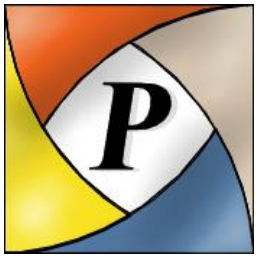
Institute for Energy Technology
OECD Halden Reactor Project



ProcSee

Graphical User Interface Management System

Plugins

Reference Manual

1.2

This document will be subjected to revisions in the future as the development of ProcSee continues. New versions will be issued at new releases of the ProcSee system.

Published by : Institute for Energy Technology, OECD Halden Reactor Project
Date : May 2016
Revision : 1.2

Contents

Introduction

About ProcSee Plugins

The ProcSee Run Time Manager (RTM) can be expanded with plugins. By using plugins, platform specific functionality can be provided without including platform specific code in the RTM. Plugins can also be used to provide functionality that is only relevant for specific applications.

At the moment the plugin interface, i.e. the software interface between any plugin and the RTM, is undocumented and subject to change. However, plugins developed by the ProcSee team will be released as part of ordinary ProcSee releases.

About this manual

This manual describes the ProcSee plugins available at the moment. Additional plugins may be added in later versions.

Using Plugins

To use the functionality provided by a plugin, the plugin must be loaded into the RTM. This is done by adding the plugin to the application or library where the plugin is needed.

On the Microsoft Windows platform, plugins are loaded by right clicking the application or library and selecting **Plugins...** in the context menu. In the window that opens, select the plugin in the list of available plugins, and click on the load button, ref Figure 1. The plugin will then be displayed in the list of loaded plugins.

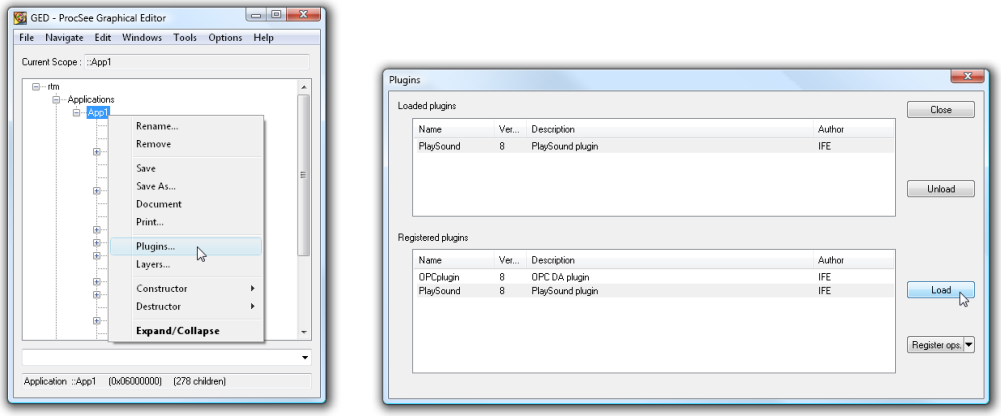


Figure 1 Loading a plugin in an application using GED.

On the other platforms, the addition of plugins must be done by editing the application or library .Tdoc file.

In the .Tdoc file, the use of plugins is indicated by the keyword **plugin** followed by the name of the plugin, ref Figure 2. The plugin must be located in the **plugins/$(ARCH)** directory of the ProcSee installation, and be registered in the **plugins.pprd** file. The plugins.pprd file contains information about the plugins available, and what version of the plugin-interface they use. The version number of the plugin interface of the plugin must match the version number expected by the RTM.

```
application App1
{

   plugin "PlaySound";
   ...
}
```

Figure 2 Including a plugin in an application, .Tdoc syntax.

Plugins are loaded together with an application or library, and normally register functions that can be called directly from pTalk.

Troubleshooting plugin loading

When the ProcSee RTM has a problem loading a plugin there are several messages that can be given, as described in the table below. The origin of these messages are either the pTALK function

loadPlugin("*pluginName*");

or the

plugin "*pluginName*";

statement in an application or library Tdoc file.

RTM message	**Possible action**
The loadPlugin function have been disabled, start rtm with '-x pluginadd' to enable.	To allow: change RTM options, by removing -x nopluginadd, or by adding -x pluginadd.
Loading of plugins have been disabled, start rtm with '-x pluginload' to enable.	To allow: change RTM options, by removing -x nopluginload, or by adding -x pluginload.
Plugin '*plugin*' not found.	Check that the name of the plugin is correct.
Plugin '*plugin*' has directory separators in the plugin name.	To allow: start the RTM with the option -x plugindir. This gives this message as a warning instead.
Plugin '*plugin*' was not registered.	Run `registerPlugin -a` *`plugin`* to register the plugin. To allow unregistered plugins: start the RTM with the option -x pluginnotreg. This gives this message as a warning instead.
Plugin '*plugin*' has changed on disk from the one registered	This happens when the date of the plugin file has changed from the date registered in the plugins.pprd file. Run `registerPlugin -u` *`plugin`* to update the information for the plugin, or run `registerPlugin -u` to update all plugins. To allow changed plugins: start the RTM with the option -x pluginregdiff. This gives this message as a warning instead.
Plugin '*plugin*' has wrong interface version	The plugin is not made for this version of the plugin interface. This can be because the PROCSEE_DIR environment variable is set to another version of ProcSee. ProcSee RTM will crash if plugins with wrong interface version was allowed.

When the program `registerPlugin` is run, it must be in the same directory as the plugins, and for the same ARCH as the plugins, as it has to load the plugins in order to get the information it needs. Run `registerPlugin -?` to list its options.

2

Plugins

OPC Plugin

ProcSeeôs OPC plugin enables the RTM to act as an OPC Data Access client. The plugin uses OPC Data Access 2.0, allowing the RTM to access any server compliant with the OPC Data Access 2.0 specification.

The OPC plugin is available in the Microsoft Windows version only.

The OPC plugin is documented in a separate document: OPCpluginConfig.pdf.

PlaySound Plugin

The PlaySound plugin is a very simple plugin, which makes the Microsoft Windows Multimedia API function PlaySound available to pTalk. This is for playing simple event sounds. At the moment this plugin is available only on Windows, since the PlaySound function is a Windows function.

The PlaySound plugin makes the following two functions available to pTalk:

```
int playSound( char* fileName, char* options=0 );
int stopSound();
```

The **playSound** function plays a **.wav** sound file. The sound-file is played asynchronously by default.

The *fileName* is either an absolute file-name, or a file-name relative to the application that it is called from, or a name of some of the event sounds in Windows, like ñSystemAsteriskò (Check the Windows documentation of PlaySound for details).

The *options* argument can contain:

- **loop** to make the sound play repeatedly. To stop the playback, use the stopSound function or play another sound.
- **sync** to make the playSound function not return until the sound has been played (the return value is then 0 if there was any errors, in other cases the function returns 1).
- **async** to make the playSound return immediately (before the sound is played). This is the default.
- **nodefault** to make the system not play a default sound if the fileName is not found.
- **alias** to only look for system event sounds.
- **filename** to only look for sound-files.

The **stopSound** function stops the currently playing sound.

Example of usage:

```
playSound( "sounds/tada.wav" );  // plays the wave file.
playSound( "SystemAsterisk", "loop" );  // plays the event sound.
stopSound();
```

FilePlugin

The FilePlugin adds file and directory operations to pTALK. It provides read and write operations for (text) files, and functionality to check file and directory permissions.

The FilePlugin reads a configuration file when it is loaded. This configuration file contains settings that can be used to limit the set of files that the plugin can read from or write to. It is possible to configure it to have a smaller set of files that can be written to than the set of files that can be read. More information about this configuration file is given on page 12.

The FilePlugin provides the following pTALK functions:

```
char* ::FilePlugin.read( char* fileName, char* flags=0 );
int ::FilePlugin.write( char* fileName, char* data, char* flags=0 );
int ::FilePlugin.access( char* fileName, char* flags );
```

The **read** function reads a file, and returns the content as a string. It returns 0 if there are any problems reading the file. The filename has to be in the set of files allowed to be read from, as defined by the configuration file. The flags argument can contain the following options:

- **text** to indicate that the file read is a text file. This is the default. All newlines in the file will be converted to the ónô character.
- **binary** to indicate that no changes should be made to the bytes read.

The **write** function writes the content of the data string to the specified file. It returns 0 if it failed to write the file, and a value different from 0 on success. The filename has to be in the set of files allowed to be written to, as defined by the configuration file. The flags argument can contain the following options:

- **text** or **text=sys** to indicate that ónô characters are written as system dependent newlines, i.e. it will write ñ\r\nò for each newline on MS Windows systems, and it will write ñ\nò on unix and linux systems. This option is the default if no text or binary option is specified.
- **text=win** to indicate that ónô characters are written as ñ\r\nò for each newline.
- **text=unix** to indicate that ónô characters are written as ñ\nò for each newline.
- **binary** to indicate that no changes should be made to the bytes written.
- **length=number** to specify the number of bytes to write. If not specified, all of the data argument before the string terminator ó0ô is written to the file.

The **access** function returns a number different from 0 if the access permissions specified in the flags argument is ok, and 0 if the access is not ok. The filename has to be in the set of files allowed to be read from, as defined by the configuration file. The flags argument can contain the following:

- **file** to check if the file exists and is a regular file.
- **dir** to check if the filename specified exists and is a directory.
- **execute** or **x** to check if the file exists and its file mode has the execute bit set. (This option is not useful on MS Windows, as MS Windows does not have an execute bit in the file mode).
- **read** to check if the file exists and is readable.
- **write** to check if the file exists and is writable.
- **exists** to check if the file exists.

- **create** to check if the file can be created. This is checked by checking if the file exists and is writable, or if the file does not exist, by checking that the directory that the file would be placed in exists, and that the permissions on this directory is so that a new file can be added to the directory. The filename only has to be in the set of files that can be read, so the file plugin may not be able to create the file even if this option returns ok.
- **allowWrite** to check if the file exists and is writable, including checking for the file being in the set of writable files as specified in the file plugin configuration file, i.e. the return value indicates if the file plugin can write to the file.
- **allowCreate** to check if the file can be created using the file plugin. This is checked the same way as for the **create** option, but in addition the filename has to be in the set of files that can be written.

FilePlugin configuration file

The default settings for the FilePlugin is to allow all read operations, and disable write operations of certain files, like executable files (*.exe). These settings can be changed by modifying the configuration file the plugin reads at start up. The name of this file is **FilePlugin.conf** and it is located in the same directory as the **FilePlugin.pplg** plugin, i.e. in the **plugins/$ARCH** directory under the ProcSee installation. If a file named **FilePlugin.conf** is found in the application directory, it is used instead, so that application specific settings can be configured.

Modify the contents of this configuration file to change the set of files the plugin can access. The following configuration settings are recognized by the FilePlugin:

- **write** = **true** or **false** # if the plugin is allowed to write to files.
- **overwrite** = **true** or **false** # if the plugin is allowed to write over existing files.
- **extAllow** = list of extensions # extensions that the plugin can access, unless also in extDeny
- **extDeny** = list of extensions # extension the plugin cannot access
- **extAllowRead** = list of extensions # additional extensions the plugin can access when reading
- **extAllowWrite** = list of extensions # extensions the plugin can access when writing (limiting the extensions from extAllow)
- **extDenyWrite** = list of extensions # extensions the plugin cannot access when writing
- **dirAllow** = list of directories # directories that the plugin can access, unless also in dirDeny
- **dirDeny** = list of directories # directories that the plugin cannot access
- **dirAllowRead** = list of directories # additional directories that the plugin can access when reading
- **dirAllowWrite** = list of directories # directories that the plugin can access when writing (limiting the directories from dirAllow)
- **dirDenyWrite** = list of directories # directories that the plugin cannot access when writing

Some of the settings can be specified on separate lines several times in the configuration file, like the é Allow and é Deny settings. The file plugin will concatenate these arguments. The set of files that can be read by the file plugin is specified with the **extAllow**, **extAllowRead**, **extDeny**, **dirAllow**, **dirAllowRead**, and **dirDeny** settings. The combination of the settings **extAllow**, **extDeny**, **extAllowWrite**, **extDenyWrite**, **dirAllow**, **dirDeny**, **dirAllowWrite**, and **dirDenyWrite** gives the set of files the file plugin can write to. To prevent all write operations, set the **write** setting to false. The setting **overwrite** can inhibit the file plugin from overwriting existing files.

Notice that the set of files/directories available for reading includes all files/directories available for writing. A directory specified in the configuration file includes all sub-directories as well. Typically the dirDeny setting is used to deny access to sub-directories specified with the setting dirAllow.

The read function uses the read set of files, and the write function uses the write set of files. The access function uses the read set of files, except it uses the write set of files when the allowWrite or allowCreate options are specified.

The **FilePlugin.conf** is line based where each setting must start on a new line, and the arguments for the setting must follow before a new line break. Empty lines are ignored, and comments must start with the character ó#ô. The arguments for the settings can be a boolean value (i.e. true or false), or a list of text strings separated by comma. The text strings are either file extensions or directories, depending on the setting. The text strings can contain wildcard characters, i.e. * and ?. If the text strings contain space they must be specified in quotes, like ñc:/user settings/ò. When using quotes, use forward slashes ó/ô, or double the back-slashes ô\\ô.

Next follows an example of a FilePlugin configuration file.

```
write=true
overwrite=false
extAllow=*
extDenyWrite=exe
dirAllow=C:/myProj/log
dirAllowRead=C:/myProj/data1, C:/myProj/data2
```

This example results in making the FilePlugin being able to create and write new files, but not to overwrite existing files. It can read files with any extension, but can not write files with the exe extension. It can read files from the directories: C:/myProj/log, C:/myProj/data1, and C:/myProj/data2, and all their subdirectories, but it can only write files to the C:/myProj/log directory and its subdirectories.