pipe plugin for PowerPro: version .66 18 February 2008

A plugin to connect PowerPro installations across network, receive input from redirected console programme output, and connect to named pipes generally

by
Alan Campbell

This plugin creates named pipes. Named pipes are a remote communication mechanism, which will connect your workstation to other workstations on your LAN (but probably not to the WAN: see Section 11.1 **"Security and Authentication"). Your system should be firewalled to prevent baddies from talking to your pipes.**

1.0 Overview

This PowerPro plugin allows communication:

- between Powerpro installations on different computers on the same network. Communication is via named pipes.
- from command-line programmes that can redirect their (generally stdout) output; you can redirect to e.g. \\.\pipe\Powerpro.
- with any named pipe accessible on your LAN, though I've yet to find a use for this. I gather some programmes expose named pipes, e.g. some serious database servers. Wikipedia says "Windows NT's entire NT Domain protocol suite of services are implemented as DCE/RPC services over Named Pipes, as are the Exchange 5.5 Administrative applications."
- from and to most of the major scripting languages with a *nix tradition (Tcl, Python, Perl) support named pipes, usually through the appropriate win32 package.

1.1 This Document

There are two version of this document, with the same content. There's an RTF file, which looks nice in Word but is something like 200k in size; and there's a compiled help (CHM) document, which is much smaller if somewhat uglier.

In my experiments I've found the RTF file doesn't display correctly in anything but Word (not Keynote, even Wordpad: you've think Microsoft could at least get their rtf engines consistent). So if you don’t have Word, better use the chm file.

Both documents have extensive hyperlinks. The table of contents at the front of each document is a set of them.

The chm file has no index.

1.2 What’s New In This Version

- fixed bug that crashed Powerpro when long strings sent to listening pipe
- bit of a crashing problem on unload, fixed I think

2.0 Requirements

According to http://world.std.com/~jmhart/pipenp.htm , "Windows 95/98 is limited to being a named pipe client; it can not be a server".

So named pipes can't be created on win 9x boxes. So if you execute the listen_on service, you better do it on a NT/2000/XP machine.

You should be able to send messages to a named pipe from win 9x.

Requires Powerpro version 3.4 or later. Test scripts require at least 3.8.15. Edit them to get rid of ?c…c syntax to use with previous versions. They work in standard configuration.

If you want to send or receive binary data via pipes, you'll need the handles version of the binary plugin. Get it here (a file of the form binaryWithHandlesPlugin*.zip):
http://groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons/0_Funny_Strings/

If you want to send or receive unicode strings via pipes, you'll need the handles version of the binary plugin. Get it at the url above. You need at least unicode plugin version 0.65.

If you want to receive data into a vector, you'll need to use the vec plugin (part of the Powerpro distribution).

Those of you who are tetchy about MFC will be happy to know that, unlike the registry plugins, pipe.dll *doesn't* require MFC support (e.g. MFC42.DLL).

Tested on W2000 sp4.

2.1 Related plugins

If you pass a handle to a vector in, the pipe plugin accesses the vec plugin.

I'm working on a spread plugin, which will allow connection to anything connected to a spread server. Spread is "A Reliable Multicast and Group Communication Toolkit" available at http://www.spread.org/. Among other things, using spread would mean you can communicate with other Powerpro installations on your LAN and, if you set security settings appropriately, across the internet.

3.0 File list

plugins\pipe.dll
doc\pipePluginReadme.rtf
doc\pipePluginReadme.chm
scripts\pipePluginDemoScriptListenAndSendToSelf.powerpro
scripts\pipePluginDemoScriptListenToDOS.powerpro
scripts\pipePluginDemoScriptPipeVs2ppTiming
scripts\pipePluginDemoScriptListen.powerpro
scripts\pipePluginDemoScriptToOther.powerpro
scripts\pipe.ini

4.0 Installation

Copy pipe.dll from pipePluginX.XX.zip archive into your PowerPro directory, or into its Plugins subfolder. If you want to provide an initial configuration of the plugin (see Section 9.2), edit pipe.ini and put it in the folder pointed to by pprofolder; or add its edited contents to plugins.ini in the same folder.

All other files can go wherever you want.

5.0 Uninstall

Remove all files listed in the above section ("3.0 File list") from wherever they went..

6.0 Acknowledgements

Me me me me.

7.0 Testing: the Demo Scripts

I've provided the following demo scripts:

pipePluginDemoScriptListenAndSendToSelf.powerpro:

Creates pipes and send to them on the same Powerpro installation. A pointless exercise (why send data to yourself?) you illustrates how to listen on and send to pipes, with and without authorisation, in Powerpro.

pipePluginDemoScriptListenToDOS.powerpro:

Demonstrates how to redirect output from consol programmes to a Powerpro pipe. Most of the ways Powerpro pipes can be set up to listen are illustrated here.

Normally one would be redirecting output from interesting console programmes that can get information not easily available within Powerpro, these illustrations use standard win32 console commands like vol and echo.

pipePluginDemoScriptPipeVs2ppTiming:

Compares how long it takes to receive redirected console programme output using pipes versus 2pp.exe. Runs the same command through a pipe a hundred times, then through 2pp.exe 100 times, and reports timings.

On my machine the former takes about four seconds, the latter about eight.

pipePluginDemoScriptListen.powerpro:

No edits required. Run this script on the workstation you want to send a message to *before* running **pipePluginDemoScriptToOther.powerpro**. (Or: just execute **pipe.listen_to** via the ARB, tiny command box, or any method you prefer.)

pipePluginDemoScriptToOther.powerpro:

Pick a workstation on the LAN you want to send a message to (which may be the workstation you're running on) in the dialog thrown up at the beginning of the script. The workstation you pick must be running **pipePluginDemoScriptListen.powerpro**.

To run the scripts:

Either:

- Put them in your <PowerPro configuration>\scripts directory
- Run scripts with a command (menu item?) in PowerPro like

  *Script RunFile pipePluginDemoScriptListenAndSendToSelf *or*
  .pipePluginDemoScriptListenAndSendToSelf.powerpro

Or:

- Put the scripts anywhere and double click on it (as long as PowerPro is already running). This didn't work on my machine until I manually set up an association with **.powerpro**, but if your Powerpro installation went correctly it should work for you.

The scripts will output both to the Debug window and (if you remove one comment in each script) a log file. By default that file is "pipePluginDemoOutput.log" in same directory as your pcf file. You can edit the script to point it another path/file if you wish: it's the first line of code in the script, and assigns a path/file name to the variable **pipe_test_logfile**. If you assign the null string to **pipe_test_logfile**, no log file will be created.

None of the test scripts use the evaluate-expression operator "&" so are not dependent on your choice for it. It uses the ?c…c syntax to avoid problems with your declared escape character, so you should have no problems whether that's ' or \.

There are further comments on how the scripts work embedded in the scripts itself.

8.0 List of Services And General Notes on Usage

Ensure pipe.dll is in your PowerPro installation directory, or in the plugins subfolder thereof.

There are numerous services in this plugin. They are (aliases are in italics):

service	description	section
create	creates a pipe object	10.1
destroy, *release*	destroys a pipe object	10.2
listen_on, *listen*, *on* listen_off, *off*	begin listening/cease listening for incoming messages	10.3 10.4
send_to, *send*, *to*	send a message	10.5
get_computer_names	get names of all workstations on the LAN	10.6
get_ip_addr	get IP address of this workstation	10.7
returns_values returns_status returns_nothing	determine what if anything all other services return via the calling form returnval = pipe.service	10.8
error_dialog_on error_dialog_off	turn PowerPro script error dialog on and off	10.9
config	set location of configuration ini file	10.10
version	returns the plugin version number as four digit number, last two to be taken as right of decimal.	
unload	remove plugin from memory	

These are described below, in section 10 and its subsections.

9.0 Writing scripts using the pipe plugin

Call the appropriate plugin service as follows:

retval = **pipe.<service>(**arguments if any**)**

Results, if any, are generally available as the returned value from the above expression, though there are alternatives**.**

The essential services are pipe.listen_on and pipe.send_to**.** If you want a machine to receive messages from others, you must run pipe.listen_on on that machine. That machine will then listen for messages sent to it until pipe.listen_off is called.

You send a message to a listening machine with pipe.send_to**.**

After you are finished using the pipe plugin in your script, you can if you wish unload it with

pipe.unload

It's probably best done if you do not foresee the plugin being used again for a while.

On the other hand, be very careful not to unload the plugin when a plugin pipe may be in the middle of receiving data. In particular you should probably never unload from a script which is triggered in response to data coming in (i.e. a script set with the <command_to_call> parameter of listen_to).

By default, data, if there is any to be returned, comes back as the result of the service expression (returnval in the above line). By default again, all services set a result string in variable **pipe_status**.

If you wish different behaviour, call either returns_status**()** or returns_nothing**()**.

If you call returns_status**()**, all subsequent service calls will return a result string as the result of the call. **pipe_status** won't be altered, and data will be returned via a variable (generally **pipe_value**, but depends on the service: see details below).

Use returns_values**()** to return to default behaviour.

If you unload the plugin, it's behaviour returns to the default in any subsequent call, i.e. further pipe service calls will return values.

There are other services which affect the behaviour of the pipe plugin (see sections 10.3-10.7). You can also customise the behaviour of plugin services by providing a configuration ini file (see section 9.2).

If you do an *Exec ChangeConfiguration, and the new and old pcf files are in different folders, you should unload this plugin before using any of its services with the new configuration. (only necessary if you use relative paths to specify the location of ini files).

9.1 Interfacing With Command Line stdout

You can redirect stdout of a command line programme in a console to a pipe with e.g.

```
echo win.debug(?.hello.) > \\MACHINENAME\pipe\<pipe_name>
```

A pipe created with Powerpro using

```
pipe.listen_on("powerpro", 0)
```

on the same machine can be reached with e.g.

```
echo win.debug(?.hello.) > \\.\pipe\powerpro
```

If you don’t like the syntax, you could define an environment variable e.g.

```
set ppPipe=\\.\pipe\powerpro
```

You can then do

```
echo win.debug(?.hello.) > %ppPipe
```

You can run a console programme (and send it's output to a Powerpro pipe) from within Powerpro with something like:

```
file.runwait(0, env("COMSPEC"), ?"/c blah.exe > \\.\pipe\powerpro", "", "hide")
```

An alternative way of communicating with Powerpro, you could use the command line utilities 2pp.exe, 2ppvar.exe and stdin2pp.exe written by Julien Pierrehumbert and available at

http://tech.groups.yahoo.com/group/power-pro/files/Plug-ins_and_add-ons/0_Julien%27s_Plugins/.

Example of usage:

```
echo win.debug(?.hello.) | 2pp.exe
```

If you have 2pp.exe you can run **pipePluginDemoScriptPipeVs2ppTiming.powerpro** to compare which technique is faster. It looks like if you close a pipe and reopen around every invocation of %COMSPEC, using a pipe takes longer than 2pp (50% longer?). If you open a listening pipe and listen on it continuously over a 100 invocations of %COMSPEC, takes less time (30% less?).

ASAIK there’s no way to update a static variable with Julien’s utilities.

listen_on parameter *<leave_crlf>*: Most console programmes appended \r\n to each lines output. Generally it's useful to have them stripped of data sent to Powerpro, so leaving it's default value of 0 us probably good.

listen_on parameter *<break_msgs>*: Sometimes works sensibly, sometimes not, depending on the DOS command. "vol" seems to send out two messages, one for volume name, one for serial number. But "type" seems to break a file into a series of messages, neither of fixed length nor representing text lines) listen_off parameter <*waitForDisconnectedPipe*> is often useful if your console programme is piping a substantrial amount of data to your pipe. It means disconnection won;'t take effect until the sending process (the command line programme) disconnects.

Here are some examples of using a pipe to pick up redirected data from console programmes:

Invoke a function in same script file with incoming message as an argument; only safe and workable if redirected data is a single, fairly short (< 40—450 characters) line:

```
  pipe.listen_on("powerpro", 0, "", "", cbx("@usingFuncArg", "#"))
  file.runwait(0, env("COMSPEC"), ?"/c echo some text > \\.\pipe\Powerpro", "", "hide")
  pipe.listen_off
```

Execute a Powerpro command; with incoming message as an argument: only safe and workable if redirected data is a single, fairly short (< 40—450 characters) line:

```
  pipe.listen_on("powerpro", 0, "", "", "win.debug(#)")
  file.runwait(0, env("COMSPEC"), ?"/c echo some text > \\.\pipe\Powerpro", "", "hide")
  pipe.listen_off
```

Invoke a function in same script file:

```
  pipe.listen_on("powerpro", 0, "st_sPipeData", scriptname, cbx("@usingStaticVar"))
  file.runwait(0, env("COMSPEC"), ?"/c echo some text > \\.\pipe\Powerpro", "", "hide")
  pipe.listen_off
```

echo a Powerpro command:

```
  pipe.listen_on("powerpro", 0)
  file.runwait(0, env("COMSPEC"), ?"/c echo win.debug(?#hi#) > \\.\pipe\Powerpro", "", ;;
     "hide")
  pipe.listen_off
```

Put incoming data in a global variable, and pick up its value as soon as console command completes:

```
global g_sPipeData
pipe.listen_on("powerpro", 0, "g_sPipeData")
file.runwait(0, env("COMSPEC"), ?"/c echo some text  > \\.\pipe\Powerpro", "", "hide")
win.debug(g_sPipeData)
pipe.listen_off
```

Same as above: but "vol" returns two messages; only the first ends up in the global:

```
global g_sPipeData
pipe.listen_on("powerpro", 0, "g_sPipeData")
file.runwait(0, env("COMSPEC"), ?"/c vol c: > \\.\pipe\Powerpro", "", "hide")
win.debug(g_sPipeData)
pipe.listen_off
```

Return results in a global vector:

```
global g_hVec = vec.create(20, 1)
pipe.listen_on("powerpro", 0, g_hVec)
file.runwait(0, env("COMSPEC"), ?"/c vol c: > \\.\pipe\Powerpro", "", "hide")
win.debug("following vector filled from dos via pipe using vector in global variable")
local n = g_hVec.length
for (local i = 0; i lt n; i++)
 win.debug("vector element " ++ i ++ ": " ++ g_hVec[i])
endfor
pipe.listen_off
```

9.2 The Configuration ini File

On the first call to any pipe service, the plugin checks for a file called pipe.ini in the folder pointed to by the Powerpro variable pprofolder (usually the folder in which the currently active pcf file is found). It then looks for a [pipeConfig] section in that ini file. If that's not found, it looks for the same section in the file plugins.ini in the same folder. If either are found, it looks for the following values in the section:

key	possible values	default value	can also change using service	meaning
messageIdPrefix	string (max 30 chars)	none of your business	none	see Section 11.1
allowedSources	string (max 255 chars)	null string	none	see Section 11.1
messageIdPrefixRequired	d a n	d	none	see Section 11.1
insertionMarkerAll	string (max 10 chars)	#	listen_on	see listen_on, *<command_to_call>* arg
insertionMarkerLines	string (max 10 chars)	£	listen_on	see listen_on, *<command_to_call>* arg
raiseErrors	0 1 y n t f [*]	1	error_dialog_on error_dialog_off	determines if errors in syntax format, service arguments etc cause Powerpro to raise the script error dialog.
whatToReturn	s d n [†]	d	returns_values returns_status returns_nothing	for services which return data, determines what is returned by the service is the data, the status (whether there was an error or not), or nothing.

[*] Only the first non-whitespace character of the key's value is checked, no matter how many there are. So values "yes", "no", "true" or "false" will work. As usual the ini value is case-insensitive, so "Y" and "FALSE" are also valid.

[†] Only the first non-whitespace character of the key's value is checked, so "status", "data" and "none" will work.

If no pipe.ini or plugin.ini files are found with a [pipeConfig] section, the pipe plugin will initially just use the compiled-in, default values (specified in the third column above) to configure itself.

If an ini file with a [pipeConfig] section is processed, and there's a possible value that could be in ini but isn't, the pipe plugin reverts to the default value for that value.

Once an ini file with a [pipeConfig] section is processed, its values stay in force until the pipe plugin unloaded or the config service is run.

If the first service called is one which itself changes the configuration (e.g. **use_long_vars**, error_dialog_on), the configuration ini file will be found and evaluated *before* the service is applied.

If you want to change your initial configuration, you can use the ini plugin to do it. E.g.:

ini.set(pprofolder ++ "plugins.ini", "pipeConfig", "messageIdPrefix", "HITHERE")

After making your changes, either:

- unload the pipe plugin. The next time you use one of its services, the new configuration will kick in. Or

- run pipe.config**(<path to configuration ini file>)**.

If you do an *Exec ChangeConfiguration; and if the new and old pcf files are in different folders; and if there's a pipe.ini or plugins.ini file in the new pprofolder, the configuration it specifies won't take effect until you take one of the steps above.

10.0 The Services

Details of specific services follow.

The next sections describe services that affect the behaviour of all others.

10.1 create

pipe.create([<*default_pipe_name*>]**)**

used in demo script: **pipePluginDemoScriptListenAndSendToSelf.powerpro**

Creates a handle to a pipe object and returns a handle to it. You can (but don't have to) invoke the listen_on, listen_off and send_to services with a handle returned by **create** as their first arguments. You can also use the object.service notation, e.g.

```
local hPipe = pipe.create
hPipe.listen_on("funny",…..)
….
hPipe.listen_off
```

<*default_pipe_name*> is the name used by listen_on and send_to services if they have no <*pipe_name*> argument. It may be no longer than 255 characters in length.

If you call without any arguments, you'll get the handle to the default pipe (the one for the pipe used if you call listen_on, listen_off or send_to without a handle).

Note that no two pipes concurrently listening can listen on the same <*pipe_name*>. So if you create a pipe handle with the <*default_pipe_name*>, listen_on the default pipe (the one without a handle) with no explicit <*pipe_name*>, and also attempt to listen_on the pipe handle pipe, again with no explicit <*pipe_name*>, you will attempting to listen on two pipes both with the address \\.\pipe\Powerpro, and your second call to listen_on will fail.

You can create up to 20 non-default pipes.

10.2 destroy

pipe.destroy(<*pipe_handle*>**)**

or

pipe_handle.destroy

alias: *release*

used in demo script: **pipePluginDemoScriptListenAndSendToSelf.powerpro**

Deletes a handle returned by the create service. If you attempt to destroy the handle to the default pipe, you'll not get an error message, but nothing will happen: the default pipe andle always exists.

10.3 listen_on

pipe.listen_on([<*pipe_handle*>] [, <*pipe_name*> [, <*authenticate*>
[, <*var_name*> | <*vec_handle*> | <*binary_handle*> | <*unicode_handle*>
[, <*static_var_script*> [, <*command_to_call*>
[, <*leave_crlf*>
[, <*break_msgs*> [, <*markers*>
[, <*insertionMarkType*> [, <*cmd_on_dead_pipe*>
[, <*dead_pipe_folder*>]]]]]]]]]]]]**)**

or

pipe_handle.listen_on([<*pipe_name*> [, <*authenticate*>
[, <*var_name*> | <*vec_handle*> | <*binary_handle*> |
<*unicode_handle*>
[, <*static_var_script*> [, <*command_to_call*>
[, <*leave_crlf*> [, <*break_msgs*> [, <*markers*>
[, <*insertionMarkType*> [, <*cmd_on_dead_pipe*>
[, <*dead_pipe_folder*>]]]]]]]]]]]**)**

aliases: ***listen, on***

used in demo scripts: **pipePluginDemoScriptListenAndSendToSelf.powerpro,
pipePluginDemoScriptListenToDOS.powerpro,
pipePluginDemoScriptPipeVs2ppTiming,
pipePluginDemoScriptListen.powerpro**

Starts listening for messages.

All parameters are optional.

<*pipe_handle*>:	A handle returned by the create service. If you don't supply one (or use one via the handle.listen_on(…) syntax) the default pipe that's always available will be used.
<*pipe_name*>:	If no pipe name is given or it's the null string, but there's a <*pipe_handle*>, the pipe object's default pipe name is used. If there's no <*pipe_handle*> or it's the null string, the pipe created with the name specified by the **defaultPipeName** key in pipe.ini/plugin.ini. This defaults to "Powerpro" if no key is specified. If you attempt to listen on two pipes simultaneously using the same <*pipe_name*>, you'll get an error. <*pipe_name*> may be no longer than 255 characters in length.

<*authenticate*>: 0 or 1, or a string beginning "n" or "y". See below. Messages may be accepted in *authenticated message format* (See Section 11.1, "Security and Authentication"). If you omit the <*authenticate*> parameter:

- If you don't specify a non-null <*pipe_name*> or explicitly specify one identical to the default, authenticated message format will be used.

- If you specify any other explicit <*pipe_name*>, it won't be expected: the incoming data will be assumed to be all message.

You can override these assumptions by explicitly specifying the <*authenticate*> parameter as 0 or a string beginning "n" (no authenticated message format); or 1 or a string beginning "y" (use authenticated message format).

Normally you would avoid authenticated message format if messages were coming from some source other then Powerpro, so you'll want <*authenticate*> set to 0.

If a message is received in authenticated message format, the variable <*pipe_name*> is set to the name of the originating workstation; therefore the variable <*pipe_name*> should probably be declared global before **pipe.listen_on** is called.

<*var_name*>: If present and not the null string, received data will be placed in this variable. It should be declared before data is received. It should be declared global, unless <*static_var_script*> is present.

<*vec_handle*>: If the variable named by <*var_name*> contains a handle to a PowerPro vector, or in place of <*var_name*> you supply PowerPro vector handle, data received by a listening pipe will be used to fill that vector, one line per vector item. It's your responsibility to make sure the vector accessed by <*vec_handle*> is big enough for the number of lines of data, or can grow to be big enough.

<*binary_handle*>: If the variable named by <*var_name*> contains a handle to a binary block generated by the handles version of the binary plugin; or in place of <*var_name*> you supply such a handle, data received by a listening pipe will be used to fill that binary block. If you pass a <*binary_handle*>, the <*leave_crlf*> parameter (see below) is ignored.

Typically you'll be listening for a binary block if you expect data to be send by pipe.send_to, also using a <*binary_handle*>.

Used in pipePluginDemoScriptListenAndSendToSelf.powerpro

<unicode_handle>: If the variable named by *<var_name>* contains a handle to a unicode string generated by the unicode plugin; or in place of *<var_name>* you supply such a handle, data received by a listening pipe will be used to modify that string. You need at least unicode plugin version 0.65. If you pass a handle, it must *not* be to the null unicode string. If you pass a variable name containing a handle, it can be the handle to the null string; it will be replaced with another handle when data is received.

If you pass a *<unicode_handle>*, the *<leave_crlf>* parameter (see below) is ignored.

Typically you'll be listening for a unicode string if you expect data to be send by pipe.send_to, also using a *<unicode_handle>*.

Used in pipePluginDemoScriptListenAndSendToSelf.powerpro

<static_var_script>: The name (not path to) the script file in which *<var_name>* is a *static* variable. To be useful *<command_to_call>* (see below) should be a script in the same file

<command_to_call>: This will be executed when data is received by the pipe. Typically a script, though it could be e.g. "win.debug(?#….#)" or "*message data received #", or even "myProg.exe".

If *<command_to_call>* is a script, *don't* within that script call **pipe.listen_off** or **pipe.unload** in that script, or you'll end up stuck in a loop or crashing.

Providing an empty string as *<command_to_call>* leaves it unspecified but allows successive arguments to be appended.

If *<command_to_call>* contains a substring corresponding to the parameter *<insertionMark>* parameter with *<insertionMarkType>* beginning "a", *or*, if *<markers>* parameter is absent, corresponding to the configuration ini key **insertionMarkerAll** (*or*, if that's absent, "#"), the whole of the received data will be inserted as a literal string enclosed in the esc() function at that point in *<command_to_call>*. If *<insertionMarkAll>* appears to already be enclosed in ?c…c delimiters, they'll be removed.

If you've supplied a *<vec_handle>* or *<binary_handle>* and you include an "a"-type insertion mark in *<command_to_call>*, the handle will be inserted in the command in place of the insertion mark.

Be careful using an insertion mark without an handle, i.e. inserting raw message data. The command sent will be <*command_to_call*>, plus the incoming data as a literal string, plus a call to the Powerpro esc() function, plus the starting folder path. Powerpro can only cope with commands of 530 characters or less; if your incoming message is longer than about 400 characters, you're in danger of hitting that limit.

One alternative is to include instead a substring in <*command_to_call*> corresponding to the <*insertionMark*> parameter with <*insertionMarkType*> beginning "l", *or*, if the <*markers*> parameter is absent or null , corresponding to the configuration ini key **insertionMarkerLines** (*or*, if that's absent, "£"), the received data will be broken into lines (delimited by \n or \r, and each line inserted as a literal string enclosed in the esc() function at that point in <*command_to_call*>. <*command_to_call*> will be called once for each line. (if the insertion marker appears to already be enclosed in ?c…c delimiters, they'll be removed.). For an alternative, see <*break_msgs*> parameter below.

If <*command_to_call*> contains the string <*folder_marker*> ("¦" by default), all text after that string is taken as the folder in which is <*command_to_call*> started. That generally only has effect if <*command_to_call*> is an executable, batch file, link, etc; in particular it will have no effect if <*command_to_call*> is a PowerPro script or command.

<*leave_crlf*>: When piping in from a dos command line, \r\n is appended to the end of each redirected line. The pipe plugin will strip these characters off the endo of each message, unless you include this argument and its value is 1 (valid values are 0, 1, or any string beginning "y" or "n").

<*break_msgs*>: Normally when you receive input from an external pipe, all messages coming through are accumulated until the external pipe disconnects, then the resulting concatenated string sent on to PowerPro as you specified in other arguments to listen_on.

When you redirect data from a console programme into a pipe, that data sometimes arrives as a set of messages, one message per line of output (that's not universal; nor do the messages necessarily make sense).

You can request messages to be dealt with separately, so that each message is separately forwarded to PowerPro by specifying "1" or any string beginning "y" for <*break_msgs*>. If absent, <*break_msgs*> is taken to be "0". Valid values are 0, 1, or any string beginning "y" or "n").

<*markers*>: May contain up to two white-space separated strings:

<*insertionMark*> <*folder_marker*>

<*insertionMark*> indicates where to insert incoming data in a <*command_to_call*>; see above. Can be any string up to ten characters in length. You'll probably want a single character, and of course you'll want a string that can’t legitimately occur in <*command_to_call*>.

<*folder_marker*> (“¦” by default) indicates, in <*command_to_call*>, the beginning of the path to the folder in which <*command_to_call*> will be executed. Can be any string, but again you'll probably want a single character, and of course you'll want a string that can’t legitimately occur in <*command_to_call*>.

<*insertionMarkType*>: Begins “a” if previous argument <*markers*> is to be replaced with all data, or “l” if it’s to be replaced with one line of data at a time. Taken to be “a” if absent.

<dead_pipe_cmd>: If a sending process makes a connection to a listening PowerPro pipe, and then drops the connection, this script, if specified, will fire. It's not always clear whether it's a good idea to use it or not. The idea is to have a way of knowing when a sender completes data transmission, but some senders may drop a pipe and re-establish connection in the process of sending the "same" data, and others may never relinquish connection. Use at your own risk.

If *<dead_pipe_cmd>* contains the string *<folder_marker>* ("¦" by default), all text after that string is taken as the folder in which is *<dead_pipe_cmd>* started. That generally only has effect if *<dead_pipe_cmd>* is an executable, batch file, link, etc; in particular it will have no effect if *<dead_pipe_cmd>* is a PowerPro script or command.

<dead_pipe_folder>: the folder from which *<dead_pipe_cmd>* is executed; ignored if *<dead_pipe_cmd>* is unspecified, or if it's a script call or

On successful return, **pipe_status** contains "OK", otherwise returns an error message beginning "ERROR:".

10.3.1 How To Listen

When a message is received, what happens obviously depends on the parameters you used with listen_on. Usually the object is to get Powerpro to respond when data arrives, and for the response to have the message data available to work on.

A simple solution, if there's a small amount of data and a simple response, is to execute a Powerpro command with inserted data e.g.:

```
pipe.listen_on("powerpro", 0, "", "", "*message data received: #")
```

That'll work only if there's a small amount of data.

If you expect to receive largish messages (greater than 400-450 characters), you're best to put data in a variable using *<var_name>*, *<vec_handle>* or *<binary_handle>*. There's no limit on the size of strings that can be put in vector elements, variables, or binary blocks.

My preferred approach is to set a static variable in *<var_name>*, set *<static_var_script>* to the name of the file in which the variable is declared, and *<command_to_call>* to a function in the same file.

I've found that the static variable and *<command_to_call>* can be the same file as the one from which the listening pipe is launched. Seems to create no conflict.

10.3.2 Listening for Non-Powerpro Sources

The pipe plugin listens on a message-type pipe. Message-type pipes treat the bytes written in each write operation to the pipe as a message unit. That should work fine with any application that writes to pipes using the ::CreateFile function.

If a sending application writes to a pipe and then hangs, waiting for Powerpro to read the pipe, won't work. The pipe plugin as listener assumes it's the server , not the sender.

Non-Powerpro sources don't know about the authentication rules used by Powerpro pipes (See Section 11.1, "Security and Authentication"), so you probably want to listen on a pipe without them.

See here for a more detailed discussion of sending data from console applications to Powerpro pipes.

10.4 listen_off

pipe.listen_off([<*pipe_handle*>])

or

pipe_handle.**listen_off()**

alias: ***off***

used in demo scripts: **pipePluginDemoScriptListenAndSendToSelf.powerpro,**
pipePluginDemoScriptListenToDOS.powerpro,
pipePluginDemoScriptPipeVs2ppTiming,
pipePluginDemoScriptListen.powerpro

Terminates listening and closes the listening pipe.

<*pipe_handle*>: A handle returned by the create service.

On success, returns "OK", otherwise returns an error message beginning "ERROR:".

Be careful not to call **listen_off** on a pipe currently receiving data. At best you'll just lose data; at worst you get an error message.

In some situations using the **listen_on** parameter <*cmd_on_dead_pipe*> may be the soluction: either make <*cmd_on_dead_pipe*> "pipe_listen_off", or an invocation of a script that calls **pipe_listen_off**. That may not always work, or do what you want it to; some processes break connection with a pipe and then re-establish it immediately to send more data.

10.5 send_to

pipe.send_to([<*pipe_handle*>,] <*binary_handle*> | <*unicode_handle*> | <*data*>
[, <*machine_name*> [, <*pipe_name*> [,<*authenticate*>]]]**)**

or

pipe_handle.send_to(<*binary_handle*> | <*unicode_handle*> | <*data*>
[, <*machine_name*> [, <*pipe_name*> [,<*authenticate*>]]]**)**

aliases: ***send, to***

used in demo scripts: **pipePluginDemoScriptListenAndSendToSelf.powerpro,**
pipePluginDemoScriptListenToDOS.powerpro,
pipePluginDemoScriptPipeVs2ppTiming,
pipePluginDemoScriptToOther.powerpro

Pipes a message to <*pipe_name*> on <*machine_name*>.

If no pipe name is given, but there's a <*pipe_handle*>, the pipe object's default pipe name is used. If there's no <*pipe_handle*>, the message is sent to the pipe whose name is specified by the **defaultPipeName** key in pipe.ini/plugin.ini. This defaults to "Powerpro" if no key is specified..

If <*machine_name*> is absent or the null string, the name of the machine on which this copy of Powerpro is running used.

The receiving pipe most commonly will be one on which the pipe plugin is listening, i.e. for which a pipe.listen_on has previously been issued. There's nothing to prevent you from sending a message to yourself, i.e. for the sending plugin and receiving pipe to be the same. (I do it for demo purposes in the demo script **pipePluginDemoScriptListenAndSendToSelf.powerpro**). Normally not much point in that; I think you'd normally be sending data from powerpro on one workstation to powerpro running on another.

<*pipe_handle*>:	A handle returned by the create service.
<*binary_handle*>:	If this parameter is a handle to a binary block generated by the handles version of the binary plugin; *or* the name of a variable containing such a handle, the binary data in the byteblock will be sent. The receiving pipe should of course also have been set up with a handle to a binary block. Used in pipePluginDemoScriptListenAndSendToSelf.powerpro
<*unicode_handle*>:	If this parameter is a handle to a unicode string generated by the the unicode plugin; *or* the name of a variable containing such a handle, the unicode string will be sent. You need at least unicode plugin version 0.65. Used in pipePluginDemoScriptListenAndSendToSelf.powerpro

<*data*>:	If this parameter is neither a <*unicode_handle*> nor a <*binary_handle*> the string you supply as data will be sent.
<*machine_name*>:	If absent or the null string, the target pipe is assumed to be on the same machine as instance of Powerpro is running..
<*authenticate*>:	Messages may be sent in *authenticated message format* (See Section 11.1, "Security and Authentication"). If you omit the <*authenticate*> parameter:

- If you don't use any pipename or explicitly specify one identical to the default, authenticated message format will be used.

- If you specify any other explicit pipename, authenticated message format won't be used: the provided <*data*> will be sent as-is.

You can override these assumptions by explicitly specifying the <*authenticate*> parameter as 0 (no authenticated message format) or 1 (use authenticated message format).

Normally you would avoid authenticated message format if messages were being sent to a destination other then a Powerpro installation (since recxipient wouldn't recognise the authentication string.

On success, returns "OK", otherwise returns an error message beginning "ERROR:".

10.5 to_console

pipe.to_console([<*pipe_handle*>,] <*console_command*> [, <*executable*>
[, <*current_directory*>]]**)**

or

pipe_handle.to_console(<*console_command*> [, <*executable*>
[, <*current_directory*>]]**)**

aliases: ***console***

used in demo scripts: **pipePluginDemoScriptListenToDOS.powerpro,**

The <*pipe_handle*> or default pipe must already be listening before you call **to_console**.

First time you call, the console process is created. <*executable*> and <*current_directory*> on that call. <*executable*> assumed to be cmd.exe if omitted or null string.

Call with no args except <*pipe_handle*> destroys the console process.

10.6 get_computer_names

pipe.get_computer_names()

used in demo script: **pipePluginDemoScriptToOther.powerpro**

On success, returns (or sets **pipe_value** to) a list of all available workstation names on the LAN, separated by spaces. You can use the word() function to retrieve individual computer names.

On success, sets **pipe_status** (or returns) "OK", otherwise returns an error message beginning "ERROR:".

10.7 get_ip_addr

pipe.get_ip_addr(<*machine_name*>)

On success, returns (or sets **pipe_value** to) the IP address of the named machine.

On success, sets **pipe_status** (or returns) "OK", otherwise returns an error message beginning "ERROR:".

10.8 returns_values(), returns_status(), returns_nothing()

These services determine what if anything the **enum_xxx** and **get** services return as retval in:

returnval = pipe.service(….)

By default, data, if there is any to be returned, comes back as the result of the service expression (returnval in the above line). By default again, all services set a result string in variable pipe_status.

If you wish different behaviour, call either **returns_status()** or **returns_nothing()**.

If you call **returns_status()**, all subsequent service calls will return a result string as the result of the call. **pipe_status** won't be altered, and data will be returned via a variable (generally **pipe_value**, but depends on the service: see details below).

The listen_on, listen_off and send_to services returns no data, only a result status, so they will *always* return that result status as the return value of the service, regardless of which of the returns_xxx services has been called.

Use **returns_values()** to return to default behaviour.

If you unload the plugin, it's behaviour returns to the default in any subsequent call, i.e. further pipe service calls will return values.

10.9 error_dialog_on(), error_dialog_off()

Used in all demo scripts

Some pipe services can result in errors. For instance, you might try to use a duplicate pipe name when listening on one. In addition to such errors setting a PowerPro variable or returning a value with a status message the prefixed "ERROR:" (see previous section); they will also trigger the standard PowerPro script error dialog, allowing you to cancel all running scripts.

Under some conditions you might no want pipe plugin errors to be treated as scripting errors, and you would therefore not want to see the PowerPro script error dialog. If that's what you want (maybe because you're testing for the presence or absence of a section/key pair), invoke **pipe.error_dialog_off()**, or make sure the **raiseErrors** key in the configuration ini file is false or 0. Invoke **pipe.error_dialog_on()** to turn error dialogs back on after you turn them off.

If you unload the plugin, it's behaviour returns to the default in any subsequent call, i.e. errors on further pipe service will cause the error dialog to pop up.

Invoking **pipe.error_dialog_off()** only affects error dialogs appearing when a pipe service call goes wrong. The normal Powerpro error dialog will appear if anything else goes wrong in Powerpro.

10.10 config

pipe.config(name_of_pipe_file)

specifies a configuration ini file, with format, section and keys as described in section 9.2

The ini file can either be given as an absolute path, or a path relative to the folder returned by the pprofolder variable (generally the folder containing the currently running Powerpro configuration file).

Returns "OK" if file and found and there are no keys with illegal values, or a message beginning "ERROR:" if there is one.

If you unload and reload a plugin, it's behaviour returns to the default or to that defined by a default config ini file (see section 9.2).

11.1 Security and Authentication

Pipes created by pipe plugin have unlimited access: they can be accessed by anybody.

Turns out this is the norm for named pipes on Windows. The issue is discussed at

http://www.beyondlogic.org/solutions/pipesec/pipesec.htm

where you can download the "Win32 Pipe Security Editor" which will allow you to see what pipes are running on your machine, and examining their access rights (all, on my machine, without exception, wide open).

So, for the moment, named pipes created by the pipe plugin have unlimited access rights.

I've included some rudimentary authentication via settings in pipe.ini or plugins.ini. If you decide to require it, messages are sent prefixed by a series of characters to specify in the ini key **messageIdPrefix** (which defaults to…well, never mind, you don't need to know). This prefix is immediately followed by the name of the originating workstation, followed by a delimiting character.

The ini key **messageIdPrefixRequired** determines whether authentication is required. It may begin:

first letter	**require authenticated message format?**
d	only for the default pipe name, typically the one used to send messages between Powerpro installations, typically "Powerpro"
a	apply to all pipe names, whether default or not
n	never

Even if authenticated message format isn't required, if an incoming message string conforms to it, it will be assumed that's what it is. So better make sure **messageIdPrefix** is a series of characters that can't possible occur in a "raw" incoming message. In particular it shouldn't look like a Powerpro command.

If you want to send messages between Powerpro installations, both must either use authenticated message format or both not. If they use it, both must use the same **messageIdPrefix**.

You may specify a ini file key **allowedSources**, which is a list of allowed message originators, the names separated by one or more blanks. If **allowedSources** isn't an empty string (which it is, by default, if not set otherwise in pipe.ini/plugins.ini); and if an incoming message is in authenticated message format; then the originating workstation name must be in **allowedSources**. If it isn't, the message is silently dropped.

For further security, if anyone wants it, I think I can add an option for pipes to inherit the access rights of Powerpro.exe or pipe.dll. You would then shut down rights to either of those using the usual file properties dialog, and thereby limit access to the pipes created by pipe.listen_on.

11.2 Handles to Pipes

There's a default pipe over which listen_on and send_to operate. But you can listen on and send to additional pipes if you use the create service, which returns a *handle* to the created pipe. A handle is just a simple string beginning "p\x07" followed by a number from 3000 to 3020.

Once you've got a <*pipe_handle*>, you can use it as the first argument in calls to listen_on, listen_off and send_to (if you call those services without a <*pipe_handle*>, you're using the default pipe). Or you can use the handle.service notation instead:

```
local hPipe = pipe.create(“fruit”)
hPipe.listen_on("banana",…..)
….
hPipe.listen_off

hPipe.release
```

If you have PowerPro version 4.4.05 or later, and *if* a handle to a pipe is held in a *local* variable, there's no need to use release it; it will automatically be released when the variable goes out of scope at the end of the containing routine. You can override that automatic override by using the localcopy service.

Scripting tip: if you're going to use this facility, *don't* unload the pipe plugin at the end of your script. PowerPro will need the plugin dll loaded in memory to do it's thing with local handles.

If you do release a handle held in a local variable, best invalidate it by doing e.g.:

```
myLocalVar = pipe.release(myLocalVar)
```

On the other hand, don't lose a handle to a pipe by e.g. overwriting it without first releasing the underlying object. This is a bad idea.

```
local hPipe = pipe.create("xxx")
hPipe  = ""  ;; oops, lost the handle, and the plot
```

So is this:

```
local hPipe = pipe.create("xxx")
….
hPipe = pipe.create("yyy")
```

because the handle returned by pipe.create, though it will work in context, will be lost after the expression is evaluated, so there'll be no handle to pass to pipe.release

However, I would imagine in most cases you will want a pipe to persist for a period of time, not just for the duration of a script, so you'll normally want to assign pipe handles to statics or globals.

Unloading the pipe plugin will cause all structs and arrays to be released.

11.3 pipe_ Variables

These are the variables set by various services. The equivalent xn variable is given in case you wish to invoke **pipe.use_x_vars().**

pipe_ var	set by
pipe_status	all services
pipe_value	get_computer_names get_ip_addr
pipe_name	set when message received
pipe_msg	may be set when message received

12.0 Restrictions

13.0 Possible Enhancements

At the moment pipes have no security: they can be accessed by anybody. I think I can add an option for pipes to inherit access rights of Powerpro.exe or pipe.dll. You would then shut down rights to either of those using the usual file properties dialog.

14.0 Change History

.66:

.64:

- <*insertionMarker*> parameter of listen_on is now <*markers*>, made up of <*insertionMark*> and <*folder_marker*>.
- <*command_to_call*> parameter of listen_on can now include a <*folder_marker*> followed by an initial folder.
- Corrected handling of text inserted as a literal string in a <*command_to_call*> when a listening pipe receives data. In particular plugin now figures out current PowerPro configuration escape character and uses that as necessary.

.62:

- You can now listen and send on more than one pipe at once; there's a default pipe over which listen_on and send_to operate if you don't create a handle and use it as an argument to or object for those services. Therefore…
- Added optional argument <*pipe_handle*> to services listen_on, listen_off and send_to. Also…

- Added optional further arguments <*var_name*>, <*static_var_script*>, <*command_to_call*>, and <*leave_EOL*>, <*break_msgs*>, <*markers*>, <*insertionMarkType*> and <*cmd_on_dead_pipe*> to the listen_on service….

- …and therefore removed pipe.exe from distribution. It's easier to pipe from stdout in a console command to e.g. \\.\pipe\Powerpro, which will listen as you specified in your call to the listen_on service.

- You can specify a handle to a vector or a binary object in <*var_name*>.

- Documentation now and again named the service send_to as send_to_Powerpro. Oops, sorry.

- Added version service.

- Added instructions on use \\.\pipe\Powerpro to do dos stdout redirection

- Added new configuration ini keys **insertionMarkerLines** and **insertionMarkerAll**.

- Renamed new configuration ini key **powerProFormatRequired** to **messageIdPrefixRequired**

- Added new demo script pipePluginDemoScriptListenToDOS.powerpro.

- renamed all scripts pipePluginTest*.powerpro scripts to pipePluginDemo*.powerpro

.60:

- Fixed problem in pipe.listen_off

- Reduced size of dll

.59:

- Fixed bug in send_to, communicating back to PowerPro

.57:

- Added hyperlinks to documentation, and added doc in HTML help format; tidied up and corrected documentation.

.50: First version