

XML Elements for Defining GUI Screens for Project54 Applications W. Thomas Miller, III

Overview

The *Project54* XML GUI Specification provides a framework for implementing application GUIs which interact with the application code by fielding status text messages from the application code and by sending command text messages back to the application code. Each XML file contains the description of a single GUI interface, based on XML element definitions described in this document. The appearance, functionality and speech grammar associated with a GUI are all built at run-time as described in the XML file. As a result, Project54 applications can be created for which the application GUI can be customized significantly in appearance and grammar (and to some extent functionality) simply by editing the XML file, without modifying the application code.

The logical flow of a XML GUI is based on responding to text messages from the application. Messages are classified into two categories consistent with the standard Project54 model for application-to-application feedback. *Status* messages include all messages which begin with "STATUS <rest of message>" (e.g. "STATUS STROBES OFF", "STATUS SIRENMODE WAIL"). All other messages are interpreted as *command* messages (e.g. "STROBES OFF", "WAIL"). Speech input messages (e.g. "SPEECHIN STROBES OFF", "SPEECHIN WAIL") are stripped of the "SPEECHIN" prefix and processed like other command messages.

The major functional XML elements (e.g. <button>, <lightset>, <textfield>, <textarea>, <image>) examine all status messages and change their visible state whenever a status message matches one of their internal message templates. <action> XML elements examine all command messages and send specified messages whenever a command message matches one of their internal command templates. <action> XML elements can send specified command or status messages to the GUI itself, to the parent application, or to a different application. In response to button press or release on the GUI, <button> elements send specified command messages both to the GUI itself and to the parent application.

All interface elements which are capable of displaying status contain member <state> elements and member <statusmessage> elements. Each <state> element defines the appearance and behavior of the parent element when in that state. The <state> elements are unique for each parent element type (i.e. the properties needed to draw a status light are different from those needed to draw a button or a textfield). Each <statusmessage> element provides a template for one or more expected status messages, along with a selector for a <state> element. The <statusmessage> elements are consistent for all parent element types. When the template of a <statusmessage> element is matched by an incoming status message, the corresponding <state> is selected for the parent element, and the element is redrawn to reflect the new state.

Each <action> element contains member <command> elements and member <message> elements. Each <command> element provides a template for one or more expected command messages. Each <message> element provides a template for a new message to be sent. When the template of a <command> element within an <action> element is matched by an incoming command message,

one or more messages are sent according to the specifications in the corresponding <message> elements.

The <screen> element

The root element is the <screen>. Each XML source file contains a single <screen> element which defines both the appearance and the functionality of one application GUI. The fundamental structure of the <screen> element is shown below.

```
- <screen id="NHDS PatrolScreen">
  <title>Project54: Patrol Information Screen</title>
  <grammar>patrol_screen.txt</grammar>
+ <device x="0" y="0" id="PATROL SCREEN BASE">
+ <device x="0" y="0" id="STALKER RADAR">
+ <device x="0" y="37" id="ASTRO RADIO">
+ <device x="0" y="56" id="WHELEN SERIAL LIGHTS">
  </screen>
```

Every <screen> has an “id” attribute, one <title> element, up to one <grammar> element, and up to 32 <device> elements. The “id” attribute of <screen> is the logical name of the GUI which is returned to the application upon GUI creation. This name is used as both the source name and destination name for all messages sent from the GUI to the application (except for messages sent by <action> elements), and must be used by the application as the destination name for all messages sent from the application to the GUI. In this way, an application with multiple GUI screens can determine which GUI sent a message and can target a message to a specific GUI. Note that the GUI name must be unique over all GUIs of all applications, since a single COM dll manages all GUIs.

The <title> element

Every <screen> must have a <title> element. This is the title placed in the Windows title bar when this screen is displayed.

The <grammar> element

Every <screen> can have a <grammar> element. This is the name of the grammar file which is loaded automatically when this screen is displayed. Note that this grammar file is automatically generated from the XML script at run time. Any existing grammar file with the same name will be overwritten. If the grammar file specification is missing or blank, no grammar file is created and the application grammar must be created separately and managed explicitly by the application code.

The <device> element

```
- <device x="0" y="0" id="STALKER RADAR">
+ <buttonset>
+ <textfield x="10" y="3" id="TARGET">
+ <textfield x="40" y="3" id="LOCK">
+ <textfield x="70" y="3" id="PATROL">
+ <lightset x="15" y="25" id="ANTENNAS">
+ <lightset x="45" y="25" id="RADAR MODES">
+ <action>
+ <action>
+ <action>
  </device>
```

Every <screen> must have at least one, and not more than 32, <device> elements. Each <device> element contains a collection of the XML elements which describe the message processing functionality relative to a device. The <device> element must have the x="n" and y="m" integer attributes, where n and m specify the offset of the contained elements in the screen display (0 to 100). The "id" attribute of <device> is only implemented to assist readability, and has no functional purpose.

A <device> element has no inherent functionality other than its base location for GUI screen drawing. Rather, the <device> element serves as a container for other functional elements of the GUI. Each <device> element can contain one <buttonset> and a collection of <textfield>, <textarea>, <lightset>, <image>, <label> and <action> elements (0 to 32 elements of *each* type). The association of individual <device> elements with individual application functionalities (e.g. lights versus siren, or radio state control/display versus radio channel changing) is encouraged to the extent that it simplifies GUI organization and ease of support. However, identical GUI functionality and run-time efficiency can be achieved by using a single <device> element or by using several <device> elements.

The <label> element

Every <device> can have up to 32 <label> elements. Each <label> element represents static text which is displayed on the screen.

```
= <label x="50" y="94">
  <text>Project54 Patrol Information Screen</text>
  <fontsize>1</fontsize>
  <font>BOLDITALIC</font>
  <just>cb</just>
</label>
```

The <label> element must have the x="n" and y="m" integer attributes, where n and m specify the location of the label relative to the <device>. The actual coordinates of the label are determined from the sum of the coordinates of the device and label (final result in the range 0 to 100).

Every <label> element must have a text <element>. The text string is the static text to be displayed on the page. The string can contain either “\n” or “|” to encode the newline character, and “\+” to encode the “&” character (“&” is a reserved operator in XML scripts and can not be used directly in text strings).

Every <label> must have a <fontsize> element. The integer value is the size (1 to 5) used to draw the label characters on the page.

Every <label> must have a element. The string value defines the font used to draw the label characters on the page (NORMAL, BOLD, ITALIC, BOLDITALIC).

Every <label> must have a <just> element. The string value defines the positioning of the label relative to the specified x,y screen coordinate. Each string contains two characters controlling horizontal (r, c, l = right, center, left) and vertical (t, c, b = top, center, bottom) string justification respectively.

The <buttonset> element

Every <device> can have one <buttonset> element. The <buttonset> element is a simple container for the <button> elements for the device.

```
- <buttonset>
+ <button col="1" row="3" id="FRONT ANTENNA">
+ <button col="2" row="3" id="REAR ANTENNA">
  </buttonset>
```

The <button> element in <buttonset>

Each <button> element defines the properties of a single button while this screen is displayed.

```
- <button col="1" row="3" id="FRONT ANTENNA">
  <behavior>sticky</behavior>
+ <state>
+ <state>
+ <statusmessage>
+ <statusmessage>
+ <statusmessage>
  </button>
```

The <button> element must have the col="n" and row="m" integer attributes, where n and m specify the absolute location in the button columns on the screen. Each column has 6 buttons. The number of columns of buttons for a screen is determined automatically by the maximum column number used in all of the <button> elements in all of the <device> elements in the <screen>. The "id" attribute of <button> is the logical name of the control as referenced in GetGuiElementHandle.

The <button> can have a <behavior> element which specifies “sticky”. This creates a button which must be pressed once to depress, and then pressed again to release. Alternatively the <button> can have a <behavior> element which specifies “repeat”. This creates a button which sends the same command repeatedly while depressed. A command is sent when it is first pressed, and then after about 1 second the command is repeated every 600 milliseconds, and finally after about 3 seconds the command is repeated every 300 milliseconds. If the <behavior> element is missing, the button automatically releases when not pressed on the touchscreen.

The <state> element in <button>

Each <button> element must have at least one and no more than 32 <state> elements. The first <state> element listed is used to define the initial state. The <value> element is an identifier used to select the state. The <onoff> element specifies (0 or 1) whether the button appears released or depressed while in that state. The <text> item defines the button label while in that state. The string can contain either “\n” or “|” to encode the newline character, and “\+” to encode the “&” character (“&” is a reserved operator in XML scripts and can not be used directly in text strings). Finally, the <command> item defines the command message that will be sent the next time that the button is activated (physically pressed or released via the touchscreen) while in that state. The example below shows a two state button which generates the “FRONT ANTENNA” command when depressed and generates the “FRONT ANTENNA OFF” command when released. In the example, the label does not change between the states (but it could).

```
- <state>
  <value>0</value>
  <onoff>0</onoff>
  <command>FRONT ANTENNA</command>
  <text>Front\nAntenna</text>
</state>
- <state>
  <value>1</value>
  <onoff>1</onoff>
  <command>FRONT ANTENNA OFF</command>
</state>
```

Note that the command message associated with a button state is not sent when the <button> element enters that logical state. Rather, the command message for a state is sent when the <button> element is in that logical state at the time that the physical button is activated (pressed or released) on the touchscreen.

Also note that the logical button state may change as the result of the physical button being activated (pressed or released) on the touchscreen. Immediately after a physical button is pressed, the logical state of the <button> element automatically changes to the first state with an <onoff> element of “1” (if such a state exists). Immediately after a physical button is released, the logical state of the <button> element changes to the first state with an <onoff> element of “0” (if such a state exists).

Finally, a `<button>` element may have only a single logical state, in which case it is always in that state by default (and the `<onoff>` element is not relevant). In this situation, the command for the state is sent whenever the physical button is released, unless the button has a “repeat” behavior, in which case the command is sent periodically while the button is pressed.

The `<statusmessage>` element in `<button>`

Each `<button>` element can have up to 32 `<statusmessage>` elements. Each `<statusmessage>` element defines the template for matching to incoming Project54 status messages.

```
= <statusmessage>  
  <msrc>Radar</msrc>  
  <mid>* </mid>  
  <mtext>FRONT ANTENNA</mtext>  
  <value>1</value>  
  </statusmessage>
```

The `<msrc>` element defines the source application for the matching Project 54 status message. The name specified is first referenced through the `.\Project54\app_registry_name\Messaging` registry key to determine the actual application messaging name. If no match is found, the `<msrc>` element value is assumed to be the actual messaging name of the application. If it is not important to match the source application name, the `*` wildcard value could be used, or the `<msrc>` element could simply be omitted. The `<msrc>` wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The `<mid>` element defines the message ID string for the matching Project 54 status message. If it is not important to match the message ID string, the `*` wildcard value can be used as in the example, or the `<mid>` element could simply be omitted. The `<mid>` wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The `<mtext>` element specifies the text of the message body for the matching Project 54 status message. The “STATUS” prefix is implied and should not be included (i.e. `<mtext>FRONT ANTENNA</mtext>` matches the actual message text “STATUS FRONT ANTENNA”).

The `<value>` element specifies a string for identifying the corresponding `<state>` element. When a matching message is detected, the button switches to the `<state>` with the matching `<value>` element.

A simple `<button>` which does not reflect the state of a device, and which issues a single fixed command every time the physical button is activated, can default many of these values. The example below shows a simple button which issues the “MAIN SCREEN” command when the physical button is pressed and then released:

```

= <button col="1" row="1" id="MAIN SCREEN">
  = <state>
    <command>MAIN SCREEN</command>
    <text>Main\nScreen</text>
  </state>
</button>

```

A complex button could have up to 32 <state> elements, each with different <text> labels and different <command> elements, selected by different incoming status messages matching different <statusmessage> elements.

```

<button col="3" row="3" id="TEST">
  <state>
    <value>1</value>
    <command>ADVISOR 2</command>
    <text>Advsr 2</text>
  </state>
  <state>
    <value>2</value>
    <command>ADVISOR 3</command>
    <text> Advsr 3</text>
  </state>
  <state>
    <value>3</value>
    <command>ADVISOR 4</command>
    <text> Advsr 4</text>
  </state>
  <state>
    <value>4</value>
    <command>ADVISOR 1</command>
    <text> Advsr 1</text>
  </state>
  <statusmessage>
    <mtext>ADVISOR 1</mtext>
    <value>1</value>
  </statusmessage>
  <statusmessage>
    <mtext>ADVISOR 2</mtext>
    <value>2</value>
  </statusmessage>
  <statusmessage>
    <mtext>ADVISOR 3</mtext>
    <value>3</value>
  </statusmessage>
  <statusmessage>
    <mtext>ADVISOR 4</mtext>
    <value>4</value>
  </statusmessage>
</button>

```

The <button> in the example above sends the command to change to a new traffic advisor mode (1 of 4) each time it is pressed. The button label always shows the next mode to be selected. The <button> element synchronizes to the actual traffic advisor mode when it receives a “STATUS ADVISOR n” message.

The <mtext> element can also contain the * wildcard character at the end of the indicated message text. In such cases, the message is assumed to match if the incoming status message text matches everything in the template up to the *. The corresponding <value> element in <statusmessage> can reference this variable portion of the message text using the ? wildcard. In the <value> element ? is replaced by the variable portion of the text.

```
<statusmessage>
  <mtext>ADVISOR * </mtext>
  <value>?</value>
</statusmessage>
```

In this example, `<mtext>ADVISOR *</mtext>` matches all messages of the form “STATUS ADVISOR *more_text*”. The corresponding `<value>` is the string *more_text*. Thus, the single `<statusmessage>` element above could be used to replace all four `<statusmessage>` elements in the prior example.

The `<textfield>` element

Every `<device>` can have up to 32 `<textfield>` elements. Each `<textfield>` element defines the properties of a single text field where variable text is displayed based on incoming status messages.

```
- <textfield x="10" y="3" id="TARGET">
  <fontsize>3</fontsize>
  <width>3</width>
  <behavior>edit</behavior>
+ <state>
+ <statusmessage>
+ <label x="9" y="14">
  </textfield>
```

The `<textfield>` element must have the `x="n"` and `y="m"` integer attributes, where `n` and `m` specify the location of the field relative to the base coordinates for the `<device>` element. The actual coordinates of the field are determined from the sum of the coordinates of the device and the textfield (final result in the range 0 to 100). The “id” attribute of `<textfield>` is the logical name of the control as referenced in `GetGuiElementHandle`.

Every `<textfield>` must have a `<fontsize>` element. The integer value is the size (1 to 5) used for characters in the form field. Every `<textfield>` must have a `<width>` element. The integer value is the width in characters of the visible form field. This impacts the number of visible characters but does not limit the number of characters that can be entered into the field.

The `<textfield>` can have a `<behavior>` element which specifies “edit”. This creates a textfield which can be used for user text entry. Alternatively, the `<behavior>` element can specify “password”, in which case the textfield can be edited, but all typed in characters are displayed as ‘*’. In these cases, the textfield contents are reported to the application every time the contents are modified (every keystroke). The application will receive messages with the “id” attribute of the `<screen>` element in the source and destination fields, the “id” attribute of the `<textfield>` element in the message id field, and the contents of the `<textfield>` in the message text field. If the `<behavior>` element is missing or does not contain one of the values above, the textfield contents can not be edited by the user.

The <state> element in <textfield>

Each <textfield> element must have at least one and no more than 32 <state> elements. The first <state> element listed is used to define the initial state. The <value> element is an identifier used to select the state based on matching incoming status messages.

```
= <state>  
  <value>0</value>  
  <text>?</text>  
  <onoff>1,1,1,1,0,0,0,0</onoff>  
  <textcolor>0x00008f</textcolor>  
  <offcolor>BLUE</offcolor>  
</state>
```

The <text> element specifies the text that should be written to the text field whenever that state is selected by an incoming status message. The text can be a constant text string <text>DEVICE ERROR!!!</text> or can include the * or ? wildcard characters. See the following section on <statusmessage> elements for more information on the processing of wildcards in <textfield> elements.

The <onoff> element can describe an arbitrary flashing pattern. Sequential steps in the pattern are separated by “,” characters. Pattern transitions occur every 150 milliseconds. The pattern repeats until a different <state> is selected. In the above example, the text would be “on” for 4 tics (600 milliseconds) and “off” for 4 tics.

The optional <textcolor> element defines the corresponding color for the text when it is “on”. The color should be a 24 bit color in hexadecimal format (0xbbggrr) where bb, gg, and rr are the individual blue, green and red color components (00 to ff). The following keywords can also be used: BLACK, WHITE, GRAY, RED, GREEN, BLUE, YELLOW, and DEFAULT. The optional <offcolor> element defines the corresponding color for the text when it is “off”, while blinking as described above. If the <offcolor> element is omitted for blinking text, the text alternates between visible and invisible.

The <statusmessage> element in <textfield>

Each <textfield> element can have up to 32 <statusmessage> elements. Each <statusmessage> element defines the template for matching to incoming Project54 status messages.

```
= <statusmessage>  
  <msrc>Radar</msrc>  
  <mid>*</mid>  
  <mtext>TARGET SPEED *</mtext>  
  <value>0</value>  
</statusmessage>
```

The <msrc> element defines the source application for the matching Project 54 status message. The name specified is first referenced through the .\Project54\app_registry_name\Messaging registry key to determine the actual application messaging name. If no match is found, the <msrc> element value is assumed to be the actual messaging name of the application. If it is not important to match the source application name, the * wildcard value could be used, or the <msrc> element could simply be omitted. The <msrc> wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The <mid> element defines the message ID string for the matching Project 54 status message. If it is not important to match the message ID string, the * wildcard value can be used as in the example, or the <mid> element could simply be omitted. The <mid> wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The <mtext> element specifies the text of the message body for the matching Project 54 status message. The “STATUS” prefix is implied and should not be included (i.e. <mtext>FRONT ANTENNA</mtext> matches the actual message text “STATUS FRONT ANTENNA”). The <mtext> element can also contain the * wildcard character at the end of the indicated message text. In such cases, the message is assumed to match if the incoming status message text matches everything in the template up to the *. In the example, <mtext>TARGET SPEED *</mtext> matches all messages of the form “STATUS TARGET SPEED *more_text*”.

The <text> element in the corresponding <state> can reference this variable portion of the message text using the * and ? wildcards. In the <text> element, ? is replaced by the variable portion of the text (“*more_text*”) while * is replaced by the entire matching message (“TARGET SPEED *more_text*”). Thus, in the examples above the incoming target speed value would be written into the <textfield>. Note that either wildcard character could be embedded inside a string. For example <text>Trg = ? mph</text> would cause “Trg = *more_text* mph” to be written to the <textfield> when a matching status message was received.

The <value> element specifies a string for identifying the corresponding <state> element. When a matching message is detected, the button switches to the <state> with the matching <value> element. The <value> element in <statusmessage> can also reference the variable portion of the message text using the ? wildcard: <value>?</value>. However, this aspect is not generally useful for <textfield> processing.

The <label> element in <textfield>

Every <textfield> can have up to 32 <label> elements. Each <label> element represents static text which is displayed on the screen.

```
= <label x="9" y="14">  
  <text>Target</text>  
  <fontsize>2</fontsize>  
  <font>BOLD</font>  
  <just>ct</just>  
  </label>
```

The <label> element must have the x="n" and y="m" integer attributes, where n and m specify the location of the label relative to the <textfield>. The actual coordinates of the label are determined from the sum of the coordinates of the device, textfield and label (final result in the range 0 to 100).

Every <label> element must have a text <element>. The text string is the static text to be displayed on the page. The string can contain either "\n" or "|" to encode the newline character, and "\+" to encode the "&" character ("&" is a reserved operator in XML scripts and can not be used directly in text strings).

Every <label> must have a <fontsize> element. The integer value is the size (1 to 5) used to draw the label characters on the page.

Every <label> must have a element. The string value defines the font used to draw the label characters on the page (NORMAL, BOLD, ITALIC, BOLDITALIC).

Every <label> must have a <just> element. The string value defines the positioning of the label relative to the specified x,y screen coordinate. Each string contains two characters controlling horizontal (r, c, l = right, center, left) and vertical (t, c, b = top, center, bottom) string justification respectively.

The <textarea> element

Every <device> can have up to 32 <textarea> elements. Each <textarea> element defines the properties of a single text area where variable text is displayed based on incoming status messages.

```
= <textfield x="10" y="3" id="CADMSG">  
  <fontsize>3</fontsize>  
  <width>20</width>  
  <height>8</height>  
  <behavior>edit</behavior>  
+ <state>  
+ <statusmessage>  
+ <label x="9" y="14">  
  </textfield>
```

The <textarea> element must have the x="n" and y="m" integer attributes, where n and m specify the location of the field relative to the base coordinates for the <device> element. The actual coordinates of the field are determined from the sum of the coordinates of the device and the textarea (final result in the range 0 to 100). The "id" attribute of <textarea> is the logical name of the control as referenced in GetGuiElementHandle.

Every <textarea> must have a <fontsize> element. The integer value is the size (1 to 5) used for characters in the text area. Every <textarea> must have a <width> element. The integer value is the width in characters of the visible form field. Every <textarea> must have a <height> element. The integer value is the height in characters of the visible form field. These values impact the number of visible characters but do not limit the number of characters that can be displayed in the control.

The `<textarea>` can have a `<behavior>` element which specifies “edit”. This creates a textarea which can be used for user text entry. The text area contents are reported to the application every time the contents are modified (every keystroke). The application will receive messages with the “id” attribute of the `<screen>` element in the source and destination fields, the “id” attribute of the `<textarea>` element in the message id field, and the contents of the `<textarea>` in the message text field. If the `<behavior>` element is missing or does not contain one of the values above, the text area contents can not be edited by the user.

The `<state>` element in `<textarea>`

Each `<textarea>` element must have at least one and no more than 32 `<state>` elements. The first `<state>` element listed is used to define the initial state. The `<value>` element is an identifier used to select the state based on matching incoming status messages.

```
= <state>
  <value>0</value>
  <text>?</text>
  <textcolor>0x00008f</textcolor>
  <mode>add</mode>
</state>
```

The `<text>` element specifies the text that should be written to the text area whenever that state is selected by an incoming status message. The text can be a constant text string `<text>DEVICE ERROR!!!</text>` or can include the * or ? wildcard characters. See the following section on `<statusmessage>` elements for more information on the processing of wildcards in `<textarea>` elements. The optional `<textcolor>` element defines the corresponding color for the text. The color should be a 24 bit color in hexadecimal format (0xbbggrr) where bb, gg, and rr are the individual blue, green and red color components (00 to ff). The following keywords can also be used: BLACK, WHITE, GRAY, RED, GREEN, BLUE, YELLOW, and DEFAULT. The optional `<mode>` element sets the mode for processing the text. The valid modes are “set” (the default), “add” and “scroll”. In “set” mode, the new text replaces the current text area text. In “add” mode, the new text is appended to the current text area text. In “scroll” mode, the text must be a positive or negative integer value. The current text area text is scrolled down or up, respectively, by the integer number of lines.

The `<statusmessage>` element in `<textarea>`

Each `<textarea>` element can have up to 32 `<statusmessage>` elements. Each `<statusmessage>` element defines the template for matching to incoming Project54 status messages.

```
= <statusmessage>
  <msrc>Radar</msrc>
  <mid>*</mid>
  <mtext>TARGET SPEED *</mtext>
  <value>0</value>
</statusmessage>
```

The <msrc> element defines the source application for the matching Project 54 status message. The name specified is first referenced through the .\Project54\app_registry_name\Messaging registry key to determine the actual application messaging name. If no match is found, the <msrc> element value is assumed to be the actual messaging name of the application. If it is not important to match the source application name, the * wildcard value could be used, or the <msrc> element could simply be omitted. The <msrc> wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The <mid> element defines the message ID string for the matching Project 54 status message. If it is not important to match the message ID string, the * wildcard value can be used as in the example, or the <mid> element could simply be omitted. The <mid> wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The <mtext> element specifies the text of the message body for the matching Project 54 status message. The “STATUS” prefix is implied and should not be included (i.e. <mtext>FRONT ANTENNA</mtext> matches the actual message text “STATUS FRONT ANTENNA”). The <mtext> element can also contain the * wildcard character at the end of the indicated message text. In such cases, the message is assumed to match if the incoming status message text matches everything in the template up to the *. In the example, <mtext>TARGET SPEED *</mtext> matches all messages of the form “STATUS TARGET SPEED *more_text*”.

The <text> element in the corresponding <state> can reference this variable portion of the message text using the * and ? wildcards. In the <text> element, ? is replaced by the variable portion of the text (“*more_text*”) while * is replaced by the entire matching message (“TARGET SPEED *more_text*”). Thus, in the examples above the incoming target speed value would be written into the <textarea>. Note that either wildcard character could be embedded inside a string. For example <text>Trg = ? mph</text> would cause “Trg = *more_text* mph” to be written to the <textarea> when a matching status message was received.

The <value> element specifies a string for identifying the corresponding <state> element. When a matching message is detected, the <textarea> switches to the <state> with the matching <value> element. The <value> element in <statusmessage> can also reference the variable portion of the message text using the ? wildcard: <value>?</value>.

The following set of <state> and <statusmessage> elements provides to the application complete control of the <textarea> using status messages of the form “STATUS SET TEXT text_to_set”, “STATUS ADD TEXT text_to_add”, “STATUS SCROLL UP number_of_lines”, and “STATUS SCROLL DOWN number_of_lines”.

```

- <state><value>0</value><text>?</text><mode>set</mode></state>
- <state><value>1</value><text>?</text><mode>add</mode></state>
- <state><value>2</value><text>-?</text><mode>scroll</mode></state>
- <state><value>3</value><text>?</text><mode>scroll</mode></state>
- <statusmessage><mtext>SET TEXT *</mtext><value>0</value>
  </statusmessage>
- <statusmessage><mtext>ADD TEXT *</mtext><value>1</value>
  </statusmessage>
- <statusmessage><mtext>SCROLL UP *</mtext><value>2</value>
  </statusmessage>
- <statusmessage><mtext>SCROLL DOWN *</mtext><value>3</value>
  </statusmessage>

```

The <label> element in <textarea>

Every <textarea> can have up to 32 <label> elements. Each <label> element represents static text which is displayed on the screen.

```

- <label x="9" y="14">
  <text>Target</text>
  <fontsize>2</fontsize>
  <font>BOLD</font>
  <just>ct</just>
</label>

```

The <label> element must have the x="n" and y="m" integer attributes, where n and m specify the location of the label relative to the <textarea>. The actual coordinates of the label are determined from the sum of the coordinates of the device, textfield and label (final result in the range 0 to 100).

Every <label> element must have a text <element>. The text string is the static text to be displayed on the page. The string can contain either "\n" or "|" to encode the newline character, and "\+" to encode the "&" character ("&" is a reserved operator in XML scripts and can not be used directly in text strings).

Every <label> must have a <fontsize> element. The integer value is the size (1 to 5) used to draw the label characters on the page.

Every <label> must have a element. The string value defines the font used to draw the label characters on the page (NORMAL, BOLD, ITALIC, BOLDITALIC).

Every <label> must have a <just> element. The string value defines the positioning of the label relative to the specified x,y screen coordinate. Each string contains two characters controlling horizontal (r, c, l = right, center, left) and vertical (t, c, b = top, center, bottom) string justification respectively.

The <lightset> element

Every <device> can have up to 32 <lightset> elements. Each <lightset> element defines the behavior of a related set of status lights.

```
- <lightset x="10" y="0" id="WIG WAGS">
+ <light x="0" y="0">
+ <light x="48" y="0">
+ <state>
+ <state>
+ <statusmessage>
+ <statusmessage>
- </lightset>
```

The <lightset> element must have the x="n" and y="m" integer attributes, where n and m specify the location of the <lightset> relative to the base coordinates for the <device> element. The actual coordinates are determined from the sum of the coordinates of the <device> and the <lightset> (final result in the range 0 to 100). The "id" attribute of <lightset> is only implemented to assist readability, and has no functional purpose.

The <light> element in <lightset>

Every <lightset> can have up to 32 <light> elements. Each <light> element defines the basic appearance of a single status light.

```
- <light x="0" y="0" id="WIG WAG 1">
  <text>WW</text>
  <textcolor>BLACK</textcolor>
  <oncolor>WHITE</oncolor>
  <offcolor>DEFAULT</offcolor>
- </light>
```

The <light> element must have the x="n" and y="m" integer attributes, where n and m specify the location of the <light> relative to the base coordinates for the <lightset> element. The actual coordinates are determined from the sum of the coordinates of the <device>, the <lightset> and the <light> (final result in the range 0 to 100). The "id" attribute of <light> is the logical name of the control as referenced in GetGuiElementHandle.

The <text> element defines the initial text label for the status light. This should generally be limited to 1 or 2 characters (or no characters). The <textcolor>, <oncolor> and <offcolor> elements define the corresponding colors for this status light. The colors should be a 24 bit color in hexadecimal format (0xbbggrr) where bb, gg, and rr are the individual blue, green and red color components (00 to ff). The following keywords can also be used: BLACK, WHITE, GRAY, RED, GREEN, BLUE, YELLOW, and DEFAULT.

The <state> element in <lightset>

Each <lightset> element must have at least one and no more than 32 <state> elements. The first <state> element listed is used to define the initial state. The <value> element is an identifier used to select the state based on matching incoming status messages.

```
= <state>
  <value>0</value>
  <onoff>00</onoff>
</state>
= <state>
  <value>1</value>
  <onoff>10,10,01,01</onoff>
</state>
```

The <onoff> element describes the condition of the individual <light> elements while in this state. 0 corresponds to “off” and 1 corresponds to “on”. The first character corresponds to the first <light> in the set, the second character to the second <light>, and so forth. In the example above, both status lights in the <lightest> are off when in state 0.

The <onoff> element can describe an arbitrary flashing pattern. Sequential steps in the pattern are separated by “,” characters. Pattern transitions occur every 150 milliseconds. The pattern repeats until a different <state> is selected. In state 1 of the above example, the <lights> alternate, each being on for 300 milliseconds and off for 300 milliseconds. The example below shows a typical lightbar pattern for <lightest> of 6 <lights>.

```
= <state>
  <value>1</value>
  <onoff>001100,011110,110011,100001</onoff>
</state>
```

The <state> element can optionally include <text>, <textcolor>, <oncolor> and <offcolor> elements, so that the labels and colors of the status lights change when the <state> changes. While each <light> element in a <lightest> can have different specifications when created, the new values for these items in the <state> element apply equally to all <light> elements in the <lightset>. In the more complex example below, the lightbar is off with no text labels in state 0, the lightbar flashes a blue strobe pattern with “S” text labels in state 1, and flashes a yellow warning pattern with “W” text labels in state 2.

```
= <state>
  <value>0</value>
  <onoff>000000</onoff>
  <text></text>
</state>
= <state>
  <value>1</value>
  <onoff>001100,011110,110011,100001</onoff>
  <textcolor>WHITE</textcolor>
```

```

    <oncolor>BLUE</oncolor>
    <offcolor>DEFAULT</offcolor>
    <text>S</text>
    </state>
= <state>
    <value>2</value>
    <onoff>101010,101010,010101,010101</onoff>
    <textcolor>BLACK</textcolor>
    <oncolor>YELLOW</oncolor>
    <offcolor>DEFAULT</offcolor>
    <text>W</text>
    </state>

```

The <statusmessage> element in <lightset>

Each <lightset> element can have up to 32 <statusmessage> elements. Each <statusmessage> element defines the template for matching to incoming Project54 status messages.

```

= <statusmessage>
    <msrc>Lights</msrc>
    <mid>*</mid>
    <mtext>WIG WAGS</mtext>
    <value>1</value>
    </statusmessage>

```

The <msrc> element defines the source application for the matching Project 54 status message. The name specified is first referenced through the .\Project54\app_registry_name\Messaging registry key to determine the actual application messaging name. If no match is found, the <msrc> element value is assumed to be the actual messaging name of the application. If it is not important to match the source application name, the * wildcard value could be used, or the <msrc> element could simply be omitted. The <msrc> wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The <mid> element defines the message ID string for the matching Project 54 status message. If it is not important to match the message ID string, the * wildcard value can be used as in the example, or the <mid> element could simply be omitted. The <mid> wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The <mtext> element specifies the text of the message body for the matching Project 54 status message. The “STATUS” prefix is implied and should not be included (i.e. <mtext>WIG WAGS</mtext> matches the actual message text “STATUS WIG WAGS”).

The <value> element specifies a string for identifying the corresponding <state> element. When a matching message is detected, the <lightset> switches to the <state> with the matching <value> element.

The `<mtext>` element can also contain the `*` wildcard character at the end of the indicated message text. In such cases, the message is assumed to match if the incoming status message text matches everything in the template up to the `*`. The corresponding `<value>` element in `<statusmessage>` can reference this variable portion of the message text using the `?` wildcard. In the `<value>` element, `?` is replaced by the variable portion of the text.

```
<statusmessage>  
  <mtext>ADVISOR * </mtext>  
  <value>?</value>  
</statusmessage>
```

In this example, `<mtext>ADVISOR * </mtext>` matches all messages of the form “STATUS ADVISOR *more_text*”. The corresponding `<value>` is the string *more_text*. Thus “STATUS ADVISOR 1” would select state 1, “STATUS ADVISOR 2” would select state 2, and so forth.

The `<label>` element in `<lightset>`

Every `<lightset>` can have up to 32 `<label>` elements. Each `<label>` element represents static text which is displayed on the screen.

```
= <label x="-2" y="3">  
  <text>Cntrl\nHead</text>  
  <fontsize>1</fontsize>  
  <font>BOLDITALIC</font>  
  <just>rc</just>  
</label>
```

The `<label>` element must have the `x="n"` and `y="m"` integer attributes, where `n` and `m` specify the location of the label relative to the `<lightset>`. The actual coordinates are determined from the sum of the coordinates of the `<device>`, the `<lightset>` and the `<label>` (final result in the range 0 to 100).

Every `<label>` element must have a text `<element>`. The text string is the static text to be displayed on the page. The string can contain either `"\n"` or `"|"` to encode the newline character, and `"\&"` to encode the `"&"` character (`"&"` is a reserved operator in XML scripts and can not be used directly in text strings).

Every `<label>` must have a `<fontsize>` element. The integer value is the size (1 to 5) used to draw the label characters on the page.

Every `<label>` must have a `` element. The string value defines the font used to draw the label characters on the page (NORMAL, BOLD, ITALIC, BOLDITALIC).

Every `<label>` must have a `<just>` element. The string value defines the positioning of the label relative to the specified `x,y` screen coordinate. Each string contains two characters controlling horizontal (`r`, `c`, `l` = right, center, left) and vertical (`t`, `c`, `b` = top, center, bottom) string justification respectively.

The <image> element

Every <device> can have up to 32 <image> elements. Each <image> element defines the properties of bitmap images to be drawn on the screen.

```
- <image x="50" y="58" id="SHIELDS">
  <just>ct</just>
+ <state>
+ <state>
+ <statusmessage>
+ <statusmessage>
  </image>
```

The <image> element must have the x="n" and y="m" integer attributes, where n and m specify the location of the <image> relative to the base coordinates for the <device> element. The actual coordinates are determined from the sum of the coordinates of the <device> and the <image> (final result in the range 0 to 100). The "id" attribute of <image> is the logical name of the control as referenced in GetGuiElementHandle.

Every <image> must have a <just> element. The string value defines the positioning of the bitmap image relative to the specified x,y screen coordinate. Each string contains two characters controlling horizontal (r, c, l = right, center, left) and vertical (t, c, b = top, center, bottom) string justification respectively.

The <state> element in <image>

Each <image> element must have at least one and no more than 32 <state> elements. The first <state> element listed is used to define the initial state. The <value> element is an identifier used to select the state based on matching incoming status messages.

```
= <state>
  <value>0</value>
  <file>proj54.bmp</file>
  </state>
```

The <file> element specifies the path to the bitmap file to be displayed. Unless the path includes a drive specification (e.g. c:\folder\file.bmp) the string is assumed to be a partial path relative to the base specified in the .\Project54\Paths\Bitmaps\BitmapDefaultPath registry variable. Note that the bitmap file is not a background image. Rather, it has equal status to other controls on the screen and should not generally overlap with other visual elements.

The <statusmessage> element in <image>

Each <image> element can have up to 32 <statusmessage> elements. A static image need not have any <statusmessage> elements. Each <statusmessage> element defines the template for matching to incoming Project54 status messages.

```

= <statusmessage>
  <mtext>DAYMODE</mtext>
  <value>0</value>
  </statusmessage>

```

The <msrc> element defines the source application for the matching Project 54 status message. The name specified is first referenced through the .\Project54\app_registry_name\Messaging registry key to determine the actual application messaging name. If no match is found, the <msrc> element value is assumed to be the actual messaging name of the application. If it is not important to match the source application name, the * wildcard value could be used, or the <msrc> element could simply be omitted. The <msrc> wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The <mid> element defines the message ID string for the matching Project 54 status message. If it is not important to match the message ID string, the * wildcard value can be used as in the example, or the <mid> element could simply be omitted. The <mid> wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The <mtext> element specifies the text of the message body for the matching Project 54 status message. The “STATUS” prefix is implied and should not be included (i.e. <mtext>DAYMODE</mtext> matches the actual message text “STATUS DAYMODE”.

The <value> element specifies a string for identifying the corresponding <state> element. When a matching message is detected, the <lightest> switches to the <state> with the matching <value> element. The image drawn to the screen can thus be switched in response to incoming status messages. In the example below, different images are displayed in day-mode versus night-mode of the system.

```

= <state>
  <value>0</value>
  <file>c:\Project54\Bitmaps\proj54.bmp</file>
  </state>
= <state>
  <value>1</value>
  <file>c:\Project54\Bitmaps\nproj54.bmp</file>
  </state>
= <statusmessage>
  <mtext>DAYMODE</mtext>
  <value>0</value>
  </statusmessage>
= <statusmessage>
  <mtext>NIGHTMODE</mtext>
  <value>1</value>
  </statusmessage>

```

The <mtext> element can also contain the * wildcard character at the end of the indicated message text. In such cases, the message is assumed to match if the incoming status message text matches

everything in the template up to the *. The corresponding <value> element in <statusmessage> can reference this variable portion of the message text using the ? wildcard. In the <value> element, ? is replaced by the variable portion of the text.

```
<statusmessage>  
  <mtext>ADVISOR *</mtext>  
  <value>?</value>  
</statusmessage>
```

In this example, <mtext>ADVISOR *</mtext> matches all messages of the form “STATUS ADVISOR *more_text*”. The corresponding <value> is the string *more_text*. Thus “STATUS ADVISOR 1” would select state 1, “STATUS ADVISOR 2” would select state 2, and so forth.

The <label> element in <image>

Every <image> can have up to 32 <label> elements. Each <label> element represents static text which is displayed on the screen.

```
= <label x="-2" y="3">  
  <text>Participating\nAgencies</text>  
  <fontsize>1</fontsize>  
  <font>BOLDITALIC</font>  
  <just>rc</just>  
</label>
```

The <label> element must have the x="n" and y="m" integer attributes, where n and m specify the location of the label relative to the <image>. The actual coordinates are determined from the sum of the coordinates of the <device>, the <image> and the <label> (final result in the range 0 to 100).

Every <label> element must have a text <element>. The text string is the static text to be displayed on the page. The string can contain either “\n” or “|” to encode the newline character, and “\+” to encode the “&” character (“&” is a reserved operator in XML scripts and can not be used directly in text strings).

Every <label> must have a <fontsize> element. The integer value is the size (1 to 5) used to draw the label characters on the page. Every <label> must have a element. The string value defines the font used to draw the label characters on the page (NORMAL, BOLD, ITALIC, BOLDITALIC). Every <label> must have a <just> element. The string value defines the positioning of the label relative to the specified x,y screen coordinate. Each string contains two characters controlling horizontal (r, c, l = right, center, left) and vertical (t, c, b = top, center, bottom) string justification respectively.

The <action> element

Every <device> can have up to 32 <action> elements. Each <action> element defines the mapping of internal or external command strings to specific messages to be sent to the application (or to other applications).

```

- <action>
  <command>RADAR</command>
+ <message>
  </action>

```

The <command> element in <action>

Every <action> element has at least one and as many as 32 <command> elements. Each <command> element specifies a template for matching incoming command messages. If an incoming command message matches any of the <command> elements, the Project54 messages described by the <message> elements are sent. The template format is the same as that for matching the text of <statusmessage> elements. The template can be an absolute string, in which case an absolute match is required, or can have a trailing * wildcard character, in which case only the beginning of the command is required to match.

Incoming command strings originate from one of three classes of sources:

1. Commands issued by <button> elements when the corresponding touchscreen button is activated.
2. Any Project54 message from the application which does not begin with the “STATUS “ prefix.
3. Special internally generated command strings (described later).

Note that the <action> element only processes the text of the incoming command. It is ignorant as to the source of the command. In contrast, status messages can be differentiated by source and by message ID, as well as by message content.

The <message> element in <action>

Every <action> element has at least one and as many as 32 <message> elements. Each <message> element defines a single message to be sent to the application (or to other applications). If multiple messages are specified, they are sent in the order that they appear in the XML script.

```

- <message>
  <msrc>* </msrc>
  <mdest>Radar</mdest>
  <mid>message id</mid>
  <mtext>SHOW WINDOW</mtext>
  </message>

```

The <msrc> element defines the source name for the Project 54 message to be sent. The <mdest> element defines the destination name for the Project 54 message to be sent. The names specified are first referenced through the .\Project54\app_registry_name\Messaging registry key to determine the actual messaging name. If no match is found, the <mdest> element value is assumed to be the actual messaging name of the destination. The * wildcard value can also be used, or the <msrc> and

<mdest> elements can be omitted. If both <msrc> and <mdest> are omitted, the message is sent to the application with both names set to the <screen> element “id” attribute (the GUI name), and is also automatically looped back to the GUI for processing. This is useful for “translating” incoming command messages to status messages to change the state of a <button>, <lightest>, etc. in response to a command message. If only the <msrc> name is omitted, the message is sent with the source name field set to the messaging name of the application. In this case, <mdest> should generally be set to the messaging name of a *Project54* application as well.

The <mid> element defines the message ID string for the Project 54 message to be sent. If it is not important to specify the message ID string, the <mid> element can simply be omitted. In this case the messaging name of the application itself is used as a default message ID.

The <mtext> element specifies the text of the message body for the Project 54 message to be sent. If the * wildcard character is used in the <command> element, the <mtext> element can reference this variable portion of the message text using the * and ? wildcards. In the <mtext> element, ? is replaced by the variable portion of the text while * is replaced by the entire matching command. Either wildcard character can be embedded inside a string in <mtext> to generate more complicated messages. This wildcard processing directly mirrors that implemented for <textfield> elements.

While it is not normally necessary, each <message> element can have a <value> element. In this case, the <value> is matched to the <value> of incoming status messages as explained below in the subsequent section on <statusmessage> elements inside <action> elements.

Some complete example <action> elements are included below for clarity.

Send the “SHOW WINDOW” command to the “Lights” application in response to the “EMERGENCY SIGNALS” command:

```
= <action>
  <command>EMERGENCY SIGNALS</command>
= <message>
  <mdest>Lights</mdest>
  <mtext>SHOW WINDOW</mtext>
  </message>
</action>
```

Forward the “FRONT ANTENNA”, “FRONT ANTENNA OFF”, ”REAR ANTENNA”, and “REAR ANTENNA OFF” messages to the Radar application:

```
= <action>
  <command>FRONT ANTENNA*</command>
  <command>REAR ANTENNA*</command>
= <message>
  <mdest>Radar</mdest>
  <mtext>*</mtext>
  </message>
</action>
```

Translate all variations of the system wide “BROADCAST NIGHTMODE” message into an internal “NIGHTMODE” status message to change the visible state of one or more of the screen control elements:

```
= <action>  
  <command>BROADCAST NIGHTMODE* </command>  
= <message>  
  <mtext>STATUS NIGHTMODE</mtext>  
  </message>  
</action>
```

Send “FEEDBACK ON” messages to the Radar, Lights, and Radio applications in response to the internally generated “P54 WINDOW SHOW” command:

```
= <action>  
  <command>P54 WINDOW SHOW</command>  
= <message>  
  <mdest>Radar</mdest>  
  <mtext>FEEDBACK ON</mtext>  
  </message>  
= <message>  
  <mdest>Lights</mdest>  
  <mtext>FEEDBACK ON</mtext>  
  </message>  
= <message>  
  <mdest>Radio</mdest>  
  <mtext>FEEDBACK ON</mtext>  
  </message>  
</action>
```

The <statusmessage> element in <action>

Each <action> element can have up to 32 <statusmessage> elements. Each <statusmessage> element defines the template for matching to incoming Project54 status messages.

```
= <statusmessage>  
  <msrc>Lights</msrc>  
  <mid>*</mid>  
  <mtext>WIG WAGS</mtext>  
  <value>1</value>  
</statusmessage>
```

The <msrc> element defines the source application for the matching Project 54 status message. The name specified is first referenced through the .\Project54\app_registry_name\Messaging registry key to determine the actual application messaging name. If no match is found, the <msrc> element value is assumed to be the actual messaging name of the application. If it is not important to match

the source application name, the * wildcard value could be used, or the <msrc> element could simply be omitted. The <msrc> wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The <mid> element defines the message ID string for the matching Project 54 status message. If it is not important to match the message ID string, the * wildcard value can be used as in the example, or the <mid> element could simply be omitted. The <mid> wildcard simply indicates a *don't care* condition and is unrelated to the wildcard processing described below.

The <mtext> element specifies the text of the message body for the matching Project 54 status message. The “STATUS” prefix is implied and should not be included (i.e. <mtext>WIG WAGS</mtext> matches the actual message text “STATUS WIG WAGS”).

The <mtext> element can also contain the * wildcard character at the end of the indicated message text. In such cases, the message is assumed to match if the incoming status message text matches everything in the template up to the *. The corresponding <value> element in <statusmessage> can reference this variable portion of the message text using the ? wildcard. In the <value> element, ? is replaced by the variable portion of the message text.

The <value> element specifies a string for identifying the corresponding <message> elements. When a matching status message is detected, the <action> element saves the <value> associated with that element. Subsequently, when a <command> element is matched, only <message> elements with <value> elements that match the saved value are actually sent. In this way, incoming <statusmessages> can control what messages are sent in response to commands. If no <message> elements have matching <value> elements, then no messages will be sent.

```
= <action>
  <command>DO IT</command>
= <message>
  <mdest>AnApp</mdest>
  <mtext>DO THING 1</mtext>
  <value>1</value>
  </message>
= <message>
  <mdest>AnApp</mdest>
  <mtext>DO THING 2</mtext>
  <value>2</value>
  </message>
= <message>
  <mdest>AnApp</mdest>
  <mtext>DO THING 3</mtext>
  <value>3</value>
  </message>
= <statusmessage>
  <msrc>AnApp</msrc>
  <mtext>MODE *</mtext>
  <value>?</value>
  </statusmessage>
```

```

- <statusmessage>
  <msrc>AnApp</msrc>
  <mtext>STOP SENDING MESSAGES</mtext>
  <value>0</value>
  </statusmessage>
</action>

```

In the above example, the “DO IT” command sends one of three different messages to AnApp, depending on the mode of operation most recently reported by AnApp: “STATUS MODE 1”, “STATUS MODE 2”, or “STATUS MODE 3”. If AnApp reports “STATUS STOP SENDING MESSAGES”, subsequent “DO IT” commands are ignored since no <message> elements have value 0.

The <extra> element

Every <device> can have one <extra> element. The <extra> element defines additional command strings that need to be added to the grammar for the application.

```

- <extra>
  <command>LIGHTS OFF</command>
  <command>SIREN OFF</command>
</extra>

```

The <extra> element can contain any number of <command> elements. These commands are added to the application grammar but have no other direct functionality.

The <screen> grammar

Every <screen> can have one <grammar> element. This is the name of the grammar file which is automatically loaded when this GUI is displayed. This grammar file is automatically generated by the XML GUI parser code from the XML script at run time. Any existing grammar file with the same name will be overwritten.

The command set for the grammar is created by parsing all <device> elements in the <screen>. A simple grammar list is compiled containing all <command> elements inside the <state> elements of <button> elements, and all <command> elements inside <extra> elements. <command> elements inside <action> elements are not added to the grammar.

Any command which has the format “P54 *rest_of_command*” is not added to the grammar. These represent silent internal commands. A button which issues only internal commands is automatically drawn using the “no speech command” color scheme. There are several such internal messages described in a subsequent section. The XML script can define any number of additional internal silent commands.

If a grammar file was created, a “GRAMMAR file_name” message to the Speechio application is automatically generated whenever the GUI is brought to the top of the Project54 GUI window stack.

Special <value> Processing

Normally, the <value> of a detected status message is compared to the <value> of each <state> element to determine which <state> to activate in response to the message. This is a simple string comparison, so that any matching string names will work in the <value> elements of <statusmessage> and <state> elements. Time efficiency in value comparison can be optimized by using 1 character <value> settings such as <value>0</value> or <value>A</value>. However, in some circumstances it may be advantageous to use more readable values such as <value>ok</value> versus <value>error</value>, or <value>on</value> versus <value>off</value>. Again, it is a general string comparison so any alpha-numeric characters can be used. It is generally not advisable to use non-alphanumeric characters in value strings since they may have special meaning to the XML parser.

The <value> associated with a <statusmessage> element can be specified using the * and ? wildcards. If the <mtext> element of a <statusmessage> contains a trailing * wildcard, it will match anything which matches the message prefix. In addition, the * and ? wildcards can be used in the <value> element to create variable <value> settings. In this context, * is equivalent to the entire input message text while ? is equivalent to only the variable part following the prefix.

```
<statusmessage>  
  <mtext>ADVISOR * </mtext>  
  <value>?</value>  
</statusmessage>
```

In the example above, if the incoming message was “STATUS ADVISOR TWO” the <value> element string would be “TWO”. It would thus select a <state> element that contained <value>TWO</value>.

There is also a special value processing that can occur using multi-character value settings. Each element that has member <state> and <statusmessage> elements actually keeps a current setting for its value setting. When a match occurs to a <statusmessage> element, the parent updates its current value string based on the <value> element of the <statusmessage>. This process is done on a character by character basis, in which a ‘%’ in any character position in the <value> element of the <statusmessage> means don’t change the current value in that character position. This allows for a level of logical processing in which the current value is the net result of recently received messages, rather than completely determined by the most recent message.

In the example below for a <textfield>, the status messages individually reflect the state of each of two different devices (on or off messages arrive for each device individually). However, the selected <state> of the <textfield> reflects the combined state of both devices.

```

= <state>
  <value>00</value>
  <text>NONE ON</text>
</state>

= <state>
  <value>01</value>
  <text>DEV 1 ON</text>
</state>

= <state>
  <value>10</value>
  <text>DEV 2 ON</text>
</state>

= <state>
  <value>11</value>
  <text>BOTH ON</text>
</state>

= <statusmessage>
  <msrc>AnApp</msrc>
  <mtext>DEVICE 1</mtext>
  <value>%1</value>
</statusmessage>

= <statusmessage>
  <msrc>AnApp</msrc>
  <mtext> DEVICE 1 OFF</mtext>
  <value>%0</value>
</statusmessage>

= <statusmessage>
  <msrc>AnApp</msrc>
  <mtext>DEVICE 2</mtext>
  <value>1%</value>
</statusmessage>

= <statusmessage>
  <msrc>AnApp</msrc>
  <mtext>DEVICE 2 OFF</mtext>
  <value>0%</value>
</statusmessage>

```

Special Internal Messages

There are two command messages expected by the XML GUI code which should be sent to the GUI by the application when appropriate. These built-in messages are needed to trigger implicit functions in the underlying GUI code. However, they can not be fielded explicitly in `<action>` elements within the XML description of the GUI in order to perform custom GUI operations.

1. The “SHOW WINDOW” command should be sent to the GUI by the application when the application desires that the window be brought to the top of the Project54 GUI window stack.
2. A message with blank message id and blank message text (“”) should be sent to the GUI by the application in order to force any prior changes in the window appearance be painted. Normally, changes in the visual states of GUI elements are recorded as status messages are processed, but these visual changes are not actually painted until a subsequent update message is received.

There are two internal command messages which are generated automatically by the XML GUI code and which are returned to the application via the *MessageCallback* procedure of the *IUNH_XmlGuiCallback* interface. These built-in messages can also be fielded explicitly in `<action>` elements within the XML description of the GUI in order to perform custom GUI operations.

1. The “P54 WINDOW SHOW” command is issued to the application (and to the GUI) when the window is brought to the top of the Project54 GUI window stack. This is usually a good time at which to send “FEEDBACK ON” messages to other applications (see the example in the section on `<action>` elements) and to put in place the window grammar.

2. The “P54 WINDOW HIDE” command is issued to the application (and to the GUI) when the window is taken off of the top of the Project54 GUI window stack. This is usually a good time at which to send “FEEDBACK OFF” messages to other applications.

Finally, the “STATUS P54 TIME *time_of_day*” status message and the “STATUS P54 DATE *today's_date*” status message are both automatically issued to the GUI once each second, if the XML script for the GUI includes a `<statusmessage>` element which matches either of them. The example below puts the time of day in a `<textfield>`.

```
= <textfield x="0" y="10" id="TIME" >
  <fontsize>1</fontsize>
  <width>10</width>
= <state>
  <text>?</text>
  </state>
= <statusmessage>
  <mtext>P54 TIME *</mtext>
  </statusmessage>
</textfield>
```

Standard Registry Settings

.\Project54\app_registry_name\Messaging

Each variable in this registry key maps a logical name for an application into an actual messaging name (similar to most other Project54 applications). The only required logical name mappings are “self” and “Speechio”. Note that “self” refers to the actual application and must indicate the messaging name for the application as provided in the *.\Project54\AppManager\Applist* registry key. “Speechio” of course must indicate the messaging name of the application which controls speech recognition and synthesis.

All other logical names must be as specified in the `<statusmessage>` and `<message>` elements of the XML scripts for the GUI.

Note each registry setting referenced is resolved only once at the time that the GUI is created (any subsequent registry changes are ignored).

Custom Registry Settings

XML GUI scripts can directly reference registry parameters, making it possible to define GUI interfaces which can be configured via registry settings. Any XML element or attribute value can be set to defer to the value of a registry variable. The simple format for referencing registry values is as follows:

```
@subkey\variable
@key\subkey\variable
```

Here, @ indicates it is a registry reference, *key* is the name of the registry key (relative to `.\Project54\`), *subkey* is the name of the registry subkey (relative to `.\Project54\key\`) and *variable* is the registry variable name. If the *key* is not included, it is assumed to be *app_registry_name* (the registry path is `.\Project54\ app_registry_name\ subkey`). For example, the `<button>` element below gets its command from the registry variable “B11Command” and gets its label from the registry variable “B11Label”, both in the registry path `.\Project54\app_registry_name\Buttons\`.

```
= <button col="1" row="1">
= <state>
= <command>@Buttons\B11Command</command>
  <text>@Buttons\B11Label</text>
  </state>
</button>
```

In the next example, the `<button>` element gets its command from the registry variable “B11Command” and gets its label from the registry variable “B11Label”, both in the registry path `.\Project54\NewPScreen\Buttons`.

```
= <button col="1" row="1">
= <state>
= <command>@NewPScreen\Buttons\B11Command</command>
  <text>@ NewPScreen\Buttons\B11Label</text>
  </state>
</button>
```

In the following example, several “extra” speech commands are loaded from the registry:

```
= <extra>
  <command>@Speech\Ex1</command>
  <command>@Speech\Ex2</command>
  <command>@Speech\Ex3</command>
  <command>@Speech\Ex4</command>
  <command>@Speech\Ex5</command>
  <command>@Speech\Ex6</command>
  <command>@Speech\Ex7</command>
  <command>@Speech\Ex8</command>
  <command>@Speech\Ex9</command>
</extra>
```

Note that if the registry variable does not exist, the substitution results in a blank string. Thus, in the above example, the registry could be set to define from 0 to 9 additional speech commands since blank commands are not added to the grammar.

The full format for referencing registry values is as follows:

```
@subkey\variable:nn
@key\subkey\variable:nn
```

This format is the same as the simple format except that in this case it is assumed that the registry variable contains multiple subvalues separated by commas (value_1, value_2, ...) or semi-colons (value_1; value_2; ...). In this format, *nn* represents an integer index for the subvalue (starting at 1).

```
= <button col="@PScreen\Buttons\BtnSpecs:1"
      row="@PScreen\Buttons\BtnSpecs:2">
= <state>
  <command>@PScreen\Buttons\BtnSpecs:3</command>
  <text>@PScreen\Buttons\BtnSpecs:4</text>
</state>
</button>
```

In the above example, the `<button>` element gets its location, command and label from the “BtnSpecs” registry variable, which must have the format:

col_m,row_n,command_text,label_text (e.g. 1,1,MAIN SCREEN,Main\nScreen)

or

col_m;row_n;command_text;label_text (e.g. 1;1;MAIN SCREEN;Main\nScreen)

If *nn* is zero, the reference is to the variable name itself. Thus, the reference

```
<text>@PScreen\Buttons\BtnSpecs:0</text>
```

is the same as:

```
<text>BtnSpecs</text>
```

which isn't very useful. However, the variable name can be replaced by an *integer value*, in which case the reference is to a variable in the *subkey* selected by its position. For example, the reference:

```
<text>@PScreen\Buttons\2:0</text>
```

returns the name of the second variable in the `.\Project54\PScreen\Buttons` registry path. The reference:

```
<text>@PScreen\Buttons\2 </text>
```

returns the value of the second variable in the .\Project54\PScreen\Buttons registry path. The reference:

```
<text>@PScreen\Buttons\2:3 </text>
```

returns the third subvalue (comma or semi-colon separated) of the second variable in the .\Project54\PScreen\Buttons\ registry path.

The following example adds extra speech commands to the vocabulary by enumerating the first 10 registry variables. In this case, the commands are in the variable values and the variable names are not referenced explicitly.

```
= <extra>  
  <command>@PScreen\ExtraSpeech\1 </command>  
  <command>@PScreen\ExtraSpeech\2 </command>  
  <command>@PScreen\ExtraSpeech\3 </command>  
  <command>@PScreen\ExtraSpeech\4 </command>  
  <command>@PScreen\ExtraSpeech\5 </command>  
  <command>@PScreen\ExtraSpeech\6 </command>  
  <command>@PScreen\ExtraSpeech\7 </command>  
  <command>@PScreen\ExtraSpeech\8 </command>  
  <command>@PScreen\ExtraSpeech\9 </command>  
  <command>@PScreen\ExtraSpeech\10 </command>  
</extra>
```

The following <buttonset> has 2 buttons defined by two registry variables in the .\Project54\app_registry_name\Buttons\ registry path:

```
= <buttonset>  
  = <button col="@Buttons\1:1"  
    row="@Buttons\1:2">  
    = <state>  
      <command>@Buttons\1:3 </command>  
      <text>@Buttons\1:4 </text>  
    </state>  
  </button>  
  = <button col="@Buttons\2:1"  
    row="@Buttons\2:2">  
    = <state>  
      <command>@Buttons\2:3 </command>  
      <text>@Buttons\2:4 </text>  
    </state>  
  </button>  
</buttonset>
```

Note each registry setting referenced is resolved only once at the time that the GUI is created (any subsequent registry changes are ignored).

Appendix A: XML Script for the WhelenSerial Lights and Siren GUI

```
<screen id="Whelen Serial">
<title>Project54: XML Emergency Signals</title>
<grammar>whelen_serial.txt</grammar>
<device x="0" y="0" id="SCREEN BASE">
<buttonset>
<button col="1" row="1" id="MAIN SCREEN">
<state><command>MAIN SCREEN</command><text>Main\nScreen</text></state></button>
<button col="2" row="1" id="PATROL SCREEN">
<state><command>PATROL SCREEN</command><text>Patrol\nScreen</text></state></button>
<button col="3" row="1" id="LIGHTS AND SIREN"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>LIGHTS AND SIREN</command><text>Lights\n+ Siren</text></state>
<state><value>1</value><onoff>1</onoff><command>LIGHTS AND SIREN OFF</command></state>
<statusmessage><mtext>LIGHTS AND SIREN</mtext><value>1</value></statusmessage>
<statusmessage><mtext>LIGHTS AND SIREN OFF</mtext><value>0</value></statusmessage>
</button>
</buttonset>
<lightset x="88" y="68" id="CONTROL HEAD">
<light x="0" y="0" ><text>CH</text><textcolor>WHITE</textcolor><oncolor>BLUE</oncolor><offcolor>RED</offcolor></light>
<state><value>0</value><onoff>0</onoff></state>
<state><value>1</value><onoff>1</onoff></state>
<statusmessage><mtext>LIGHTS CONTROL HEAD</mtext><value>1</value></statusmessage>
<statusmessage><mtext>LIGHTS CONTROL HEAD OFF</mtext><value>0</value></statusmessage>
<label x="-2" y="1"><text>Control Head</text><fontsize>1</fontsize><font>BOLDITALIC</font><just>rt</just></label>
</lightset>
<action>
<command>MAIN SCREEN</command>
<message><mdest>Appmanager</mdest><mtext>SHOW WINDOW</mtext></message>
</action>
<action>
<command>PATROL SCREEN</command>
<message><mdest>PScreen</mdest><mtext>SHOW WINDOW</mtext></message>
</action>
<label x="50" y="94"><text>Whelen MPC01 Lights and
Siren</text><fontsize>2</fontsize><font>BOLDITALIC</font><just>cb</just></label>
<extra>
<command>EMERGENCY SIGNALS</command>
</extra>
</device>
<device x="0" y="0" id="LIGHTS">
<buttonset>
<button col="1" row="2" id="FRONT STROBES"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>FRONT STROBES</command><text>Front\nStrobes</text></state>
<state><value>1</value><onoff>1</onoff><command>FRONT STROBES OFF</command></state>
<statusmessage><mtext>FRONT STROBES</mtext><value>1</value></statusmessage>
<statusmessage><mtext>FRONT STROBES OFF</mtext><value>0</value></statusmessage>
</button>
```

```

<button col="2" row="2" id="REAR STROBES"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>REAR STROBES</command><text>Rear\nStrobes</text></state>
<state><value>1</value><onoff>1</onoff><command>REAR STROBES OFF</command></state>
<statusmessage><mtext>REAR STROBES</mtext><value>1</value></statusmessage>
<statusmessage><mtext>REAR STROBES OFF</mtext><value>0</value></statusmessage>
</button>

```

```

<button col="3" row="2" id="STROBES"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>STROBES</command><text>Strobes</text></state>
<state><value>1</value><onoff>1</onoff><command>STROBES OFF</command></state>
<statusmessage><mtext>STROBES</mtext><value>1</value></statusmessage>
<statusmessage><mtext>STROBES OFF</mtext><value>0</value></statusmessage>
</button>

```

```

<button col="@GUIButtons\WigWags:1" row="@GUIButtons\WigWags:2" id="WIG WAGS"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>WIG WAGS</command><text>Wig\nWags</text></state>
<state><value>1</value><onoff>1</onoff><command>WIG WAGS OFF</command></state>
<statusmessage><mtext>WIG WAGS</mtext><value>1</value></statusmessage>
<statusmessage><mtext>WIG WAGS OFF</mtext><value>0</value></statusmessage>
</button>

```

```

<button col="@GUIButtons\TakeDowns:1" row="@GUIButtons\TakeDowns:2" id="TAKE DOWNS"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>TAKE DOWNS</command><text>Take\nDowns</text></state>
<state><value>1</value><onoff>1</onoff><command>TAKE DOWNS OFF</command></state>
<statusmessage><mtext>TAKE DOWNS</mtext><value>1</value></statusmessage>
<statusmessage><mtext>TAKE DOWNS OFF</mtext><value>0</value></statusmessage>
</button>

```

```

<button col="@GUIButtons\RearFloods:1" row="@GUIButtons\RearFloods:2" id="REAR FLOODS"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>REAR FLOODS</command><text>Rear\nFloods</text></state>
<state><value>1</value><onoff>1</onoff><command>REAR FLOODS OFF</command></state>
<statusmessage><mtext>REAR FLOODS</mtext><value>1</value></statusmessage>
<statusmessage><mtext>REAR FLOODS OFF</mtext><value>0</value></statusmessage>
</button>

```

```

<button col="@GUIButtons\LeftAlley:1" row="@GUIButtons\LeftAlley:2" id="LEFT ALLEY"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>LEFT ALLEY</command><text>Left\nAlley</text></state>
<state><value>1</value><onoff>1</onoff><command>LEFT ALLEY OFF</command></state>
<statusmessage><mtext>LEFT ALLEY</mtext><value>1</value></statusmessage>
<statusmessage><mtext>LEFT ALLEY OFF</mtext><value>0</value></statusmessage>
</button>

```

```

<button col="@GUIButtons\RightAlley:1" row="@GUIButtons\RightAlley:2" id="RIGHT ALLEY"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>RIGHT ALLEY</command><text>Right\nAlley</text></state>
<state><value>1</value><onoff>1</onoff><command>RIGHT ALLEY OFF</command></state>
<statusmessage><mtext>RIGHT ALLEY</mtext><value>1</value></statusmessage>
<statusmessage><mtext>RIGHT ALLEY OFF</mtext><value>0</value></statusmessage>
</button>

```

```

<button col="@GUIButtons\HighLow:1" row="@GUIButtons\HighLow:2" id="HIGH LOW"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>HIGH LOW</command><text>High\nLow</text></state>
<state><value>1</value><onoff>1</onoff><command>HIGH LOW OFF</command></state>
<statusmessage><mtext>HIGH LOW</mtext><value>1</value></statusmessage>
<statusmessage><mtext>HIGH LOW OFF</mtext><value>0</value></statusmessage>
</button>

```

```

<button col="@GUIButtons\RapidRate:1" row="@GUIButtons\RapidRate:2" id="RAPID RATE"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>RAPID RATE</command><text>Rapid\nRate</text></state>
<state><value>1</value><onoff>1</onoff><command>RAPID RATE OFF</command></state>
<statusmessage><mtext>RAPID RATE</mtext><value>1</value></statusmessage>
<statusmessage><mtext>RAPID RATE OFF</mtext><value>0</value></statusmessage>
</button>

```

```

</buttonset>

```

```

<lightset x="20" y="5" id="WIG WAGS">
<light x="8" y="0" ><text>W</text><textcolor>BLACK</textcolor><oncolor>WHITE</oncolor><offcolor>DEFAULT</offcolor></light>
<light x="56" y="0" ><text>W</text><textcolor>BLACK</textcolor><oncolor>WHITE</oncolor><offcolor>DEFAULT</offcolor></light>
<state><value>0</value><onoff>00</onoff></state>

```



```

<statusmessage><mtext>STROBES OFF</mtext><value>0</value></statusmessage>
<label x="-5" y="0"><text>R</text><fontsize>2</fontsize><font>BOLD</font><just>rt</just></label>
</lightset>

<extra>
<command>LIGHTS OFF</command>
</extra>

</device>

<device x="0" y="0" id="SIREN">

<buttonset>

<button col="3" row="4" id="RADIO MODE">
<state><value>1</value><command>MANUAL MODE</command><text>Manual\nMode</text></state>
<state><value>2</value><command>HANDS FREE MODE</command><text>HF\nMode</text></state>
<state><value>3</value><command>RADIO MODE</command><text>Radio\nMode</text></state>
<state><value>4</value><command>P A MODE</command><text>PA\nMode</text></state>
<statusmessage><mtext>SIRENMODE Manual</mtext><value>2</value></statusmessage>
<statusmessage><mtext>SIRENMODE HF</mtext><value>3</value></statusmessage>
<statusmessage><mtext>SIRENMODE Radio</mtext><value>4</value></statusmessage>
<statusmessage><mtext>SIRENMODE PA</mtext><value>1</value></statusmessage>
<statusmessage><mtext>SIRENMODE Wail</mtext><value>4</value></statusmessage>
<statusmessage><mtext>SIRENMODE Yelp</mtext><value>4</value></statusmessage>
<statusmessage><mtext>SIRENMODE Piercer</mtext><value>4</value></statusmessage>
<statusmessage><mtext>SIRENMODE Off</mtext><value>1</value></statusmessage>
</button>

<button col="1" row="5" id="WAIL"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>WAIL</command><text>Wail</text></state>
<state><value>1</value><onoff>1</onoff><command>WAIL OFF</command></state>
<statusmessage><mtext>WAIL</mtext><value>1</value></statusmessage>
<statusmessage><mtext>WAIL OFF</mtext><value>0</value></statusmessage>
</button>

<button col="2" row="5" id="YELP"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>YELP</command><text>Yelp</text></state>
<state><value>1</value><onoff>1</onoff><command>YELP OFF</command></state>
<statusmessage><mtext>YELP</mtext><value>1</value></statusmessage>
<statusmessage><mtext>YELP OFF</mtext><value>0</value></statusmessage>
</button>

<button col="3" row="5" id="PIERCER"><behavior>sticky</behavior>
<state><value>0</value><onoff>0</onoff><command>PIERCER</command><text>Piercer</text></state>
<state><value>1</value><onoff>1</onoff><command>PIERCER OFF</command></state>
<statusmessage><mtext>PIERCER</mtext><value>1</value></statusmessage>
<statusmessage><mtext>PIERCER OFF</mtext><value>0</value></statusmessage>
</button>

<button col="2" row="6" id="AIR HORN">
<state><value>0</value><onoff>0</onoff><command>AIR HORN</command><text>Air\nHorn</text></state>
<state><value>1</value><onoff>1</onoff><command>AIR HORN OFF</command></state>
<statusmessage><mtext>AIR HORN</mtext><value>1</value></statusmessage>
<statusmessage><mtext>AIR HORN OFF</mtext><value>0</value></statusmessage>
</button>

<button col="3" row="6" id="MANUAL SIREN">
<state><value>0</value><onoff>0</onoff><command>P54 MANUAL SIREN</command><text>Manual\nSiren</text></state>
<state><value>1</value><onoff>1</onoff><command>P54 MANUAL SIREN OFF</command></state>
<statusmessage><mtext>MANUAL SIREN</mtext><value>1</value></statusmessage>
<statusmessage><mtext>MANUAL SIREN OFF</mtext><value>0</value></statusmessage>
</button>

</buttonset>

<textfield x="45" y="47" id="SIREN">
<fontsize>2</fontsize><width>7</width>
<state><value>0</value><text>?</text><textcolor>DEFAULT</textcolor></state>
<state><value>1</value><text>Wail</text><textcolor>RED</textcolor></state>

```

```
<state><value>2</value><text>Yelp</text><textcolor>RED</textcolor></state>
<state><value>3</value><text>Piecer</text><textcolor>RED</textcolor></state>
<statusmessage><mtext>SIRENMODE Wail</mtext><value>1</value></statusmessage>
<statusmessage><mtext>SIRENMODE Yelp</mtext><value>2</value></statusmessage>
<statusmessage><mtext>SIRENMODE Piecer</mtext><value>3</value></statusmessage>
<statusmessage><mtext>SIRENMODE *</mtext><value>0</value></statusmessage>
<label x="-3" y="1"><text>Siren</text><fontsize>2</fontsize><font>BOLD</font><just>rt</just></label>
</textfield>
```

```
<extra>
<command>SIREN OFF</command>
</extra>
```

```
</device>
```

```
</screen>
```