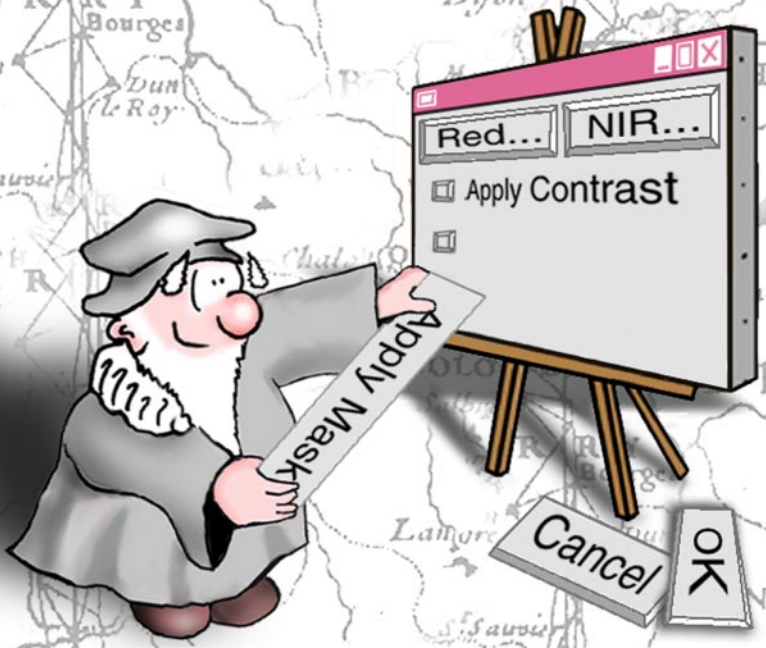
SML DIALOGS

Tutorial

Building Dialogs in SML



in

TNTmips®

TNTedit™

TNTview®

Before Getting Started

For SML scripts that process multiple input objects or require user input for numerous parameter settings, consider creating custom dialog windows to streamline user interaction with the script. Dialog setup is streamlined by using a simple XML-based dialog specification. This booklet shows you how to construct custom dialogs and how to use them with an SML script.

Prerequisite Skills This booklet assumes that you have completed the exercises in the *Displaying Geospatial Data*, *TNT Product Concepts*, and *Writing Scripts with SML* tutorial booklets. Those exercises introduce essential skills and basic techniques that are not covered again here. Please consult those booklets and the TNTmips Reference Manual for any review you need.

Sample Data The exercises presented in this booklet use sample data that is distributed with the TNT products. If you do not have access to a TNT products DVD, you can download the data from the MicroImages web site. In particular, this booklet uses sample files and scripts in the SMLDLG, SCRIPTING, and CB_DATA sample data collections.

More Documentation This booklet is intended only as an introduction to creating dialog windows in SML scripts. The Reference Guide to SML Dialog Specifications in XML is available from the MicroImages web site and is also included as an appendix in electronic versions of this tutorial booklet.

More Documentation This booklet is intended only as an introduction to the TNT Geospatial Scripting Language. Descriptions of sample scripts can be found in a variety of Technical Guides that are available from MicroImages' web site.

TNTmips Pro, Basic, and Free TNTmips comes in three versions: TNTmips Pro (which requires a software license key), low-cost TNTmips Basic, and TNTmips Free. TNTmips Basic and TNTmips Free provide nearly all the capabilities of TNTmips Pro but limit the size of the geospatial objects and attribute tables that can be used in your project. TNTmips Basic and TNTmips Free allow you to create and use scripts in the Display process (database queries, style scripts, macroscripts, and others) and the Geoformula process, but only TNTmips Pro allows you to create and run complex standalone geospatial scripts.

Randall B. Smith, Ph.D., 5 January 2012
©MicroImages, Inc., 2003- 2012

You can print or read this booklet in color from MicroImages' Web site. The Web site is also your source for the newest Tutorial booklets on other topics. You can download an installation guide, sample data, and the latest version of TNTmips.

http://www.microimages.com

Welcome to Building Dialogs in SML

You can use the geospatial scripting language (SML) in the TNT products to write many types of custom programs that operate on the geospatial data objects in your TNT Project Files. If you are writing a script for a one-time processing operation, the processing parameters and names of input and output objects and files can be written explicitly into the script. But any script you plan to reuse or provide to others should include interactive dialogs to let the user select objects and enter program parameters.

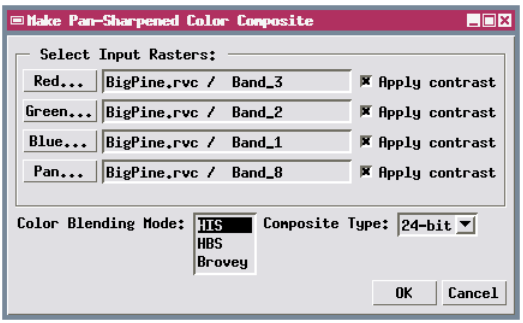
SML provides several ways for you to include interactive dialogs in your scripts. The Raster, Vector, CAD, TIN, and File function groups each include GetInput...() , GetOutput...(), and other functions that pop up a dialog so the user can select or create objects and files as needed as the script is executed. The functions in the Popup Dialog group provide dialogs that prompt the user to enter numeric and other types of parameter values. These functions are easy to use, but each opens a separate transient dialog window, so a complex processing script might require a barrage of popup dialogs.

You can make complex scripts easier to use by creating one or more custom dialog windows that bring together object selection and parameter inputs. Custom dialog windows in SML can include text and icon pushbuttons, toggle and radio buttons, text and numeric fields, labels, menu buttons, listboxes, and comboboxes, among others. This booklet provides an introduction to building custom dialog windows for use in your SML scripts. It provides many sample dialog scripts and several complete scripts that incorporate complex custom dialog windows.

STEPS

☑ choose Script / Edit Script... from the TNTmips main menu to open the Spatial Manipulation Language editor window

Dialog specifications in XML are introduced on pages 4-15 with examples of all basic dialog elements. Pages 16-20 describe how to set up the script to process the dialog specification and open the dialog and the optional use of an XML Editor. Techniques for wiring the dialog controls to script actions are described on pages 21-27. Pages 28-31 provide examples of complex dialog windows and dialogs with views. Use of a drawing canvas, dialog localization, and use of modeless dialogs in event-driven scripts are coveredon pages 32-35.



An example of a custom dialog window. We will examine this dialog and its components in more detail on a later page.

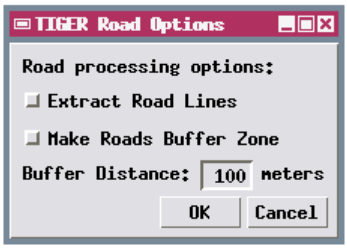
Dialogs from XML Text

Custom dialogs in SML make use of classes in the User Interface (GUI) class group shown in the Script Reference window. This group includes class GUI_DLG, which represents a dialog window, and a class for each type of layout element and control that can be used in a dialog window. To use a custom dialog with an SML script, you must construct the dialog by designating what controls it contains and how they are arranged in the dialog. You must also "wire-up" the dialog by incorporating code in your script to get the values and settings the script user designates in the dialog and to specify what actions should be taken by the script when particular controls (such as a dialog's OK button) are activated.

In order to simplify the process of laying out a dialog window, SML lets you create and arrange dialog components using XML-formatted text, which we will refer to as a *dialog specification*. The dialog specification can be embedded in the script as a string variable or read in from a separate XML file. The dialog specification is interpreted by SML and used to set up the GUI_DLG class and the component controls your script will use.

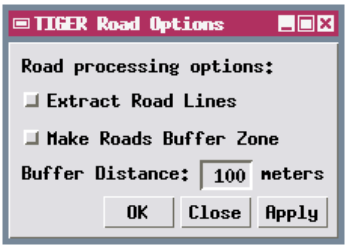
The first part of this booklet focuses on using XML to set up a dialog specification using different types of dialog controls. We will then discuss how to use the specification in a script, how to open the dialog and obtain settings from it, and how to use the dialog to initiate script actions.

Dialog windows you create with a dialog specification are automatically supplied with pushbuttons at the bottom of the window (such as an OK button) that you can use to initiate script actions. You can open a dialog in either modal or modeless form, which are described and illustrated below.



Modal Dialog

A modal dialog suspends other script operations as long as the dialog is open. It is automatically supplied with OK and Cancel buttons. A modal dialog is typically used in a standalone script to gather all inputs and process settings prior to running the script process.



Modeless Dialog

A modeless dialog can remain open while other script operations take place. A modeless dialog is automatically supplied with OK, Close, and Apply buttons. A modeless dialog is required in event-driven scripts such as tool scripts and display control scripts.

A Simple Dialog Specification in XML

XML is a form of structured text that makes use of *tags* (enclosed between < and > characters) to delimit and identify *elements* of text data. In an SML dialog specification, the data elements identified by the XML tags correspond to specific components of a dialog window, such as buttons and labels. Tags occur in pairs, with a *start tag* and *end tag* enclosing the relevant text.

As an example, look at the dialog specification for the Hello World dialog window shown below. This modal dialog window consists of a *label* element and two pushbuttons (OK and Cancel, which are supplied automatically). The two <label> tags enclose and identify the text to be used for the label on the dialog. Note that start and end tags use the same tag name, but in the end tag it is preceded by the forward slash (/) character.

STEPS

- ☑ choose File / Open / *.SML File..., navigate to the SMLDLG directory, and select GENDLG.SML.
- ☑ run the script
- ☑ when prompted to select the dialog specification, select HELLO.XML in the SMLDLG directory and click [OK]
- ☑ compare the dialog window to its specification
- ☑ keep the Hello World dialog open and proceed to the next page

```
<?xml version="1.0"?>
<root>
  <dialog id="hello" Title="Hello World">
    <label>Sample Dialog Window</label>
  </dialog>
</root>
```

Data elements in XML can be nested inside other data elements (in one or more levels) to indicate membership in a group. This is a good match to the structure of a dialog window, in which some window components are contained within other components. In this example the dialog element (which identifies the dialog window as a whole) contains the label element. The dialog is in turn contained within the root element, which is the required top-level element in any dialog specification.

For ease of editing and reading a dialog specification, the start and end tags for “container” dialog elements should be placed on separate text lines, with the contained elements identified on intervening lines. Different levels of indents should also be used to clarify this nested, tree-like structure.

XML, the Extensible Markup Language, is a generic meta-language that allows the creation of structured, self-describing text. XML supplies a structure and syntax, but a specific set of tags and attributes is required to define the meaning of the elements in an XML file. MicroImages has created such a set of tags and attributes to identify the various types of dialog components and their properties. SML interprets these tags in order to create your dialog window.

Using Attributes for Tags

STEPS
- ☑ examine the attributes for the elements in the dialog specification
- ☑ click [OK] on the Hello World dialog window to end the script

Start-tags can also include one or more *attributes*, which are predefined keywords to which you can assign a value. The assignment has the form:

```
attribute = "value"
```

In a dialog specification, each type of dialog component has a predefined set of attributes. Each time you use that component type, you can assign attribute values to define the specific properties you want that component to have. Attribute names are case-sensitive, but you can list attributes in any order within the start-tag.

Any dialog component can have an *id* attribute, a unique identifier for that element. The value of an id attribute can be any character string, but it must be unique within the dialog. Use an *id* attribute for the dialog element and any other element that your script needs to access, such as to set or read values.

To conform to XML format standards, the dialog specification should begin with a standard XML Declaration specifying the version of XML.

```
<?xml version="1.0"?>
<root>
  <dialog id="hello" Title="Hello World">
    <label>Sample Dialog Window</label>
  </dialog>
</root>
```

In the Hello World dialog, the dialog element has an *id* attribute that the script uses to create and open the actual dialog window. It also has a *Title* attribute that supplies the text for the window title. The label element in the dialog could also have an *id* attribute, but here it doesn't need one. The label text is simply a static part of the dialog, so there is no need for the script to have any interactive access to it. Therefore the *id* attribute is omitted.

In order for SML to correctly interpret the dialog specification, it must follow the simple syntax rules that define a *well-formed* XML document:
- every element must have both start and end tags
- all attribute values must be in quotes
- elements may not overlap

In the next few pages we will examine several additional dialog windows and their specifications in XML to illustrate the use of other dialog components.

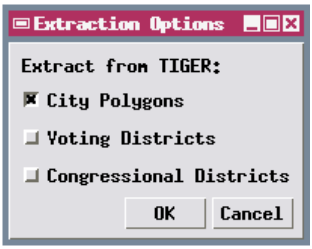
The *Reference Guide to SML Dialog Specifications in XML* contains complete lists of the attributes available for use with each type of dialog element and guidelines for using dialog specifications in SML scripts. It is available from the MicroImages web site and is appended to PDF versions of this booklet.

Togglebuttons and Empty Tags

The Extraction Options sample dialog window has a label and three *togglebuttons*. Each of these components is represented by an element within the dialog element in the dialog specification. Note that these components are laid out from top to bottom in the window, rather than from left to right. This is the default layout order for components in a dialog. Later we will see how you can control the layout orientation.

STEPS
- ☑ run the GENDLG script again
- ☑ when prompted to select the dialog specification, select TIGEROP.XML in the SMLDLG directory and click [OK]
- ☑ compare the dialog window to its specification
- ☑ click [OK] on the Extraction Options dialog window to end the script

Each togglebutton element in this example has three attributes: *id*, *Name*, and *Selected*. The *id* attribute is needed so that the SML script can find out whether each togglebutton has been set or not. The *Name* attribute specifies text for the label that is automatically placed to the right of the button. The *Selected* attribute lets you set the default state for each button when the dialog window opens. The button is pushed in (on) if you set *Selected* = "true" and pushed out (off) if you set *Selected* = "false".




```
<?xml version="1.0"?>
<root>
  <dialog id="tigerop" Title="Extraction Options">
    <label>Extract from TIGER:</label>
    <togglebutton id="citybtn" Name="City Polygons"
       Selected="true"/>
    <togglebutton id="votebtn" Name="Voting Districts"
       Selected="false"/>
    <togglebutton id="congbtn" Name="Congressional Districts"
       Selected="false"/>
  </dialog>
</root>
```

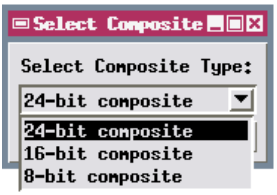
Note that all of the information required to set up a togglebutton is included in the start tag name and its attributes. Unlike the label element, there is no other text to be enclosed by separate start and end tags. In this case the two tags can be combined into a single *empty tag*. The slash character (/) that normally begins an end tag is placed before the closing > character of an empty tag.

The dialog specifications shown in this booklet are color-coded for ease of reading. Standard XML formatting characters are in blue, tag names are in purple or red, attribute keywords are in green, and attribute values and other text are in black.

Combobox and Items

STEPS

- ☑ run the GENDLG script again
- ☑ when prompted to select the dialog specification, select SELCOMP.XML in the SMLDLG directory and click [OK]
- ☑ compare the dialog window to its specification
- ☑ click [OK] on the Select Composite dialog window to end the script



You can use the TNTmips Text File Editor or any other text editor to open and examine the dialog specification XML files used in these exercises.

The Select Composite sample dialog window includes a label and a type of menu control called a *combobox*. A recessed box resembling an editable text field shows the current menu selection, and a small icon at the right side of the control is used to drop down the menu choices.

The possible selections for any menu-type control (combobox, listbox, and menubutton) are created in a dialog specification by adding *item* elements within the parent control element. In this example the combobox control element includes three item elements that specify different types of color composite raster. An item element resembles a label element in that its start and end tags enclose the text for the menu entry. However, an item element also has a *Value* attribute. The text string you assign to this attribute becomes the "value" of the combobox control when that menu item is selected. Each item in a menu control therefore should have a different character string assigned for its value. The item value strings in this example are numbers corresponding to the bit-depths of the composites.

Menu-type controls that show the selected menu item (combobox and listbox) have a *Default* attribute that you can use to indicate which item should be the default selection. Simply assign that item's value string as the value of the *Default* attribute. As an alternative, you can use the item attribute *Selected*, which is either "true" or "false". If *Selected*="true", that item is initially selected (overriding the Default attribute of the parent control, if any).

```
<?xml version="1.0"?>
<root>
 <dialog id="selcomp" Title="Select Composite">
  <label>Select Composite Type:</label>
  <combobox id="comptype" Default="24">
   <item Value="24">24-bit composite</item>
   <item Value="16">16-bit composite</item>
   <item Value="8">8-bit composite</item>
  </combobox>
 </dialog>
</root>
```

Radiogroup and Groupbox

The Select Process sample dialog window includes a hybrid type of control called a *radiogroup*. A radiogroup is a group of small buttons (resembling togglebuttons) in which only one button can be turned on at a time. Turning on one button automatically turns off any other button that was previously turned on. So a radiogroup functions like a menu in which all selections are constantly visible.

The specification for a radiogroup has the same structure as the other menu-type controls. The item elements inside the radiogroup element provide the names for the buttons (analagous to the entries in a menu) and the value for the control when that button is turned on. This example shows an alternative method for specifying the text string for a menu item. In the example on the previous page, the item name was provided as text between start and end tags. In the Select Process dialog, the items are specified using empty tags with a *Name* attribute. Either method can be used for any menu-type control.

This dialog also includes a *groupbox*, a simple rectangular frame around one or more other controls, which can also be automatically provided with a label inside the top edge using a *Name* attribute. A groupbox is one type of *layout* component in a dialog. A layout component does not provide any program control, but merely aids in the layout of the dialog window. Other layout components are shown in later sample dialogs.

STEPS

- ☑ run the GENDLG script again
- ☑ when prompted to select the dialog specification, select RADIOGP.XML in the SMLDLG directory and click [OK]
- ☑ compare the dialog window to its specification
- ☑ click [OK] on the Select Process dialog window to end the script

In the sample dialog specifications in this booklet, tags for control components are shown in red and tags for layout and main-level dialog elements are shown in purple.

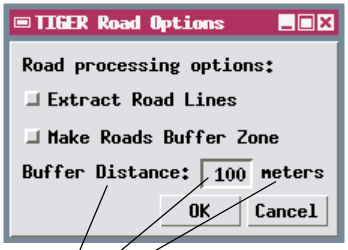
```
<?xml version="1.0"?>
<root>
 <dialog id="radiogp" Title="Select Process">
  <groupbox Name=" Process: " ExtraBorder="4">
   <radiogroup id="processgp" Default="entire">
    <item Value="entire" Name="Entire Scene"/>
    <item Value="polygon" Name="AOI Polygon"/>
    <item Value="manual" Name="Manual Selection"/>
   </radiogroup>
  </groupbox>
 </dialog>
</root>
```

Numeric Fields and Layout Panes

STEPS

- ☑ run the GENDLG script again
- ☑ when prompted to select the dialog specification, select TIGRDS.XML in the SMLDLG directory and click [OK]
- ☑ compare the dialog window to its specification
- ☑ click [OK] on the TIGER Road Options dialog window to end the script

The TIGER Road Options sample dialog window introduces the layout element you will probably use most often, the *pane*. A layout pane is simply an invisible container in which you can place multiple dialog components and control their arrangement. The most important attribute of a pane element is *Orientation*, which can be either "horizontal" or "vertical". The three components near the bottom of this dialog window (two labels surrounding an editable numeric field) are placed within a pane with horizontal orientation, so the components are arranged in order from left to right across the pane. If the pane orientation is set to "vertical", the elements within the pane are arranged from top to bottom.



An invisible layout pane with horizontal orientation contains these dialog components.

The numeric field is created using an *editnumber* element in the dialog specification. The *Width* attribute sets the width in typical characters, while the *Precision* attribute sets the number of digits after the decimal point (the default is 6). The *Default* and *MinVal* attributes set the value initially displayed in the field and the minimum allowed value, respectively. There is also an available *MaxVal* attribute to set the maximum allowed value. This control has no built-in label, so the dialog uses a label element to the left of the field to label its purpose and another to the right to identify the units.

```
<?xml version="1.0"?>
<root>
  <dialog id="tigrds" Title="TIGER Road Options">
    <label>Road processing options:</label>
    <togglebutton id="getrds" Name="Extract Road Lines"/>
    <togglebutton id="mkbuf" Name="Make Roads Buffer Zone"/>
    <pane Orientation="horizontal">
      <label>Buffer Distance:</label>
      <editnumber id="buffdist" Width="4" Precision="0"
        Default="100" MinVal="0"/>
      <label>meters</label>
    </pane>
  </dialog>
</root>
```

Pushbuttons and Listbox

The Languages sample dialog window illustrates the use of *pushbuttons* and a *listbox*. The upper pane of the dialog includes two pushbuttons, one with a text label and one with an icon. Use the *Name* attribute for the pushbutton tag to specify the text for the button label or the *Icon* attribute to specify the name of the icon to be used. The icon name must match an *iconid* listed in the internal TNT reference files. To see a complete list of the valid *iconids*, open the Script Reference window from the SML editor and select the GUI_CTRL_TOGGLEBUTTON class. The valid iconid values are listed under the iconid parameter of the CreateIcon() class method.

A listbox provides a list of selectable items. If the value for the *Height* attribute is less than the number of items, the box is provided with a vertical scrollbar. A listbox can be set to allow the selection of more than one of its items by using the *SelectStyle* attribute. The default value "single" for this attribute permits only one item to be selected at a time. The value "multi" used in this dialog specification allows the user to select multiple entries by simply clicking on each one (clicking on an already-selected item toggles it off). The value "extended" allows Windows-style item selection by using the mouse with the SHIFT or CTRL key.

STEPS

- ☑ run the GENDLG script again
- ☑ when prompted to select the dialog specification, select LANGUAGE.XML in the SMLDLG directory and click [OK]
- ☑ select several languages in the listbox
- ☑ compare the dialog window to its specification
- ☑ click [OK] on the Languages dialog window to end the script

```
<?xml version="1.0"?>
<root>
  <dialog id="language" Title="Languages">
    <pane Orientation="horizontal">
      <pushbutton id="dict" Name="Dictionary..."/>
      <pushbutton id="file" Icon="CREATE_FILE"/>
    </pane>
      <label>Select languages:</label>
      <listbox id="language" Height="4" SelectStyle="multi">
        <item Value="english" Selected="true">English</item>
        <item Value="french">French</item>
        <item Value="spanish">Spanish</item>
        <item Value="japanese">Japanese</item>
        <item Value="german">German</item>
      </listbox>
  </dialog>
</root>
```

Menubuttons

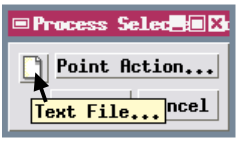
STEPS

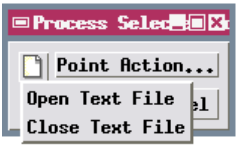
- ☑ run the GENDLG script again
- ☑ when prompted to select the dialog specification, select PROCPTS.XML in the SMLDLG directory and click [OK]
- ☑ click on the two menu buttons in the upper part of the dialog window
- ☑ compare the dialog window to its specification
- ☑ click [OK] on the Process Selected Points dialog window to end the script

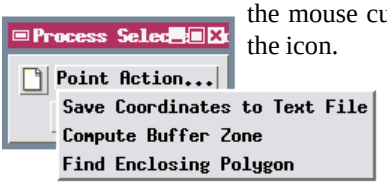
A dialog window can also include *menubuttons*, buttons that drop down a selection menu when pressed. Like a pushbutton, a menubutton can have either an icon or a static text label. Since a menubutton has no way of showing which item is currently selected, it is best used to launch one of several alternative actions from the dialog.

The Process Selected Point sample dialog might be used with a Tool Script that allows the user to select points in a View window. It has two menubuttons, one with an icon and one with a label. The two menubuttons are inside a horizontal pane so they appear side by side in the dialog window. Like other menu controls, selections for a menubutton are registered using items within the menubutton element.

When you use an icon for a pushbutton, togglebutton, or menubutton, you can use the ToolTip attribute to set the text to be shown in the popup ToolTip when the mouse cursor hovers over the icon.








```
<?xml version="1.0"?>
<root>
  <dialog id="procpts" Title="Process Selected Points">
    <pane Orientation="horizontal">
      <menubutton id="file" Icon="DESKTOP_FILE"
         ToolTip="Text File...">
        <item Value="open">Open Text File</item>
        <item Value="close">Close Text File</item>
      </menubutton>
      <menubutton id="ptaction" Name="Point Action...">
        <item Value="save">Save Coordinates to Text File</item>
        <item Value="buffer">Compute Buffer Zone</item>
        <item Value="polygon">Find Enclosing Polygon</item>
      </menubutton>
    </pane>
  </dialog>
</root>
```

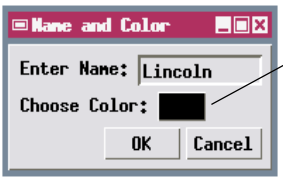
Edittext and Colorbutton

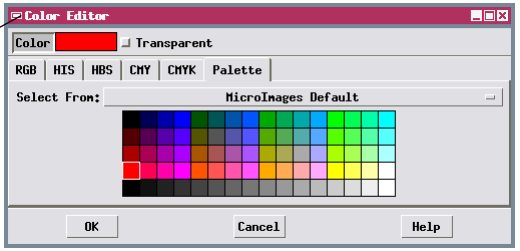
The Name and Color sample dialog window shows two additional types of controls, an *edittext* and a *colorbutton*. An edittext control allows the user to enter and/or edit a text string. The *MaxLength* attribute specifies the maximum allowed length of the string in characters, while the *Width* attribute specifies the width of the editable field (in "typical" characters). Text is aligned to the left side of the field by default. To right-align the text, use the *Justify* attribute with the value "right". The edittext control does not have a built-in label.

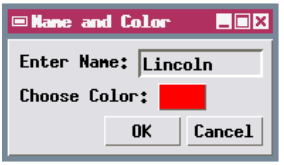
Pressing the colorbutton control opens a standard Color Editor window to allow the selection of a color from a palette or by specifying RGB, HIS or other color values. The selected color is shown on the colorbutton after the selection dialog is closed. The selected color is maintained in a COLOR class structure as red, green, and blue values (each 0 to 100) and an optional transparency value (also 0 to 100). The colorbutton control does not have a built-in label.

STEPS

- ☑ run the GENDLG script again
- ☑ when prompted to select the dialog specification, select NMCOLOR.XML in the SMLDLG directory and click [OK]
- ☑ compare the dialog window to its specification
- ☑ click [OK] on the Name and Color dialog window to end the script








```
<?xml version="1.0"?>
<root>
 <dialog Title="Name and Color" id="nmcolor">
  <pane Orientation="horizontal">
   <label>Enter Name:</label>
    <edittext id="name" MaxLength="10" Width="10"/>
  </pane>
  <pane Orientation="horizontal">
   <label>Choose Color:</label>
   <colorbutton id="colorbtn" AllowTransparent="true"/>
 </pane>
 </dialog>
</root>
```

Using Tabbed Pages

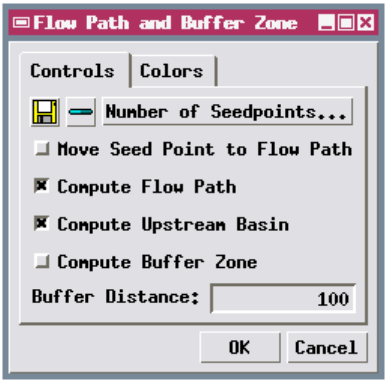
STEPS

- ☑ run the GENDLG script again
- ☑ when prompted to select the dialog specification, select FLOWPATH.XML in the SMLDLG directory and click [OK]
- ☑ click on each of the panel tabs to examine the included controls
- ☑ compare the dialog window to its specification
- ☑ click [OK] on the Flow Path and Buffer Zone dialog window to end the script

The FLOWPATH tool script performs watershed analysis operations on an elevation model shown in a View window, using seed points placed in the window interactively by the point tool invoked by the script.

If you are creating a dialog window with many controls, you can group controls together on separate tabbed panels. To do so, you first create a *book* element and create one or more *page* elements inside it. Each page element corresponds to a tabbed panel. The *Name* attribute value that you assign for each page provides the text that is placed on the panel's tab. Controls can be placed and arranged within a page just as in the main dialog or in any layout element.

The Flowpath sample dialog provides an example based on the control dialog for the Flowpath script, one of the sample tool scripts distributed with TNTmips. The two panels of this dialog are illustrated below, and the dialog specification is shown on the facing page. This dialog places the book of tabbed pages immediately inside the dialog element, but the book could be placed inside any layout element (except directly inside another book element). One page includes a number of different types of controls for general toolscript operations, while the other page provides a set of colorbuttons to designate colors for different vector overlays created by the script.



Specification for Flowpath Dialog

```
<?xml version="1.0"?>
<!DOCTYPE root SYSTEM "smlforms.dtd">
<root>
 <dialog id="flowpath" Title="Flow Path and Buffer Zone">
  <book>
   <page Name="Controls">
    <pane Orientation="horizontal">
     <pushbutton Name="Save" Icon="FILE_SAVE"
            ToolTip="Save Output Layers..."/>
     <pushbutton Name="Remove" Icon="CONTROL_SUBTRACT_CYAN"
            ToolTip="Remove Output Layers"/>
     <pushbutton Name="Number of Seedpoints..."/>
    </pane>
    <pane Orientation="vertical">
     <togglebutton id="btnSnap" Name="Move Seed Point to
            Flow Path" Selected="false"/>
     <togglebutton id="btnFlow" Name="Compute Flow Path"
            Selected="true"/>
     <togglebutton id="btnBasin" Name="Compute Upstream
            Basin" Selected="true"/>
     <togglebutton id="btnBuffer" Name="Compute Buffer Zone"
            Selected="false"/>
    </pane>
    <pane Orientation="horizontal">
     <label>Buffer Distance: </label>
     <editnumber id="buffDist" Width="5" Default="100"
            Precision="0" MinVal="0"/>
    </pane>
   </page>
   <page Name="Colors" Orientation="vertical">
    <pane Orientation="horizontal">
     <colorbutton id="fcolor"/>
     <label>  Flow path color</label>
    </pane>
    <pane Orientation="horizontal">
     <colorbutton id="bacolor"/>
     <label>  Basin color</label>
    </pane>
    <pane Orientation="horizontal">
     <colorbutton id="bucolor"/>
     <label>  Buffer zone color</label>
    </pane>
    <pane Orientation="horizontal">
     <colorbutton id="bocolor"/>
     <label>  Extents box color</label>
    </pane>
   </page>
  </book>
 </dialog>
</root>
```

Using an XML Dialog Specification

STEPS
- ☑ choose File / Open / *.SML File... from the SML editor window
- ☑ select HELLOXML1.SML from the SMLDLG directory
- ☑ examine the script
- ☑ repeat, but this time select HELLOXML2.SML from the SMLDLG directory
- ☑ run the second script
- ☑ press [OK] on the Hello World sample dialog window to end the script

Now that we have introduced the most common components of a dialog and the structure of the dialog specification, we can begin to examine how to use a specification with an SML script. The dialog specification must be read and interpreted when the script is run to create an XML document structure in memory. This structure is an instance of the class XMLDOC. The specification can either be in a separate file or embedded in the script. Different XMLDOC class methods are used in these two situations to ingest the specification and assign it to the class instance. The two scripts for this exercise implement the Hello World sample dialog and illustrate these two approaches (script excerpts below).

```
# helloXML1.sml

class STRING xmlfile$;
xmlfile$ = _context.ScriptDir + "/hello.xml";

class XMLDOC dlgdoc;
dlgdoc.Read(xmlfile$);
```

Script helloXML1.sml ingests the specification from a separate file in the same directory using the Read(xmlfile$) class method in XMLDOC. This method reads the file and parses the text to create the XML structure in memory. The filename passed to the Read() method must include the full directory path to the file as well as the file extension. The filename string is constructed in this example by a string expression using the script's CONTEXT class structure. The class member _context.ScriptDir provides the path to the directory containing the current SML script. The string expression concatenates the path with the name of the file.

```
# helloXML2.sml

class STRING xml$;
xml$='<?xml version="1.0"?>
  <root>
    <dialog id = "hello" title="Hello World">
      <label>Sample Dialog Window</label>
    </dialog>
  </root>';

class XMLDOC dlgdoc;
dlgdoc.Parse(xml$);
```

The sample script helloXML2.sml has the dialog specification embedded within it, assigned to a string variable. Note the single quotation marks enclosing the specification, which enable multiple lines of text to be used within the assignment statement. The string variable is then passed to the XMLDOC class method Parse(xml$), which interprets the text and creates the XMLDOC class instance in memory.

Creating and Opening the Dialog Window

The XML structure created in memory by the Read() or Parse() method of class XMLDOC contains all of the elements in the dialog specification in a hierarchical, tree-like structure. Each element, including the dialog element itself, can also be thought of as a "node" in the structure. To create and open the dialog window, your SML script must retrieve the dialog node from the parsed XML document structure and identify it as the source for the dialog window.

STEPS

☑ examine the second half of the HELLOXML2.SML script

A modal dialog window created using an XML specification is automatically provided with OK and Cancel buttons.

The two scripts you ran on the previous page differ only in how they ingest the dialog specification; the second half of each (excerpted below) is identical, and implements the steps outlined above. The dialog node in the Hello World specification has the unique *id* attribute "hello" that can be used to retrieve it. This is done using the XMLDOC class method GetElementByID(), which returns an instance of class XMLNODE. This is the class that represents any given node in a parsed XML document. In this script the dialog node is represented by the class variable *dlgnode*.

```
class XMLNODE dlgnode;
dlgnode = dlgdoc.GetElementByID("hello");

class GUI_DLG dlgwin;
dlgwin.SetXMLNode(dlgnode);
dlgwin.DoModal();
```

The dialog window itself is an instance of class GUI_DLG, represented in this script by class variable *dlgwin*. The class method SetXMLNode() in class GUI_DLG is used to identify the dialog node in the XML structure as the source for the dialog window. The final script statement uses the GUI_DLG class method DoModal() to create and open the dialog window. This method opens the window as a modal dialog, meaning that once the dialog is opened, no other parts of the script can be executed until the dialog is closed. Modal dialogs are easy to manage and are appropriate for selecting objects and/or gathering processing parameters for standalone SML processing scripts.

Dialog controls have a common set of attributes to allow you to control their alignment and resize behavior:

HorizAlign ("Left", "Right", or "Center") and VertAlign ("Top", "Center", or "Bottom") let you control the alignment of the controls within their parent pane.

HorizResize and VertResize determine how the control behaves when the parent is resized (such as when the user resizes the dialog window). Possible values are "Expand", "Fixed", and "Relative" (keeps same proportion of space).

Trapping XML Errors

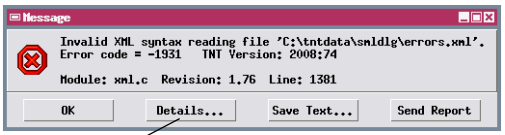
STEPS

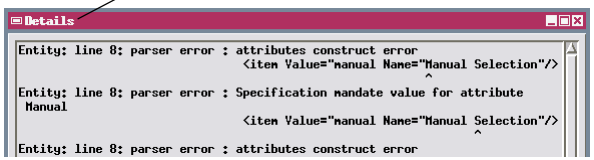
- ☑ choose File / Open / *.SML File... and select XMLERRS1.SML from the SMLDLG directory
- ☑ run the script; note the error message reporting that the XMLNODE parameter is NULL
- ☑ choose File / Open again, and select XMLERRS2.SML
- ☑ in the Message window that opens, press the Details button
- ☑ press the [OK] button on both message windows to end the script

In order for SML to interpret a dialog specification correctly, the specification must be *well-formed*, meaning that it uses the proper notation for tags and attributes and has the nested structure expected of an XML document. The syntax of the dialog specification is checked by the Read() or Parse() class method when the script is run, but XML syntax errors are not automatically reported to you by SML. The excerpt of script xmlerrs2.sml below shows how you can get an XML syntax error report.

```
numeric errXML;
class XMLDOC dlgdoc;
errXML = dlgdoc.Read(xmlfile$);

if (errXML < 0) {
  PopupError(errXML);
  Exit();
  }
```

The Read() and Parse() class methods return a numeric value that is an error code (a negative value) if there are XML syntax errors or 0 if there are no errors. The sample script assigns the returned value to a numeric variable *errXML*, then checks the value of the variable. If *errXML* is less than 0, the error value is passed to the PopupError() function, which opens a Message window reporting that there is an XML syntax error. If you press the Details button on this window, a second message window opens and lists the errors.





The specification read by the two sample scripts in this exercise (excerpted below) is a copy of the groupbox/radiobox dialog specification described previously, but the closing double quotation mark is omitted for the Value attribute value in the third radiogroup item. Subsequent portions of the specification cannot be interpreted correctly, triggering a number of error listings in the details message window.

```
<radiogroup id="processgp" Default="entire">
  <item Value="entire" Name="Entire Scene"/>
  <item Value="polygon" Name="AOI Polygon"/>
  <item Value="manual Name="Manual Selection"/>
</radiogroup>
```

The closing quotation mark is missing for the Value attribute.

Trapping Dialog ID Errors

Even if the dialog specification is well-formed XML, there may be errors in the use or spelling of attribute names and values that prevent the dialog window from being constructed correctly. If you are using the SML editor or a text editor to create the dialog specification, you should carefully examine the spelling and use of attribute names and values to ensure that they are consistent within the specification and script and that they conform to the usage rules outlined in the *Reference Guide to SML Dialog Specifications in XML*. The *Guide* lists the attributes that are predefined for use with each dialog element. Valid values are also listed for those attributes that have a fixed set of possible values.

STEPS

- ☑ choose File / Open / *.SML File... and select XMLERRS3.SML from the SMLDLG directory
- ☑ run the script; note the error messages that appear
- ☑ press [OK] on the message windows to end the script

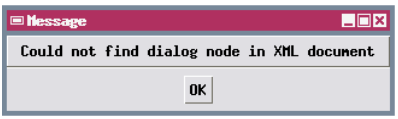
Problems with attribute names may also prevent the dialog from opening at all. In the example used for this exercise, the dialog element *id* values in the dialog specification and the SML script do not match. The script excerpt shown here illustrates how the script can explicitly check

```
<?xml version="1.0"?>
<root>
  <dialog id="dlg" Title="Select Process">
```

The dialog *id* attribute value in the specification does not match the *id* value expected by the script.

```
class XMLNODE dlgnode;
dlgnode = dlgdoc.GetElementByID("radiogp");

if (dlgnode == 0) {
  PopupMessage("Could not find dialog node in XML document");
  }
```

for this condition. The GetElementByID() method in class XMLDOC returns a valid instance of class XMLNODE if the specified dialog *id* is found. If it is not, the class instance returned is empty. In SML an empty class instance can be represented by the numeric value 0. The script excerpt compares the returned class to 0 and if the comparison is true it opens a message window stating that the dialog node was not found.



Performing a manual syntax check from the SML editor window (Syntax / Check...) does not evaluate the XML syntax of any embedded or referenced dialog specification.

Using an XML Editor

You can create embedded dialog specifications directly with the SML editor or create separate dialog specification files with any text editor. But you may find it easier to create your specifications using a specialized XML editor. A number of free, shareware, and commercial XML editors are available for download via the World Wide Web. Nearly all XML editors can check that the XML is well-formed, and many provide a graphical *Tree View* of the document structure that simplifies editing. Some editors can compare the XML to a document model that specifies the available tags, their allowed attributes, and the relationships permitted between elements (for example, the only allowed child element of a *combobox* is an *item*). An XML file that conforms to its document model is said to be *valid*, and the checking procedure is called *validation*. One widely-supported form of XML document model is a Document Type Definition (DTD) file. MicroImages has created a DTD file for SML dialog specifications (smlforms.dtd) that can be found in the SMLDLG directory. An excerpt of this file is shown below.

```
<!-- SMLFORMS DTD Version 2.0 -->
<!-- For use with SML Dialog Specifications in XML -->
<!-- MicroImages, Inc. -->

<!-- ===================================== -->
<!--              MAIN ELEMENTS            -->
<!-- ===================================== -->

<!ELEMENT root (dialog | script | strings)* >
<!ELEMENT script (#PCDATA)>
<!ELEMENT dialog (book | pane | groupbox | label | pushbutton |
  togglebutton | colorbutton | edittext | editnumber | radiogroup |
  combobox | menubutton | listbox | canvas)*>
```

Excerpt showing the beginning lines of the SMLFORMS DTD file, defining the main elements <root>, <script>, and <dialog>.

MicroImages has also provided a Dialog Specification Template File (dlgtempl.xml, shown below) that you can open in a validating XML editor to create your dialog specifications. The template includes the *root* and *dialog* elements preceded by a Document Type Declaration that defines the name of the *root* element (<root>) and the name of the DTD and its location (in the same directory as the XML file being edited). Most validating XML editors read the DTD and provide menus of valid elements that can be inserted into the current element, menus of attributes available to be added for each type of element, and menus of attribute values for those attributes that have a fixed set of valid values.

```
<?xml version="1.0"?>
<!DOCTYPE root SYSTEM "smlforms.dtd">
<root>
  <dialog>
  </dialog>
</root>
```

Dialog Specification Template File (dlgtempl.xml) containing the Document Type Declaration and <root> and <dialog> elements.

Programming the OK and Cancel Buttons

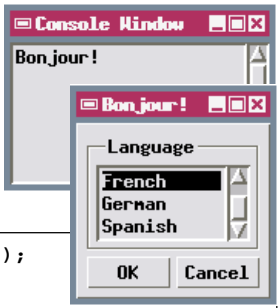
One main role of a dialog window is to record parameter choices and values and pass them on for use by the SML script after the dialog closes. There are several ways in which you can use the default OK and Cancel pushbuttons on a modal dialog to manage this interaction.

The DoModal() class method of the GUI_DLG class, which you use to open a modal dialog, returns a numeric value when the dialog is closed by either the OK or Cancel button. Pressing OK returns a value of 0 and pressing Cancel returns a value of -1. A simple way to carry out script actions using these buttons is to check the value returned by this method and take alternative actions based on this value.

The simple objective in this example is to print "Hello" to the console in the selected language when the OK dialog button is pressed. As shown in the script excerpt below, the value returned by the DoModal() method is stored in a numeric variable (dlgreturn), and its value is then checked. If the value is 0, the value of the string variable language$ is read from the dialog's listbox (more on that later) and used to set the text to be printed to the console. If the value of dlgreturn is -1, the Exit function is called to exit the script.

STEPS

- ☑ choose File / Open / *.SML File... and select BONJOUR1.SML from the SMLDLG directory
- ☑ run the script
- ☑ select a language from the listbox
- ☑ press [OK] on the Bonjour! sample dialog window; a translation of "Hello!" is printed to the Console window in the selected language, and script execution ends
- ☑ repeat using a different language
- ☑ examine the callback procedure in the script




```
dlgreturn = dlg.DoModal();

if (dlgreturn == 0) {
 local class STRING language$;
 language$ = dlg.GetCtrlValueStr("listbox");

 if ( language$ == "English" ) then
  print("Hello!");
 else if ( language$ == "French" ) then
  print("Bonjour!");
 else if ( language$ == "German" ) then
  print("Hallo!");
 else if ( language$ == "Spanish") then
  print("Hola!");
 }

if (dlgreturn == -1) {
 Exit();
 }
```

Specifying Callbacks for Dialog Buttons

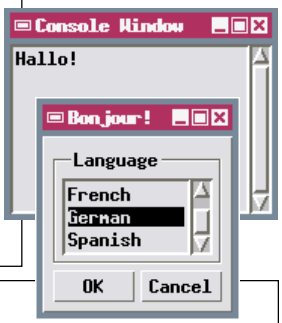
STEPS

- ☑ choose File / Open / *.SML File... and select BONJOUR2.SML from the SMLDLG directory
- ☑ run the script
- ☑ select a language from the listbox
- ☑ press [OK] on the Bonjour! sample dialog window; a translation of "Hello!" is printed to the Console window in the selected language, and script execution ends
- ☑ repeat using a different language
- ☑ examine the callback procedure in the script

A more flexible method of programming the default pushbuttons on a dialog is through the use of *callbacks*, which are pointers to procedures or functions defined in the SML script. You can write a user-defined function or procedure to be carried out by the OK button and one to be carried out by the Cancel button. You associate these procedures with their respective buttons by assigning them as attributes of the dialog element in the dialog specification. For a modal dialog, the dialog element has *OnOK* and *OnCancel* attributes; modeless dialogs have *OnOK*, *OnApply*, and *OnClose* attributes.

The script version for this exercise includes a procedure called SayHello() to print "Hello" in the selected language. The dialog specification assigns this procedure name to the *OnOK* attribute to map this code to the OK pushbutton. An OnCancel callback is not required in this instance, as the script will automatically exit after the Cancel button is pressed and the DoModal() class method returns.

```
proc SayHello ()
  {
  local class STRING language$;
  language$ = dlg.GetCtrlValueStr("listbox");

  if ( language$ == "English" ) then
    print("Hello!");
  else if ( language$ == "French" ) then
    print("Bonjour!");
  else if ( language$ == "German" ) then
    print("Hallo!");
  else if ( language$ == "Spanish") then
    print("Hola!");
  }
```




```
<?xml version="1.0"?>
<!DOCTYPE root SYSTEM "smlforms.dtd">
<root>
  <dialog id="hello" Title="Bonjour!" OnOK="SayHello()">
    <pane>
      <groupbox Name="Language" ExtraBorder="4">
        <listbox id="listbox" SelectStyle="single" Height="3"
                                                Default="French">
```

Using the Script Tag

The SML code executed by a dialog's callbacks can reside in the main SML script (as in the previous exercise) or within the dialog specification itself. The *script* element in a dialog specification is defined as a container for SML code. This is a main-level element within the root element of the XML text The script element can include the code for one or more callback procedures. Any global variables or functions you define in the main SML script are also available for use by the code in the dialog specification's script element.

The example in this exercise uses the same dialog as the preceding exercise. The only difference here is that the SayHello() procedure is in the script element of the dialog specification (excerpted below).

STEPS

- ☑ choose File / Open / *.SML File... and select BONJOUR3SML from the SMLDLG directory
- ☑ run the script as you did in the last exercise

```
<?xml version="1.0"?>
<!DOCTYPE root SYSTEM "smlforms.dtd">
<root>
  <dialog id="hello" Title="Bonjour" OnOK="SayHello()">
    ...  [layout and control specifications omitted] ...
  </dialog>
  <script>
    <![CDATA[
      proc SayHello () {
        clear();
        local string language$;
        language$ = dlg.GetCtrlValueStr("listbox");

        if ( language$ == "English" ) then
          print("Hello!");
        else if ( language$ == "French" ) then
          print("Bonjour!");
        else if ( language$ == "German" ) then
          print("Hallo!");
        else if ( language$ == "Spanish") then
          print("Hola!");
        }
    ]]>
  </script>
</root>
```

beginning CDATA delimiter

ending CDATA delimiter

The SML code for your callbacks might contain characters (such as <, >, and &) that have special significance in XML syntax. For this reason the SML code should be enclosed within CDATA delimiters as shown. They tell the XML parser that the enclosed section should be treated as regular text ("character data") that should not be interpreted with XML syntax rules.

Getting Values from the Dialog

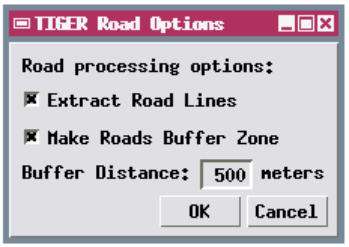
STEPS

- ☑ choose File / Open / *.SML File... and select GETDATA.SML from the SMLDLG directory
- ☑ run the script
- ☑ in the TIGER Road Options dialog, set the Road processing options and edit the Buffer Distance field, then press OK
- ☑ note the dialog values printed to the SML Console Window
- ☑ examine the script examples of the four methods of reading values from the dialog
- ☑ press [OK] on the TIGER Road Options sample dialog window to end the script

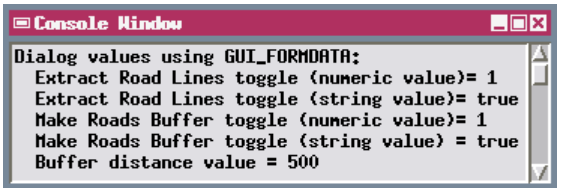
The previous three exercises used one of several available methods for getting control values and settings from the dialog. The script for this exercise shows all of these methods using the TIGER Road Options dialog described on a previous page as an example. When you press the OK pushbutton, a procedure assigned to the dialog's OnOK callback prints the control value settings to the SML console window. The settings are retrieved in four different ways that are described briefly below.

The first approach gets all of the dialog control settings at once using the GetValues() class method in class GUI_DLG (the dialog window class). The values are returned as an instance of class GUI_FORMDATA, which you must have previously declared. You can then use GetValueNum() and GetValueStr() methods in the latter class to read out the stored values (as a number or as a string, respectively) as needed.

The second approach uses methods in class GUI_DLG to get the value for each control from the dialog individually using the control's *id*: GetCtrlValueNum(*id*) and GetCtrlValueStr(*id*).



The third and fourth approaches use the GetCtrlByID(*id*) method in class GUI_DLG to get a *handle* for each control, then a GetValue(), GetSelected(), or other similar method from the individual control class to get the control's setting. The control handle can be stored as a control class instance that is then used to get the setting, or the methods to get the control handle and its value can be strung together, omitting the control handle class variable (a more compact but perhaps less obvious approach).



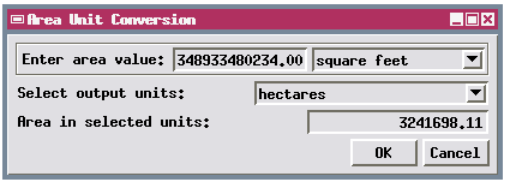
Changing Control Settings in Callbacks

You can also use callbacks for individual controls to show the results of computations in the dialog and to change the status of other controls. The script for this exercise performs area unit conversions using a value you enter in an editnumber control and input and output units you select from combobox controls. The output area is computed and shown in an editnumber control when you press the [OK] button. The output editnumber is set to be read-only, which disables manual editing of the output value.

STEPS

- ☑ choose File / Open / *.SML File... and select AREACALC.SML from the SMLDLG directory
- ☑ examine the procedure and function definitions in the script
- ☑ run the script
- ☑ enter an area value in the upper numeric field
- ☑ select input and output area units from the combobox controls
- ☑ press [OK] to compute the output area
- ☑ when you are finished, press [Cancel] to close the dialog and exit the script

You can change control settings using class methods in either the dialog window's class (an instance of GUI_DLG) or the individual control classes. The *OnOK* callback function for this dialog window [called Recalculate()] uses the method SetCtrlValueNum(*id*) in class GUI_DLG to set the value for the output area editnumber control. The callback procedures for the input area editnumber control and the two combobox controls use the ClearValue() method in class GUI_CTRL_EDIT_NUMBER to blank the editnumber control when you change any of the other control settings.



In order to force the Area Unit Conversion window to stay open after the OK button is pressed, the *OnOK* callback for the dialog element is defined in the script as a function that returns the value 0 (after its other operations are complete). In order to use this return value, the attribute assignment for the *OnOK* callback in the dialog specification must have the keyword "return" preceding the function name, as shown in the excerpt to the right.

The AREACALC script also prints the output units, scale factor for the units conversion, and the output area value to the Console Window.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root SYSTEM "smlforms.dtd">
<root>
  <dialog id="areacalc"
      Title="Area Unit Conversion"
      OnOK="return Recalculate()">
```

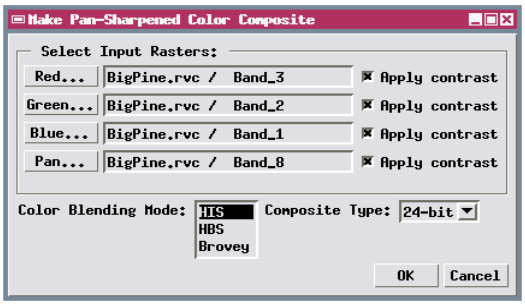
Managing a Complex Dialog Window

STEPS

- ☑ choose File / Open / *.SML File... and select PANSHARPCOMP.SML from the SMLDLG directory
- ☑ run the script
- ☑ in the Make Pan-Sharpened Color Composite window, press [Red...]
- ☑ use the Select Object dialog to navigate into the BIGPINE Project File in the SMLDLG directory and select raster BAND_3
- ☑ press [Green...] and select raster BAND_2
- ☑ press [Blue...] and select raster BAND_1
- ☑ press [Pan...] and select raster BAND_8
- ☑ choose a Color Blending Mode and a Composite Type
- ☑ press [OK] to name an output color composite raster and run the process
- ☑ examine the script

The script in this exercise provides an example of a fully-functioning processing script that creates a relatively complex custom dialog window to manage its inputs. The script creates a pan-sharpened color-composite raster from three bands of a multispectral image and a higher-resolution panchromatic (grayscale) image, each of which is selected using a pushbutton. Read-only edittext controls are used to show the file and object names for each of the selected input objects. The dialog provides togglebuttons that allow you to apply a saved contrast table (if present) to each input raster, a listbox to select from three color blending modes, and a combobox to choose the bit-depth of the output color composite.

The dialog specification for the Make Pan-Sharpened Color Composite window is embedded in the SML script. The dialog controls and their callbacks are designed to lead the user through the controls in proper sequence. Only the Red... and Cancel buttons are intially active. (The dialog specification sets an *Enabled* attribute to 0 for the Green..., Blue..., and Pan... pushbuttons and for all of the Apply Contrast toggle buttons so that these controls are initially dimmed and disabled. An *OnOpen* callback for the dialog window disables the OK button when the window first opens.) The *OnPressed* callback for the Red... pushbutton enables its Apply Contrast toggle button as well as the Green... pushbutton. Pressing each successive input raster button activates the next one, until pressing the Pan button activates the dialog's OK button.



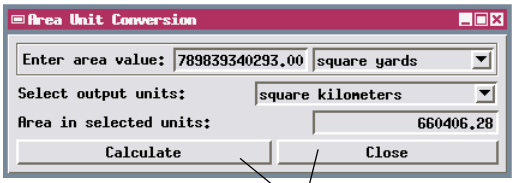
Suppressing Default Pushbuttons

Although dialog windows are supplied with action pushbuttons by default, you may want to use your own pushbuttons with more appropriate names in their place. The dialog element in the dialog specification has a *Buttons* attribute that can be used to specify which of the default buttons to use on the dialog. If you use this attribute but assign it an empty string (simply a pair of quotation marks as shown in the specification excerpt), none of the default buttons are supplied. In this version of the Area Unit Conversion script, Calculate and Close buttons are used in place of the default OK and Cancel buttons.

STEPS

- ☑ choose File / Open / *.SML File... and select AREACALCUNITS.SML from the SMLDLG directory
- ☑ examine the procedure and function definitions in the script
- ☑ run the script

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root SYSTEM "smlforms.dtd">
<root>
  <dialog id="areacalc"
      Title="Area Unit Conversion"
      Buttons="">
```



The input and output units comboboxes are also automatically populated with area choices in this script version, rather than having menu items explictly listed in the dialog specification. The GUI_CTRL_COMBOBOX class has an AddUnitItems() method that allows you to specify the type of unit to use (in this case, area units); the combobox is then automatically supplied with all of the unit names that are directly supported in the TNT products. This method must be called after the comboboxes have been created by the SetXMLNode() method on the dialog. To do so, the script makes use of another dialog class method, SetOnInitDialog(). This method lets you identify a procedure to be called when the modal dialog is created but before it is opened by the DoModal() method. In this example an OnInitDialog() procedure is used to get the control handles required in the control callbacks and to add the area items to the two area unit comboboxes.

The width of pushbuttons relative to their text can be controlled using the *HorizResize* attribute of the pushbutton element. If not specified, this attribute defaults to “Expand”, which allows the button to fill the available horizontal space in the parent pane. If you instead set this attribute to “Fixed”, the button width will be slightly larger than the width of its text.

In addition to area units, you can add standard lists of the following types of units to a combobox: length (distance), plane angle, time, temperature, volume, and mass.

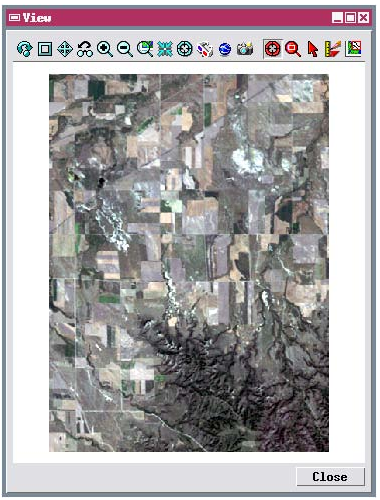
Creating a View in a Dialog Window

STEPS

- ☑ select File / Open / *.SML File and select VIEW.SML from the SMLDLG directory
- ☑ run the script using as input the raster object _8_BIT from the CB_COMP Project File in CB_DATA sample data directory
- ☑ press [Close] to close the window

A dialog window created by an SML script can display input or output objects in a geospatial view. The script for this exercise provides a simple example of the required steps.

A dialog specification cannot directly specify a view; rather, it should create a layout pane (called "viewpane" in this example) that will subsequently contain the view. An OnInitdialog() procedure (see previous page) is then used to set up the view within this layout pane. The sample code for this procedure is shown below. The procedure creates a spatial group, adds the selected raster object to the group, and calls a method on class GRE_GROUP to create a view to display the group. This method specifies the parent of the view along with its height and width in screen pixels. The parent of the view must be an instance of class WIDGET, the base class for the Motif dialog components used in X Windows, which are the hidden underpinnings of the GUI classes in SML. Prior to creating the view, the procedure calls a method on class GUI_LAYOUT_Pane to get a widget corresponding to the pane to serve as the parent for the view. Another procedure called when the dialog opens is used to draw the view.




```
proc OnInitDialog ()
  {
  local class GUI_LAYOUT_PANE viewpane;  # layout pane to contain the view
  local class widget viewpaneWidget;     # widget for pane to serve as parent for the view
  local class GRE_GROUP viewgp;          # group to be shown in the view
  local class GRE_LAYER_RASTER tempLayer;          # raster layer in the group

  viewpane = dlgview.GetPaneByID("viewpane");      # get handle for layout pane from dialog
  viewpaneWidget = viewpane.GetWidget();           # get Xm widget for pane

  viewgp = GroupCreate();                             # create display group
  tempLayer = GroupQuickAddRasterVar(viewgp, Rast);   # add the raster to the group

  # create 2D view of group in dialog
  view = viewgp.CreateView(viewpaneWidget, "View", 400, 200, "NoDftMarkButtons,
       NoStatusBar,NoLegend");

  # set white as background color for view
  color.red = 100;   color.green = 100;   color.blue = 100;
  view.BackgroundColor = color;
  }
```

A Complex Dialog with a View

The script for this exercise presents a more complex sample dialog that incorporates process controls and a geospatial view. The sample application is an interactive "boxcar" classification of cells in a set of three image bands (raster objects). Numeric fields are provided for the user to enter minimum and maximum cell values for each input raster. When the Process button is pressed, the script identifies cells whose value sets fall within all of the specified ranges. Default limit values are automatically computed from the raster statistics when the input rasters are selected, and an OnInitDialog() procedure is used to set these default values before the dialog opens.

STEPS

- ☑ select File / Open / *.SML File and select BOXCAR3.SML from the SMLDLG directory
- ☑ run the script, selecting for input rasters RED, GREEN, and BLUE from the CB_TM Project File in the CB_DATA directory
- ☑ press [Process] on the Boxcar Classification window to run using the default values
- ☑ study the script to see how the various window components are constructed and how actions are controlled

The view in the dialog initially displays a temporary composite raster created from the three input rasters. Each time the Process button is pressed, a temporary binary classification raster with color palette is created and displayed over this composite to portray the results of the classification. Cells with values within all of the specified ranges are shown in yellow in the classification overlay. A read-only text field below the range controls is used to show status messages. The Clear button clears the previous clasification results, while the Save button saves the temporary classification raster to a Project File.

In the dialog specification, the *ChildSpacing* attribute is used for the pane containing the pushbuttons to set the spacing in screen pixels between the pushbuttons (the "children" of the layout pane).

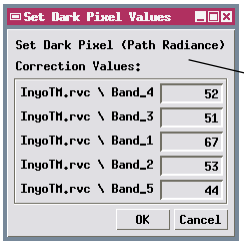
Dynamically Adding Dialog Components

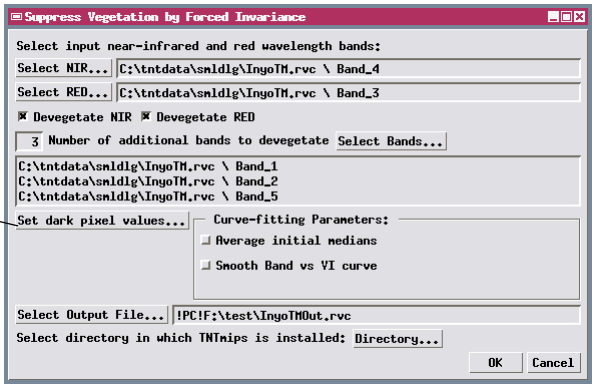
STEPS

- ☑ choose File / Open / *.SML File... and select DEVEG.SML from the SMLDLG directory
- ☑ run the script
- ☑ in the dialog window that opens, press [select NIR...], navigate into the INYOTM Project File in the SMLDLG directory, and select BAND_4
- ☑ press [Select RED...] and select BAND_3
- ☑ turn on the Devegetate NIR and Devegetate RED toggle buttons
- ☑ type "3" into the "Number of additional bands to devegetate" field and press <Enter>
- ☑ press [Select Bands...]
- ☑ select BAND_1, BAND_2, and BAND_5
- ☑ press [Set dark pixel values...]
- ☑ press [OK] in the Set Dark Pixel Values window that opens
- ☑ press [Select Output File...] and name a new Project File to contain the output objects
- ☑ press [Directory...] and select the directory in which TNTmips is installed
- ☑ press [OK] to start processing

The script in this exercise is designed to modify cell values in a set of multispectral image bands to suppress the contribution of vegetation. Most processing parameters are set on the main dialog window (shown below). However, each band to be processed must also be corrected for additive brightness effects of atmospheric haze by subtracting a "dark pixel value" (determined by the user) from each cell value. These values are set using a second dialog opened by the *OnPressed* callback procedure [SetDP()] for the Set dark pixel values... pushbutton. The Set Dark Pixel Values window lists the bands (filename and object name) and provides an editnumber field showing a default dark pixel value (the minimum cell value for the raster). The tricky part is that the number and selection of bands to be processed can vary.

The SetDP() procedure includes only a skeletal dialog specification with the dialog title, labels at the top, and the groupbox with a contained pane to hold all other controls. The procedure parses this skeletal specification, then modifies the XML structure in memory to add a series of horizontal panes, each containing a text label and an editnumber control. These changes are made using methods in class XMLNODE: NewChild(), SetAttribute(), and SetText(). The completed dialog is then opened.





Raster Intervals Script Dialogs

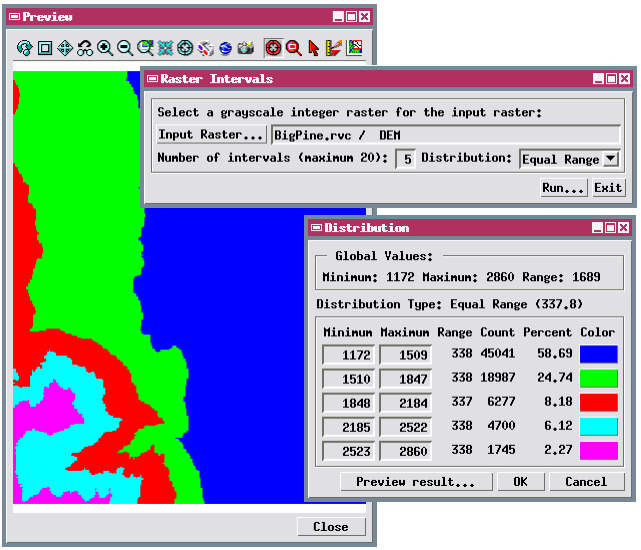
The Raster Intervals sample script creates several dialog windows including a preview of the result, using methodology introduced on the preceding several pages. This script categorizes a grayscale raster into the specified number of value intervals using either equal value range or equal cell count per interval.

The main dialog window provides the basic input controls and is set up by a complete dialog specification embedded in the script. Pressing the Run button calls the setUpIntervals() procedure in the script, which computes the required intervals and opens a Distribution window that shows the numerical values for the resulting range intervals and allows editing of the range boundaries. Since the number of intervals can vary, the procedure includes only a skeletal specification for this window. The XML structure for this dialog is modified on-the-fly by this procedure (as described on the previous page) to provide the controls and value labels for each interval.

Pressing the Preview result... button on the Distribution window calls the onPreview() script procedure, which creates and opens a Preview window that displays a temporary raster with values categorized using the current distribution parameters. The techniques used to create and populate this view were described on pages 28 and 29.

STEPS

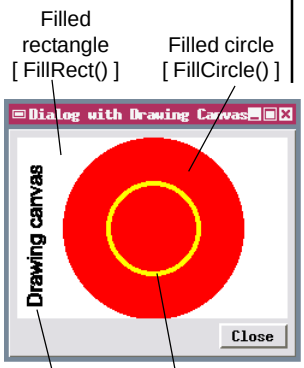
- ☑ choose File / Open / *.SML File... , navigate to the StandAlone directory in the SML sample script collection, and select RasterIntervals.sml
- ☑ run the script
- ☑ press [Input Raster...] on the Raster Intervals window
- ☑ select raster DEM from the BigPine Project File in the SMLDLG directory
- ☑ press [Run...]
- ☑ press [Preview result...] on the Distribution window
- ☑ press [Close] on the Preview window
- ☑ press [OK] on the Distribution window
- ☑ use the Select Object dialog to name a new Project File and output raster object
- ☑ press [Exit] on the Raster Intervals window



Using a Drawing Canvas

STEPS

- ☑ choose File / Open / *.SML File... , navigate back to the SMLDLG directory and choose DRAWDLG.SML
- ☑ run the script
- ☑ note the graphics and text drawn in the dialog window created by the script



- ☑ press [Close] on the Dialog with Drawing Canvas window
- ☑ examine the script and its dialog specification
- ☑ choose File / Exit from the SML Editor window

In some instances you may want to design a dialog window that incorporates a graph created from your input data or from the process output, or some other graphic. Various types of graphics and text can be drawn within a drawing canvas control in the dialog. To do so you use the *canvas* element, which maps to the GUI_CANVAS class in SML, in the dialog specification.

The simple modal dialog created by the DRAWDLG.SML sample script illustrates the use of a drawing canvas. The dialog specification for this window is shown at the bottom of the page. The canvas element has Height and Width attributes that you must use to set the dimensions of the canvas in screen pixels.

The sample script draws a filled white rectangle, filled red and open yellow circles, and vertical text within the canvas. Placement of these elements in the canvas is referenced to an X-Y coordinate system with units of screen pixels and an origin (0,0 position) at the upper left corner of the canvas. Drawing within the canvas requires a *graphics context* (class GC) that is created using the CreateGC() method of the GUI_CANVAS class. The GC class provides methods for drawing points, polylines, various geometric shapes, and text, as well as methods to set colors, fonts, and font sizes. The class methods used to draw the graphics in this dialog are shown in the illustration. In this script the graphic elements are drawn in an OnRedraw() procedure that is called within the dialog's OnOpen callback.

```
<?xml version="1.0"?>
  <root>
    <dialog id="target" Title="Dialog with Drawing Canvas" Buttons=""
       OnOpen="OnOpenDlg()">
      <canvas id="canvas" Height="150" Width="225"/>
      <pushbutton id="closeBtn" Name=" Close " VertResize="Fixed"
      HorizResize="Fixed" HorizAlign="Right" OnPressed="OnClose()"/>
    </dialog>
  </root>
```

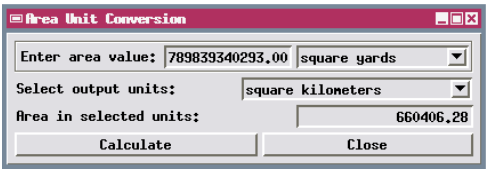
Localizing a Script Dialog

A script dialog can be set up to automatically localize the dialog text when the script is run under different TNTmips language interfaces (locales). The sample script for this exercise includes English and French localizations.

STEPS

- ☑ choose File / Open / *.SML File... and select AREACALCLOC.SML from the SMLDLG data directory
- ☑ examine the dialog specification in the script

Setting the *ResourceLookup* attribute of the dialog element to "true" replaces any text element in the dialog with its translated text based on the current language locale. Translations for custom text must be provided within the dialog specification using *string* elements within *strings* sections, with one section for each language; an empty *strings* section is used to identify the original language of the dialog text. The *id* attribute for each *string* element is simply the original text string. In this example translations are provided for six custom text strings (window title, labels, and button names); the selections for the two standard area unit menus are automatically provided with translations using the French language text resource package provided with TNTmips.



Dialog window when script is run with TNTmips set to use the English language interface.

Dialog window when script is run with TNTmips set to use the French language interface.

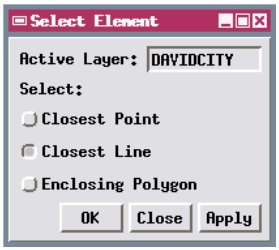
The apostrophe (single quote) is a special character in XML. The predefined string "'" is used to encode an apostrophe in a text string in XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root SYSTEM "smlforms.dtd">
<root>
   <strings language="enu">          <!-- English -->
   </strings>
   <strings language="frc">          <!-- French -->
     <string id="Area Unit Conversion">Conversion des unités d&apos;aire
        </string>
     <string id="Enter area value:">Porte une valeur d&apos;aire:</string>
     <string id="Select output units:">Choix d&apos;unité nouvelle:</string>
     <string id="Area in selected units:">L&apos;aire dans les unités choisis:
        </string>
     <string id=" Calculate "> Calculer </string>
     <string id=" Close "> Fermer </string>
   </strings>
  <dialog id="areacalc" Title="Area Unit Conversion" Buttons=""
     ResourceLookup="true">
```

Modeless Dialogs

STEPS

- ☑ choose Main / Display from the TNTmips menu
- ☑ press the Open Display icon button on the Display Manager, navigate to the SML sample data folder, and choose ToolGroup from the TOOLS Project File
- ☑ press the Select Element tool script icon button on the View window toolbar
- ☑ on the Select Element window that appears, use the radio buttons to choose which type of element to select



- ☑ left-click in the View to select an element
- ☑ close the Select Element window
- ☑ choose Options / Scripts / Tool Scripts in the View
- ☑ in the Customize Tool Scripts window, click on the Select Element entry in the list and press the Edit icon button to open the SML window to show the tool script
- ☑ close the SML window and Customize Tool Scripts window when you have finished examining the script

Tool scripts and display control scripts are specialized scripts that are activated from within a standard View window and allow continued interaction with data being displayed. These scripts are event-driven: they consist primarily of prenamed procedures that are called by events in the display window (such as left or right mouse click). If such a script requires a custom dialog window, the open dialog must not interfere with the processing associated with these various events. Modeless dialogs are therefore the appropriate choice for these types of scripts. The tool script in this exercise provides a modeless dialog that is used to choose what type of element (point, line, or polygon) will be selected by a left-click action in the View.

A modeless dialog can be set up with a dialog specification just like a modal dialog, but different GUI_DLG class methods are then used to initialize and control the window. A modal dialog is created and opened in one step using the DoModal() GUI_DLG class method, but for modeless dialogs these steps are separate. A CreateModeless() method is used to create (initialize) the dialog, and Open() and Close() methods are used to open and close it when needed.

Tool scripts are explicitly activated by pressing the script's icon button on the View window's tool bar and are deactivated by selecting another tool (such as the Recenter or Zoom tool). The OnInitialize() prenamed procedure is called automatically the first time the tool is activated; this is the appropriate place to process the dialog specification and create the modeless dialog. The dialog should be opened within the OnActivate() procedure, which is called each time the tool is activated, and closed within the OnDeactivate() procedure, which is called when another tool is selected.

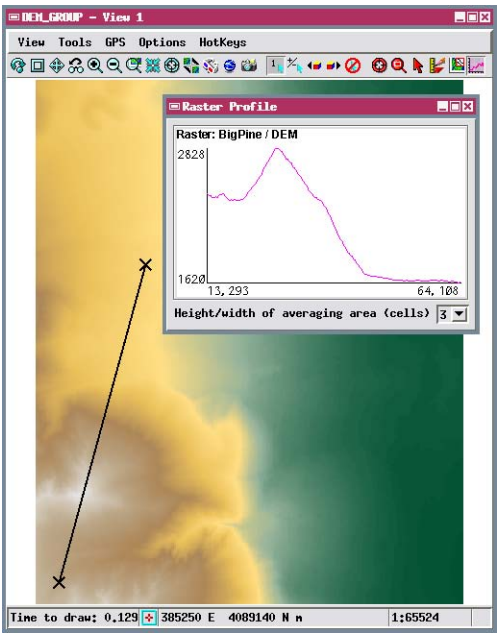
Using a Canvas with a Modeless Dialog

The Raster Profile tool script provides a line tool for drawing a line in the View and a modeless Raster Profile window that shows a graph of cell value (elevation in meters in this example) versus distance along the profile line. The profile is recreated and redrawn whenever the profile line is changed or redrawn in the View (the callback for the line tool, cbToolApply(), also calls the cbRedraw() procedure that handles drawing the graphics).

The graphics context for a drawing canvas in a modeless dialog window is automatically destroyed whenever the window closes and even when the window loses focus (by making another window active). For these reasons the graphics context for the canvas must be recreated each time the canvas is redrawn. Thus you will find the statement

```
gc = canvas.CreateGC()
```

near the beginning of the cbRedraw() procedure in this sample script.



STEPS

- ☑ press the Open Display icon button on the Display Manager, navigate to the SMLDLG sample data folder, and choose DEM_GROUP from the BIG PINE Project File
- ☑ press the Raster Profile tool script icon button on the View window toolbar
- ☑ left-click and drag to draw a profile line in the View; note the resulting profile graphic in the Raster Profile window
- ☑ choose Options / Scripts / Tool Scripts in the View
- ☑ in the Customize Tool Scripts window, click on the Raster Profile entry in the list and press the Edit icon button to open the SML window to show the tool script
- ☑ examine the cbRedraw() procedure in the script
- ☑ choose Display / Exit / from the Display Manager

Advanced Software for Geospatial Analysis

MicroImages, Inc. publishes a complete line of professional software for advanced geospatial data visualization, analysis, and publishing. Contact us or visit our web site for detailed product information.

TNTmips Pro TNTmips Pro is a professional system for fully integrated GIS, image analysis, CAD, TIN, desktop cartography, and geospatial database management.

TNTmips Basic TNTmips Basic is a low-cost version of TNTmips for small projects.

TNTmips Free TNTmips Free is a free version of TNTmips for students and learning professionals with small projects.

TNTedit TNTedit provides interactive tools to create, georeference, and edit vector, image, CAD, TIN, and relational database project materials in a wide variety of formats.

TNTview TNTview has the same powerful display features as TNTmips and is perfect for those who do not need the technical processing and preparation features of TNTmips.

TNTatlas TNTatlas lets you publish and distribute your spatial project materials on CD or DVD at low cost. TNTatlas CDs/DVDs can be used on any popular computing platform.

Index

book element....14
callback....22-26
canvas....32,35
colorbutton....13,14
combobox....8,27
control settings....25
dialog element....5,6,33
editnumber....10,25
edittext....13,26
groupbox....9
id attribute....6,7,19,24,25,33
item....8,9,11,12,18
label....5,8,10-13,28,29,31
layout component....9,10
listbox....8,11,26
menubutton....8,12
modal dialog....4,17,21,34
modeless dialog....4,34,35
page....14,15
pane....10-12
Parse(xml$)....16,17
popup dialog....3
pushbutton....5,11,12,21,22,27,28
radiogroup....9
Read(xmlfile$)....16,17
root element....5,20
script element....23
togglebutton....7,12,26
view....28,29,31
widget....28
XML....4,5
attributes....6
editors....20
elements....5
errors....18-19
tags....5,7
valid....20
well-formed....6,18,20

MicroImages, Inc.

11th Floor - Sharp Tower
206 South 13th Street
Lincoln, Nebraska 68508-2010 USA

Voice: (402) 477-9554
FAX: (402) 477-9559

email: info@microimages.com
internet: www.microimages.com

Reference Guide to SML Dialog Specifications in XML

This document is a reference to the creation of custom SML dialog window specifications in XML format. A dialog specification is a simple text file (or text character string embedded in the SML script) that conforms to the basic format and syntax rules of the Extensible Markup Language (XML). The XML text consists of a set of predefined XML *tags* and *attributes* (described in this document) in a nested, hierarchical structure that mirrors the grouping of components in the dialog window. In comparison to the previously-available X-Motif class structures in SML for custom dialogs, XML specifications provide a simpler, more highly-structured approach to constructing custom SML dialogs, as well as streamlined methods for accessing information input via the dialog. Custom dialog windows created using an XML specification can be used in either SML for X or SML for Windows. Dialogs created using the X-Motif classes can be used only in SML for X.

About XML

XML is a markup meta-language that is designed to allow the creation of structured, self-describing text. XML makes use of *tags* (enclosed between < and > characters) to delimit and identify pieces (elements) of text data. In an SML dialog specification, the data elements identified by the tags correspond to specific components of a dialog window, such as buttons and labels. Delimiting tags occur as a pair, a start tag and end tag, that enclose the relevant text. The start and end tag use the same tag name, but in the end tag the name is preceded by the forward slash "/" character. For example, here is the entry for a text label in an SML dialog specification in XML:

```
<label>Select Composite Type:</label>
```

In this case the <label> tags bracket the text to be used for the label on the dialog.

Data elements in XML can be nested inside other data elements to indicate membership in a group. There may be multiple levels of membership, forming a hierarchical or tree-like data model. This model is well suited to describe a dialog window, because many dialog elements, such as panes, list boxes, and menu buttons, act as "containers" that have other elements as their "contents". (Another way of stating this is that the container element is the "parent" and the contained elements are the "children"). In a dialog specification in XML the contained elements are nested inside the tag pair for the container element. For example, the excerpt below shows the specification for a layout pane that includes two text labels sandwiched around a numeric entry control:

```
<pane Orientation="horizontal">
   <label>Buffer Distance:</label>
   <editnumber id="buffdist" Width="4" Precision="0" Default="100" MinVal="0"/>
   <label>meters</label>
</pane>
```

For ease of editing and reading the dialog specification, the start and end tags for a parent element should be placed on separate lines in the text, with the contained elements identified on the intervening lines. You should also indent lines corresponding to child elements by an equal amount relative to their parent element tags.

Start tags can also include *attributes*. A specific, predefined set of attributes are available for each type of dialog element in an SML dialog specification. You use these attributes to define the individual characteristics and settings for each dialog element, to provide a control handle for the SML script to use to access the control, and to specify a callback procedure associated with a control (if needed). To use an attribute within an element tag, you list the attribute name and assign it a value (in double quotation marks) using an equals sign (=). In the specification excerpt immediately above, the layout pane has an attribute called *Orientation* that has been assigned the value *horizontal*. Attribute names and predefined nonnumeric values are case-sensitive.

Some dialog elements can be completely specified using the tag name and a list of start-tag attributes. There is no other "text" associated with the element to be bracketed by separate start and end tags. In this case the two tags are combined into a so-called *empty tag*. In an empty tag the forward slash character (/) normally used to begin an end tag is placed just before the final character (>) of the tag. The *editnumber* tag in the specification excerpt immediately above is an example of an empty tag.

© 2003 - 2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

In order for SML to correctly parse and interpret a document specification, the specification must follow some simple syntax rules that define a *well-formed* XML document:

1) An element containing text or other elements must have both start and end tags.
2) An empty element tag must have a forward slash (/) before the ending bracket
3) All attribute values must be in quotes.
4) Elements may not overlap.

In general uses of XML, the meanings of tags and attributes are specified in a document type definition (DTD), which can be either contained in the XML file or referenced by it, or alternatively in an external XML Schema. However, the XML text describing an SML dialog does not need to reference a DTD or Schema, because the tag and attribute meanings are processed by the SML parser.

To conform to XML standards, the dialog specification should begin with an XML Declaration containing the version of XML:

```
<?xml version="1.0"?>.
```

For further information about XML, please consult the following:

Learning XML, by Erik T. Ray. O'Reilly & Associates, Sebastopol, CA, 2001. 354 pages.

Mastering XML, by Ann Navarro, Chuck White, and Linda Burman. SYBEX, San Francisco, CA, 2000, 882 pages.

XML Handbook, by Charles F. Goldfarb and Paul Prescod. Prentice-Hall PTR, Upper Saddle River, NJ, 4th Ed., 2002, 1147 pages.

www.w3.org/XML/: XML web site maintained by the World Wide Web Consortium (W3C), the organization that formulates and publishes XML standards. Includes technical specifications and links to other web resources on XML, including tutorials and books.

www.xml.org: Industry web portal on XML maintained by OASIS, the nonprofit Organization for the Advancement of Structured Information Systems. Includes sections on XML basics.

xml.coverpages.org/xml.html: XML section of Cover Pages, maintained by Robin Cover and hosted by OASIS.

www.xml.com: XML site by O'Reilly & Associates.

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

Overview of Tag Set and Classes for SML Dialog Specifications

A dialog created using an XML dialog specification makes use of a set of Graphical User Interface (GUI) classes in SML. Most of the XML tag types in a dialog specification correspond to specific GUI classes. In essence, the XML specification provides a shorthand way to set up and access the various class structures that underlie a custom dialog window. The following is a list of the available dialog elements with their tag name and corresponding GUI class. Each element type is documented fully in subsequent sections of this document.

Tag	Description	Class
	Main Elements	
<root>	root element of the document	None
<dialog>	dialog window	GUI_DLG
<script>	container for an SML script	None
<strings>	container for list of text resource strings	None
<string>	a single text resource string	None
	Layout Elements	
<book>	book of tabbed pages	GUI_LAYOUT_BOOK
<page>	tabbed page in a book	GUI_LAYOUT_PAGE
<pane>	window layout pane	GUI_LAYOUT_PANE
<groupbox>	frame around other controls	GUI_CTRL_GROUPBOX
	Control Elements	
<label>	simple text label	GUI_CTRL_LABEL
<pushbutton>	push button with text or icon	GUI_CTRL_PUSHBUTTON
<togglebutton>	independent toggle button	GUI_CTRL_TOGGLEBUTTON
<colorbutton>	color selection button	GUI_CTRL_COLORBUTTON
<edittext>	text entry control	GUI_CTRL_EDIT_STRING
<editnumber>	numeric entry control	GUI_CTRL_EDIT_NUMBER
<radiogroup>	group of radio (mutually exclusive) buttons	GUI_FORM_RADIOGROUP
<combobox>	combobox menu	GUI_CTRL_COMBOBOX
<menubutton>	button with drop-down menu	GUI_CTRL_MENUBUTTON
<listbox>	scrolled list for selection	GUI_CTRL_LISTBOX
<item>	item in a list, menu, combobox, or radiogroup	None
<canvas>	drawing canvas in a dialog	GUI_CANVAS

In addition to the classes listed above, two other SML classes provide methods to interact with dialog specifications in XML:

class XMLDOC: an XML document. Methods in this class are used to read and parse the XML text.

class XMLNODE: an element (node) in an XML document. Methods in this class allow the script to access the XML elements that correspond to particular components of the dialog, including the node for the dialog window itself.

Callbacks

All dialog elements that represent actual controls can have callback attributes. The attribute is named in such a way as to indicate the type of action happening, like OnClicked, or OnValueChanged.

The value of a callback attribute is a small fragment of SML code, usually a call to a function or procedure defined in the main SML script or in the dialog specification within a <script> element. If the value used for a callback attribute is a call to a function or procedure, include the full procedure name (including parentheses) as the callback value in the dialog specification, as for the *OnOK* attribute in the following example:

```
<dialog id="tigerops" title="Extraction options" OnOK="CheckTogs()">
```

All callbacks can return a numeric value and return 1 by default. Some callbacks, like OnOK for dialogs can prevent the default action by returning 0 instead. If the returned value is required for use in the main script, the value for the callback should be a statement with the keyword "return" followed by the name of the associated user-defined function, as in the following example:

```
<dialog id="tigerops" title="Extraction options" OnOK="return CheckTogs()">
```

The SML code within a <script> element can consist of one or more callback procedures for individual dialog controls, including the dialog's OK and Cancel buttons. In some cases the code within the <script> tag could carry out most of the intended data processing. However, a separate SML script is required to read in the dialog specification file and open the dialog window described in the specification.

Within the SML code in a <script> element, the following conditions apply:

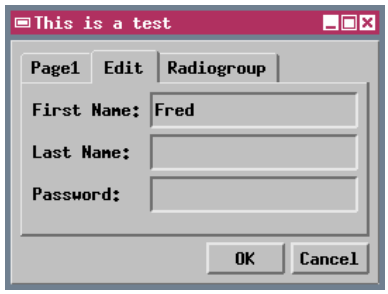
- Global variables and functions defined in the main SML script are available within the <script> element code.
- The default MDLGPARENT for the SMLCONTEXT is set to the LAYOUT_PANE containing the control so that any dialogs popped up while within the callback will automatically appear over the dialog, not somewhere else.
- The variable `this` is predefined to be the control causing the event. The variable corresponds to a class variable of one of the classes derived from GUI_CTRL. The control will already reflect the changes made, so if the event is a toggle button being toggled, `this.GetValue()` will return true if the button was toggled on and false if it was toggled off. To use the variable, you need to do two things: (1) the value you assign for the callback attribute should include `this` as a parameter of the procedure or function; (2) in the definition of the procedure or function, declare `this` as an instance of the relevant GUI_CTRL class. The excerpt below illustrates this structure for a listbox control whose OnChangeSelection callback calls a procedure called SayHello().

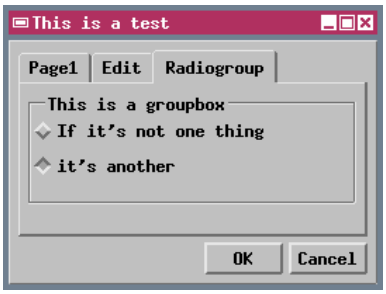
```
<root>
   <dialog id="hello" Title="Bonjour!"
      ...
      <listbox id="listbox" SelectStyle="single" Height="3" Default="French"
         OnChangeSelection="SayHello(this)">
      ...
   <\dialog>
   <script>
      <![CDATA[
         proc SayHello (class GUI_CTRL_LISTBOX this) {
               local string $language;
               language$ = this.GetSelectedItemID();
               ...
            }
         ]]>
   </script>
</root>
```

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

Example Dialog Specification in XML

This dialog specification describes a dialog window with three tabbed panels that include various types of controls.






```
<?xml version="1.0"?>
<root>
   <dialog id="test" title="This is a test" Orientation="vertical" OnOK="GetDlgValues()">
      <book>
         <page Name="Page1">
            <label>This is some text in a label</label>
            <pushbutton Name="Ignore This" OnPressed="TestFunc2();"/>
            <combobox id="combo">
               <item Value="item1">If it's not one thing</item>
               <item Value="item2">it's another</item>
            </combobox>
            <pane Orientation="horizontal">
               <pushbutton Name="Test" Icon="CHECKBOX_BLACK" ToolTip="Check This!"/>
               <togglebutton id="tbutton" Name="This is a togglebutton"/>
            </pane>
         </page>
         <page Name="Edit">
            <pane Orientation="horizontal">
               <label WidthGroup="1">First Name:</label>
               <edittext id="fname" width="20"/>
            </pane>
            <pane Orientation="horizontal">
               <label WidthGroup="1">Last Name:</label>
               <edittext id="lname" width="20"/>
            </pane>
            <pane Orientation="horizontal">
               <label WidthGroup="1">Password:</label>
               <edittext id="password" Opaque="true" width="20"/>
            </pane>
         </page>
         <page Name="Radiogroup">
            <groupbox Name="This is a groupbox">
               <radiogroup id="radiogroup">
                  <item Value="button1" Name="If it's not one thing"/>
                  <item Value="button2" Name="it's another"/>
               </radiogroup>
            </groupbox>
         </page>
      </book>
   </dialog>
   <script language="SML">
      <![CDATA[
         func TestFunc2() {
            PopupMessage("I said to ignore this!");
            return (0);
         }
      ]]>
   </script>
</root>
```

Common Dialog Element Attributes

This is a list of attributes that are common to many of the dialog elements. They are documented here to avoid redundancy. In the succeeding documentation of individual elements, the attributes unique to that element are documented in full, but the available common attributes are merely listed for each element.

`id` – All elements can have an id attribute. The value of an id attribute can be any string, but must be unique within a given XML document. You are not allowed to have two tags with the same id. This attribute allows you to gain access to the dialog elements within SML to retrieve or set their values or state. You don't have to assign ids to elements that you don't need to read or change, such as labels and icons.

`Orientation` – Either "horizontal" or "vertical". The default is the opposite of the parent's orientation or vertical for the main form. All children of the form are laid out from left to right or top to bottom. To make more complex layouts, nest <panes> with alternating orientations.

`HorizAlign` – Controls how the control is aligned horizontally in the available space. Possible values are "Left" (the default), "Right" and "Center".

`VertAlign` – Controls how the control is aligned vertically in the available space. Possible values are "Top" (the default), "Bottom" and "Center".

`HorizResize` – Controls how the control behaves when the parent is resized. Possible values are:
"Expand" – Expands horizontally to fill the available space.
"Fixed" – Width stays the same.
"Relative" – Expands horizontally, keeping the same percentage of space as it had before.

`VertResize` – Controls how the control behaves when the parent is resized. Possible values are:
"Expand" – Expands vertically to fill the available space.
"Fixed" – Height stays the same.
"Relative" – Expands vertically, keeping the same percentage of space as it had before.

`ChildSpacing` – Space between children. The default value is 4.

`ExtraBorder` – Extra border around inside of pane, in addition to ChildSpacing. The default value is 0.

`WidthGroup` – If specified, all controls with the same WidthGroup value will be adjusted to be as wide as the widest control in the group.

`ResourceLookup` – Either "true" or "false". If "false", the string is used as-is. If "true", any string shown on the dialog for the element (such as a label or dialog name) is replaced with a translated text resource version (if found) based on the language locale currently set for the TNT products. Translated versions of the strings used in the dialog must be provided within the dialog specification using <string> elements within one or more <strings> sections at the beginning of the dialog specification (one section for each language). The id for each translated string element should be set to the string actually used in the dialog specification. The specification must also include an empty <strings> section identifying the language in which the strings are initially set for the elements. If no translated resource is found, the original string in the dialog specification is used as-is. The default value for this attribute is inherited from the parent, except for <form> and <dialog>, which default to "true".

`Enabled` – Either "true" or "false". If "true" (the default), the control is enabled. If "false", it's initially disabled and grayed out. The control stays this way unless changed via SML script or C++ code.

NOTE. Any attribute that has "true" and "false" as valid values also accepts "yes" and "no" or "1" and "0".

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

Dialog (Form) Elements

Main Elements

<root>

Description: The required root element of the document. All elements described subsequently must be contained within <root> or one of its children.

Valid inside: nothing

Valid children: `<script>, <strings>, <dialog>`

Attributes: none

SML Class: none

<dialog>

Description: A dialog window.

A dialog can be modal or non-modal, but that is up to the SML code that opens the dialog, not the XML specification. A modal dialog is one that takes control and won't let the program do anything else until the dialog closes. Modal dialogs are automatically provided with "OK" and "Cancel" buttons. If there is a HelpID attribute in the XML specification of the dialog, it will also have a "Help" button. Non-modal dialogs permit program operations and user interactions to continue while they're open. In this case, an "Apply" button is provided along with the "OK" and "Cancel" buttons.

Valid inside: `<root>`

Valid children: Any layout element except `<page>`; any control element except `<item>`

Attributes:

`id, HorizAlign, VertAlign, HorizResize, VertResize, ChildSpacing, ExtraBorder, ResourceLookup, Orientation`

`OnApply` – SML Code to execute when user presses the "Apply" button.

`OnOK` – SML code to execute when user presses the "OK" button. After your OnOK callback is called, the dialog will be closed. You can prevent it from closing by having your callback return 0.

`OnCancel` – SML code to execute when the user presses the "Cancel" button. After your OnCancel callback is called, the dialog will be closed. You can prevent it from closing by having your callback return 0.

`OnOpen` – SML code to execute when the dialog opens.

`OnClose` – SML code to execute when the dialog closes. After your OnClose callback is called, the dialog will be closed. You can prevent it from closing by having your callback return 0.

`OnCloseRequest` – SML Code to execute when the dialog receives a request to close. A return value of 0 prevents the dialog from closing.

`Title` – The title of the dialog.

`HelpID` – A key into tnthelp.txt. If set, the dialog gets a "help" button that opens a help window.

`Buttons` – The buttons to create. Use to specify which of the default buttons are provided. If the value is an empty string (""), none of the default buttons are provided.

SML Class: `GUI_DLG`

<script>

Description: Container for an SML script. An SML script can be enclosed in an XML wrapper. If the file has a .sml extension but an XML header, the SML parser will look for a script tag and run that. The rest of the XML document will be parsed and accessible to the script.

Note that the contents of the script tag must start with `<![CDATA[` and end with `]]>`. Without this, you would have to escape every ampersand (&), greater-than (>), less-than (<) and both single and double quotes.

Valid inside: `<root>`

Valid children: Only a CDATA containing SML code

Attributes:

`id` – Unique identifier for the script (not usually necessary)

`language` – TNTmips only recognizes "SML", which is the default. Include this attribute to ensure that other XML viewers don't try to interpret the CDATA as some other language such as Javascript.

`Usage` – One of the following values specifying the type of script:
"standalone"
"style-point", "style-line", "style-poly"
"select-point", "select-line", "select-poly"
"geoformula", "macroscript", "toolscript"

`StrictSyntax` – either "true" or "false". If "true" (the default) the parser enforces stricter syntax rules (requires semicolons at the end of lines, predeclared variables, etc.).

SML Class: none

<strings>

Description: Container for string elements that provide dialog text resources for a particular language.

Valid inside: `<root>`

Valid children: <string>

Attributes:

`id` – Unique identifier for the string section (not usually necessary)

`language` – The three-letter abbreviation of the language locale. Possible values are:

"are"–	Arabic	"fin" –	Finnish	"plk" –	Polish
"bah"–	Indonesian	"frc" –	French	"ptg" –	Portugese
"ben"–	Bengali	"hrv" –	Croatian	"rom" –	Romanian
"bgr" –	Bulgarian	"hun" –	Hungarian	"rus" –	Russian
"bos" –	Bosnian	"ita" –	Italian	"shc" –	Serbian
"chs" –	Chinese	"jpn" –	Japanese	"sky" –	Slovakian
"deu" –	German	"kor" –	Korean	"tgl" –	Tagalog
"ell" –	Greek	"mys" –	Malaysian	"tha" –	Thai
"enu" –	English	"nld" –	Dutch	"trk" –	Turkish
"esn" –	Spanish	"nor" –	Norwegian	"urd" –	Urdu

SML Class: none

<string>

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

Description: A string that specifies a dialog text resource for a particular language.

Valid inside: `<strings>`

Valid children: text

Attributes:

`id` – Unique identifier for the string (same as text in the specification for the element that uses the resource).

SML Class: none

Layout Elements

<pane>

Description: A layout pane

Valid inside: `<dialog>, <page>, <pane>, <groupbox>`

Valid children: Any layout element except `<page>`; any control element except `<item>`

Attributes:

`id, HorizAlign, VertAlign, HorizResize, VertResize, ChildSpacing, ExtraBorder, ResourceLookup`

`Orientation` – Either "horizontal" or "vertical" (default is the opposite of the parent pane). All children of the form will be laid out from left to right or top to bottom. For more complex layouts, nest `<pane>`s with alternating orientations.

SML Class: `GUI_LAYOUT_PANE`

<page>

Description: A tabbed page in a `<book>`

Valid inside: `<book>`

Valid children: Any layout element except `<page>`; any control element except `<item>`

Attributes:

`id, Orientation, ResourceLookup`

`Name` – the label on the page's tab

`OnSetActive` – SML code to execute when the page is activated

SML Class: `GUI_LAYOUT_PAGE`

<book>

Description: book of tabbed pages

Valid inside: `<dialog>, <page>, <pane>, <groupbox>`

Valid children: `<page>`

Attributes:
`id, HorizAlign, VertAlign, HorizResize, VertResize, ChildSpacing, ExtraBorder, ResourceLookup`

SML Class: `GUI_LAYOUT_BOOK`

<groupbox>

Description: A frame around other controls

Valid inside: `<dialog>, <page>, <pane>, <groupbox>`

Valid children: Any layout element except `<page>`; any control element except `<item>`

Attributes:
`id, HorizAlign, VertAlign, HorizResize, VertResize, ChildSpacing, ExtraBorder, Orientation, ResourceLookup, Enabled`
`Name` – the label on the upper left edge of the box (optional)

SML Class: `GUI_CTRL_GROUPBOX`

Control Elements

<label>

Description: A simple text label

Valid inside: any layout element except `<book>`

Valid children: text

Attributes:
`id, HorizAlign, VertAlign, HorizResize, VertResize, ResourceLookup, WidthGroup, Enabled`
`TextAlign` – How to align the text of the label. Possible options are:
"LeftNoWrap" – Left justify without word wrapping
"Left" – Left justify, MFC may word-wrap if too long, but X won't
"Center" – Center, MFC may word-wrap if too long, but X won't
"Right" – Right justify, MFC may word-wrap if too long, but X won't

SML Class: `GUI_CTRL_LABEL`

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

<pushbutton>

Description: A push button with either text or icon.

Valid inside: any layout element except <book>

Valid children: none

Attributes:

id, HorizAlign, VertAlign, HorizResize, VertResize, WidthGroup, ResourceLookup, Enabled

Name – the text to be placed on the button.

Icon – Icon to display based on ICONID values. For example, to use ICONID_CREATE_FILE, use the string "CREATE_FILE". Possible values are those listed for the "iconid" parameter of the CreateIcon() method of class GUI_CTRL_BUSHBUTTON.

ToolTip – ToolTip text to display for the button (only if it has an icon)

OnPressed – SML code to execute to call when the button is pressed. See section on callbacks for details.

SML Class: GUI_CTRL_PUSHBUTTON

<togglebutton>

Description: A toggle button.

Valid inside: any layout element except <book>

Valid children: none

Attributes:

id, HorizAlign, VertAlign, HorizResize, VertResize, WidthGroup, ResourceLookup, Enabled

Name – the label to be placed next to the button

Icon – Icon to display. Based on ICONID values. For example, to use ICONID_CREATE_FILE, use the string "CREATE_FILE". You can get a full list of valid Icon IDs by looking at the documentation for the GUI_CTRL_TOGGLEBUTTON class in SML.

ToolTip – tooltip text to display for the button (only if it has an icon)

Type – "check" or "radio". Radio buttons are round. A check box is square. The default is "check". Note that radio behavior is not automatic. If you want automatic radio behavior, create a <radiogroup> with an <item> for each radio button.

Selected – Either "true" or "false". If true, the button is initially selected. This can be overridden through commands in the SML script.

OnChanged – SML code to execute when the button's state changes (when the user toggles the button on or off). For radio buttons, an OnChanged callback is also triggered for the button that is being toggled off. See section on callbacks for details.

Note: to create mutually exclusive radio buttons, create a <radiogroup> with <item>s instead of <togglebutton>s

SML Class: GUI_CTRL_TOGGLEBUTTON

<colorbutton>

Description: A color selection button.

The button has no text, but the body of the button will show the selected color. Pushing the button pops up a color selection dialog.

Valid inside: any layout element except `<book>`

Valid children: none

Attributes:

`id, HorizAlign, VertAlign, HorizResize, VertResize, WidthGroup, Enabled`

`ReadOnly` – Either "true" or "false". If true, the button shows a color but the user cannot change it. The default is "false"

`AllowTransparent` – Either "true" or "false". If true, the color selection dialog will allow transparency settings. The default is "false".

`OnChangeColor` – SML code to execute to call when color is changed. See section on callbacks for details.

SML Class: `GUI_CTRL_COLORBUTTON`

<edittext>

Description: A text entry control.

The control has no built-in text label.

Valid inside: any layout element except `<book>`

Valid children: none

Attributes:

`id, HorizAlign, VertAlign, HorizResize, VertResize, WidthGroup, ResourceLookup, Enabled`

`OnChanged` – SML code to execute when the value is edited.

`Width` – Width of the text control in "typical" characters.

`MaxLength` – Maximum length of text to allow.

`Justify` – Text alignment. Either "left" or "right". The default is "left".

`OnActivate` – SML code to execute when the user presses <Enter> with the cursor in the edit control.

`ReadOnly` – Either "true" or "false". If "true", the user cannot change the value, but the control does not look disabled. The default is "false".

`Opaque` – Either "true" or "false". If "true", the control shows "*" for password input. Default is "false".

SML Class: `GUI_CTRL_EDIT_STRING`

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

<editnumber>

Description: A numeric entry control.

The control has no built-in text label.

Valid inside: any layout element except `<book>`

Valid children: none

Attributes:

`id, HorizAlign, VertAlign, HorizResize, VertResize, WidthGroup, ResourceLookup, Enabled`

`OnChanged` – SML code to execute when the value is edited.

`Width` – Width of the text control in "typical" characters.

`MaxLength` – Maximum length of text to allow.

`Justify` – Text justification. Either "left" or "right". The default is "right".

`OnActivate` – SML code to execute when the user presses <Enter> in the edit control.

`ReadOnly` – Either "true" or "false". If "true", the user cannot change the value, but the control does not look disabled. The default is "false"

`Default` – The default value (optional)

`MinVal` – Minimum allowed value. Default is no minimum.

`MaxVal` – Maximum allowed value. Default is no maximum.

`AddOne` – Either "true" or "false". If "true", the value shown in the control is always one more than the actual data value. Default is "false".

`BlankZero` – Either "true" or "false". If "true", blank the field if the value is 0.0. The default is "false". Note, the IEEE NaN value will always cause the control to show blank.

`Format` – One of the following:

"Decimal" – Normal decimal format (default)
"Exponential" – Show in scientific notation.
"Latitude" – A latitude value. Actual prompt value is decimal degrees.
"Longitude" – A longitude value. Actual prompt value is decimal degrees.
"DegMinSec" – An angle in Degrees, Minutes and Seconds. Actual prompt value is decimal degrees.

`Precision` – Number of digits after the decimal place (Default is 6)

SML Class: `GUI_CTRL_EDIT_NUMBER`

<item>

Description: An item in a listbox, combobox, menubutton, or radio group.

Valid inside: `<combobox>, <listbox>, <menubutton>, <radiogroup>`

Valid children: `text`

Attributes:

`ResourceLookup`

`Name` – The button label for a radiogroup button or the item's menu entry for the other parent controls.

`Value` – The value of this item. This is the value that is returned when calling GetValue() on the parent object if this item is selected. If omitted, the default value is the item's index, starting at 1.

`Selected` – Either "true" or "false". If true, the item is selected by default. This overrides the `Default` attribute of the parent. Default is false. If more than one item is marked as selected, the last one marked is the one selected unless the parent is a multi-select list.
`Icon` – An icon ID (only valid if the `<item>` is inside a `<radiogroup>`).

SML Class: none

<radiogroup>

Description: A group of radio buttons (mutually exclusive).

Valid inside: any layout element except `<book>`

Valid children: `<item>`

Attributes:
`id, HorizAlign, VertAlign, HorizResize, VertResize, WidthGroup, Orientation, ResourceLookup, Enabled`
`Default` – The `Value` of the button that is to be on by default. This can be overridden by setting the `Selected` attribute of one of the items in the group
`OnSelection` – SML code to execute when the selected button changes.

SML Class: `GUI_FORM_RADIOGROUP`

<combobox>

Description: A combo-box.

Valid inside: any layout element except `<book>`

Valid children: `<item>`

Attributes:
`id, HorizAlign, VertAlign, HorizResize, VertResize, WidthGroup, ResourceLookup, Enabled`
`Default` – The `Value` of the item that is to be selected by default. This can be overridden by setting the `Selected` attribute of one of the items in the combobox.
`OnSelection` – SML code to execute when the selected item changes.
`Width` – Width of the control in "typical" characters. The default is to base the width on the width of the widest item in the list.
`Height` – The maximum number of items to show when the list is opened. Default is 7. If there are more items than that, the list has a scrollbar.
`Sort` – Either "true" or "false". If "true", then the values are sorted. The default is false.

SML Class: `GUI_CTRL_COMBOBOX`

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

<listbox>

Description: A simple scrolled list from which the user can view and select items.

Valid inside: any layout element except `<book>`

Valid children: `<item>`

Attributes:

`id, HorizAlign, VertAlign, HorizResize, VertResize, WidthGroup, ResourceLookup, Enabled`

`Default` – The `Value` of the item that is to be selected by default. This can be overridden by setting the `Selected` attribute of one of the items in the combobox.

`OnChangeSelection` – SML code to execute when the selected item changes.

`OnDoubleClick` – SML code to execute when the user double-clicks on a listitem.

`Width` – Width of the control in "typical" characters. The default is to base the width on the width of the widest item in the list.

`Height` – Height of the list in lines (default is 5).

`Sort` – Either "true" or "false". If "true", then the values are sorted. The default is false.

`SelectStyle` – One of the following:

"single" – (the default) allows one item to be selected.
"multi" – Multiple items selectable by simple toggle of each item
"extended" – Multiple items selected by SHIFT/CTRL key and mouse

SML Class: `GUI_CTRL_LISTBOX`

<menubutton>

Description: A pushbutton (text or icon) with a drop-down menu.

Valid inside: any layout element except `<book>`

Valid children: none

Attributes:

`id, HorizAlign, VertAlign, HorizResize, VertResize, WidthGroup, ResourceLookup, Enabled`

`Name` – the text to be placed on the button

`Icon` – Icon to display. Based on ICONID values. For example, to use ICONID_CREATE_FILE, use the string "CREATE_FILE". You can get a full list of valid Icon IDs by looking at the documentation for the GUI_CTRL_TOGGLEBUTTON class in SML.

`ToolTip` – tooltip text to display for the button (only if it has an icon)

`OnSelection` – SML code to execute when a menu item is selected.

`OnMenuPopup` – SML code to execute just before the menu is shown.

SML Class: `GUI_CTRL_MENUBUTTON`

<canvas>

Description: A drawing canvas.

Valid inside: any layout element except `<book>`

Valid children: none

Attributes:

`id, HorizAlign, VertAlign, HorizResize, VertResize, WidthGroup, ResourceLookup, Enabled`

`Height` – The height of the drawing area in screen pixels.

`Width` – The width of the drawing are in screen pixels

`OnLeftDown` – SML code to execute when the left mouse button is pressed.

`OnLeftUp` – SML code to execute when the left mouse button is released.

`OnRightDown` – SML code to execute when the right mouse button is pressed.

`OnRightUp` – SML code to execute when the right mouse button is released.

`OnMouseMove` – SML code to execute when the mouse cursor is moved.

`OnPaint` – SML code to execute when the canvas needs to be drawn.

`OnSize` – SML code to execute when the canvas size changes.

SML Class: GUI_CANVAS

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

Using an XML Dialog Specification with an SML Script

To use a dialog specification in XML in conjunction with an SML script, the script must first declare an instance of the class XMLDOC. Methods within this class are then used to ingest and parse the XML text, creating the necessary structures in memory for use throughout the remainder of the SML script.

The XML dialog specification used with a particular SML script can be either embedded in the script or in a separate file. Different methods in class XMLDOC are used to ingest the XML text in these two cases. If you want to incorporate the dialog specification directly within the SML script, you assign the XML-formatted text containing the specification to a string variable. In most cases this specification will span multiple lines of text. You can include multiple text lines in an assignment statement by enclosing the text in single quotation marks, as in the following example:

```
xml$ = '<?xml version="1.0"?>
<root>
   <strings language="enu"/>     <!-- English   -->
   <strings language="frc">      <!-- French translations -->
      <string id="Sample">Échantillon</string>
      <string id="Hello">Bonjour</string>
   </strings>
   <strings language="esn">      <!-- Spanish translations -->
      <string id="Sample">Muestra</string>
      <string id="Hello">Hola</string>
   </strings>
   <dialog id="hello" title="Sample" ResourceLookup="true" OnOK="OnOK()">
      <label>Hello</label>
   </dialog>
</root>';

class XMLDOC doc;
doc.Parse(xml$);
```

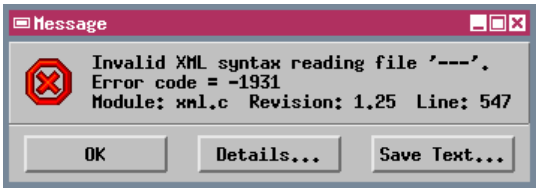
As the above example shows, when the XML text has been assigned to a text string, the XMLDOC class method `Parse(xml$)` is used to parse the XML text.

If you prefer to compose the XML text in an external editor and keep it as a separate .xml file, you use the XMLDOC class method `Read(filename$)` to read the external file and automatically parse its contents. The filename passed to this method must include the full directory path to the file as well as its filename and extension. If you keep the SML script and the XML file in the same directory, you can simplify the task of specifying the path (and make the script more readily portable) by using the CONTEXT class structure. When a script is run, an instance of this class called `_context` is created automatically. The class member `_context.ScriptDir` finds the path to the directory the script is in and can be used as a substitute for this path in string expressions, as in the following example that refers to a dialog specification file “text.xml” in the same directory as the SML script. A string expression is used to concatenate the script’s directory path with the name of the target file:

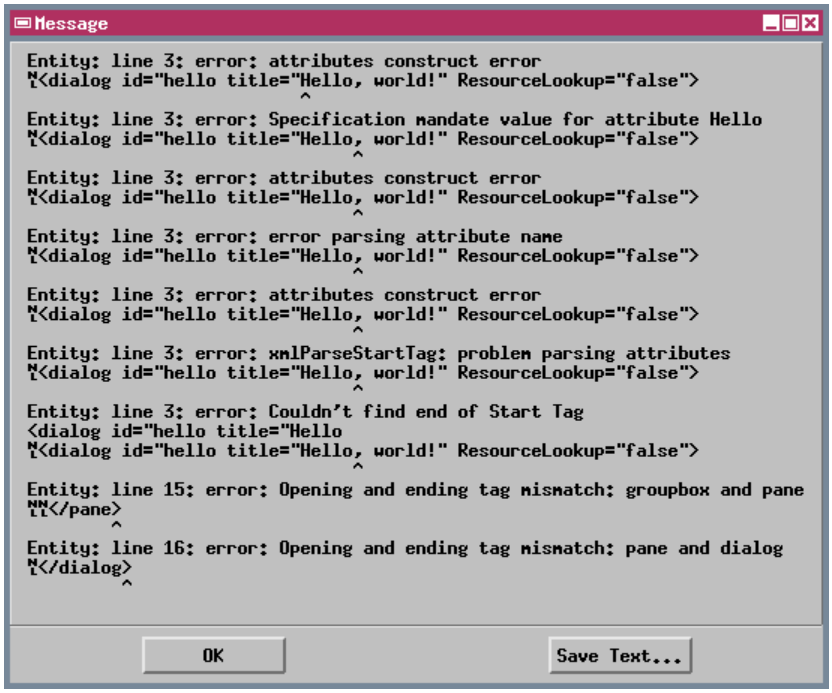
```
class XMLDOC doc;
xmlfile$ = _context.ScriptDir + "/test.xml";
doc.Read(xmlfile$);
```

Both the `Parse(xml$)` and `Read(xmlfile$)` class methods also check the XML syntax of the dialog specification and return an error code (a negative integer) if syntax errors are found. You can write the script to check the returned value and pop up an error message dialog if needed, as shown below:

```
numeric err;
class XMLDOC doc;
xmlfile$ = _context.ScriptDir + "/test.xml";

err = doc.Read(xmlfile$);

if (err < 0) {
   PopupError(err);
   Exit();
   }
```



Pressing the Details... button on the error message dialog opens another window listing the syntax errors. Note that a single error (such as missing closing quotes on an attribute value or a missing end tag) may trigger a number of listings in this window.



Opening the Dialog

Opening the dialog specified by the XML text requires several steps:

1) Use the XMLDOC class method `GetElementByID()` to get the dialog element from the parsed XML and assign it to an instance of class XMLNODE, the class used to represent elements in a parsed XML structure.

2) Set that XMLNODE instance as the source for an instance of class GUI_DLG, the SML class that actually represents the dialog window.

3) Use a GUI_DLG class method to open the dialog window as either a modal or nonmodal dialog. A modal dialog takes control and won't let the SML program do anything else until the dialog closes. Modal dialogs are automatically provided with "OK" and "Cancel" buttons. Nonmodal dialogs allow other program operations and user interactions to continue while they are open. They are automatically provided with "OK", "Apply", and "Cancel" buttons.

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

The script excerpt below carries out these steps for a modal dialog:

```
class XMLNODE dlgnode;
dlgnode = doc.GetElementByID("hello");     # “hello” is the value assigned to the id attribute
                                                                    # for the dialog element in the
XML text
if (dlgnode == 0) {           # pop up an error message if dialog node can’t be found, then exit
   PopupMessage("Could not find dialog node in XML document");
   Exit();
}

class GUI_DLG dlg;
dlg.SetXMLNode(dlgnode);    # set this XML node as the source for the dialog window
numeric ret;
ret = dlg.DoModal();          # open the window as a modal dialog
```

When the `DoModal()` class method is called in the script, the method remains active until the user closes the modal dialog. The method then returns the value -1 if the user pressed the Cancel button or 0 if the user pressed the OK button. The script can assign the returned value to a numeric variable (the variable `ret` in the above example), then check this value to determine succeeding actions. This strategy provides an easier alternative to writing explicit `OnCancel()` and `OnOK()` callback procedures. Control/settings dialogs in standalone scripts are best set up as modal dialogs; all user-input can be recorded using the modal dialog, then the remainder of the script can read these settings and process the data accordingly.

Modeless dialogs are required in event-driven scripts such as tool scripts and display control scripts that provide script interaction with geospatial data displayed in TNTmips Display process or in TNTatlas. These scripts consist of a series of procedures or functions, each driven by an event such as a mouse-button press. To create and open a dialog as a nonmodal dialog, you would use the GUI_DLG class method `CreateModeless()` to create the dialog in memory and `Open()` to open the dialog. The dialog might be created in one event-driven procedure (such as when a tool script’s tool is initialized by opening the group or layout) and opened in another (such as each time the tool is activated).

Reading Dialog Control Values

Once the dialog has been opened and the user has set its controls, the script needs to get the control settings to continue with further processing. Each control in a dialog has a “value” that is either a number or a string. For `editnumber` and `edittext` controls, the value is simply the number or string (respectively) entered by the user. For a `togglebutton`, the value can be retrieved as either a number or a string; it is 1 or "yes" if set and 0 or "no" if not set. For controls that involve selection of an `item` (`combobox`, `listbox`, `menubutton`, and `radiogroup`), the value of the control is the value of the selected item. The value of each item is set by the *Value* attribute you assigned to that item in the dialog specification. You can use any character string for the item attribute, but obviously each item in a particular control should have a unique value. If you use only numerals in the attribute string, you can read the value as either a number or a string. If you omit Value attributes for the items, the default value is the numeric index (position) in the list, beginning with 1.

For modal dialogs, dialog settings should be retrieved before the dialog closes and the controls are destroyed. Settings can be retrieved within callback procedures for individual controls or within the OnOK callback procedure for the dialog window. There are several methods that can be used to get control settings. The simplest and most direct way to get the control settings is to use the GUI_DLG class method `GetValues()` to read the values of all of the controls at once. These values are returned as an instance of the class GUIFORMDATA. You can then use GUIFORMDATA class methods `GetValueNum(ctrl_id$)` or `GetValueStr(ctrl_id$)`, where `ctrl_id$` is the *id* attribute you assigned for the control, to read each control value out of this structure as needed.

An alternate method is to use the GUI_DLG class methods `GetCtrlValueNum(ctrl_id$)` and `GetCtrlValueStr(ctrl_id$)` to retrieve the value of any control individually from the dialog. This method is less efficient than the previous one if you need to retrieve values for several controls. Internally each of these

functions calls the `GetValues()` class method to read all the control values, but returns only the requested value and discards the rest.

A third method uses the `GetCtrlByID()` class method in GUI_DLG to get the control handle for a dialog control, then a `GetValueNum()` or `GetValueStr()` method in the individual GUI_CTRL_... class to read the value.

Setting Dialog Control Values

For most dialog windows, you can set the default condition for each control (such as default value for an `editnumber` field, default state for a `togglebutton`, and default selection for a `combobox`) in the dialog specification in XML using the attribute provided for each control. However, you can also use statements in the main SML script (or in callbacks for other controls in the dialog) to set a control. For example, in a dialog that requires the selection of several input objects, you might want the dialog to show the name of the object after each has been selected. You can use an `edittext` control (set to be read-only) next to the selection button for this purpose. The callback for the selection button can construct a text string from the file name and object name of the selected object and set this as the value for the `edittext` control. Class methods in the GUI_DLG class (`SetCtrlValueNum(ctrl_id$)` and `SetCtrlValueStr(ctrl_id$)`) provide the simplest means to set control values. There are also methods in the GUI_CTRL_... class for the individual control type that can be used to set the control value or to set which item is selected by default in a group control.

Sample Script: Read and Set Control Values

The sample script below (xmldlg.sml) opens the dialog created by the specification on page 5 of this document (from a file named test.xml). After the user sets the controls on the dialog and presses the OK button, the script reads the values from the dialog using each of the methods outlined above and prints the values to the SML Console Window. It also sets default values for several of the controls before the dialog is opened. You can download the script and dialog specification from www.microimages.com/downloads/xmlscripts.htm.

```
numeric err, ret;              # variable declarations
string xmlfile$;
class GUI_DLG dlg;

func TestFunc() {              # define test callback function
   PopupMessage("I Said Ignore it!");
   }

proc GetDlgValues() {

   # Here are four ways to get settings out of the dialog.  The extracted values are printed
   # to the console window as an example.

   #  1 -- Use the GetValues() class method in GUI_DLG to get all of the control settings at
   #          at once.  They are returned to a previously-declared instance of class GUI_FORMDATA.
   #          Use GetValue...() methods in that class to read the values as needed.
   printf("Method 1...\n");
   class GUI_FORMDATA data;
```

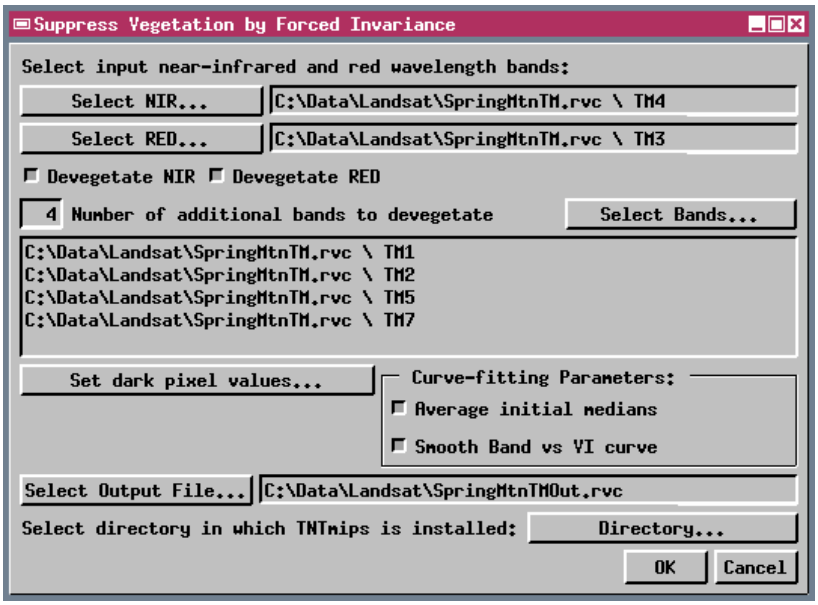
 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

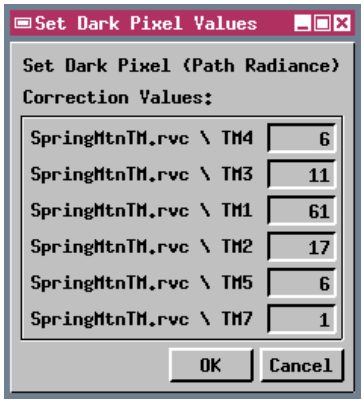
```
   data = dlg.GetValues();
   printf("  RadioGroup value = %s\n", data.GetValueStr("radiogroup") );
   printf("  FirstName: %s\n", data.GetValueStr("fname") );
   printf("  Togglebutton numeric value = %d\n", data.GetValueNum("tbutton") );
   printf("  Togglebutton string value = %s\n\n", data.GetValueStr("tbutton") );

   # 2 -- Use GetCtrlValueNum() and GetCtrlValueStr() class methods in GUI_DLG to ask the
   #       dialog for each control value individually as needed.  No GUI_FORMDATA class
   #       instance is required.  Less efficient than #1 if multiple control values are
   #       needed.  Internally it calls dlg.GetValues(), pulls out the one value asked for,
   #       and discards the rest.
   printf("Method2...\n");
   printf("  RadioGroup value = %s\n", dlg.GetCtrlValueStr("radiogroup") );
   printf("  FirstName: %s\n", dlg.GetCtrlValueStr("fname") );
   printf("  Togglebutton numeric value = %d\n", dlg.GetCtrlValueNum("tbutton") );
   printf("  Togglebutton string value = %s\n\n", dlg.GetCtrlValueStr("tbutton") );

   # 3 -- Use the GetCtrlByID() method in GUI_DLG to get the control handle for a control class,
   #       then a GetValue...() method in the individual GUI_CTRL_... class to read the control
   #       value individually as needed.
   printf("Method3...\n");
   class GUI_CTRL_EDIT_STRING fname;
   class GUI_FORM_RADIOGROUP radio;
   class GUI_CTRL_TOGGLEBUTTON tbutton;
   fname = dlg.GetCtrlByID("fname");
   radio = dlg.GetCtrlByID("radiogroup");
   tbutton = dlg.GetCtrlByID("tbutton");
   printf("  RadioGroup value = %s\n", radio.GetSelected() );
   printf("  FirstName: %s\n", fname.GetValue() );
   printf("  Togglebutton numeric value = %d\n", tbutton.GetValueNum() );
   printf("  Togglebutton string value = %s\n\n", tbutton.GetValueStr() );

   # 4 -- A more compact (but perhaps less clear) version of method 3.  Methods to get the control
   #       handle and its value are strung together, eliminating the need to declare the control handle
   #       class variable.
   printf("Method 4...\n");
   printf("  RadioGroup value = %s\n", dlg.GetCtrlByID("radiogroup").GetValueStr() );
   printf("  FirstName: %s\n", dlg.GetCtrlByID("fname").GetValueStr() );
   printf("  Togglebutton numeric value = %d\n",
   dlg.GetCtrlByID("tbutton").GetValueNum() );
   printf("  Togglebutton string value = %s\n\n",
   dlg.GetCtrlByID("tbutton").GetValueStr() );

   # NOTE: if the control value must be accessed in several places, store it as a variable for
   # reuse.  Values read from GUIFORMDATA or the dialog would be stored as numeric or string
   # variables.

   }  # end GetDlgValues()


##### Main script #####

clear();            # clear the console window

class XMLDOC doc;
xmlfile$ = _context.ScriptDir + "/test.xml";
err = doc.Read(xmlfile$);       # read and parse the dialog specification
```

```
if (err < 0) {
   PopupError(err);        # Popup an error dialog.  "Details" button will say what's wrong.
   Exit();
   }

class XMLNODE dlgnode;dlgnode = doc.GetElementByID("test");        # get the dialog element from the                                                                    #
parsed XML
if (dlgnode == 0) {
   PopupMessage("Could not find dialog node in XML document");
   Exit();
   }

dlg.SetXMLNode(dlgnode);     # set the XML dialog element as the source for dialog class
dlg.SetCtrlValueStr("fname", "Fred");     # set value for edittext control "fname"
dlg.SetCtrlValueStr("radiogroup", "button2");     # set value for radiogroup control

ret = dlg.DoModal();        # open as modal dialog
# Note: ret will be -1 if user hit cancel, 0 for OK

printf("DoModal() returned %d\n", ret);
```

Dynamic Customization of Dialogs

In a script that launches a complex processing sequence, it may not be possible to bring all the required controls together into one dialog window. In such cases, a push button on the main dialog window can be set up to open an auxiliary dialog window that is also specified in XML. One example is provided by the sample script deveg68.sml. This script is designed to process a multispectral image to suppress the expression of vegetation. The script and a color plate describing it are available from www.microimages.com/downloads/xmlscripts.htm.





The illustration above left shows the main dialog window created by the devegetation script. A numeric Dark Pixel value is required for each multispectral image band to be processed. To enter these values, the user presses the Set dark pixel values... pushbutton on the main dialog window; the callback procedure for this button creates and opens the auxiliary Set Dark Pixel Values window (above right). This window shows a list of the bands to be processed;

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com

each entry consists of a label element (with the file name and object name) and an `editnumber` control, both of which are contained in a layout pane with horizontal orientation.

Note that the file and object names required for the Set Dark Pixel Values dialog are not known in advance, and even the number of bands to be processed is not fixed. Therefore only a skeletal representation of this dialog can be provided by the embedded XML specification in the callback. This static specification sets up the dialog itself, the label at the top, the `groupbox`, and a blank layout pane within the `groupbox`. After the static specification is parsed into memory, the remaining elements of the dialog specification must be added dynamically to the XML structure in memory, using information from the input objects the user has selected, before the dialog is opened.

Methods in the class XMLNODE are used to modify the XML dialog structure in memory. You can add a new "child" element to any existing element and set their attribute values. Any element of the static XML specification that needs to be modified later should have an *id* attribute so the element can be accessed. In this case the blank layout pane inside the `groupbox` is assigned the *id* "dplist". To add each entry in the required list, a horizontal `pane` is added as a child element to dplist. The new pane for the current entry then is used as the parent for a child `label` element and a child `editnumber` element, with attribute values derived from the corresponding input object.

The script excerpt below shows the first portion of the callback procedure, which includes the static XML specification for the Set Dark Pixel Values dialog and the code that adds the first band entry to the window. The other band entries are handled in a similar manner.

```
proc SetDP () {

   ### Create string variable with XML specification of dialog
   ### to enter dark-pixel-correction values
   xmldp$ = '<?xml version="1.0"?>
   <root>
      <dialog id = "dpdlg" title = "Set Dark Pixel Values" OnOK = "dpOK()" >
         <label>Set Dark Pixel (Path Radiance)</label>
         <label>Correction Values:</label>
         <groupbox ExtraBorder = "3">
            <pane id = "dplist" Orientation = "Vertical"/>
         </groupbox>
      </dialog>
   </root>';

   ### Parse XML text for dialog into memory; return error if there are syntax errors.
   err = docdp.Parse(xmldp$);

   if ( err < 0 ) {
      PopupError( err ); # pop up an error dialog
      Exit( );
      }

   ##############################
   ### Modify the XML structure in
   ### memory before opening the dialog
   ##############################

   ### Get the id for the parent pane that will contain the list of bands and the
   ### numeric fields for the correction values
   class XMLNODE dplist;
   dplist = docdp.GetElementByID( "dplist" );
```

```
    ########################
    ### Add horizontal pane for the first row of dialog elements
    ### (label and numeric field for NIR band)
    ########################
    class XMLNODE paneNIR;
    paneNIR = dplist.NewChild( "pane" );
    paneNIR.SetAttribute( "Orientation", "Horizontal");


    # Add label with name of NIR band to the NIR pane
    class XMLNODE labelNIR;
    labelNIR = paneNIR.NewChild( "label" );
    nirprtext$ = sprintf("%s.%s \\ %s", FileNameGetName(nirfile$),
                         FileNameGetExt(nirfile$), nirobjname$);
    labelNIR.SetText(nirprtext$);

    # Add editnumber field to the NIR pane and set its attributes
    class XMLNODE prEditNIR;
    prEditNIR = paneNIR.NewChild( "editnumber" );
    prEditNIR.SetAttribute( "id", "prEditNIR" );
    prEditNIR.SetAttribute( "Width", "5" );
    prEditNIR.SetAttribute( "MinVal", "0" );
    prEditNIR.SetAttribute( "MaxVal", "255" );
    prEditNIR.SetAttribute( "Default", NumToStr(nirmin) );
    prEditNIR.SetAttribute( "Precision", "0" );

   [...code for additional list entries...]

}   # end of SetDP()
```

 © 2003-2007 MicroImages, Inc. 206 South 13th Street, Lincoln, Nebraska 68508-2010 www.microimages.com