# Tutorial - A plugin application

## Table of contents

# 1. Tutorial - A plugin application

In this tutorial we are going to build a plugin application. While the application will contain no functionality itself it can do practically anything since plugins can provide all the functionality it ever needs.

## 2. Setup

Assuming that the plugin system is setup correctly, then the only setup we will do is to create a directory for the turorial.

```
mkdir tutorial-4
cd tutorial-4
```

## 3. The C file

First we'll write the C file for the program. As usual we'll write it and then discuss what's in it afterwards. Put the file in *app.c*. For a complete and powerful application, this is pretty small file.

```
01 #include "libplugin/PluginManager.h"
02
03 int main( unsigned argc, char* argv[] ) {
04     PluginManager_setStandardOptions(
05         "APP_",
06         "-",
07         argc, argv,
08         getenv( "APP_PLUGIN_LOGGING" ) != NULL
09     );
10     PluginManager_addAppVersion( "app", NULL );
11     if( PluginManager_start( )) {
12         argc = PluginManager_argc;
13         argv = PluginManager_argv;
14
15         PluginManager_stop();
16     }
17
18     PluginManager_cleanup();
19 }
20
```

**Line 01** includes the needed header file from the plugin distribution. This contains all of the functions starting with *PluginManager_*.

**Line 04** sets some standard options. The plugin system is very flexible and will allow us, for example, to extend the plugin file format. Generally, however, we need only a few settings to be made for us. This function makes all of the standard options for us.

**Line 05** sets the environment variable prefix. The function will set the plugin path, required plugins and plugin variables from environment variables, *APP_PLUGIN_PATH*, *APP_PLUGINS* and *APP_PLUGIN_VAR*, respectively. By having a prefix, diffferent plugin appications can share the same environment variable namespace.

**Line 06** sets the command line prefix. This is used in the same way as the environment variable prefix except that command line arguments are searched. The command line arguments are, *-plugin-path*, *-plugins* and *-plugin-var* respectively.

**Line 07** passes the command line to be searched for the arguments described above. The function will copy the command line to variables *PluginManager_argc* and *PluginManager_argv*. These copies will have the plugin command line arguments stripped from them. Note that this is why we copy them back in lines 12 and 13.

**Line 08** tells the system that logging should print to the terminal only if environment variable *APP_PLUGIN_LOGGING* is defined. The default error and logging reporting can be completely replaced, but the default just prints to the terminal.

**Line 10** sets the application version which helps the plugin system work out which XML files are plugin files. Each candidate XML file is searched for top level processing instructions of the form:

```
<?application-name version="xxx"?>
```

It hopes to find one with a given *application-name* (which we've set to *app*) that the application has registered with it. Because the second argument is *NULL* the version string has to be the same as the version string of the plugin system (currently *"0.1.0"*). It is possible to have specific version strings as well by passing a conversion function as the second argument.

This version system allows you to build applications that accept only certain versions of plugin file.

Calling *PluginManager_setStandardOptions*, which we do in **line 04** also adds an application version for `<?plugin version="0.1.0"?>`. Our accepted version processing instructions are then:

```
<?plugin version="0.1.0"?>
   --- or ---
<?app version="0.1.0"?>
```

**Line 11** starts the plugin system. Here the plugin path is scanned for plugin files, eager and required plugins are loaded and started. The plugins can do all of their work in their *start* phase.

**Lines 12 and 13** save the command line back. Plugins may have altered the command line, so we should use the altered version. Note that since we aren't doing any work in the application itself we could ignore these two lines.

**Line 15** stops the plugin system. Plugins go through their *stop* and *shutdown* phases.

**Line 18** cleans up the plugin system.

## 4. Building the application

We'll write a little make file to build the library - put this file in *Makefile*:

```
01 PLUGINS_DIR =    path-to-plugins-dir
02 PLUGIN_LIBS =    $(PLUGINS_DIR)/libplugin/libplugin.a -liberty -lxml2
-ldl -lffi
03 LIBXML_DIR =     /usr/include/libxml2
04 INCLUDES = \
05                  -I$(LIBXML_DIR) \
06                  -I$(PLUGINS_DIR)/libplugin/include
07 DEFINES =        -Dbool=int -DTRUE=1 -DFALSE=0
08
09 app :      app.o
10     gcc -rdynamic -o app app.o $(PLUGIN_LIBS)
11
12 %.o :            %.c
13     gcc $(DEFINES) $(INCLUDES) -o $@ $^
14
15 clean :
16     rm *.o app
```

**Line 01** points to the directory where the plugin distribution was unpacked.

**Line 02** gives the libraries and archives we need for the application. We have to include the plugin library, of course, but we also need *libxml2* for XML processing. We have to have *libdl* to handle shared libraries and we need *libffi* to dynamic dispatch.

**Line 03** points to where ever *libxml2* is installed on your system.

**Lines 04 to 06** set up our include directories.

**Line 07** defines a *bool* type.

**Lines 09 to 10** link the application.

**Lines 12 to 13** compile the C file with the appropriate flags.

**Lines 15 to 16** clean up.

If all is well, we should be able to type `make` and everything will build.

## 5. Conclusion

That's it. You could now start writing plugins for this simple application. Perhaps the body of your application would be delivered as plugins. Most likely, however, you will do some work after successfully calling *PluginManager_start*. Perhaps you will take an existing application and wrap it's main function in the code above.

In the next tutorial we will see how to add powerful extensions to an application with minimal effort.