Tutorial - A second plugin (with C code)

Table of contents

1. Tutorial - A second plugin (with C code)

In the last tutorial we saw how to make simple plugins that used extension points from other plugins. In those cases we didn't need to write any C code because the writers of the plugins we were extending parsed our XML specifications for us.

There are times, though, when some C code has to be written. In this tutorial we'll see how to build a new extension point.

2. Setup

You're going to build a plugin file and a shared library. Our new extension point will be a cut down version of the *message* plugin. It will simply print the text contained in any extension when the plugin system starts.

We'll keep some of the setup from the last tutorial as a starting point.

```
mkdir tutorial-3
cp tutorial-2/foo.c tutorial-3/
cd tutorial-3
export GCC_PLUGIN_PATH=.:plugin-dir
```

3. The plugin file

Let's have a look at the plugin file we'll need. We have to tell the plugin system that we have a new extension point and we must let it know what function to call whenever someone extends it. Since the *message* plugin name is already taken, we use *herald* as the id of our plugin. Call this file *plugin.xml*.

```
01 <?xml version="1.0"?>
02
03 <?plugin version="0.1.0"?>
04
05 <plugin id="herald" lazy="true">
06     <library path="${plugin.dir}/plugin.so">
07         <start/>
08     </library>
09
10     <extension-point id="herald.announce-at-start">
11         <extend symbol="extendPrintTextAtStart"/>
12     </extension-point>
13 </plugin>
```

Let's look at the lines in this file which we've not seen before.

Line 01 has a *@lazy* attribute. This tells the system that it shouldn't load this plugin unless some asks for it. Lazy plugins can be asked for by specifying them directly on the command line, by having a <*requires*> element (as we saw in the last tutorial) or by extending an extension point provided by the plugin (there are a couple of ways to do it through the C API, too). So remember that this plugin won't run by default, we have to ask for it with the *GCC_PLUGINS* environment variable or the *-plugins* command line argument.

Line 06 tells the system that the plugin needs a shared library. We'll put our code into this shared library later.

Notice that we used a variable expansion, *${plugin.dir}*, to let the system know the path to the library. The system defines a number of variables for us, this one gives the absolute path of the directory in which the plugin file is found.

Note:

There is a special library *@path* attribute, *path=""* which means to use core application to find symbols in. This is often useful when we need symbols from the core application.

Note:

We can have multiple <*library*> elements at once. The system, when it needs a symbol will search through them in the order they appear in the file. You can also name libraries to disambiguate the symbol search. You can find more details in the plugin file format documentation.

Line 07 says that we would like to run the function with symbol name *Plugin_start* from the library whenever this plugin is loaded and the plugin system successfully starts up. It is in this function that we'll print out our text.

The prototype of the function is `bool Plugin_start( Plugin* plugin )`. We'll see, shortly, how to use that.

Note:

We could have changed the symbol name with a *@symbol* attribute. There are also such 'life cycle' methods for the *setup*, *stop* and *shutdown* phases - for more info, see the plugin file format documents.

Line 10 declares that we have an extension point with the given id, *herald.announce-at-start*. Other plugins will use this id to extend this extension point.

Line 11 tells the system that if someone extends this extension point which function to call. The function will be given the XML specification of the <*extension*> element which extends it. In our case, the function is called *extendPrintTextAtStart*. It's prototype is: `bool extendPrintTextAtStart( ExtensionPoint* self, Plugin* extender, xmlNodePtr specification )`.

Note:

There are more complicated ways to create extension points, through *creators* and *factories*. You can find more details in the plugin file format documentation and the C-API documentation.

4. C code

Now we'll build our C file. We'll call it *plugin.c* in the current directory. To begin with we'll just have a tiny bit in it and then we'll add some more later.

```
01 #include <stdio.h>
02 #include "libplugin/ExtensionPoint.h"
03 #include "libplugin/Plugin.h"
04 #include "libxml/tree.h"
05
06 bool Plugin_start( Plugin* plugin ) {
07     printf( "Our plugin, %s, was started!\n", Plugin_getId( plugin ));
08     return TRUE;
09 }
10
11 bool extendPrintTextAtStart( ExtensionPoint* self, Plugin* extender,
xmlNodePtr specification ) {
12     printf(
13         "Our extension point, %s, was extended by plugin, %s.\n",
14         ExtensionPoint_getId( self ),
15         Plugin_getId( extender )
16     );
17     return TRUE;
18 }
19
```

Line 01 includes the standard header so that we can use *printf*.

Lines 02 and 03 include headers so that we can use *ExtensionPoint* and *Plugin* typedefs and related functions. Note the prefix of *libplugin/*.

Line 04 includes a header from *libxml2*, the library we use to process XML. This gives us the *xmlNodePtr* typedef and will later give us functions over it.

Lines 06 to 07 give us the function that will be called when the plugin system has been successfully initialised. All we do at the moment is print a message with the plugin's id.

We return *TRUE* to indicate that we successfully started our plugin. If we hadn't the plugin system wouldn't continue.

Line 11 is our function which will be called if anyone extends our extension point, *herald.announce-at-start*. The parameters are:

- *self* - the extension point that the system creates for us.
- *extender* - the plugin that extends us.
- *specification* - the XML of the <*extension*> element in the extender plugin.

Lines 12 to 16 just print a little message about who's extending us.

Line 17 returns TRUE because we were successfully extended.

5. Building the shared library

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

```
01 PLUGINS_DIR =    path-to-plugins-dir
02 LIBXML_DIR =     /usr/include/libxml2
03 INCLUDES = \
04                  -I$(LIBXML_DIR) \
05                  -I$(PLUGINS_DIR)/libplugin/include
06 DEFINES =        -Dbool=int -DTRUE=1 -DFALSE=0
07
08 plugin.so :      plugin.o
09     gcc -shared -Wl,-export-dynamic -o plugin.so plugin.o
10
11 %.o :            %.c
12     gcc $(DEFINES) $(INCLUDES) -fPIC -o $@ $^
13
14 clean :
15     rm *.o *.so
```

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

Line 02 points to where ever *libxml2* is installed on your system.

Lines 03 to 05 set up our include directories.

Line 06 defines a *bool* type.

Lines 08 to 09 link the shared library, ensuring that symbols are exported.

Lines 11 to 12 compile the C file with the appropriate flags. Don't forget *-fPIC*.

Lines 14 to 15 clean up.

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

6. Running the plugin

In previous tutorials we've seen how to run plugins. Remember that this plugin is lazy, so

you need to explicitly ask for it.

When you run the plugin you should get the following output:

```
Our plugin, herald, was started!
```

Success, but not very exciting. Let's extend our extension point and see if we get some more. Now you could write a new plugin to do this or you could just get the *herald* plugin to extend itself. For now we'll do that later, but you might try both to see how it goes.

Add the following line into the plugin file:

```
    <extension point"herald.announce-at-start">Hello, World!</extension>
```

Now when we run the thing we should get:

```
Our extension point, herald.announce-at-start, was extended by plugin,
herald.
Our plugin, herald, was started!
```

Note that the extension point was extended before the plugin was started. This is because there are multiple phases plugins go through. The first is the *setup* phase during which time plugins create extensions, extension points and require other plugins to be loaded. The next is the *start* phase, here no more plugins can be loaded, no more extension point created or extended (extension points can be used, though). After the program has run comes the *stop* phase, during which all extension points and plugins still exist, each plugin should make their last use of other plugins in this phase. Finally, comes the *shutdown* phase, plugins can't use other extension points or plugins and should clean themselves up.

7. Parsing the extension

We'll add a little code to remember any text passed in during the extension and print it at start. Change the *plugin.c* file to be:

```
01 #include <stdio.h>
02 #include "libplugin/ExtensionPoint.h"
03 #include "libplugin/Plugin.h"
04 #include "libxml/tree.h"
05
00 static char* text = NULL;
00
06 bool Plugin_start( Plugin* plugin ) {
07     if( text ) {
00         printf( "The message was: %s\n", text );
```

```
00         free( text );
00         text = NULL;
00     } else {
00         printf( "No message\n" );
00     }
08     return TRUE;
09 }
10
11 bool extendPrintTextAtStart( ExtensionPoint* self, Plugin* extender,
xmlNodePtr specification ) {
00     if( text ) free( text );
00     text = ( char* )xmlNodeGetContent( specification );
17     return TRUE;
18 }
19
```

Now we are simply remembering the text passed to the specification and printing it when we are extended. If we run the program now, we get:

```
The message was: Hello, World!
```

8. Where to go from here

We've seen how to build a very simple extension point. From here you can look at the plugin C-API docs and at the libxml2 API docs. Good luck!