

## **Developing *Project54* Applications that Utilize the XML GUI Interface W. Thomas Miller, III**

This document summarizes various issues relative to creating applications which utilize the *Project54* XML GUI interface. Separate documents specifically describe the XML GUI Interface calls and the XML syntax and underlying functionality for the GUI interfaces.

### ***Base Project Files***

Applications which use the new XML GUI interface model require only four standard files to be included in the Visual Studio project: *P54XMLAppBase.cpp*, *P54XMLAppBase.h*, *P54XMLAppBase.def* and *AppID.h*. Applications must also have an *AppCode.cpp* file implementing application specific versions of the two required application procedures:

```
void AppMainThread(void) { ... }
```

```
void AppHandleMessage(LPWSTR source, LPWSTR dest, LPWSTR messageID, LPWSTR  
message_text) { ... }
```

These procedures retain their former meaning. Of course, application projects can have any number of other files containing application specific functionality.

The old *P54AppBase.cpp* file has been replaced by *P54XMLAppBase.cpp*. This file retains all of the former application base functionality (supporting the Application Manager interface) as well as implementing the new XML GUI interface, the *Project54* registry interface (formerly in *RegComlib.lib*), the local dynamic storage implementation (formerly in *Lheap.cpp*), and the *Project54* application feedback model (formerly implemented in *feedback.cpp* or in custom application code). The procedure prototypes for all of these functionalities have been lumped into the single file *P54XMLAppBase.h* which should be referenced through a “#include” statement in application specific code modules.

The old *P54AppBase.def* file has been replaced by *P54XMLAppBase.def* for consistency. However, the contents and purposes of these two files are identical.

The former *AppID.cpp* file has been replaced by the *AppID.h* file. This file still defines the application specific COM registration number (*CLSID\_MsgHandler*). However, it also defines the application’s base registry name (in the string *AppName*), the application’s default messaging name (in the string *self*), and the application’s version specification (in the string *AppVersion*).

## ***Standardized Application Identification***

Applications must support four different forms of identification, all defined in the *AppID.h* file. The values of these identifiers must be set by editing the file. The .h file can then be included in any application specific code file which references them.

The application specific COM registration number (*CLSID\_MsgHandler*) is used by the Application Manager to load the application, and must be unique to the application but should not normally change for new versions of the same application.

The application's base registry name (*AppName*), is needed for registry references implicit in the base code. It should generally be unique to the application and should not normally change for new versions of the same application. The string variable name *AppName* should also be used as the base registry key in any explicit registry references in the application code to read or write application specific registry values:

```
getRegStringValue(AppName, subkey, variable, value, &size);  
setRegStringValue(AppName, subkey, variable, value);
```

The application's default messaging name (*self*) should in general be unique to the device class, not to the application itself (e.g. *lights*, *radar*, *radio*, etc.). At run time, the base code automatically tries to redefine this string from the registry path:

```
./Project54/AppName/Messaging/self
```

It is no longer necessary for the application specific code to resolve the *self* messaging name from the registry. In addition, the XML GUI routines automatically resolve any messaging names built into the XML script. The application need only resolve messaging names that it uses explicitly in the application specific code.

The application's version specification (*AppVersion*) should indicate the specific hardware device (or other functionality) supported by the application, along with the version number and the date of the last edit:

```
wchar_t * AppVersion = L"Whelen Serial Lights & Siren V3.1 Jan 15, 2007";
```

All applications using the new format should have version numbers of the form V3.nn where nn is *at a minimum incremented for every change which is committed to the repository*. There is no harm in incrementing the version for intermediate edits that are not ready for the repository during the development process. New applications which are implemented using the old *P54AppBase.cpp* base file should retain the V2.nn version numbering.

The base code automatically writes the version string *AppVersion* to the registry value:

```
./Project54/Components/Versions/AppName
```

Thus, it is no longer necessary for the application specific code to write the version string to the registry at run time. The application code should still set the run-time status in the registry:

```
setRegStringValue(L"Components", L"Status", AppName, status);
```

### ***Application-to-GUI Messaging Considerations***

An application communicates with an XML GUI using text messages, in much the same way that applications communicate with each other. A special procedure is used to send messages to the GUI:

```
MessageToGui(source, gui_name, messageID, message_text);
```

The format of this procedure call is the same as for the standard *Project54* Message() procedure, except that the destination argument must be the name of an XML GUI (which is returned to the application when the GUI is created) rather than the name of another application.

Messages generated by the XML GUI script are returned to the application via the application's normal message processing procedure:

```
Void AppHandleMessage(LPWSTR source, LPWSTR dest, LPWSTR messageID, LPWSTR message_text) { ... }
```

The application can identify a message from its GUI by comparing the source argument to the name of its GUI. In order to facilitate the full capability of the XML GUI script, an application should normally relay to the GUI all messages received (other than those from the GUI itself):

```
if (wcscmp(source, gui_name) != 0)
    MessageToGui(source, gui_name, messageID, message_text);
```

If an application supports multiple GUIs, some logic is required in relaying messages. For example, it would not make sense to relay the "SHOW WINDOW" message to more than one GUI.

### ***GUI Window Management***

For a typical application there is little need for window management in the application specific code as long as all messages coming in to the application are relayed to the GUI as described in the previous section. The "SHOW WINDOW" command received from another application and relayed to the GUI will cause it to rise to the top of the *Project54* GUI stack without further action from the application code.

As was the case with the conventional GUI calls, changes to the GUI as the result of messages sent to the GUI are not normally painted immediately, in order to avoid

excessive repaint operations which can be processor time intensive. However, the XML GUI will in general repaint itself at appropriate times without explicit commands from the application. If it is important to repaint the GUI immediately after some status change has been sent to the GUI, the application can send a blank message to the GUI:

```
MessageToGui(self, gui_name, L"", L"");
```

This message will trigger an immediate repaint of the GUI. This should be avoided unless found to be necessary during application testing.

If an application has multiple GUI screens, it can force a particular GUI to come to the top of the GUI stack at any time by sending it a "SHOW WINDOW" command:

```
MessageToGui(self, gui_name, L"", L"SHOW WINDOW");
```

This is generally not necessary if an application has only a single GUI since the "SHOW WINDOW" command will generally be received from another application (such as the Main Screen application) and relayed to the GUI as described previously.

### ***Application Grammar Management***

An application which has a fixed command grammar associated with each of its GUIs can totally ignore the need to manage grammar files. The grammar file for each GUI will be created automatically at run time and will be managed entirely by the XML GUI component.

The XML GUI component creates the grammar file for a GUI automatically as the GUI window is being created from the XML script. This file is not modified again by the XML GUI component until the next time that the *Project54* application is started. Thus, if an application wishes to modify its automatic grammar (to add commands at run-time, for example) it is safe to do so any time after the call to *CreateXMLGui()*. The file will still be managed automatically by the XML GUI component as long as the file name is not changed.

If it is necessary for an application to explicitly manage grammar files, it can use the "P54 WINDOW SHOW" and "P54 WINDOW HIDE" messages to synchronize loading of the proper grammar. These messages are automatically sent by the GUI to the application when the corresponding window is brought to the top of the GUI stack or when it loses its place at the top of the stack.

### ***Status Feedback to other Applications***

In the new application framework, the feedback of status to client applications (such as a patrol screen type application) is handled automatically by the standard code in *P54XMLAppBase.cpp*. However, in order to take advantage of this functionality the application specific code should use a special procedure to send status messages to its XML GUI:

```
void StatusMessage(wchar_t * gui_name, wchar_t * status_parameter, wchar_t * status_value);
```

From the standpoint of XML GUI processing, the following two procedure calls are identical:

```
MessageToGui(self, gui_name, self, L"STATUS TARGET SPEED 72");
```

```
StatusMessage(gui_name, L"TARGET SPEED", L"72");
```

However, the latter form automatically supports feedback of status to other applications that already have or will soon register for feedback from this application. The *StatusMessage()* procedure forwards the status message both to the named GUI and to any applications that have registered for feedback. In addition, the core code keeps track of the most recent state of feedback parameters that have been transmitted via *StatusMessage()*, and automatically sends initial status messages reflecting the current state to each application that subsequently registers for feedback. Thus, the entire feedback mechanism becomes transparent to the status server application.

Under special circumstances, *MessageToGUI()* can be used to send status messages to the GUI without sending feedback to client applications, and *StatusMessage()* can be used with a NULL GUI name argument to send status feedback to client applications without sending it to the GUI. However, these options should be avoided unless absolutely necessary.