2—right arm • `byDevNo`: - 1, 2, 3, 4—Diluter - 6—Fast Wash - 7—LICOS - 8—Arm - 9—Interactive driver control or digital I/O • `lError`: error code returned from the device. 0 means no error.
Returned Parameters	None
Return Value	Always S_`OK` (0)

Notes

None.

2 Pump Communication Server

2.1 Overview

All messages transmitted between the diluter (or other Cavro device) and the control PC are handled by the Pump Communication Server. The server eliminates the need to create low-lever communication code. The Pump Communication Server is an in-process server that conforms to the Component Object Model. The functional features are:

- **Bi-directional communication** between the diluter and the control PC
- **Multi-threading communcations** between the diluter and the control PC
- **Logging of messages** to a window, a text file, or both
- **Error handling** that conforms to the standard COM interface

2.1.1 Server Name and Registry

The location of the server is not important, but the Pump Communcation Server must be registered before it can provide service to the client.

The line command for registering is:

```
C:\> Regsvr32 PumpCommServer.dll
```

The line command for un-registering is:

```
C:\> Regsvr32 PumpCommServer.dll -u
```

2.1.2 Using the Pump Communication Server

The typical steps to use the Pump Communication Server are as follows:

1. Select the log method to record communication exchanges.

   There are three ways to log the commands sent to the diluter and the replies received from the diluter:

   – Create a log file to save logged information to a text file.

   The client program can create a log file handle by means of the WIN32 API function `CreateFile`. The log file is in text format so any text editor, e.g. WordPad or NotePad, can be used to access it.

   Call `PumpSetLogFileID` to pass the log file handle to the server.

   – Create a log window to display the logged message.

   The client program can create a log window to display the logged message if you desire. Use the log window created by the communication server to display the logged information.

As long as the server is active, you can create or destroy the log window created inside the communication server using `PumpEnableLogWindow`.

The client program can share the same log file or log window. You can use `PumpAddLogMsg` to write useful information (apart from the data written to the file by the server) into the log file or log window.

2 Open the COM port.

To find the available COM ports, call `PumpDetectComm`. This is optional; it limits the choice of COM ports offered to you.

Call `PumpInitComm` to open the designated COM port.

The COM port is initialized 9600 baud or 38400 baud (this value can be set via the `DefaultBaudRate` property), 8 data bits, 1 stop bit, and no parity.

3 After the port is open, use any of the communication functions (`PumpSendCommand`, `PumpSendNoWait`, `PumpGetLastAnswer`, `PumpGetLastError`, or `PumpWaitForDevice`).

4 Close the log file.

Befre exiting, the client program must close the log file by calling the WIN32 API function `CloseHandle`, if it is created as listed in Step 1.

To close the COM port, the client program can either call `PumpExitComm` or you can simply rely upon the server to close the port automatically on exit.

2.1.3 Multiple Addressing

The Pump Communication Server allows sending commands to multiple devices at one time (up to 15 devices at one time). In order to send the same command to more than one diluter, use the values in Table 2-1 in the `byDevNo` parameter.

Table 2-1 *byDevNo Parameter Values*

Address	Dulters affected (on address...)
16	0 and 1
18	2 and 3
20	4 and 5
22	6 and 7
24	8 and 9
26	10 and 11
28	12 and 13
30	14 (one diluter only)
32	Diluters 0-3
36	Diluters 4-7

Address	Dulters affected (on address...)
40	Diluters 8-11
44	Diluters 12-14
46	All diluters

Multiple addressing cannot be used to determine device status. Therefore, you can use it with `PumpSendNoWait`, and then separately query the status of each pump, if necessary. Using multiple addressing with other commands will raise an error (invalid parameter).

2.2 Warnings

WARNING! *Read the safety notes and communication instructions in your device manuals before operating your devices.*

WARNING! *Operate your devices with caution. Risks to users, property and the environment can arise when devices are used carelessly or improperly.*

2.3 Pump Communication Server Interface

This section contains descriptions, syntax, and parameter information for properties and methods used by the Pump Communication Server.

2.3.1 Communication Server Properties

Use the following list to locate individual properties:

`BaudRate`

Represents the baud rate the communication server uses. Only two types of baud rate (9600 and 38400) can be selected. These two types of baud rate are defined in an `enum` type named `EbaudRate`.

VB	`Property BaudRate As EbaudRate`

C++	`get_BaudRate(EbaudRate *pVal);`
	`put_BaudRate(EbaudRate newVal);`

C#	`PUMPCOMMSERVERLib.EBaudRate BaudRate`
Parameter Type	Type enum (Application defined type, EbaudRate)
Input Parameters	None
Returned Parameters	None

Notes

None.

`EnableLog`

Allows the client application to turn the communication logging on or off.

VB	`Property EnableLog As Boolean`

C++	`get_EnableLog(VARIANT_BOOL *pVal);`
	`put_EnableLog(VARIANT_BOOL newVal);`

C#	`bool EnableLog`
Parameter Type	None
Input Parameters	None
Returned Parameters	None

Notes

None.

`LogComPort`

Represents whether Com Port information will be added to each log entry. If the parameter is set to True, then the COM port number will appear in the log.

VB	`Property LogComPort As Boolean`

C++	`get_LogComPort(VARIANT_BOOL *pVal);`
	`put_LogComPort(VARIANT_BOOL newVal);`

C#	`bool LogComPort`
Parameter Type	None
Input Parameters	None
Returned Parameters	None

Notes

None.

CommandRetryCount

Represents how many times to resend the command in case of communication failure. The value must be between 0 and 7, with a default value of 0.

VB	`Property CommandRetryCount As Integer`

C++	`get_CommandRetryCount(short *pVal);`
	`put_CommandRetryCount(short newVal);`

C#	`short CommandRetryCount`
Parameter Type	Integer
Input Parameters	None
Returned Parameters	None

Notes

A `CommandRetryCount` value of 0 means the command is sent only once and there is no retry in case of failure.

CommandAckTimeOut

Represents the number of the acknowledge timeout increment (in 50 ms increments). The value must be between 1 and 40, with a default value of 4.

VB	`Property CommandAckTimeOut As Long`

C++	`get_CommandAckTimeOut(long *pVal);`
	`put_CommandAckTimeOut(long newVal);`

C#	`int CommandAckTimeout`
Parameter Type	Long
Input Parameters	None
Returned Parameters	None

Notes

Since each timeout increment is 50 ms, a `CommandAckTimeOut` value of 10 means a total of 500 ms (10 * 50=500) timeout period.

2.3.2 Communication Server Methods

Use the following list to locate individual methods:

`PumpInitComm`

Opens the designated COM port.

VB	`Sub PumpInitComm(byPortNumber As Byte)`
C++	`HRESULT PumpInitComm(BYTE byPortNumber)`
C#	`void PumpInitComm(byte byPortNumber)`
Input Parameters	• `byPortNumber`: COM port number (1, 2, 3... 255)
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate the open status: • S_OK: the port open successfully • Anything other than S_OK: see [create cross-reference]Appendix A, Error Handling.

Notes

Data bits, stop bits, and parity settings are fixed and non-selectable.

A handle of the designated COM port is produced inside the server if the open function is successful. PumpGetCommID can retrieve the handle. The handle is closed by PumpExitComm or automatically closed when the server is unloaded.

A VB client can access the COM errors using the `Err` object.

PumpGetCommID

Gets the handle of the opened COM port. The communication server gets the COM port handle with a long integer. For C++, cast it as `HANDLE` before using it in the client. Inside the server, the communication server initialized the handle of the COM port as “0”. It is assigned a new value after `PumpInitComm` if the COM port is successfully opened.

VB	`Function PumpGetCommID() As Long`
C++	`HRESULT PumpGetCommID(long *plCommHandle)`
C#	`int PumpGetCommID()`
Input Parameters	None
Returned Parameters	The handle of the COM port: • Handle of the COM port if the COM port is opened successfully. • “0” if the COM port fails to open.
Return Value	Always S_OK (0)

Notes

None.

`PumpExitComm`

Closes the COM port if the COM port has been opened. The COM port is closed automatically when the server is unloaded; it is not necessary to call this function from the client program.

VB	`Sub PumpExitComm()`
C++	`HRESULT PumpExitComm()`
C#	`void PumpExitComm()`
Input Parameters	None
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

None.

`PumpDetectComm`

Determines the status of the designated COM port. This is a quick, easy way to determine whether or not a COM port is available.

VB	`Sub PumpDetectComm(byPortNumber As Byte)`
C++	`HRESULT PumpDetectComm(BYTE byPortNumber)`
C#	`void PumpDetectComm(byte byPortNumber)`
Input Parameters	• `byPortNumber`: COM port number (1, 2, 3... 255)
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate the detecting status. One of: • S_OK: the port open successfully • Anything other than S_OK: see [create cross-reference]Appendix A, Error Handling.

Notes

If access is denied, then the designated COM port has been opened before.

A VB client can access the COM errors using the `Err` object.

`PumpSendNoWait`

Sends a command to the diluter and returns without waiting for the reply. The maximum length of the command string is 255 characters.

VB	`Sub PumpSendNoWait(bstrCommand As String, byDevNo As Byte)`
C++	`HRESULT PumpSendNoWait(BSTR bstrCommand, BYTE byDevNo)`
C#	`void PumpSendNoWait(string bstrCommand, byte byDevNo)`
Input Parameters	• `bstrCommand`: Command string (BSTR string) • `byDevNo`: diluter address. For single diluter control—0, 1... 14. For multiple diluter control (multiple addressing)—16, 18, 20, 22, 24, 26, 28, 30, 32, 36, 40, 44, 46. See Section 2.1.3, ”Multiple Addressing”, on page 2-2 for more information.
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate the sending status: • `S_OK`: the command has been sent successfully • Anything other than `S_OK`: see [create cross-reference]Appendix A, Error Handling.

Notes

`PumpSendNoWait` only sends the commands to the diluter; it does not wait for the reply. Normally, using `PumpGetLastAnswer` after using `PumpSendNoWait` returns the answer from the diluter. `PumpSendCommand` can be used instead of `PumpSendNoWait` if the automatic reply from the device is desired.

A VB client can access the error using the `Err` object.

`PumpGetLastAnswer`

Gets the answer string from the designated diluter. The answer string is an in/out parameter with a maximum length of 255 characters.

VB	`Sub PumpGetLastAnswer(byDevNo As Byte, pAnswer As String)`
C++	`HRESULT PumpGetLastAnswer(BYTE byDevNo, BSTR *bstrAnswer)`

C#	`void PumpGetLastAnswer(byte byDevNo, ref string bstrAnswer)`
Input Parameters	• `byDevNo`: Diluter (0, 1... 14) • `pAnswer`: Pointer to answer BSTR string
Returned Parameters	Answer string from the designated diluter
Return Value	Always S_OK (0)

Notes

None

PumpGetLastError

Gets the diluter error information.

VB	`Sub PumpGetLastAnswer(byDevNo As Byte)`
C++	`HRESULT PumpGetLastError (BYTE byDevNo)`
C#	`Sub PumpGetLastError`
Input Parameters	• `byDevNo`: Diluter (0, 1... 14)
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate the status: • S_OK: the port opened successfully • Anything other than S_OK: see [create cross-reference]Appendix A, Error Handling.

Notes

Non-C++ users should note that this function does not return an error number. If there is an error, the function will raise an Err object that has to be handled using the standard VB “On Error GoTo...” syntax. See the VB documentation for more information.

PumpSendCommand

Sends a command to the designated diluter to get the reply message from the diluter and the diluter error detail. The answer string is an in/out parameter. The maximum length of both the command and answer strings is 255 characters.

VB	`Sub PumpSendCommand(pCommand As String, byDevNo As Byte, pAnswer As String)`
C++	`HRESULT PumpSendCommand(BSTR bstrCommand, BYTE byDevNo, BSTR *bstrAnswer)`
C#	`void PumpSendCommand(string pCommand, byte byDevNo, ref string bstrAnswer)`
Input Parameters	• `pCommand`: Command BSTR string • `byDevNo`: Diluter (0, 1... 14) • `bstrAnswer`: Pointer to answer string (BSTR string)
Returned Parameters	Diluter answer string
Return Value	An HRESULT type of long integer to indicate the status: • S_OK: the command has been sent successfully • Anything other than S_OK: see [create cross-reference]Appendix A, Error Handling.

Notes

`PumpSendCommand` combines the actions of `PumpSendNoWait`, `PumpGetLastAnswer`, and `PumpGetLastError`.

A VB client can access the error using the `Err` object.

Multiple addressing cannot be used with `PumpSendCommand`. See Section 2.1.3, ”Multiple Addressing”, on page 2-2 for more information.

PumpWaitForDevice

Waits for the designated diluter to become ready before leaving the procedure.

VB	`Sub PumpWaitForDevice(byDevNo As Byte)`
C++	`HRESULT PumpWaitForDevice(BYTE byDevNo)`
C#	`void PumpWaitForDevice(byte devNo)`
Input Parameters	• `byDevNo`: Diluter (0, 1... 14)
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

None

PumpWaitForAll

Waits for all of the diluters to become inactive before leaving the procedure.

VB	`Sub PumpWaitForAll()`
C++	`HRESULT PumpWaitForAll()`
C#	`void PumpWaitForAll()`
Input Parameters	None
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

None

PumpCheckDevStatus

Checks to see whether the diluter is still executing the previous command, or if it has finished execution. The client can use this method to check whether or not the command sent to the diluter has completed.

VB	`Function PumpCheckDevStatus(byDevNo As Byte) As Long`

C++	`HRESULT PumpCheckDevStatus(BYTE byDevNo,`
	`long *iStatus)`

C#	`int PumpCheckDevStatus(byte byDevNo)`
Input Parameters	• `byDevNo`: Diluter (0, 1... 14)
Returned Parameters	Status flag to check if the device is busy or not: • -1: COM error • 0: Busy • 1: Diluter ready • 2: Time out error
Return Value	Always S_OK (0)

Notes

None

PumpEnableLogWindow

Creates or destroys the log window. You can create a log window inside the server, which monitors the communication between the computer and the diluter.

The client calls this function to create/destroy the log window. The log window is a modeless window.

VB	`Sub PumpEnableLogWindow(byEnable As Byte, pWndParent As Long)`
C++	`HRESULT PumpEnableLogWindow(BYTE byEnable, long pWndParent)`

C#	`void PumpEnableLogWindow(byte byEnable,`
	`int pWndParent)`

Input Parameters	• `byEnable`: A byte type of integer to control whether the log window displays or not: - 1: Display the log window - 0: Destroy the log window, if it exists • `pWndParent`: Parent window handle
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate tif the log window has been created: • `S_OK`: the log window has been opened successfully • Anything other than `S_OK`: see [create cross-reference]Appendix A, Error Handling.

Notes

The formats of a message sent to the diluter or a reply received from the diluter are:

- Command sent to the diluter: `> [Device number], [Command string]`

  For example, `1, RV` means that the diluter number is `1`, and the command is `RV`.

- Answer from the diluter: `- | * [Device number], [Error number], [Answer string from the diluter, if any]`

  where `-` indicates that there is no error and `*` indicates that there is an error.

  For example, `- 1, 0, XL3000-V4.20-07/90` means that the diluter number is `1`, the error number is `0` (no error), and the answer string is “`XL3000-V4.20-07/90`”.

  For example, `* 1, 10` means that the diluter number is `1` and the error number is `10`.

The handle of the log window can be retrieved using `PumpGetLogWindowID`.

A VB client can access the error using the `Err` object.

PumpGetLogWindowID

Gets the log window handle. The communication server outputs the log window handle as a long integer.

VB	`Function PumpGetLogWindowID() As Long`
C++	`HRESULT PumpGetLogWindowID(long *hWnd)`
C#	`int PumpGetLogWindowID()`
Input Parameters	None
Returned Parameters	The log window handle, as a long integer.
Return Value	Always S_OK (0)

Notes

For C++, if you want the client to use the log window handle, change the long integer to `HWND`.

PumpSetLogFileID

Passes the log file handle from the client program to the server. Use the log file to debug the program or procedure. Inside the server, if the client program uses this function to create the log file handle and pass it into the server, the communication server writes the send command, reply message, and some other actions and results into a log file automatically.

VB	`Sub PumpSetLogFileID(lLogFileID As Long)`
C++	`HRESULT PumpSetLogFileID(long lLogFileID)`
C#	`void PumpSetLogFileID(int lLogFileID)`
Input Parameters	• `lLogFileID`: Handle of log file
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

For example, you can use the WIN32 function `CreateFile` to create a handle to the log file: `hLogFile = CreateFile("Log.txt", GENERIC_WRITE, FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL)`.

The log file must be opened in the client program. You can use the same log file for the client program and for the server. `PumpAddLogMsg` can be used to write additional information into the log file. The log file must be closed by the client program after using it. You can use the WIN32 function `CloseHandle` to close it. The log file is in ASCII text format, so you can use standard software to access it.

The log formats of the commands and answers are documented in ”PumpEnableLogWindow” on page 2-12.

PumpGetLogFileID

Gets the log file handle used in the server. Normally this function is not needed because the log file handle is created in the client program and passed into the server.

VB	`Function PumpGetLogFileID() As Long`
C++	`HRESULT PumpGetLogFileID(long *pLLogFileID)`
C#	`int PumpGetLogFileID()`
Input Parameters	None
Returned Parameters	Handle of log file
Return Value	Always S_OK (0)

Notes

None

PumpSetLogWnd

Passes a user-created log window handle to the server. The user should create a log window with a control of type `ListBox` and pass the `ListBox` control window handle to the Pump Communication Server using this method.

VB	`Sub PumpSetLogWnd(lLogHWND As Long)`
C++	`HRESULT PumpSetLogWnd(long lLogHWND)`
C#	`void PumpSetLogWndInt lLogHWND)`
Input Parameters	• `lLogHWND`: The log window handle, as a long integer
Returned Parameters	None
Return Value	Always S_OK (0)

Notes

In the server, the send command, reply message, and some other actions and results are written into the log window automatically. The client can use `PumpAddLogMsg` to log additional information into the log window.

See ”PumpEnableLogWindow” on page 2-12 for the log format of the communication commands and answers.

PumpAddLogMsg

Writes one line of text into the log file or log window if either one exists. The client application can use this function to write a line of text into the log file or log window.

VB	`Sub PumpAddLogMsg(bstrLogMsg As String, byLevel As Byte, [varWhich])`
C++	`HRESULT PumpAddLogMsg(BSTR bstrLogMsg, BYTE byLevel, VARIANT varWhich)`
C#	`void PumpAddLogMsg(string bstrLogMsg, byte byLevel, object varWhich)`
Input Parameters	• `bstrLogMsg`: The string that the communication server writes to the log file or log window • `byLevel`: The integer that determines the indent level: - 1—Indent six spaces - 2—Indent twelve spaces - 0 or other integer—No indent • `varWhich`: An optional parameter to guide the writing: - 1—Write the string only to the log window (including the window created by the server) - 2—Write the string only to the log file, if there is a log file - 0 or other integer—Write the string to the log window and the log window(s), if any
Returned Parameters	None
Return Value	An HRESULT type of long integer to indicate the writing result: • `S_OK`: the command has been executed successfully • Anything other than `S_OK`: see [create cross-reference]Appendix A, Error Handling.

Notes

The maximum string length of one line written into the log file is 512 characters.

A VB client can access the error using the `Err` object.

PumpGetServerVersion

Gets the version of the communication server. This function allows the client program to determine the revision number, which may imply whether a given feature or function is available. The version number is 1/100 of the integer value returned (e.g., “100” means “version 1.00”). It is not necessary to get the version number in this way, because the COM server has also included the version information itself.

VB	`Function PumpGetServerVersion() As Byte`
C++	`HRESULT PumpGetServerVersion(Byte *pByVersion)`
C#	`byte PumpGetServerVersion()`
Input Parameters	None
Returned Parameters	A three-digit integer for the server version
Return Value	Always S_OK (0)

Notes

This function is obsolete, and is only included for backward compatibility. We discourage the use of this function. Please use other standard ways to obtain a DLL version.

In future releases, this function might be removed.

This page is left intentionally blank.

3 Code Examples

3.1 Overview

The following code examples illustrate the use of the RSP Communication Server and the Pump Communication Server in a Microsoft .NET, Visual Basic 6.0, or Visual C++ environment.

Before using the RSP Communication Server and Pump Communication Server, they must be registered using the regsvr32 command. See ”Server Name and Registry” on page 1-1 and ”Server Name and Registry” on page 2-1 for instructions on registering the RSP Communication Server and Pump Communication Server, respectively.

Note: *The code examples use RSP Communication Server methods; use of Pump Communication Server methods is identical.*

3.2 Examples in a Microsoft .NET Environment

Perform the following tasks to use the RSP Communication Server in a Microsoft .NET environment:

1 Create a .NET project (that is, C#) using Visual Studio .NET.

2 Add a reference to the RSP Communication Server by doing the following:

- **a** Right-click on References in the Solution View.
- **b** Select Add reference.
- **c** Select the COM tab.
- **d** Select RSPCommServer 1.0 Type Library.
- **e** Click OK.

3 Create an instance of the RSP Communication Server in your class:

```
public RSPCOMMSERVERLib.RSPCommClass rspServer = new
RSPCOMMSERVERLib.RSPCommClass();
```

4 Use the new instance in your class. For example:

```
//specify baud rate
rspServer.BaudRate = RSPCOMMSERVERLib.RspBaudRate.RSP9600;
//open COM port 1
rspServer.RSPInitComm(1);
//send a command to a device, and receive the answer
//from the device
string answer = "";
```

```
rspServer.RspSendCommand("RV", 1, 8, ref answer);
```

For more details, refer to the Visual Studio .NET documentation and the sample code provided with your CD.

3.3 Examples in a Visual Basic 6.0 Environment

Perform the following tasks to use the RSP Communication Server in a Visual Basic 6.0 environment:

1. Create a new project.
2. Add a reference to "RSPCommServer 1.0 Type Library" using the Project > References menu.
3. Create a new instance of the RSP Communication Server:

   ```
   Dim rspServer As New RSPCOMMSERVERLib.RSPComm
   ```

4. Use the new instance:

   ```
   'specify the baud rate
   rspServer.BaudRate = RSP9600
   'open the COM port
   rspServer.RSPInitComm 1
   'send a command to a device
   Dim answer as String
   rspServer.RSPSendCommand "RV", 1, 8, answer
   ```

For more details, refer to the Visual Basic 6.0 documentation and the sample code provided with your CD.

3.4 Examples in a Visual C++ Environment

Perform the following tasks to use the RSP Communication Server in a Visual C++ environment:

1. Create a new C++ project.
2. Select a new MFC application type.
3. In the wizard, specify a dialog-based application.
4. In the stdafx.h file, add a line to import the RSP Communication Server library:

   ```
   #import "C:\Code\samples\MFCSample\RSPCommServer.DLL"
   no_namespace, raw_interfaces_only
   ```

   This will generate the header files necessary to use the server from Visual C++.
5. Create an instance of the server in your class:

   ```
   ComPtr<IRSPComm> rspServer;
   ```

6 Initialize the COM system and the server instance in your class constructor:

```
//initializes the COM system
::CoInitialize(NULL);
rspServerCoCreateInstance(__uuidof(RSPComm));
```

7 Uninitialize the COM system in your class destructor:

```
//uninitializes the COM system
rspServer = NULL;
::CoUninitialize();
```

8 Use the RSP Communication Server:

```
USES_CONVERSION;
//initialize the server
rspServer->RSPInitComm(3);
//send a "PI" command to arm 1, device 8
rspServer->RSPSendNoWait(A2OLE("PI"), 1, 8);
//waits for the device to complete the operation
rspServer->RSPWaitForAll();
```

For more details, refer to the Visual C++ documentation and the sample code provided with your CD.

This page is left intentionally blank.

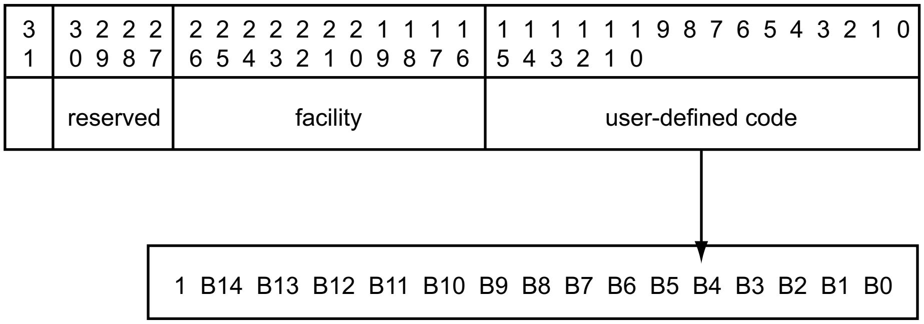
A Error Handling

A.1 Overview

The COM protocol requires that all exported functions (methods) return an HRESULT data type. This data type permits the server to define its own error code. Combining the use of HRESULT with support for the interface `IsupportErrorInfo`, the server can take advantage of a rich COM error-handling model that allows the server to report detailed error information. The Pump Communication Server and RSP Communication Server use this built-in COM error handling functionality.

Fig. A-1 shows the HRESULT structure.

Fig. A-1 *HRESULT Structure*



3 1	3 2 2 2 0 9 8 7	2 2 2 2 2 2 2 1 1 1 1 6 5 4 3 2 1 0 9 8 7 6	1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0 5 4 3 2 1 0
	reserved	facility	user-defined code

1 B14 B13 B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0

HRESULT is a 32-bit value with four fields encoded in the value; the high 16 bits are pre-defined by COM, and the low 16 bits are defined and used by the server.

The four fields are defined as follows:

- The first field, which is the highest bit, indicates whether the return value represents succes or failure. If set to 0, the value indicates succes; if set to 1, it indicates failure.
- The second field is reserved by Microsoft; the field length varies according to platform.
- The third field, the facility field of FACILITY_ITF, indicates the identity of the communication server. Refer to Microsoft's MSDN documentation on COM for additional information.
- The last field is a number that represents the error or warning by the server. The last field is defined by the user.

A.1.1 VB Client Programs

To retrieve error information from a VB client program, use the `Err` object. The property `Number` retrieves the HRESULT value, `Description` retrieves the error description string, and `Source` retrieves the server information.

A.1.2 VC Client Programs Using the RSP Communication Server

To invoke RSP Communication Server error information from a VC client program, follow this general procedure:

1. Retrieve the interface `ISupportErrorInfo` pointer using `QueryInterface`.
2. Retrieve the interface `IErrorInfo` pointer by calling `ISupportErrorInfo::InterfaceSupportsErrorInfo` and `GetErrorInfo`.
3. After retriecing the interface `IErrorInfo` pointer, use `IErrorInfo:GetDescription` or `IErrorInfo::GetSource` to retrieve the detailed error description or server information.

The following code fragment illustrates these points:

```
// Get error support interface pointer
HRESULT hr;
ISupportErrorInfo *pSupportEI = 0;
hr=pMyInterface->QueryInterface(IID_ISupportErrorInfo,
    (void**)&pSupportEI);
if (SUCCEEDED(hr))
{
    hr = pSupportEI->InterfaceSupportsErrorInfo(IID_IMyInterface);
    if(SUCCEEDED(hr))
    {
        IErrorInfo* pEI=0;
        if(SUCCEEDED(GetErrorInfo(0,&pEI)))
        {
            //get error details and handle the error
            USES_CONVERSION;    //in order to transfer BSTR string
            BSTR bstrDescription,bstrSource;

            pEI->GetDescription(&bstrDescription);
            pEI->GetSource(&bstrSource);

            cout << OLE2T(bstrDescription) << endl;
            cout << OLE2T(bstrSource) << endl;

            ::SysFreeString(bstrDescription);
            ::SysFreeString(bstrSource);

            pEI->Release();
        }
    }
    m_pSuportEI->Release();
}
```

Note: *The client can also use only the returned HRESULT value instead of invoking the error details via the above code. The client can also use _com_error to catch the error. For details, see MSDN's online Help.*

A.1.3 VC Client Programs Using the Pump Communication Server

To retrieve Pump Communication Server error information from a VC client program, use the catch mechanism to catch an error of type _com_error.

A.2 Errors in the RSP Communication Server

Table A-1 shows the definition of the low 16 bits of HRESULT for errors in the RSP Communication Server.

Tab. A-1 *Definition of Low 16 Bits of HRESULT for RSP Communication Server*

Error Type	B15	B14	B13	B12	B11-B0	Description
RSP Command Error	1	0 or 1	0	0—left arm 1—right arm	B11-B7: Device number B6-B0: RSP error code	Example: HRESULT=0x80048385 Lower 16 bits: 1000 0011 1000 0101 dev=7 error code=7
COM Port Error	1	0 or 1	1	1—COM port error	B11-B4: COM port number B3-B0: Error code	Example: HRESULT=0x8004B044 Lower 16 bits: 1011 0000 0100 0100 port=4 error code=4
General	1	0 or 1	1	0—general	B11-B0: Error code	Example: HRESULT=0x8004A003 Lower 16 bits: 1010 0000 0000 0011 error code=3

Note: *The RSP error code is defined according to the RSP Communication protocol. See the Operator's Manual for your robotic hardware.*

Table A-2 shows the errors defined in the RSP Communication Server. Note that both hexidecimal and binary numbers are used to express the HRESULT value. The numbers inside the parentheses are binary.

Tab. A-2 *Errors Defined for the Pump Communication Server*

Error Type	HRESULT Value (in Hex or Binary)	Error Description	Function that may cause the error	Note
Robot	8004(1?0Y)(XXXX)(X000)1	Initialization error	`RSPGetLastError` `RSPSendCommand`	The “Y” bit gives the arm number. The five “X” bits are used to give the device number. The error number is the same as the robotic instrument error definition
	8004(1?0Y)(XXXX)(X000)2	Invalid command		
	8004(1?0Y)(XXXX)(X000)3	Invalid operand		
	8004(1?0Y)(XXXX)(X000)4	Invalid command squence		
	8004(1?0Y)(XXXX)(X000)5	Device not implemented		
	8004(1?0Y)(XXXX)(X000)6	Timeout error		
	8004(1?0Y)(XXXX)(X000)7	Device not initialized		
Robot	8004(1?0Y)408	Command overflow	`RSPGetLastError` `RSPSendCommand`	The “Y” bit gives the arm number. The device number is 8.
	8004(1?0Y)409	No liquid detected at ZX-command		
	8004(1?0Y)40A	Z-position overrun		
	8004(1?0Y)40B	Not enough liquid detected at ZX-command		
	8004(1?0Y)40C	No liquid detected at ZZ-command		
	8004(1?0Y)40D	Not enough liquid detected at ZZ-command		
	8004(1?0Y)411	Arm collision avoided		
	8004(1?0Y)412	Clot limit exceeded at clot detection. Refer to the Operator’s Manual for the robotic instrument.		
	8004(1?0Y)413	No exit signal at all clot detection		

Tab. A-2 Errors Defined for the Pump Communication Server (cont.)

Error Type	HRESULT Value (in Hex or Binary)	Error Description	Function that may cause the error	Note
	8004(1?0Y)414	RSP 9000—step loss on X-axis		
	8004(1?0Y)415	RSP 9000—step loss on Y-axis		
	8004(1?0Y)416	RSP 9000—step loss on Z-axis		
	8004(1?0Y)417	RSP 9000—step loss on X-axis of other arm		
	8004(1?0Y)418	ALID pulse timeout		
	8004(1?0Y)419	DiTi—Tip not fetched		
	8004(1?0Y)41A	DiTi—Tip crashed		
	8004(1?0Y)4B	DiTi AC—Tip not clean		
	8004(1?0Y)4C	DiTi—Permanent tip detection		
	8004(1?0Y)4F	EEPROM read.write failure		
Robot	8004(1?0Y)0(X000)9	Plunger overload	`RSPGetLastError` `RSPSendCommand`	The “X” bit gives the device number (1, 2, 3, or 4)
	8004(1?0Y)0(X000)A	Valve blocked		
	8004(1?0Y)0(X000)B	Plunger move not allowed		
	8004(1?0Y)0(X000)F	Command overflow		
COM Port Error	8004(1?11)(XXXX)(XXXX)1	COM# port is not available	`RSPDetectComm` `RSPInitComm`	The 8 “X” bits are used to give the COM port number.
	8004(1?11)(XXXX)(XXXX)2	Access of COM# port is denied		
	8004(1?11)(XXXX)(XXXX)3	COM# port cannot be opened, though it physically exists		
	8004(1?11)(XXXX)(XXXX)4	COM# port is already opened		
	8004(1?11)(XXXX)(XXXX)9	COM# port has unclear error		

Tab. A-2 Errors Defined for the Pump Communication Server (cont.)

Error Type	HRESULT Value (in Hex or Binary)	Error Description	Function that may cause the error	Note
General	8004(1?10)001	Command not sent, `RSPSendNoWait` error	`RSPSendNoWait`	
	8004(1?10)002	Cannot create a log window	`RSPEnableLogWindow`	
	8004(1?10)003	Failed to write a string into the log file	`RSPAddLogMsg`	
	8004(1?10)004	`RSPWaitForDevice` error	`RSPWaitForDevice`	
	8004(1?10)005	No COM port is open	`RSPSendNoWait` `RSPSendCommand`	
	8004(1?10)006	Invalid parameter	`SimuAddArm` `SimuAddDevice`	

A.3 Errors in the Pump Communication Server

Table A-3 shows the definition of the low 16 bits of HRESULT for errors in the Pump Communication Server.

Tab. A-3 *Definition of Low 16 Bits of HRESULT for Pump Communication Server*

Error Type	B15	B14	B13	B12	B11-B0	Description
Pump Command Error	1	1	0	0	B11-B8: Always 0 B7-B4: Device Address B3-B0: Pump Error Code	Example: HRESULT=0x8004C0F7 Lower 16 bits: 1100 0000 1111 0111 address=F error code=7
COM Port Error	1	0	1	1	B11-B4: COM port number B3-B0: Error code	Example: HRESULT=0x8004B044 Lower 16 bits: 1011 0000 0100 0100 port=4 error code=4
General	1	0	1	0	B11-B0: Error code	Example: HRESULT=0x8004A003 Lower 16 bits: 1010 0000 0000 0011 error code=3

Table A-4 shows the errors defined in the Pump Communication Server. Note that both hexidecimal and binary numbers are used to express the HRESULT value. The number inside the parentheses are binary.

Tab. A-4 Errors Defined for the Pump Communication Server

Error Type	HRESULT Value (in Hex or Binary)	Error Description	Function that may cause the error	Note
Pump Command Error	8004(1100)(0000)(XXXX)1	Initialization error	`PumpGetLastError` `PumpSendCommand`	The 4 “X” bits are used to give the pump address.
	8004(1100)(0000)(XXXX)2	Invalid command		
	8004(1100)(0000)(XXXX)3	Invalid operand		
	8004(1100)(0000)(XXXX)4	Invalid command squence		
	8004(1100)(0000)(XXXX)5	Fluid detection		
	8004(1100)(0000)(XXXX)6	Device not implemented		
	8004(1100)(0000)(XXXX)7	Device not initialized		
	8004(1100)(0000)(XXXX)9	Plunger overload		
	8004(1100)(0000)(XXXX)A	Valve overload		
	8004(1100)(0000)(XXXX)B	Plunger move not allowed		
	8004(1100)(0000)(XXXX)F	Command overflow		
COM Port Error	8004(1011)(XXXX)(XXXX)1	COM# port is not available	`PumpDetectComm` `PumpInitComm`	The 8 “X” bits are used to give the COM port number.
	8004(1011)(XXXX)(XXXX)2	Access of COM# port is denied		
	8004(1011)(XXXX)(XXXX)3	COM# port cannot be opened, though it physically exists		
	8004(1011)(XXXX)(XXXX)4	COM# port is already opened		
	8004(1011)(XXXX)(XXXX)9	COM# port has unclear error		

Tab. A-4 Errors Defined for the Pump Communication Server (cont.)

Error Type	HRESULT Value (in Hex or Binary)	Error Description	Function that may cause the error	Note
General	8004(1010)001	Command not sent	`PumpWaitForAll`	
	8004(1010)002	Cannot create a log window	`PumpEnableLogWindow`	
	8004(1010)003	Failed to write a string into the log file	`PumpAddLogMsg`	
	8004(1010)004	`PumpWaitForDevice` error	`PumpWaitForDevice`	
	8004(1010)005	No COM port is open	`PumpSendNoWait` `PumpSendCommand`	

	8004(1010)006	Invalid parameter	`PumpSetCommandRetry`
			`PumpSendNoWait`
			`PumpSendCommand`	