Using the Glade Interface Designer for Rapid Application Development in Python

This tutorial will walk you through the steps required to build an AWIPS useful application. The application will simply be a tool to send a RWT product over to CRS. Here's a screen shot of what the application should look like after we're done with this tutorial.

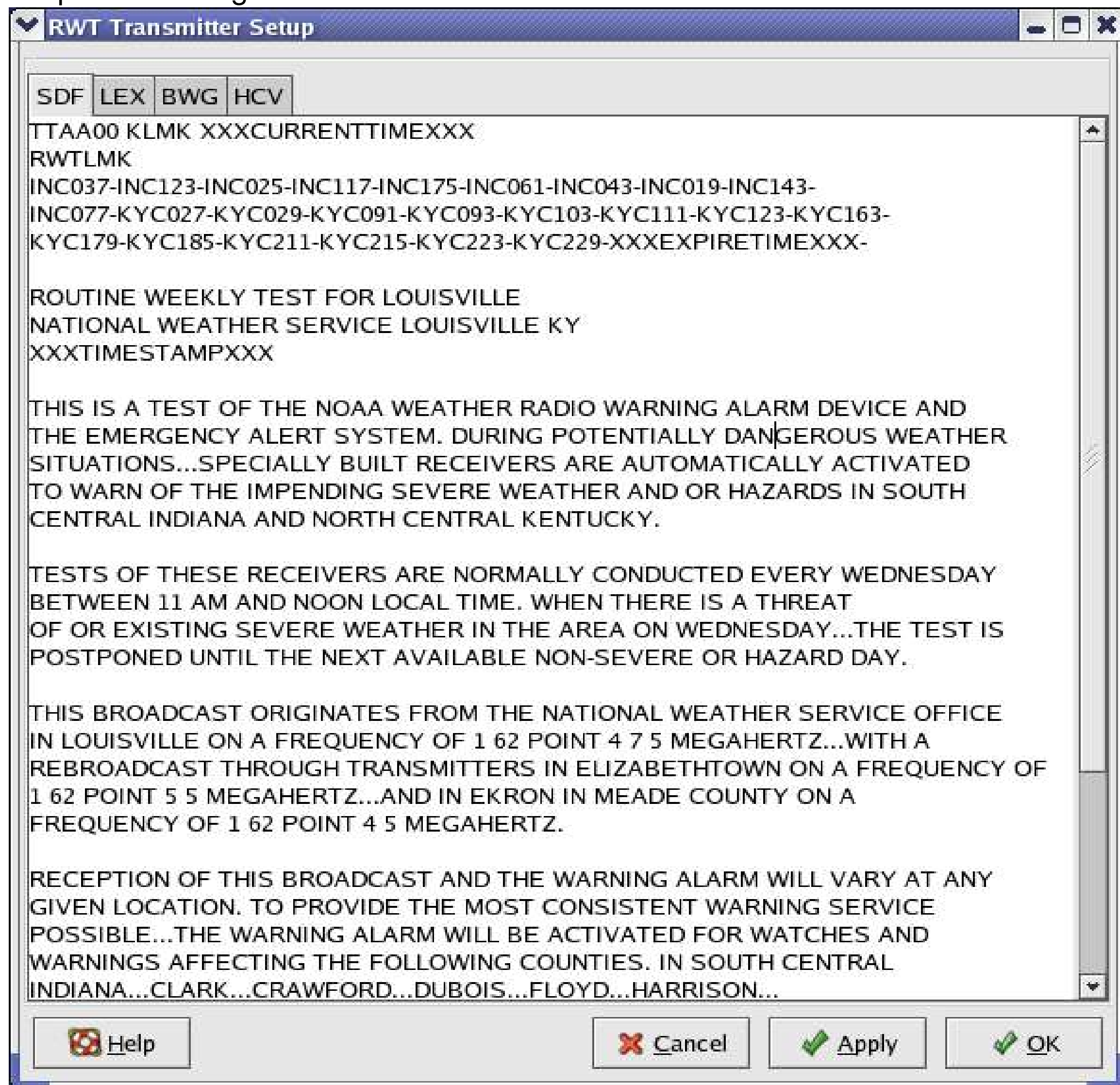
The user selects a transmitter and clicks the “Apply” button to have the RWT sent over to the pending side of NWRWAVES Browser.

There are 4 dialog boxes that go along with the main application: Properties, About, Help and Alert.

- Properties dialog is where we store our template information that can be edited by the user. The template is basically a RWT product with special sections that our program will fill in when the user clicks the main window 'Apply' button.
- About just shows information about the program.
- Help gives some general help information about how to edit the templates contained in the Properties dialog box.
- Alert is a small dialog that reminds the user to check NWRWAVES pending directory.

Screen-shots of the dialog boxes

Properties Dialog



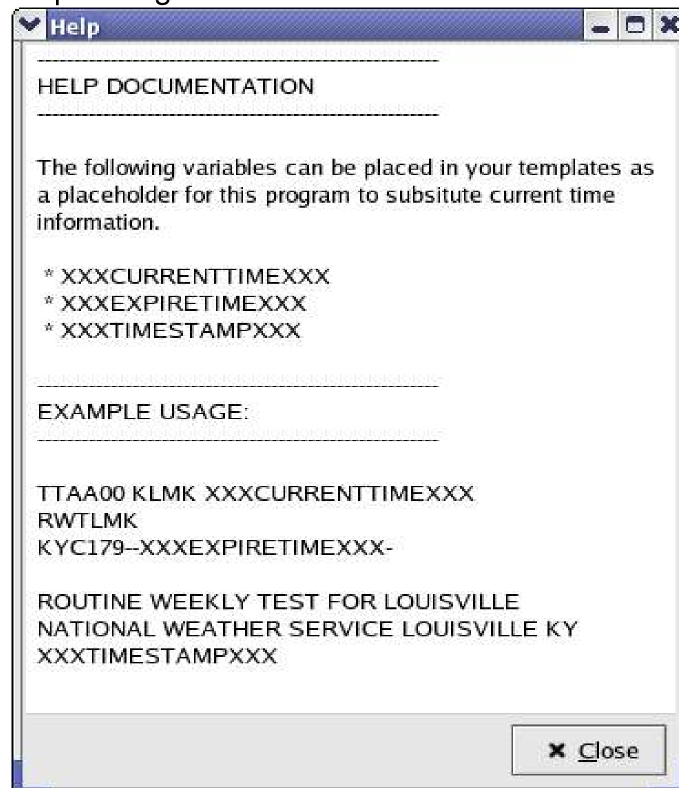
The template has special variables that begin and end with ' XXX .These variables will be filled in by the program when the user clicks the main window 'Apply' button.

The ' Apply' and 'OK' buttons save any user changes to the template. The 'Help' button provides some help on how to set up a template.

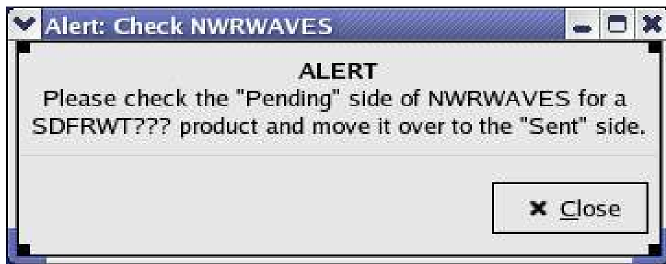
About dialog

You will be the one programming this application, so you should add your name to this list and move the names above to suite your taste.

Help Dialog

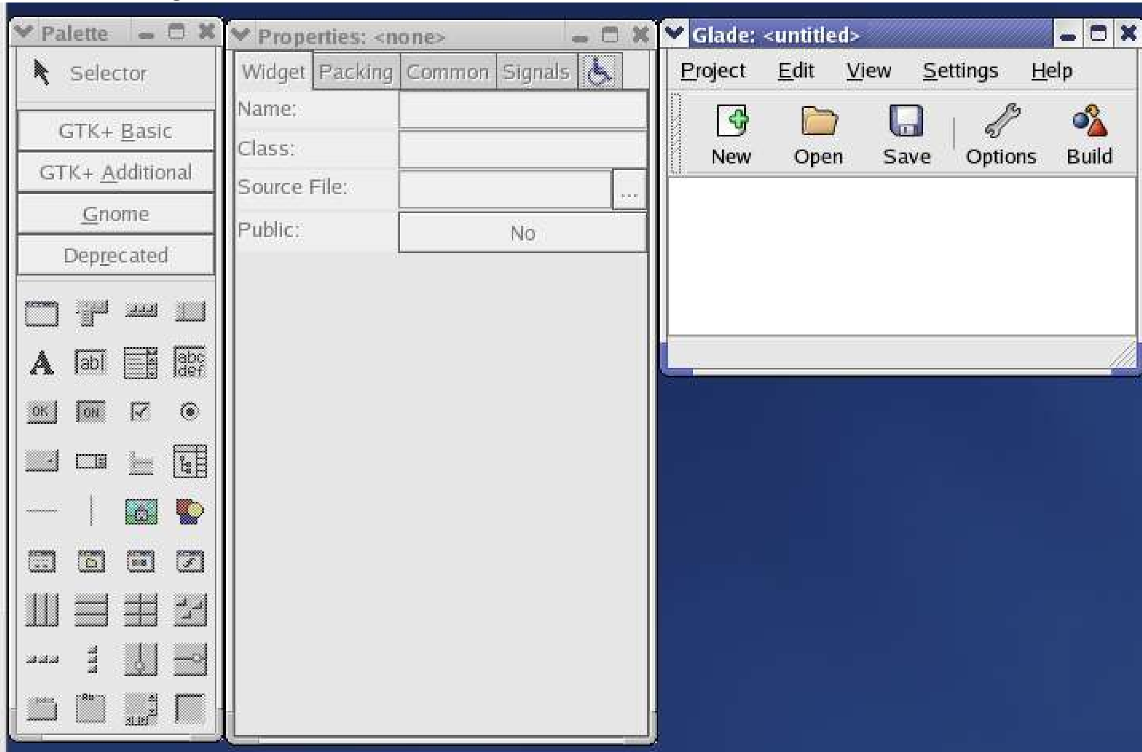


Alert Dialog

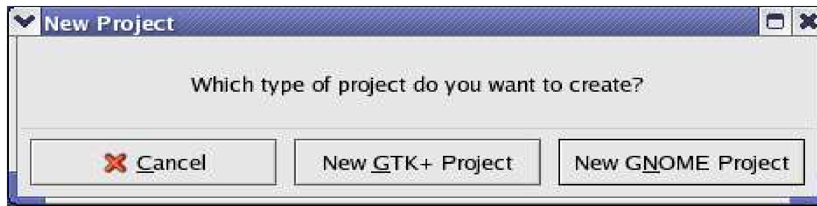


Start Glade:

You should get these three windows:



Click on the “New” icon. You should get this dialog:



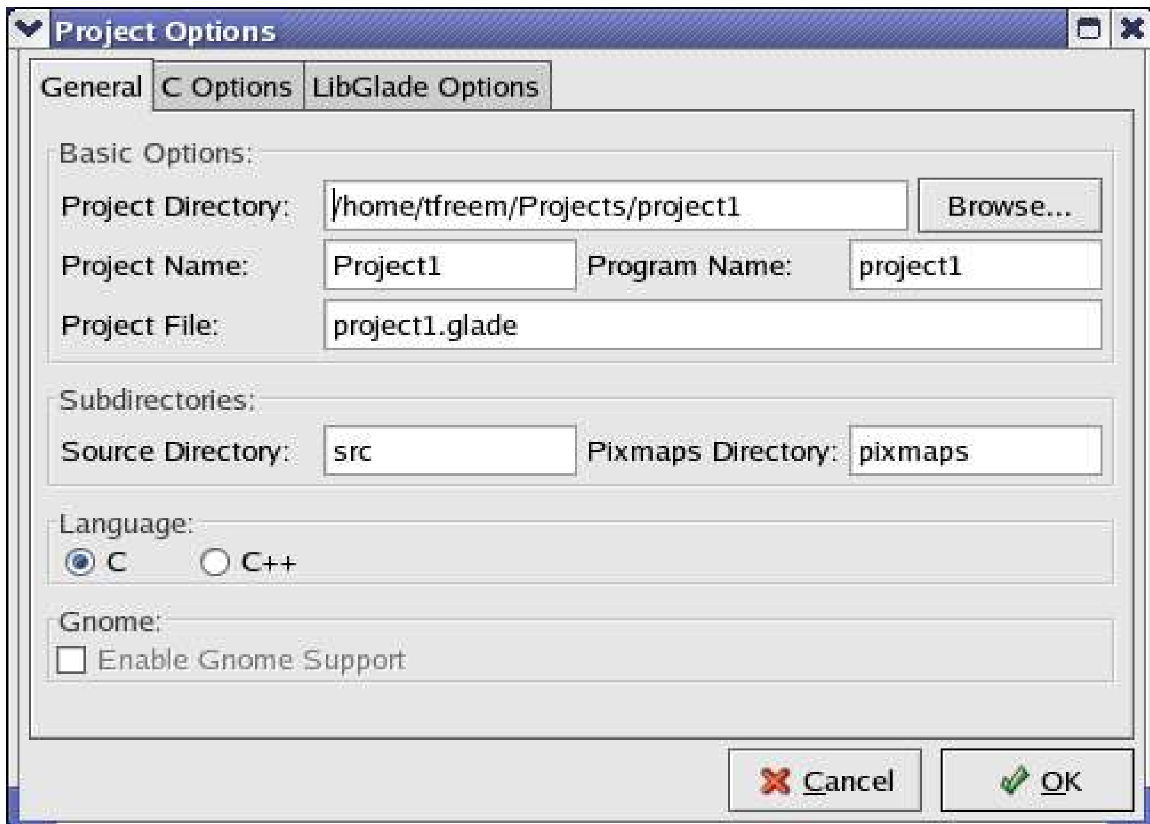
Click the “New GTK+ Project” button.

Now you should see that the “Palette” dialog box has some color to it. Hover your mouse over each of the little icons in the “Palette” dialog box.   A tool tip will tell you what the icon is for.

Save the Project

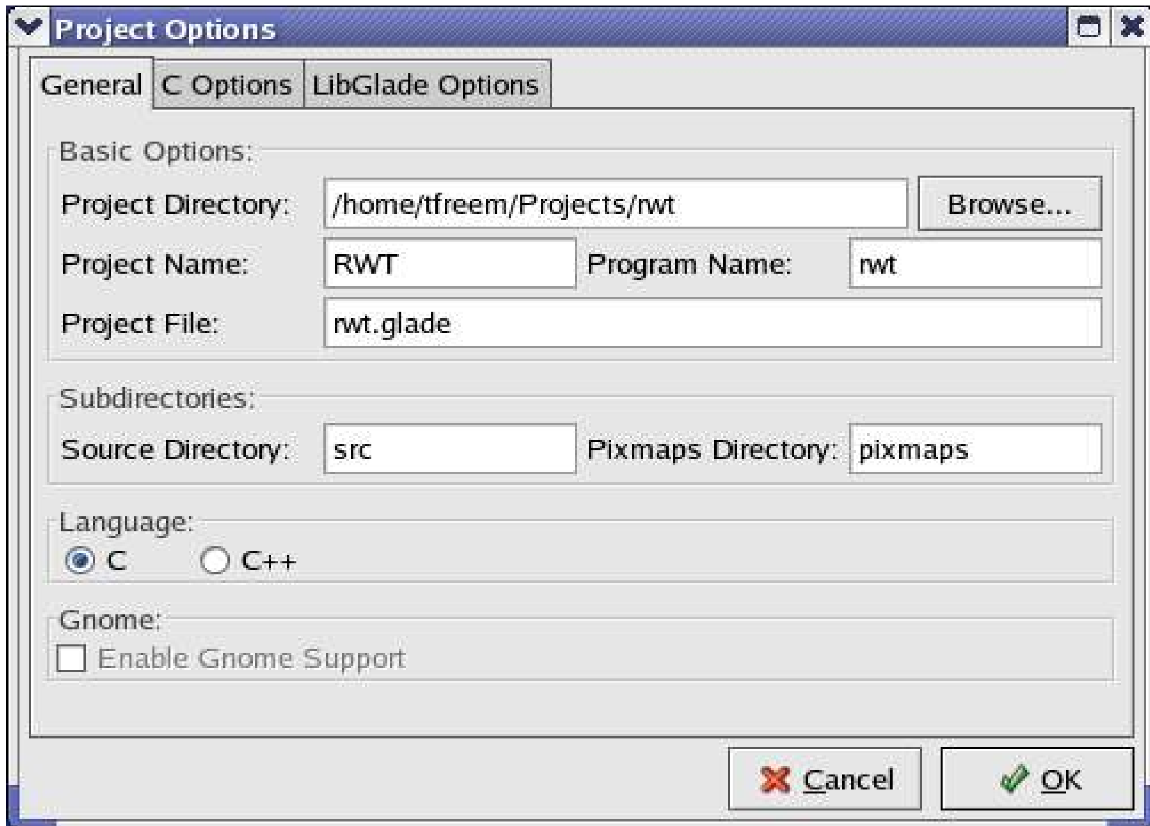
Click the “Save” button on the “Glade” dialog box. Do not click the “Build” button as this will make the c source files ... we are only interested in the .glade xml file.

You will be presented with the following dialog:



In the “Project Directory” line change “project1” to rwt. You will notice that the

“Project Name” changed to “Rwt” ... change this to all caps: “RWT”. When done the dialog should look like this:



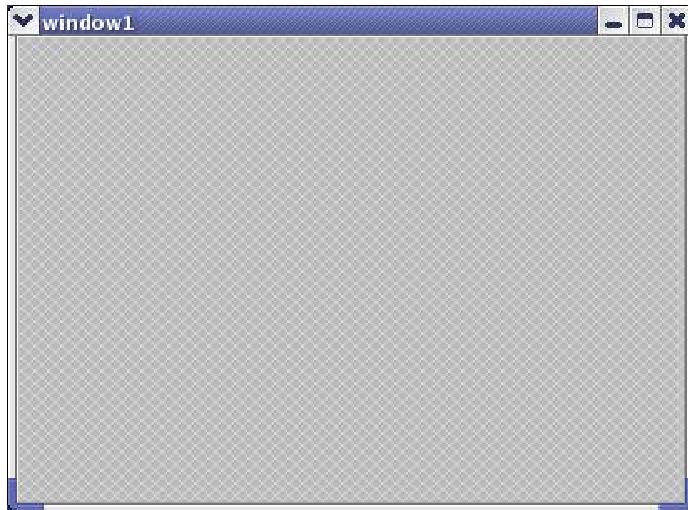
Nothing else is required, just click the “OK” button.

By saving the project at this point, it gives us a place to store our files.

Begin Building the GUI

Create an application window.

Click on the “Window” icon in the “Palette” dialog box. You should get a blank application window for us to start filling with our application controls.

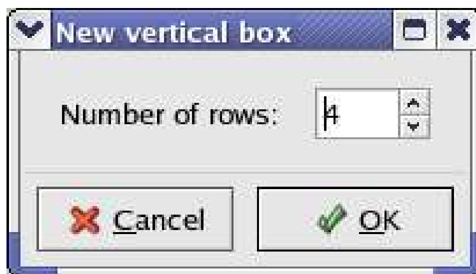


Our application will have 4 sections:

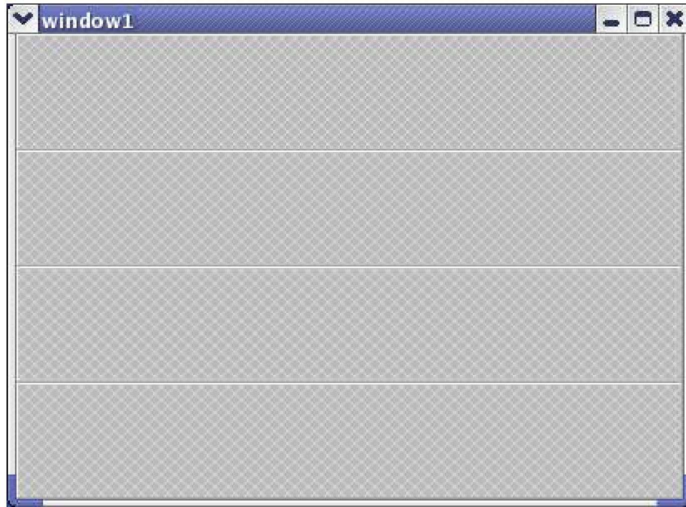
1. a menu bar at the top
2. a label to greet and give some information to the user
3. a selection tree that will display each of our transmitters
4. a button bar at the bottom (4)

Divide blank window into 4 sections.

Click on the “Vertical Box” icon in the “Palette” box and then immediately click somewhere in the blank area of our blank application window. Glade will then ask you a question:

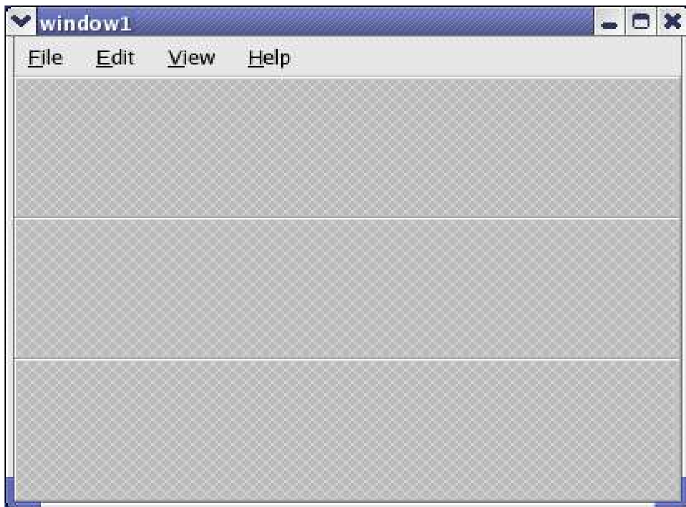


Change the number of rows from 3 to 4 and click OK. Our window should now look like this:

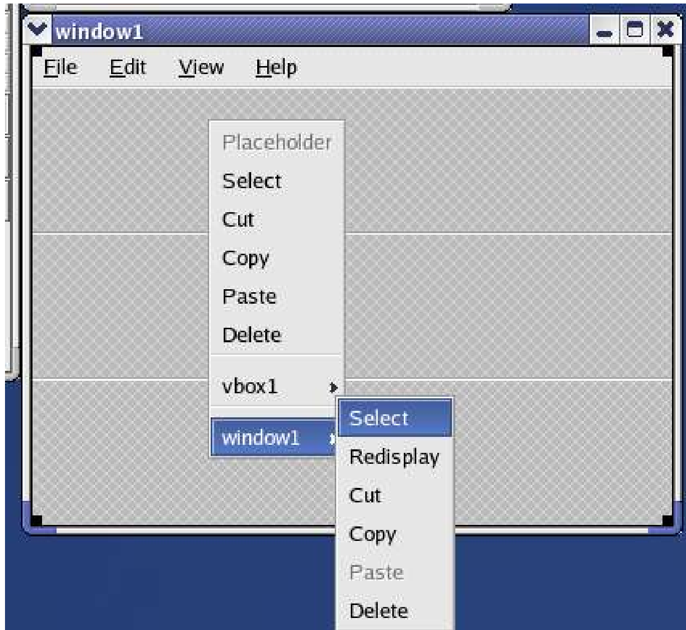


Create the menu bar section.

Click on the “Menu Bar” icon in the “Palette” box and immediately click on the top blank section of our window. It should look like this when you're done:

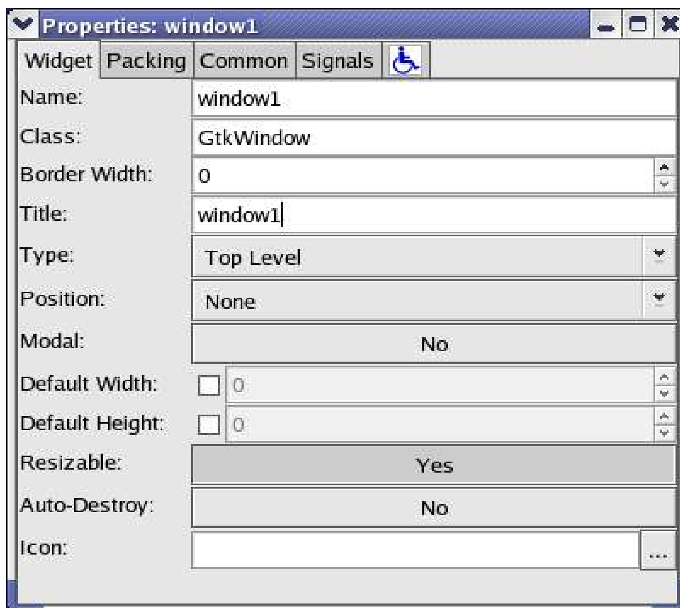


Before we go any further. Let's give our window the proper application name. To do this, we have to select window1 somehow. Three ways to do this, one is to simply click “window1” in the “Glade: <untitled>” dialog box. Another way is how we will select most all our widgets and that is to do a right click somewhere in our window:

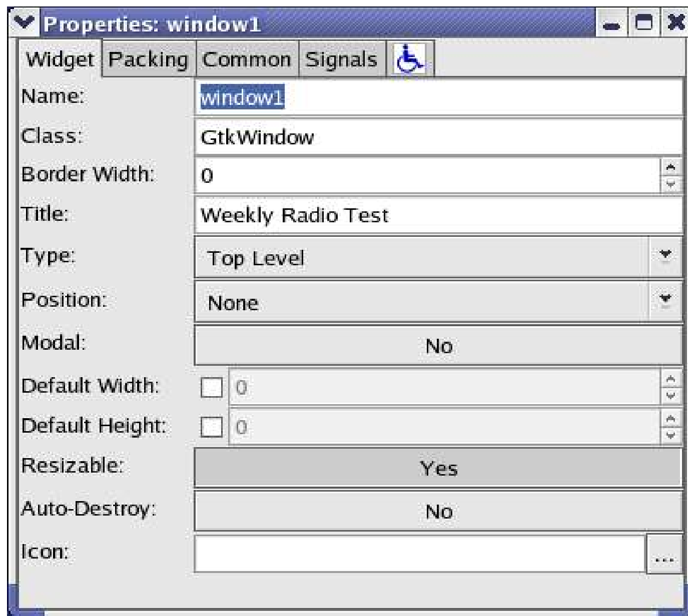


Notice we can select vbox1 or window1. Selecting vbox1 will give us the option of adding more v-sections to our window. We don't need to do this, we need window1. Click on window1 --> Select.

If you did this properly, you just selected window1. Your "Properties" box should look like this:



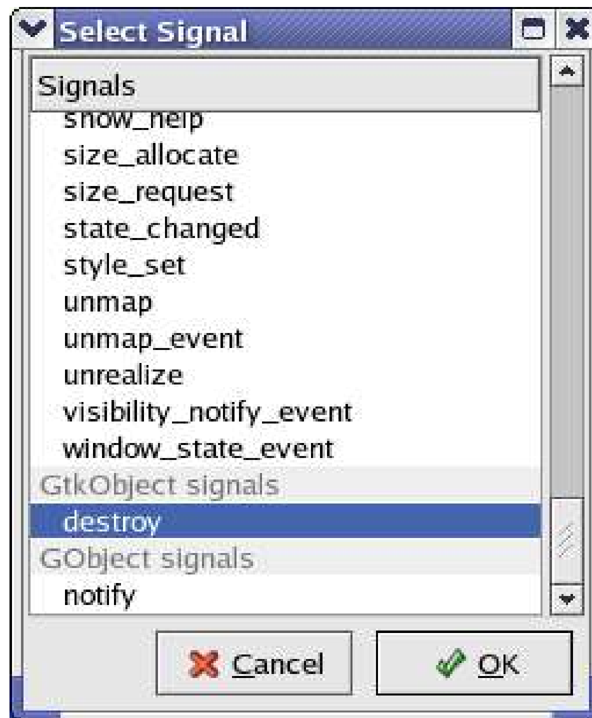
Change the "Title:" from "window1" to "Weekly Radio Test":



We now need to connect a signal to our window. This is because our application can be destroyed by the user clicking on the little “X” in the upper right hand side of our application. We need to tell our program to shut down gracefully if the user should “destroy” the application in this manner.

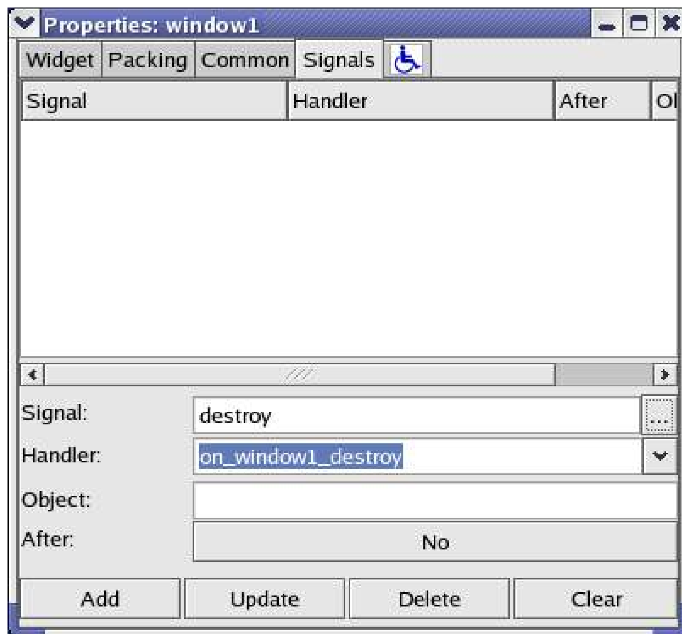
Click on the “Signals” tab.

Click on the button with the three little dots next in the “Signal” section. A “Select Signal” dialog will open. Scroll down to the bottom of the list and highlight “destroy”.

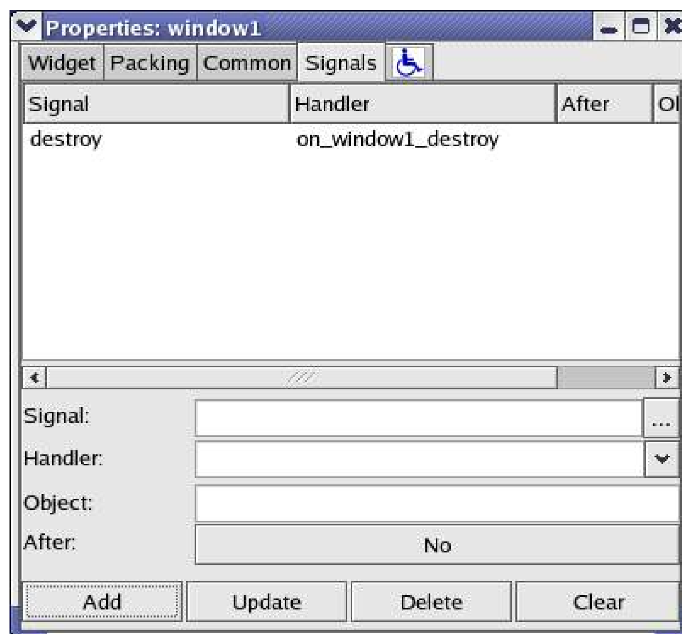


Click the “OK” button.

The “Properties” dialog box should look like this now:



DON'T FORGET: Click the “Add” button. The “Properties” dialog box should look like this:



The “on_window1_destroy” will be the name of a function that is called when someone clicks on the “X” to destroy/close the application window.

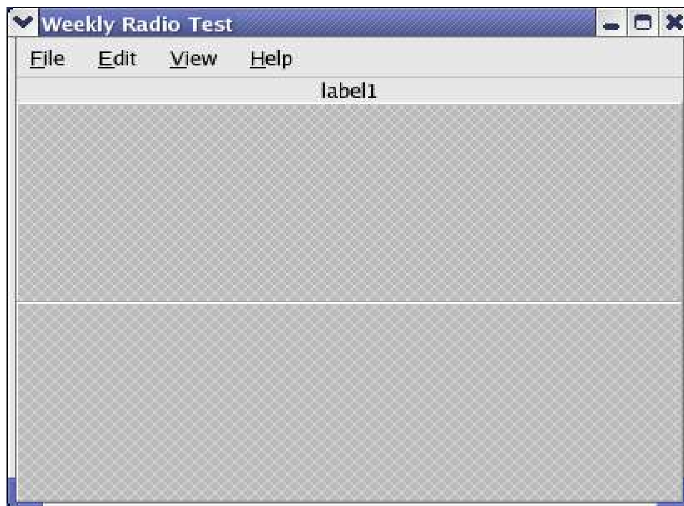
In python, we'll create a function that will handle this signal:

```
def on_window1_destroy(*args):
   gtk.main_quit()
```

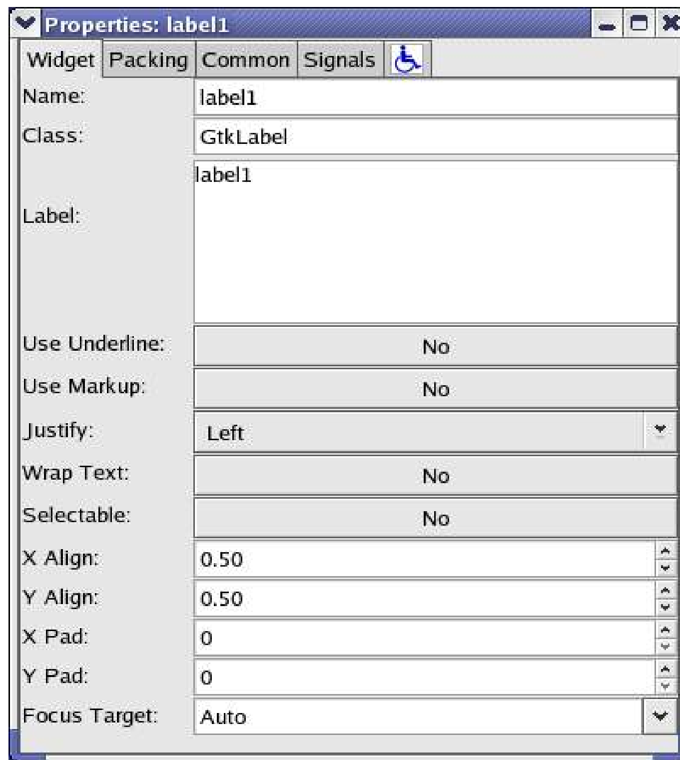
Now that's out of the way, let's press onward ....

Create the label section.

The next section of our application is a label area that greets the user and provides some general information about what is expected of the user. Since this is a label area, click on the “Label” icon in the “Palette” box and immediately click in the blank section below our menu bar. The window should look like this:



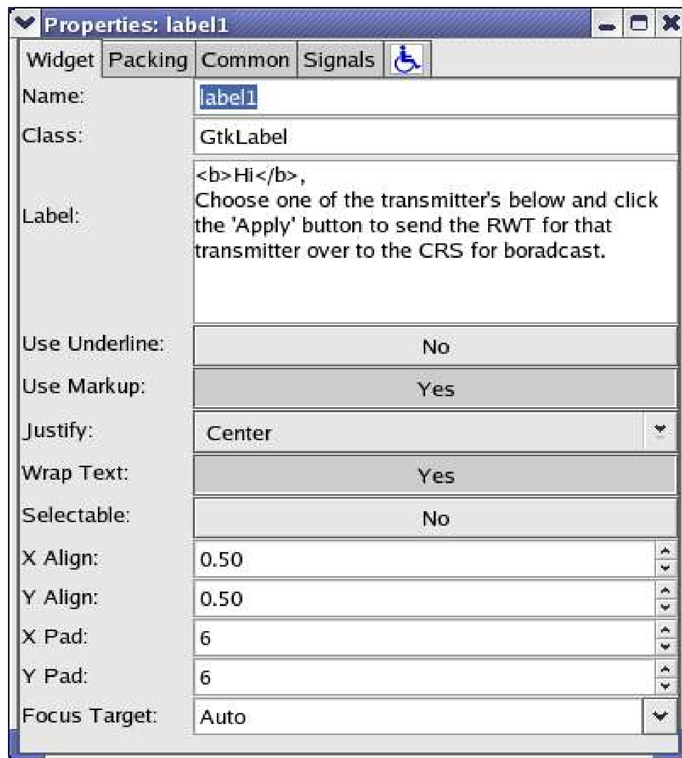
And our “Properties” dialog box should look like this:



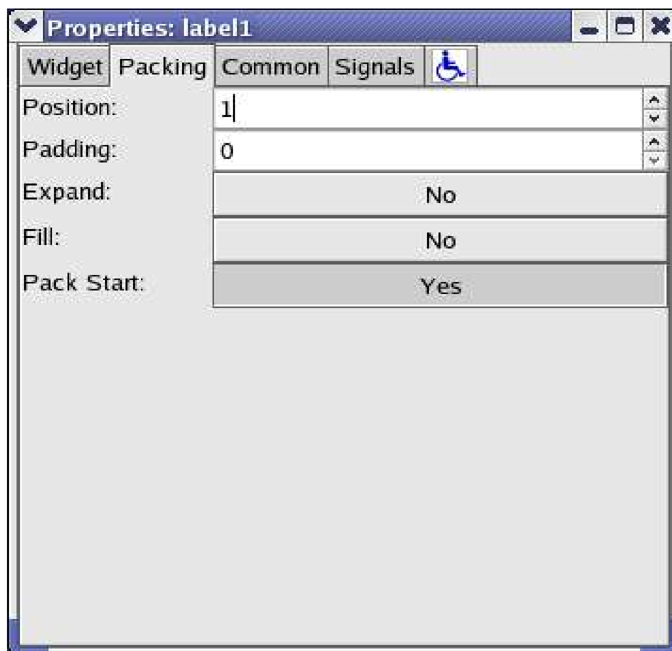
Change the X Pad to 6.
Change the Y Pad to 6.
Change the “Wrap Text” to “Center”.
Change the “Use Markup” to “Yes”.
Change the “Label” from “label1” to:

```
<b>Hi</b>,
Choose one of the transmitter's below and click the 'Apply'
button to send the RWT for that transmitter over to the CRS
for broadcast.
```

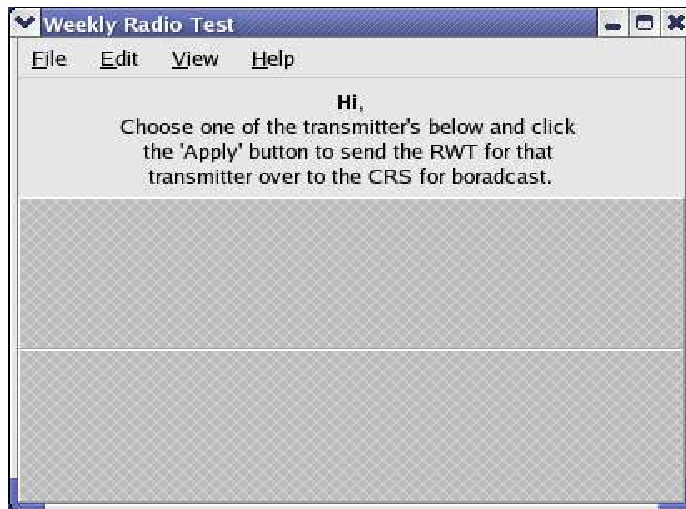
The “Properties” dialog box should now look like this after all that editing:



Click on the “Packing” tab. Make sure 'Expand' and 'Fill' are set to 'No'. We need this section of our window to maintain aspect and not fill or expand if the user should resize our application window.



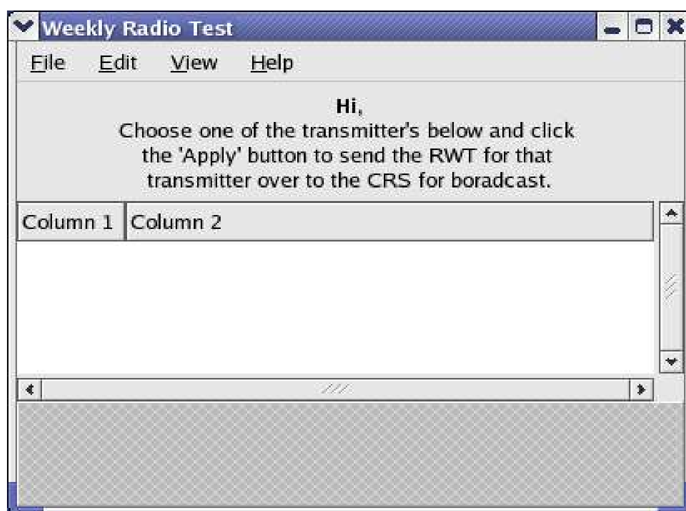
Your application window should look like this:



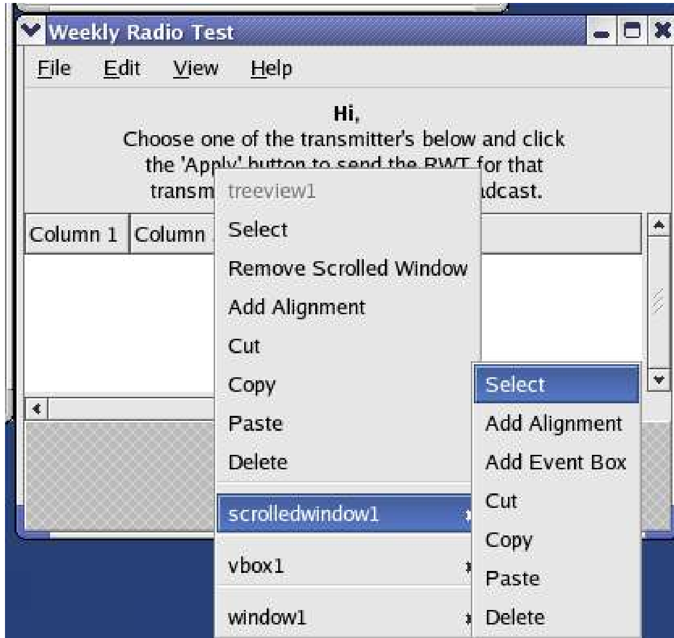
Create the transmitter selection section.

For the next section of our application window, we' llI use a “List View” that will show our transmitters.

Click on the “List or Tree View” icon in the “Palette” box and immediately click on the blank section below our label. Your window should look like this:



What actually happened was a Tree View got placed inside of a ScrolledWindow widget. We need to edit the properties of the scrolledwindow in order to remove the scrollbars. Right click on the item we just placed in our application window and click on “scrolledwindow1 --> select” .

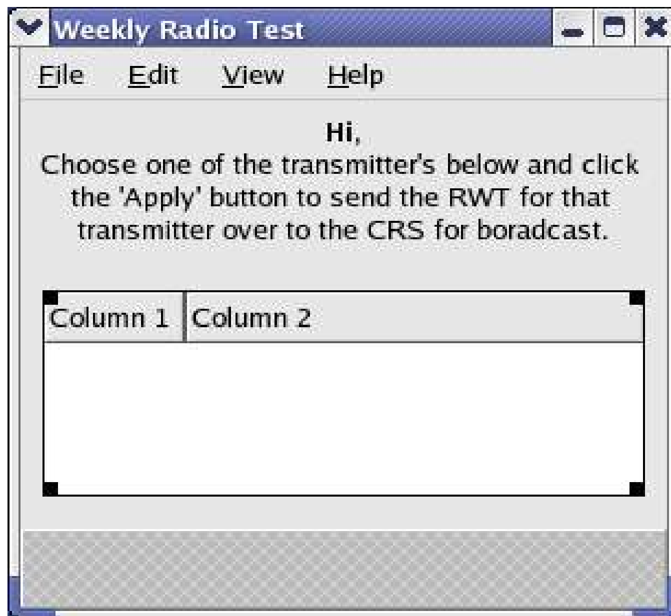


After selecting “scolledwindow1” go over to the “Properties” dialog box and change the “H Policy” and “V Policy” to “Automatic”. Change the “Border Width” to 10.

Still in the “Properties” dialog box, click on the “Packing” tab. Change the

“Padding” value from 0 to 6. Make sure “Expand” and “Fill” are both set to ' Yes' .
This will allow our “List View” to resize with the application window.

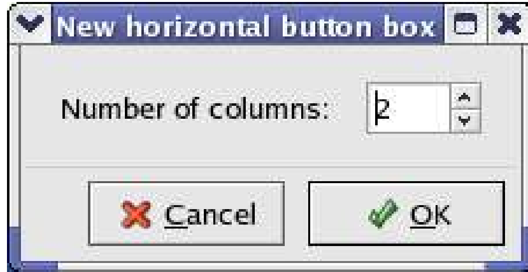
Our application window should now look like this:



We'll have to programatically place our transmitter information in the “List View” area of our application.

Create the button section.

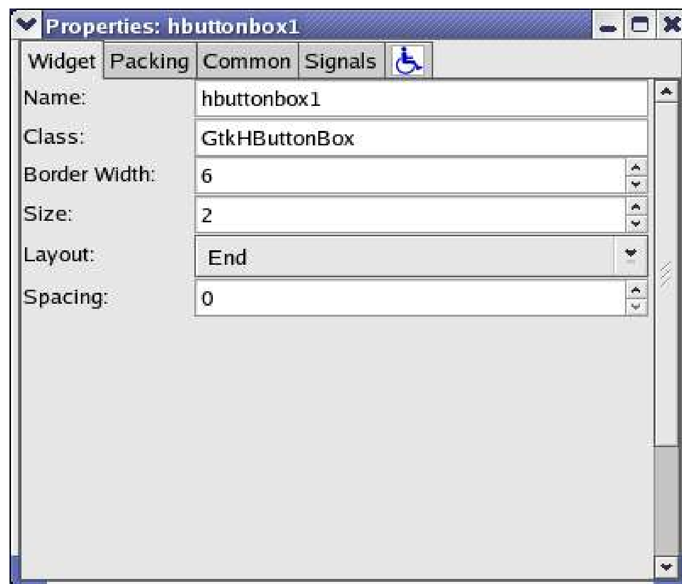
Click on the “Horizontal Button Box” in the “Palette” box and immediately click on our last blank section. You will be asked a question.



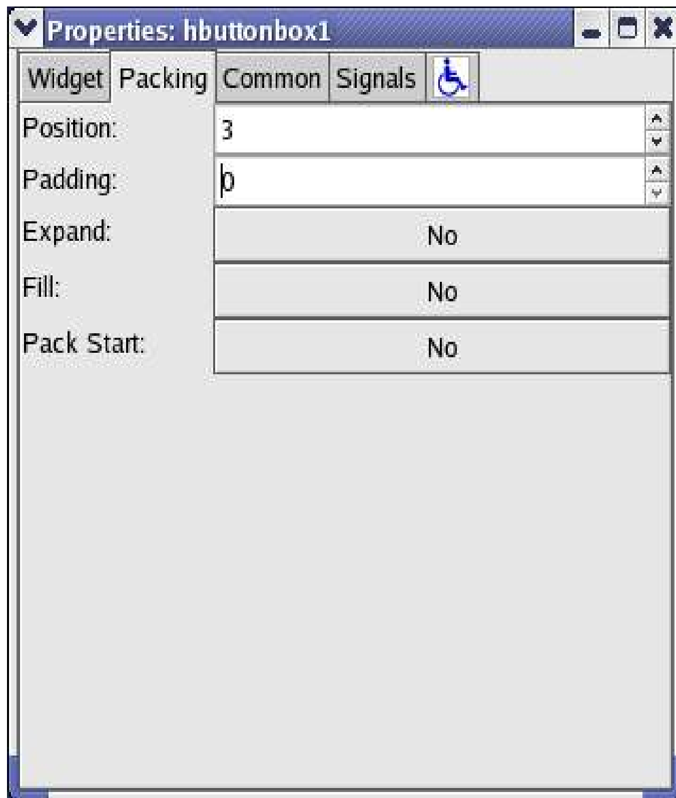
Change the “Number of columns” from 3 to 2 and click “OK”

Our buttons look pretty weird. Click in the space between the buttons in order to make sure we have “hbuttonbox1” selected (or do a right click and select the hbuttonbox1 widget).

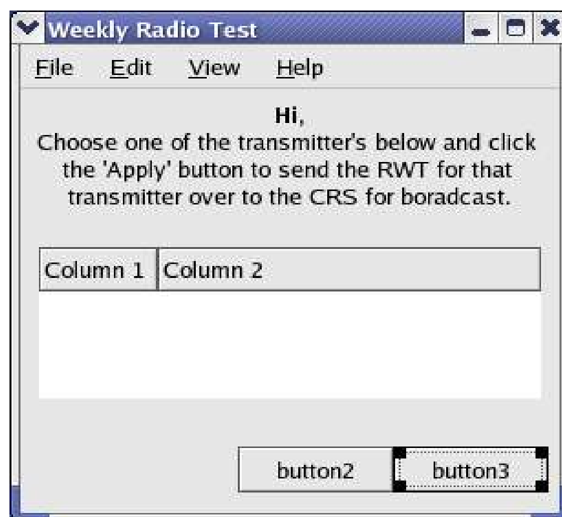
In the “Properties” dialog box, change the “Layout” from “Default” to “End”. Change the “Border Width” from 0 to 6.



Click on the “Packing” tab. Because I want the buttons to always be located at the bottom of the application window (and not get filled or expanded) make sure that “Expanded” is set to 'No' and “Fill” is set to 'No'

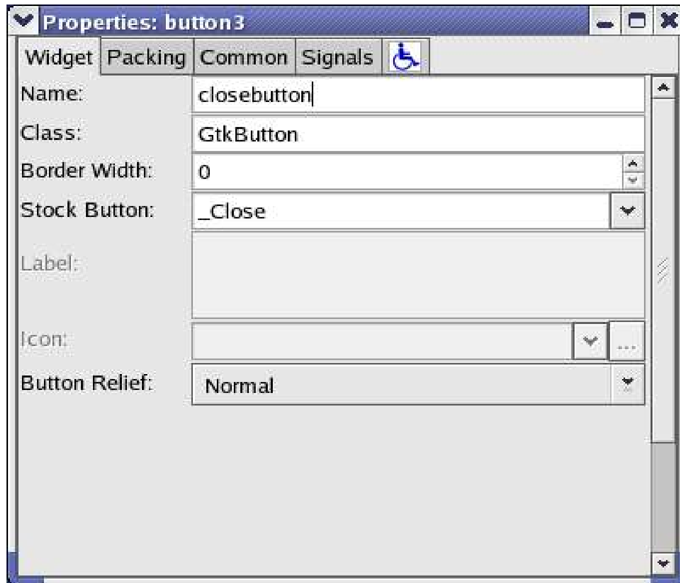


The application window is starting to shape up. Click on the first button on the right to highlight it. In my case it's button3.

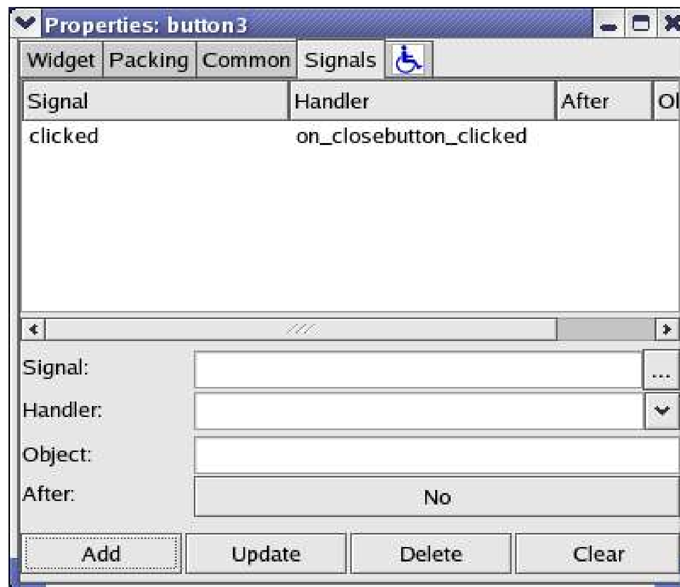


This button will be our “Close” button.

In the “Properties” dialog box change the “Stock Button” information by clicking the down arrow and selecting the “Close” stock icon. Change the “Name” of the button from “button3” to “closebutton”. The “Properties” dialog should look like this when you're done.

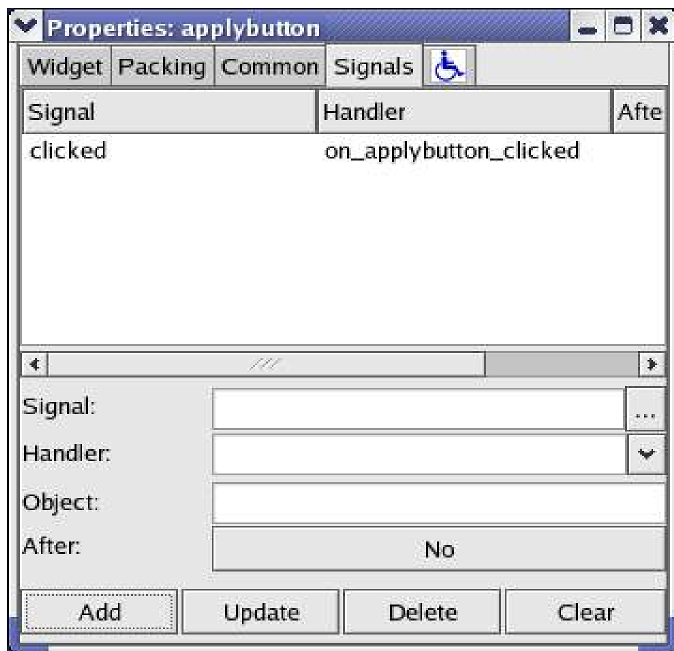


Now we need to add a “clicked” signal to the close button. Click on the “Signals” tab, click on the button in the “Signal” section that has the three little dots, highlight “clicked”, press “OK” and then click the “Add” button. Your “Properties” dialog should look like this when you're done:

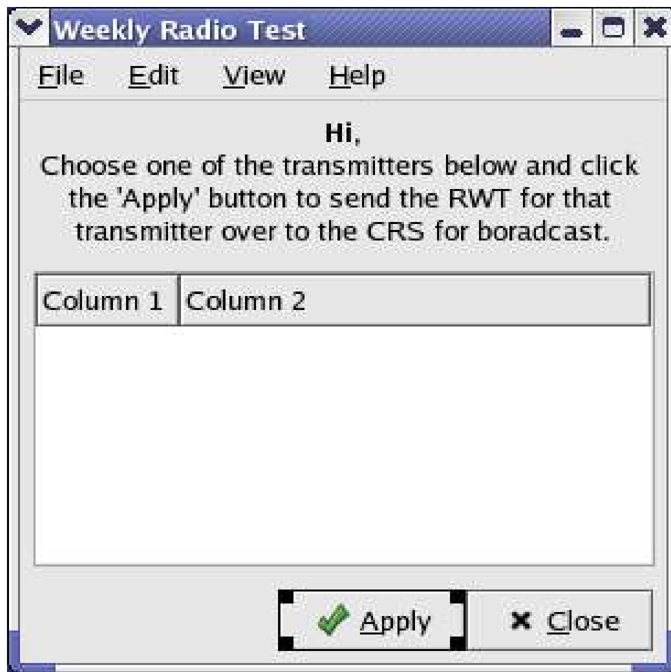


The button beside our “Close” button will be our “Apply” button. Click to highlight our future “Apply” button (In my case it's button2). Change “Name” from button2 (or whatever it is you have) to 'applybutton'. Use the drop down menu in the 'Stock Button” area and choose the “Apply” icon. The “Widget” tab in the “Properties” dialog box should look like this when you're done:

Click on the “Signals” tab and fill in the information for the “clicked” signal. The “Signals” tab should look like this when you're done:



At this point, our application window should look like this:



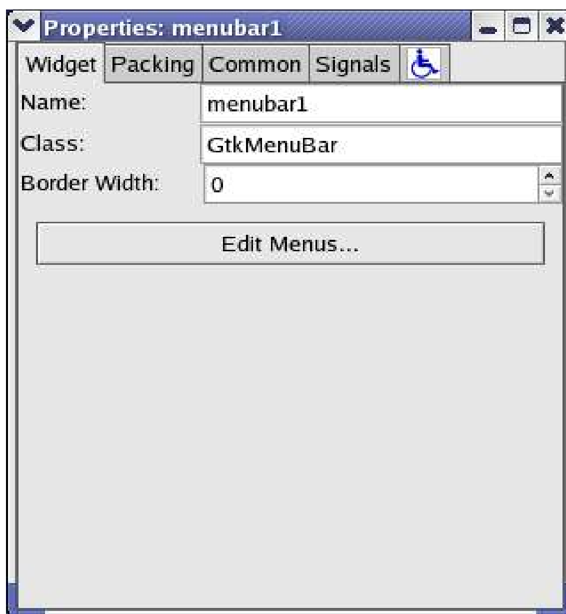
Now it's time to handle our menu items.

For this application we only need three menu items: “File”, “Edit”, and “Help”

The choice under the “File” menu item will be: “Quit”
The choice under “Edit” will be: “Preferences”
The choice under “Help” will be: “About”

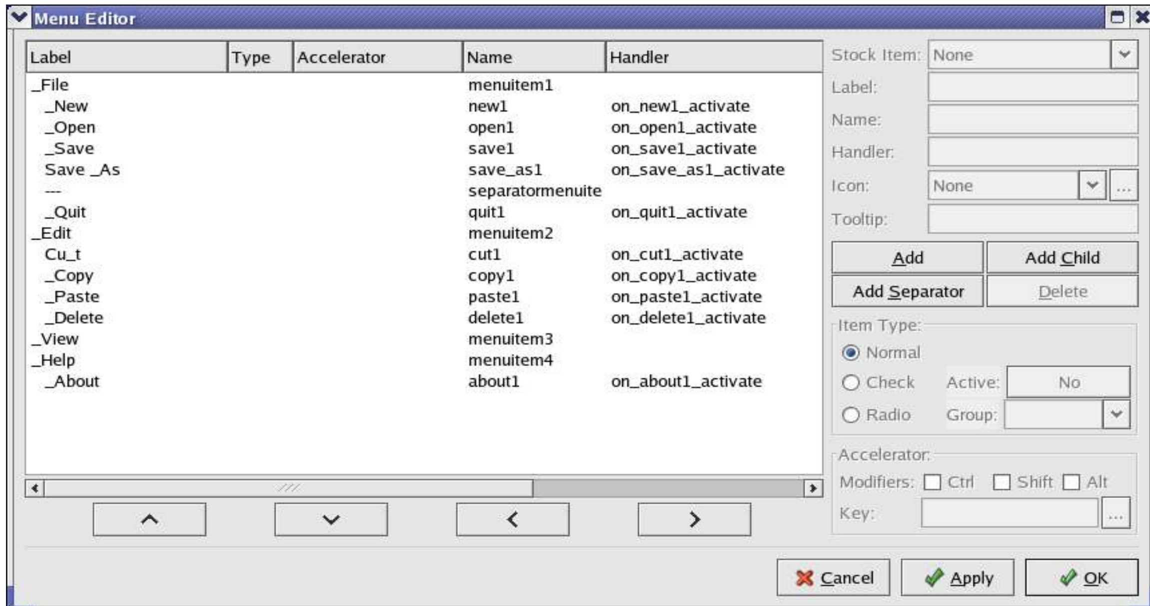
There will be no “View” menu item.

Click on the menu bar to select it. The “Properties” dialob box should look like this:



Click the “Edit Menus ... “ button.

You should get a 'Menu Editor' dialog that looks like this:

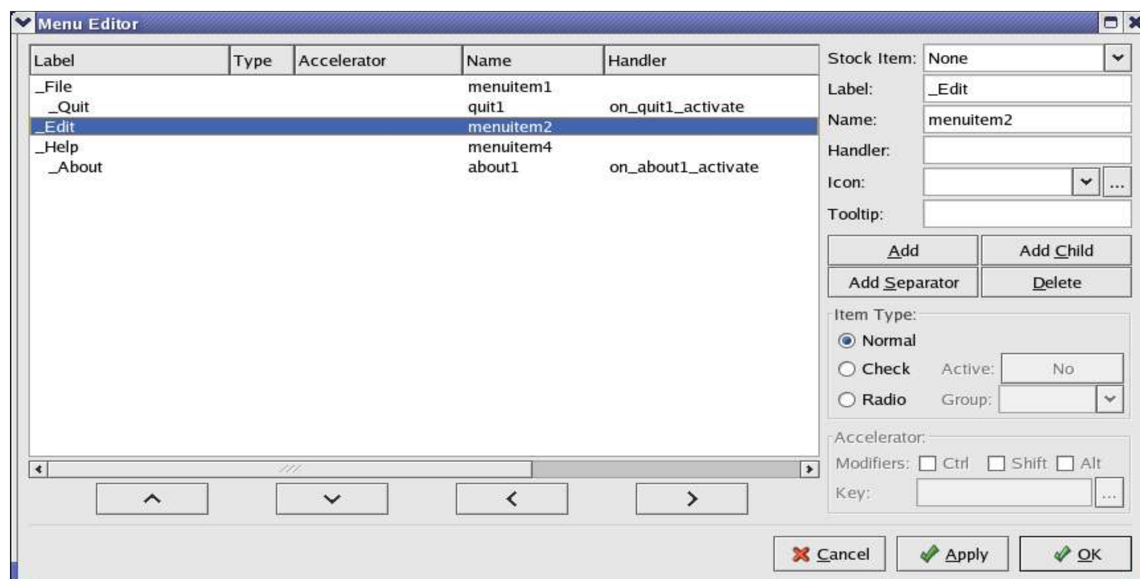


Under "_File":

- Click on "_New" and then click the "Delete" button.
- Click on "_Open" and then click the "Delete" button.
- Click on "_Save" and then click the "Delete" button.
- Click on the "---" item and then click the "Delete" button.

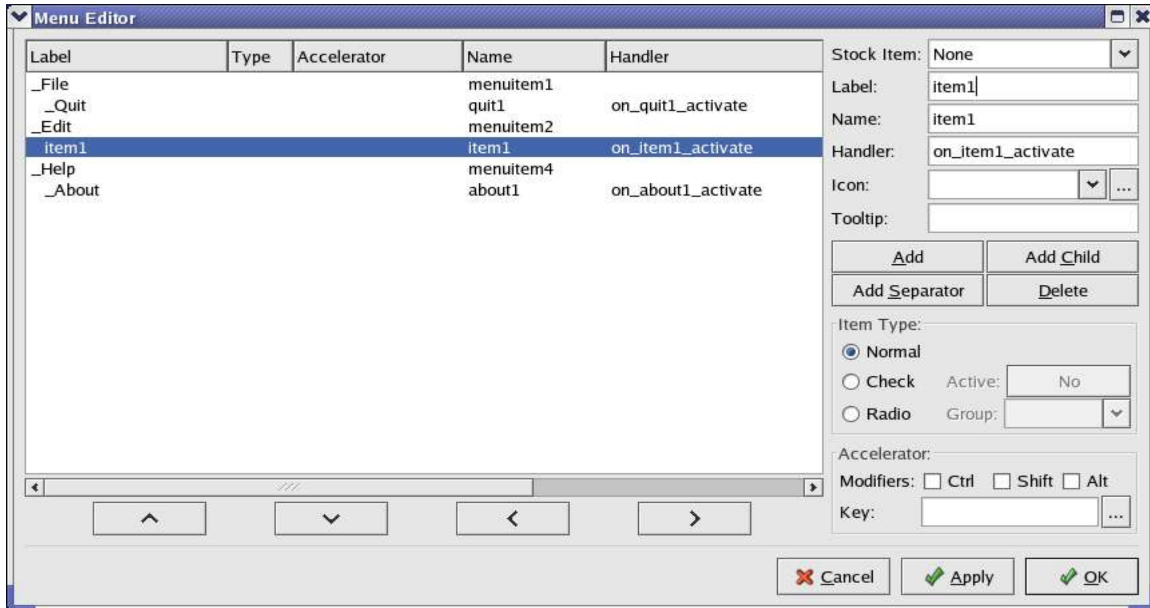
Remove all the items under "_Edit".
Remove the "_View" item.

Your dialog should look like this:



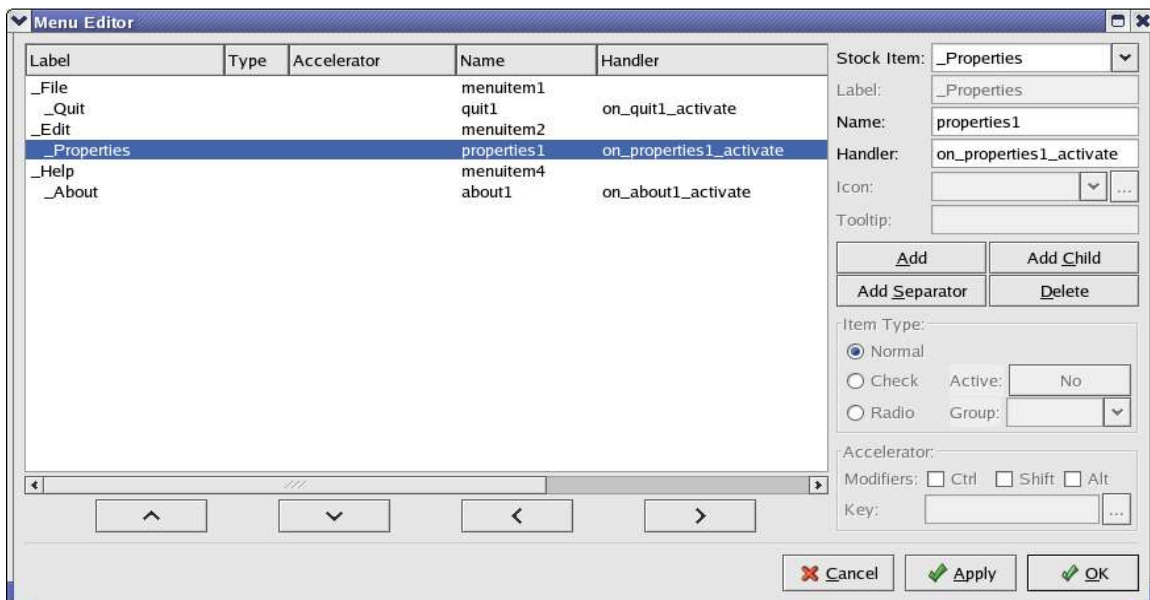
We need to add a “Preferences” menu item under the “_Edit” menu item.

Highlight “_Edit” and click on “Add Child” button. A new “item1” should be displayed.



Click the “Stock Item” selection list and choose the “Properties” entry.
“Label” should now be grayed out.
Change the value of “Name” from “item1” to “properties1”.
Change the “Handler” from “on_item1_activate” to “on_properties1_activate”.

The dialog box should look like this:



We now have three menu items to deal with in our python code:
on_quit1_activate, on_properties1_activate and on_about1_activate.

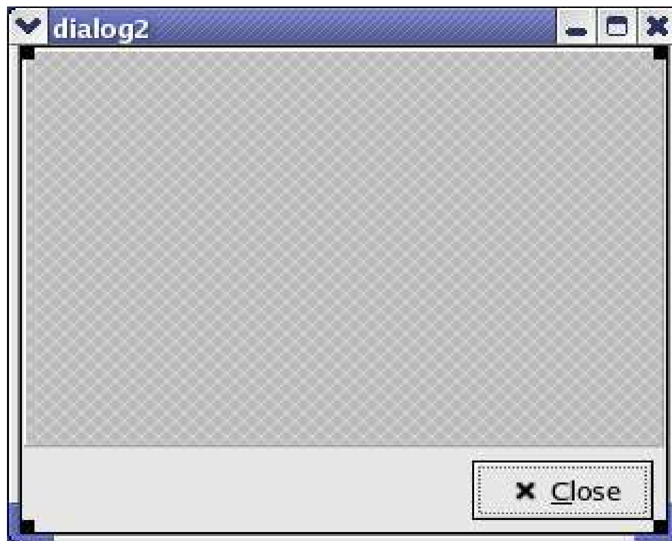
Click the “Apply” button and the “OK” button to close the dialog.

That's it for the main window. Now we need to create our supporting dialogs.
Those are: Preferences, About, Help, and Alert dialog boxes.

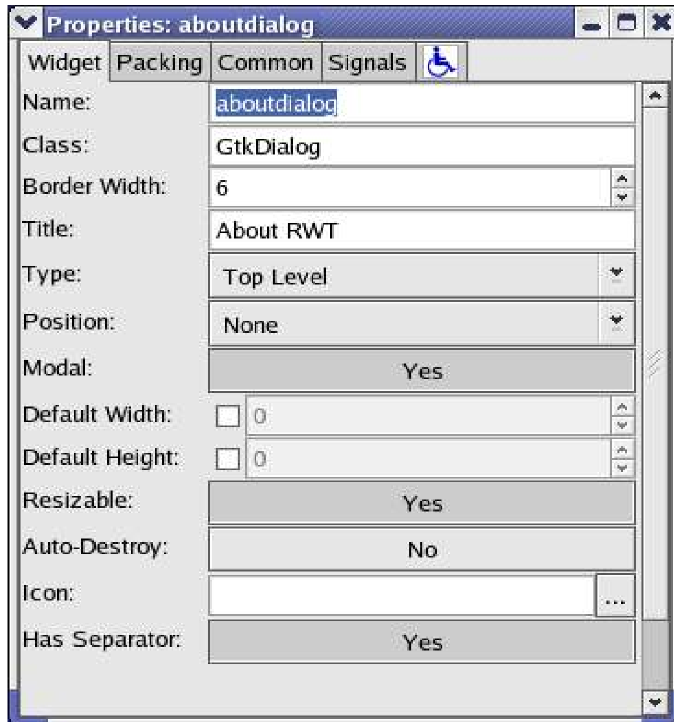
Create the "About" dialog box.

In the "Palette" dialog box click on the "Dialog" icon.

Choose "Standard Button Layout" and click on "Close" radio button. Click the "OK" button to create the dialog box.

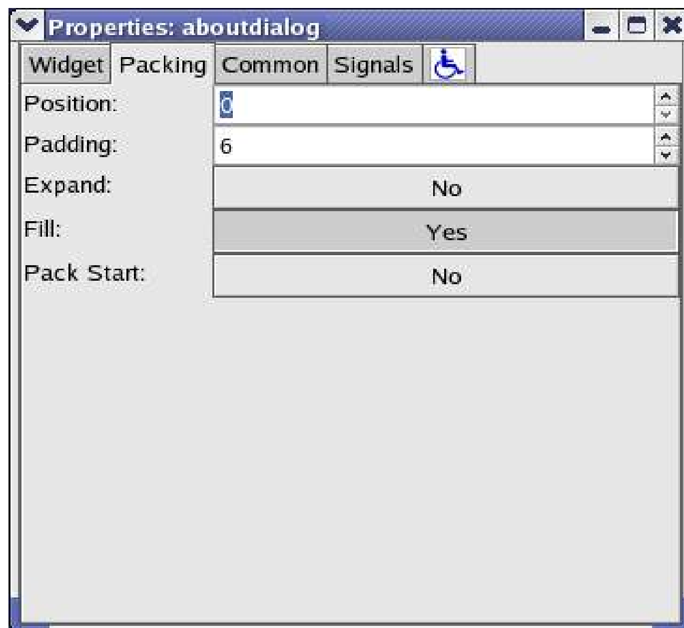


In the "Properties" dialog, change the information in the "Widget" tab to look like this:

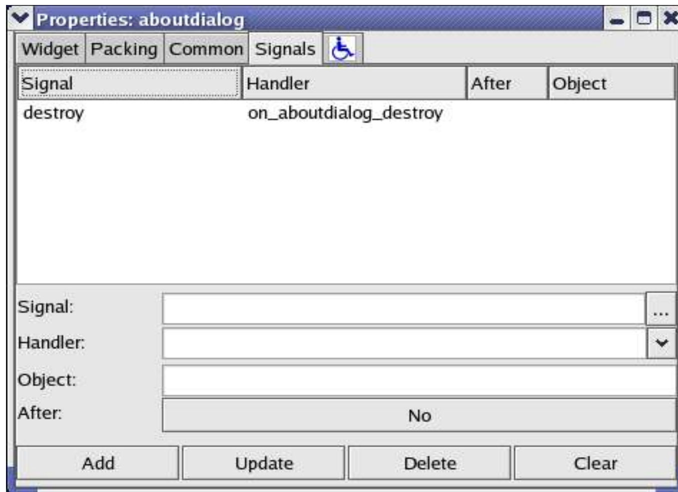


Notice that “Modal” is marked 'yes'. This makes sure that the user cannot interact with any other of our application's dialogs or windows until this one is closed.

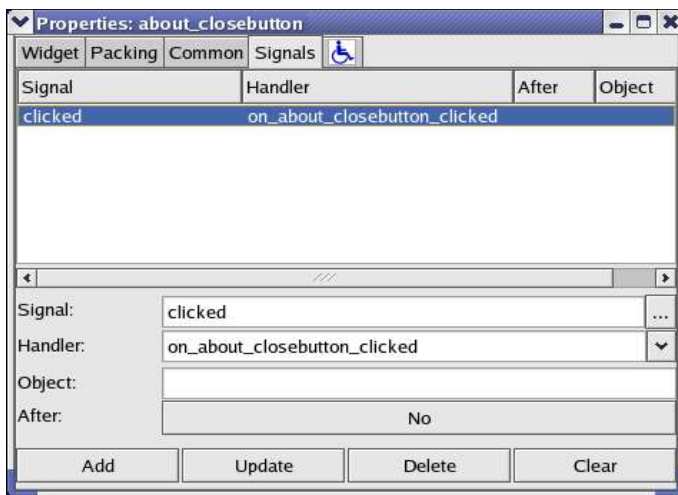
Change the “Packing” tab information to look like this:



We need to add a signal for the case where the user clicks the “X” to destroy the “About” dialog of our application, so the “Signals” tab should look like this:



The “Close” button needs a signal. Click on the “Close” button. The “Signals” tab for the close button should look like this when you are done:



Now we need to add our information to the our “About” dialog box. We'll use a label for this.

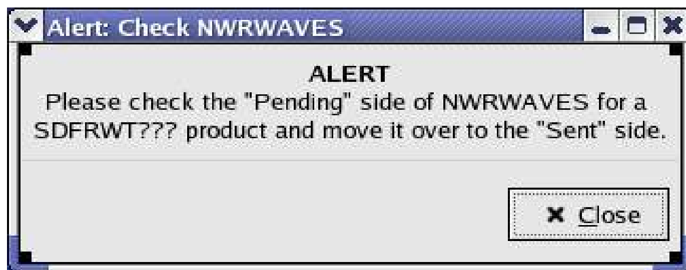
In the “Palette” dialog box click on the “Label” icon and immediately click on the blank area of our “About” dialog box. Add information to your label so that the end product looks something like this (customized for you):

Create the “Alert” dialog box

The “Alert” dialog box is created the same way as our “About” dialog box. When creating the “Alert” be sure to add signals for the ' destroy' event for the main dialog and the 'clicked' event for the close button.

- For the dialog box ' destroy' event, create “on_alertdialog_destroy” signal handler.
- For the close button 'clicked' event, create “on_alert_closebutton_clicked” signal handler.

The end result should look like this:

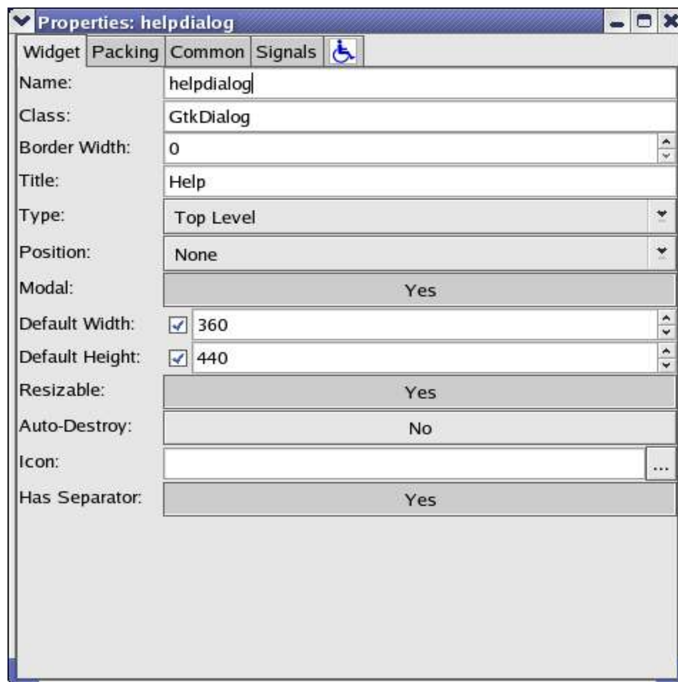


Create the “Help” dialog box

The “Help” dialog box is created the same way as the “About” and “Alert” dialog boxes, except instead of using a label to display the information, we will use a “Text View”.

On the “Palette” dialog box click on the “Dialog” icon. Make sure the “Standard Button Layout” is selected and click on “Close” radio button. Click the “OK” button to create the dialog box.

Adjust the “Properties -> Widget” tab for your help dialog to look like this:



Notice that I assigned a default width and height. This is cosmetic and is only there because I wanted to show all the help information without scrollbars. I had to experiment to get the right width and height.

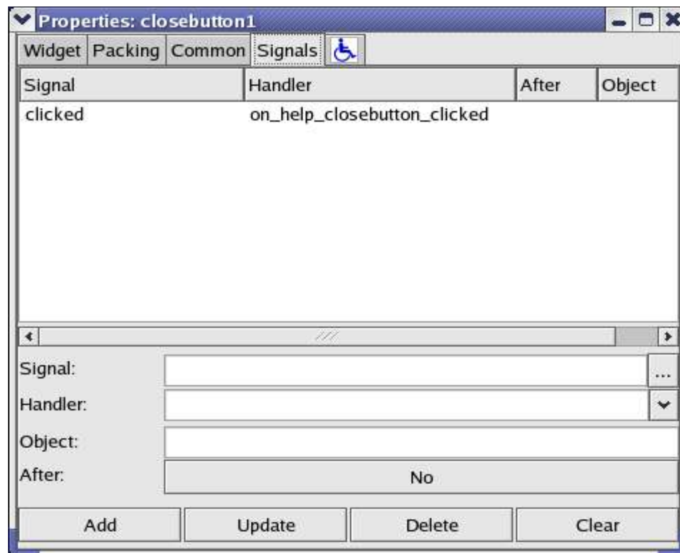
Also notice that “Modal” is marked 'yes'. This makes sure that the user cannot interact with any other of our application's dialogs or windows until this one is closed.

Create destroy signal handler.

Since this is our top level dialog, we need to add a handler for the destroy event. Click on the “Signals” tab and add 'on_helpdialog_destroy' event handler for the destroy event.

Create a signal handler for the close button.

Click on the “Close” button of our “Help” dialog box. On the “Signals” tab of the “Properties” dialog box, create a handler called “on_help_closebutton_clicked”.



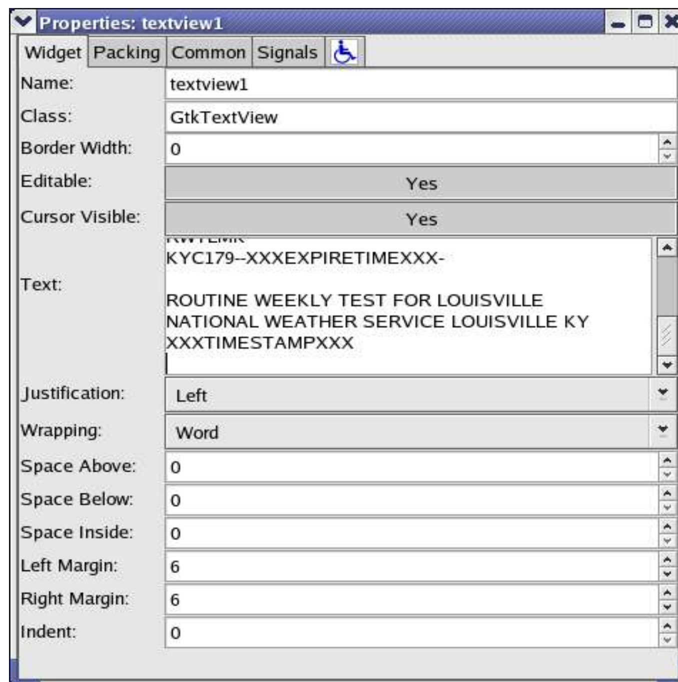
We are going to use a “Text View” for our help information rather than a simple label.

On the “Palette” dialog box, click on the “Text View” icon and immediately click on the blank area of our “Help” dialog box.

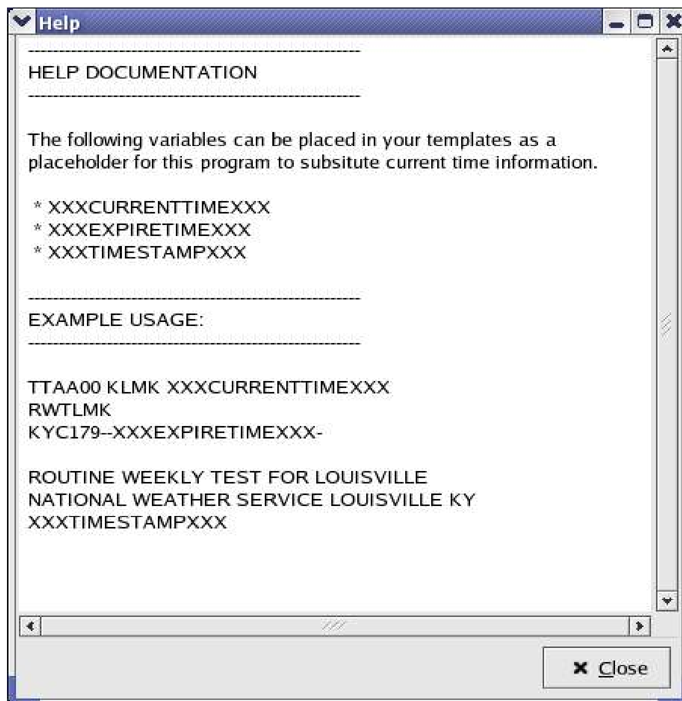
In the “Properties” dialog box, change the information in the “Widget” tab for the 'textview' that we just created. The text that I am using for the help information is:

```
------------------------------------------------------
HELP DOCUMENTATION
------------------------------------------------------

The following variables can be placed in your templates as
a placeholder for this program to subsitute current time
information.

 * XXXCURRENTTIMEXXX
 * XXXEXPIRETIMEXXX
 * XXXTIMESTAMPXXX

------------------------------------------------------
EXAMPLE USAGE:
------------------------------------------------------

TTAA00 KLMK XXXCURRENTTIMEXXX
RWTLMK
KYC179--XXXEXPIRETIMEXXX-

ROUTINE WEEKLY TEST FOR LOUISVILLE
NATIONAL WEATHER SERVICE LOUISVILLE KY
XXXTIMESTAMPXXX
```

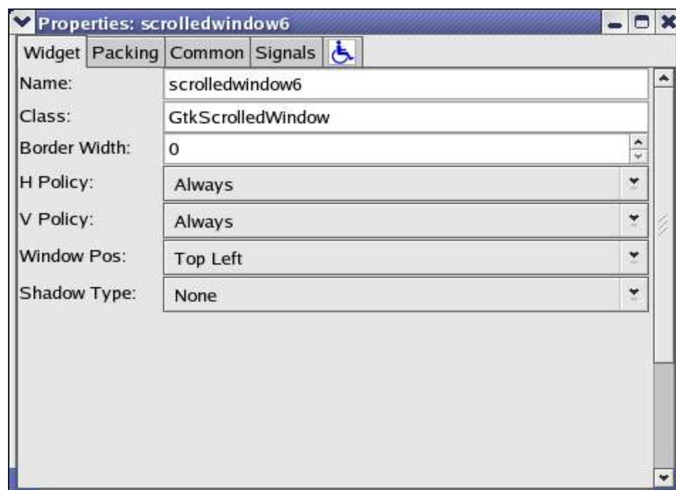
Your “Properties / Widget” information for the textview should look like this:



The main “Help” dialog should look like this:



I don' tike the scrollbars. To remove them, select the scrolledwindow by clicking once on one of the scrollbars. Your “Properties -> widget” tab should look like this:



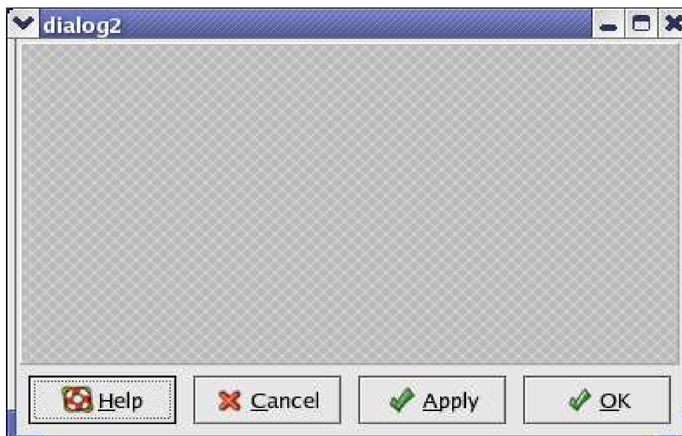
Change the “H Policy” and “V Policy” to “Automatic.

Create the “Preferences” dialog box.

On the “Palette” dialog box, click on the “Dialog” icon.

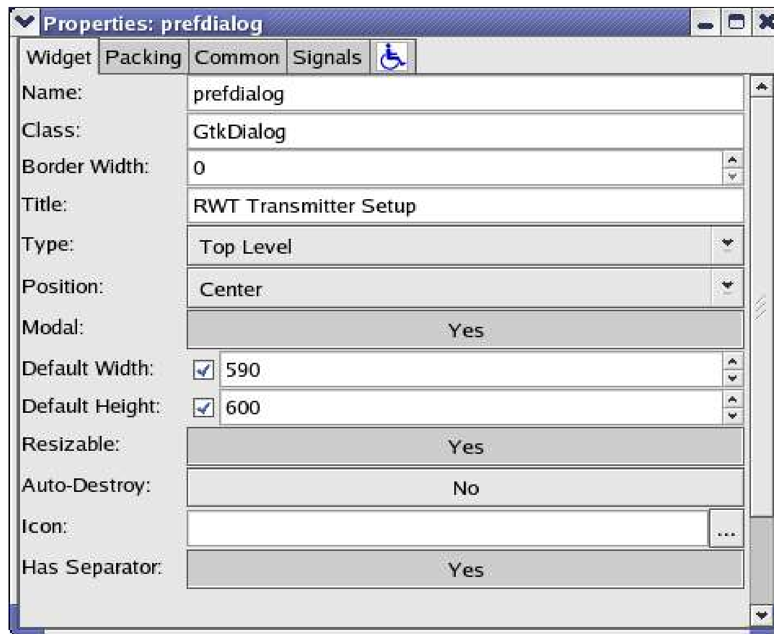
Make sure “Standard Button Layout” is selected and select the “Cancel, Apply, OK” radio button. Check the box next to “Show Help Button”. Click the “OK” button when done.

You should get a blank dialog that looks similar to this:



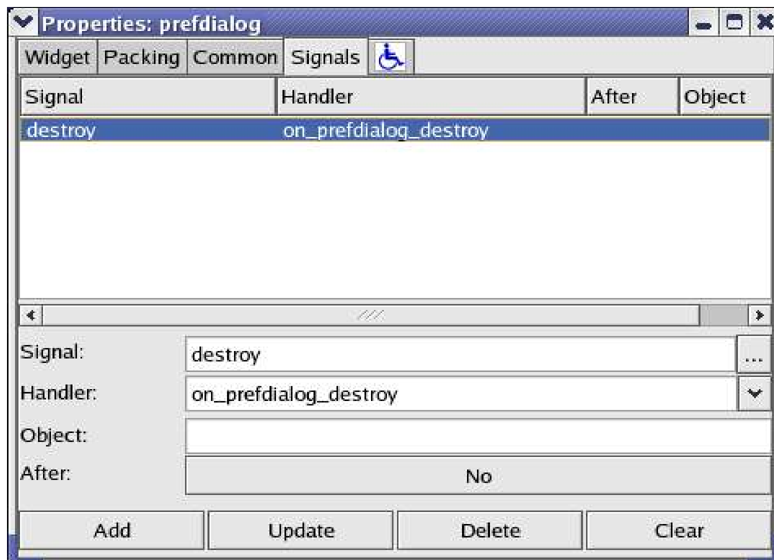
Click on the “Widget” tab of the “Properties” dialog box.

Change the “Properties --> Widget” tab information to look like this:



Click on the “Signals” tab of the “Properties” dialog box.

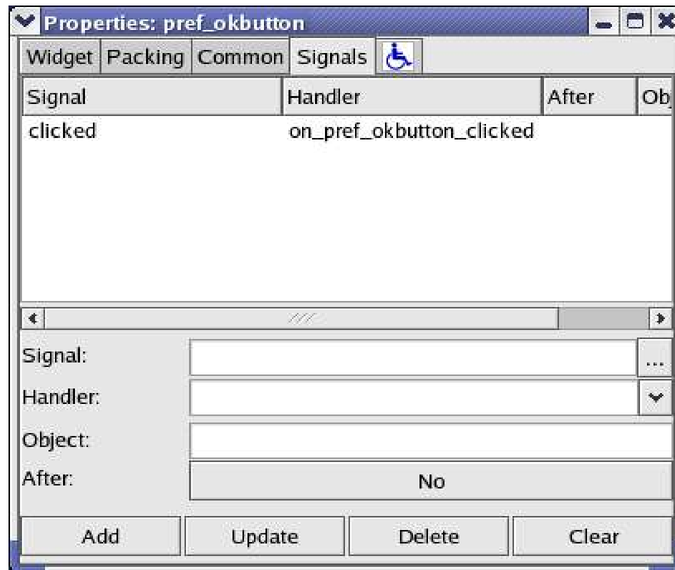
Add a 'on_prefdialog_destroy' handler for the 'destroy' event.



Create signal handlers for each of the buttons.

Click on the “OK” button.
Click on the “Widget” tab of the “Properties” dialog box.
Change the name of the button to “pref_okbutton”.
Click on the “Signals” tab of the “Properties” dialog box.
Add a 'clicked' signal handler: on_pref_okbutton_clicked

The “Properties --> Signal” dialog for the OK button should look like this:



Click on the “Apply” button.
Click on the “Widget” tab of the “Properties” dialog box.
Change the name of the button to “pref_applybutton”.
Click on the “Signals” tab of the “Properties” dialog box.
Add a 'clicked' signal handler: on_pref_applybutton_clicked

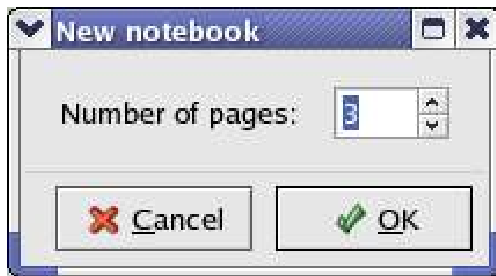
Click on the “Cancel” button.
Click on the “Widget” tab of the “Properties” dialog box.
Change the name of the button to “pref_cancelbutton”.
Click on the “Signals” tab of the “Properties” dialog box.
Add a 'clicked' signal handler: on_pref_cancelbutton_clicked

Click on the “Help” button.
Click on the “Widget” tab of the “Properties” dialog box.
Change the name of the button to “pref_helpbutton”.
Click on the “Signals” tab of the “Properties” dialog box.
Add a 'clicked' signal handler: on_pref_helpbutton_clicked

Add Notebook tabs for each transmitter.

We are going to keep things simple and hard code this section for each of our transmitters. What we'd really want to do is figure out how to pull in our transmitter information from NWRWAVES so that this section can be dynamically built. In the mean time, we will use the “Notebook” widget to hard code our transmitters to the application. Each tab of the notebook tabs will contain the information (as a template) that will eventually be read over CRS for the test.

On the “Palette” dialog box, click on the “Notebook” icon and immediately click on the blank area of our “Preferences” dialog box. You should be asked this question:



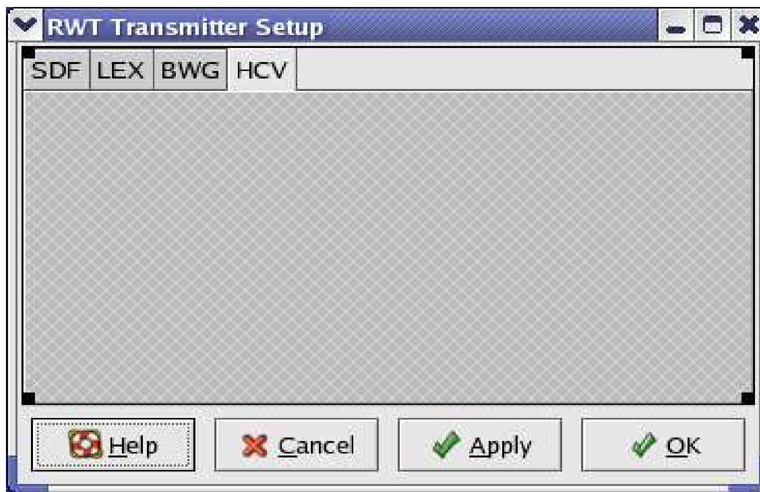
I have 4 transmitters (SDF, LEX, BWG, HCV), so I'll change the number of pages from 3 to 4 and click the “OK” button.

Your “Preferences” dialog should now look something similar to this:

Change all the label names to represent each of your transmitters.

In this case I'll change the name of “label6” to “SDF”, “label7” to “LEX”, “label8” to “BWG” and finally “label9” to “HCV”. You are just changing the label name ...

no other changes are required at this point. Your Preferences dialog should look similar to the following when done:

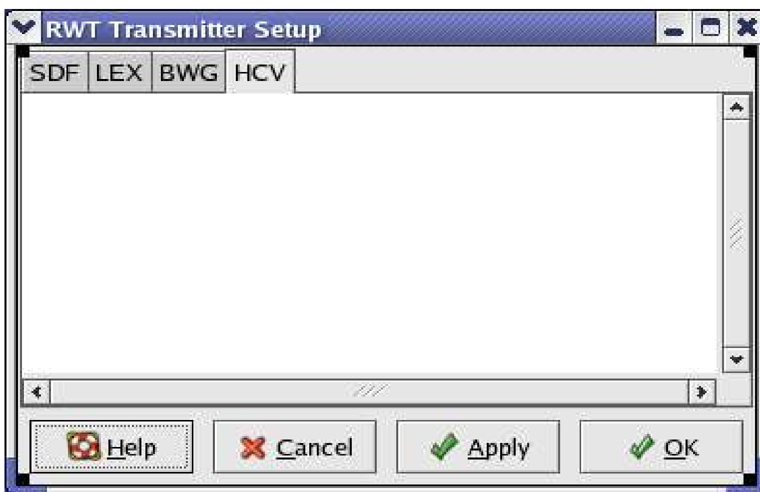


When you click on the label in our dialog box, it changes the notebook. This means that you have 4 blank areas to fill in ... one for each transmitter.

Add "Text View" widget to each transmitter page.

Click on one of the notebook tabs. In this case I clicked on "HCV".

In the "Palette" dialog box, click on the "Text View" icon and immediately click on the blank area in our Preferences dialog box. Your Preferences dialog box should look something like this:



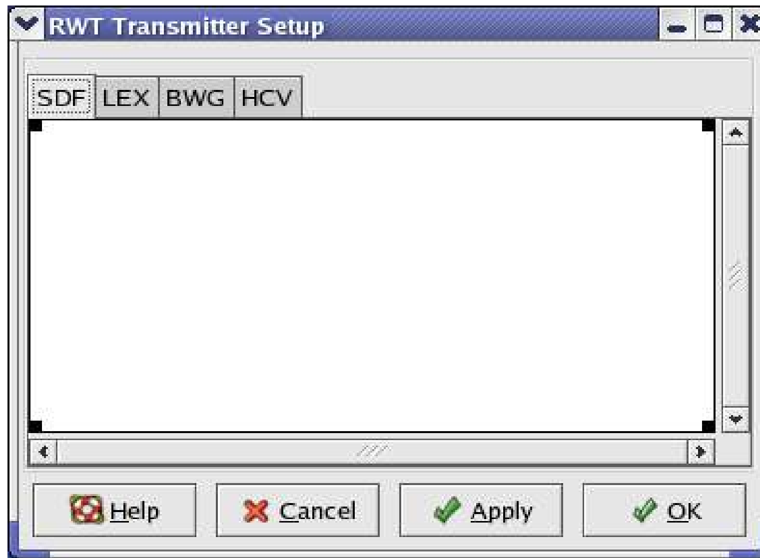
Click on “BWG” label. The Preferences dialog should look like this:

In the “Palette” dialog box, click on the “Text View” icon and immediately click on the blank area. Your window should look like the following:

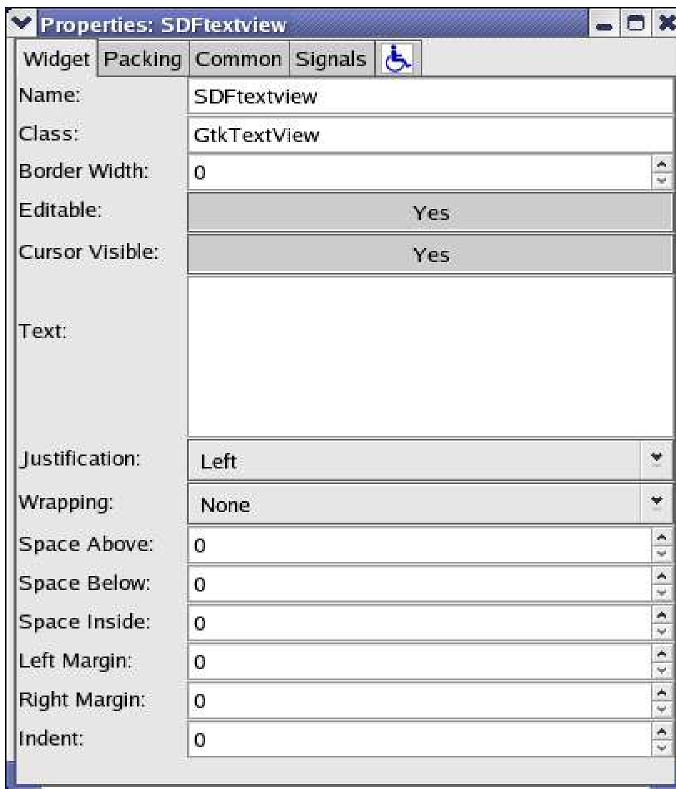
Do the same for the rest of the transmitters:

Change name of each “Text View”

Click on a notebook label and then click on the “Text View” area:



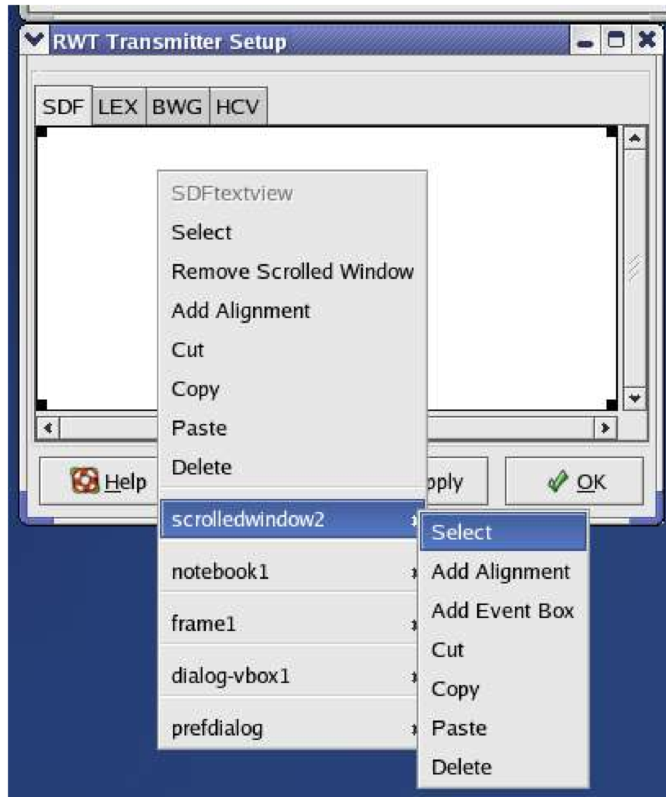
Change the Name of the “View Text” area to identify it as specific to the selected transmitter. SDF is selected, so I changed the Name from “textview1” to “SDFtextview”.



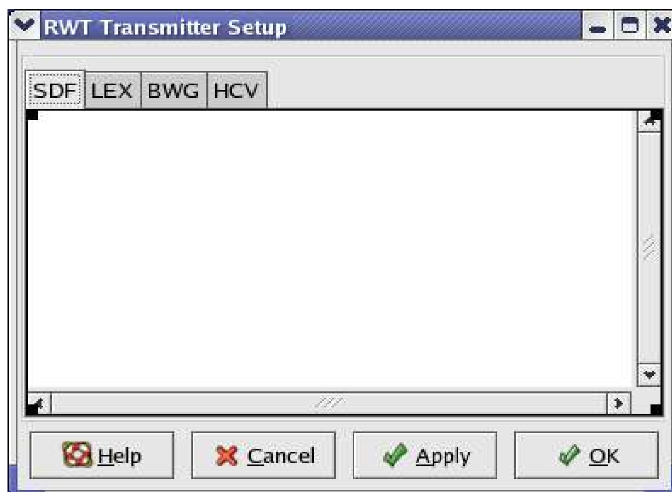
Chang the “Text View” names of the other transmitters.

Remove scrollbars

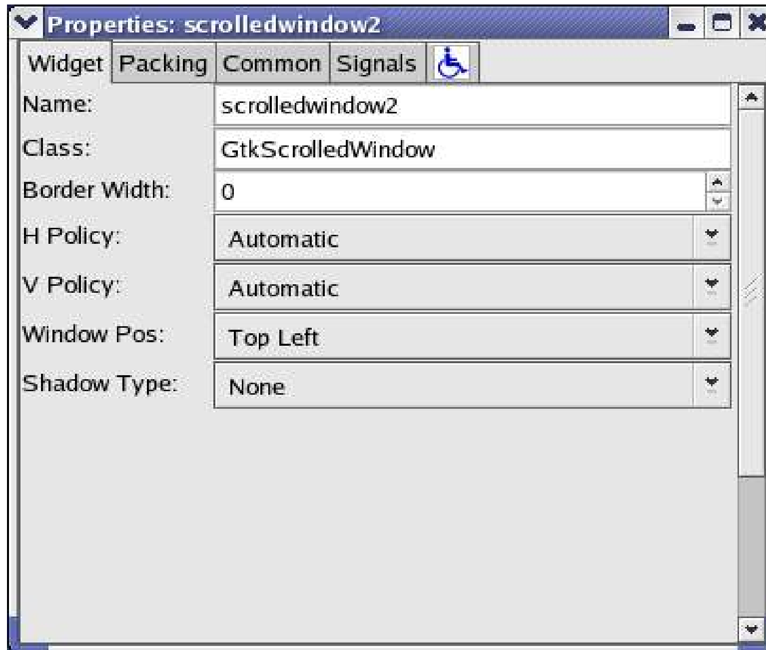
I don' tike the scrollbars, so do a right click on the “Text View” for each of your transmitters and select the “scolledwindow” entry:



Your selection should look like the following:

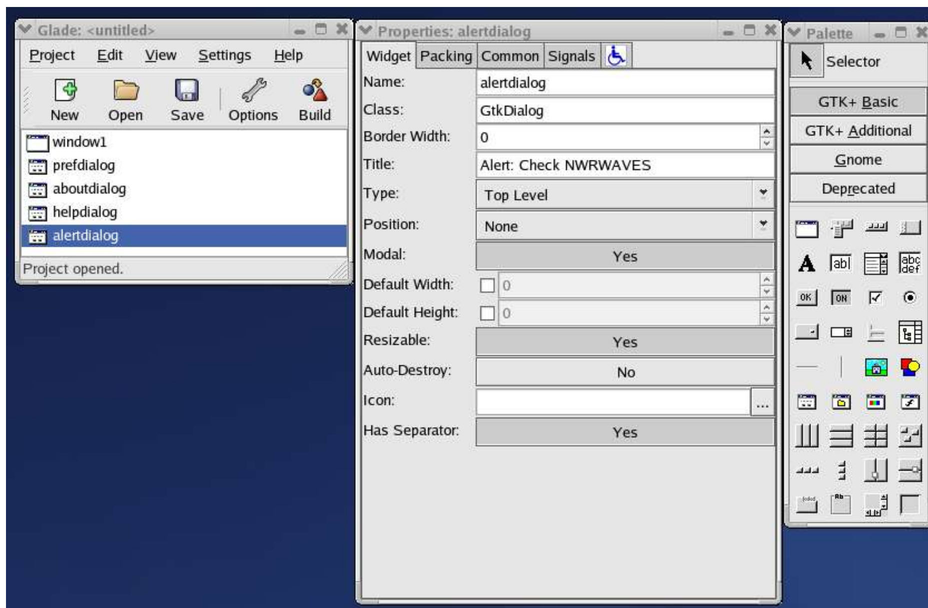


In the properties dialog box for the “scrolledwindow”, change the “H Policy” and “V Policy” to Automatic.



Do this for the other transmitters.

Done Building the GUI!
Save your work and exit glade!



Start Coding

Open a terminal window and cd over to your project directory.

```
cd /home/tfreem/Projects/rwt
```

Use your favorite editor to create the python file that will run our application. Name it 'rwt.py'.

```
gedit rwt.py
```

The first part of our code will be to set up the environment and pull in the libraries needed for this boat to float:

```
#!/usr/local/python/bin/python

####################################################
## SETUP THE ENVIRONMENT:
####################################################

import sys
import os
import time
import re

import pygtk
pygtk.require("2.0")

import gtk
import gtk.glade
```

In the next section, we need to read in the rwt.glade xml file so that we can access all our signals and have libglade build the GUI for us:

```
####################################################
### PULL IN THE GUI:
####################################################

wTree = gtk.glade.XML('rwt.glade')
```

wTree holds all of our GUI ... it's signals ... dialog boxes ... everything. We'll be referencing wTree quite a bit.

Even though we do not have our signals accounted for, we can go ahead and

connect them to our wTree. We will define our signal handlers later in the tutorial.

```
#####################################################
### CONNECT SIGNALS TO OUR HANDLERS:
#####################################################

wTree.signal_autoconnect(locals())
```

All that is left is to pass the program to gtk's main loop in order to run it.

```
#####################################################
### START UP THE PROGRAM:
#####################################################

gtk.main()
```

Save your work and run the program like this:

```
/usr/local/python/bin/python rwt.py
```

You should see your application window without the treeview displaying the transmitter information:

The application is pretty much useless as you can click on the buttons and menu items, but nothing happens. Click on a few of the buttons and menu items. Now pay attention when you click the little “x” in the window border. The GUI disappears; however the program does not return. The program is still running. In order to break out of the program you have to do a “ctrl – c” to kill it.

Code the Easy Stuff

In this section we're going to build the shell of our program. Most buttons and all menu items will be coded. The three parts we will leave for the next section will be the code for the Apply button on the main window, the code for the Apply button on the Preferences dialog and the code to build the list of transmitters on our main window.

In this section we'll handle the easy items just to get you used to thinking about how GTK likes to process and handle events.

The next section of our code will hold global type variables that a few of our functions will reference. I'm going to place our dialog boxes as global. Place the following code below "wTree = gtk.glade.XML('rwt.glade')" and above "### CONNECT SIGNALS TO OUR HANDLERS:"

```
#####################################################
### GLOBAL DEFINITIONS:
#####################################################

pref = wTree.get_widget('prefdialog')
about = wTree.get_widget('aboutdialog')
help = wTree.get_widget('helpdialog')
notebook = wTree.get_widget('notebook1')
alert = wTree.get_widget('alertdialog')
```

Next I'm going to hard code my transmitters. It's left up to you to figure out how to get this information from NWRWAVES.

```
### TODO: FIGURE OUT HOW TO USE NWRWAVES CONFIG
### TODO: FILES TO GET TRANSMITTER INFORMATION.
### TODO: HARD-CODING VALUES FOR NOW:
transmitters = (
        ['SDF', 'Standiford Field'],
        ['LEX', 'Lexington'],
        ['BWG', 'Bowling Green'],
        ['HCV', 'Horse Cave']
)
```

This program will use NWRWAVES to generate the product and place the information in the pending directory. Once NWRWAVES places the files in 'pending' we will go in and remove the 'extra' files that were generated and leave only the product for the specific transmitter that we chose from the gui.

The location of the pending directory is defined next:

```
pendingdir = '/data/fxa/workFiles/nwr/pending/'
```

This program is using a set of templates for each transmitter. They are passed to NWRWAVES after the date / time information has been updated by this program. These templates need the current Z time, the expire Z time and the local time stamp. I'm going to define these variables here as global.

```
current_time = time.localtime()
expire_time = time.localtime(time.time() + (5 * 60))
tmpl8_current = time.strftime("%d%H%M", current_time)
tmpl8_expire = time.strftime("%d%H%M", expire_time)
tmpl8_stamp = time.strftime("%H%M %p %Z %a %b %d %Y",
current_time)
```

Now we need to handle the signals that our GUI will pass to our python program. To get a list of signals that you need to program for, you can grep through rwt.glade for the word signal:

```
grep signal rwt.glade | awk '{ print $3 }'
```

Example:

```
lx2-lmk{tfreem}7: grep signal rwt.glade | awk '{ print $3 }'
handler="on_window1_destroy"
handler="on_quit1_activate"
handler="on_properties1_activate"
handler="on_about1_activate"
handler="on_applybutton_clicked"
handler="on_closebutton_clicked"
handler="on_prefdialog_destroy"
handler="on_pref_helpbutton_clicked"
handler="on_pref_cancelbutton_clicked"
handler="on_pref_applybutton_clicked"
handler="on_pref_okbutton_clicked"
handler="on_aboutdialog_destroy"
handler="on_about_closebutton_clicked"
handler="on_helpdialog_destroy"
handler="on_help_closebutton_clicked"
handler="on_alertdialog_destroy"
handler="on_alert_closebutton_clicked"
lx2-lmk{tfreem}8:
```

Notice that our main window and each dialog has a handler for the 'destroy'

event. If you see that you don't have a 'destroy' event handler for one of your dialogs or windows, go back into glade and define one.

Let's start creating our signal handler section.

```
####################################################
### CREATE SIGNAL HANDLERS:
####################################################

### MAIN WINDOW ITEMS: #############################

def on_window1_destroy(*args) : gtk.main_quit()
```

The above function handles the case where the user clicks the little 'x' on the application window.

The user can close the application by clicking the “Close” button. Let's handle for that case:

```
def on_closebutton_clicked(*args) : gtk.main_quit()
```

You can also close the application by clicking “File --> Quit”.

```
def on_quit1_activate(*args) : gtk.main_quit()
```

Another easy signal we can handle is when the user clicks on “Edit --> About”.

```
def on_about1_activate(*args) : about.show()
```

The next function we can define is in the case where the user selects “Edit --> Preferences”.

```
def on_properties1_activate(*args):
   notebook.set_current_page(0)
   pref.show()
```

This will set the current notebook page index to 0, which in my case is the SDF transmitter stuff, and then it will show the preferences dialog box.

We are left now with the apply button. Because this is a little involved, we'll just write a dummy function for it and come back later to fill it in.

```
###
### TODO:  handle the apply button function
```

```
###
def on_applybutton_clicked(*args):
    pass
```

Let's start handling the signals for our dialog boxes.

```
### ABOUT DIALOG ITEMS #############################

def on_aboutdialog_destroy(*args):
   about.hide()
```

All dialogs and windows should have a 'destroy' event that you need to code against. In this case we just want to hide the dialog.

```
def on_about_closebutton_clicked(*args):
   about.hide()
```

The About and Alert dialog boxs only have the 'close' button for the user to click. We do not want to destroy any of our dialogs, we want to hide them.

```
### HELP DIALOG ITEMS ##############################

def on_helpdialog_destroy(*args):
   help.hide()

def on_help_closebutton_clicked(*args):
   help.hide()


### ALERT DIALOG ITEMS #############################

def on_alertdialog_destroy(*args):
   alert.hide()

def on_alert_closebutton_clicked(*args):
   alert.hide()
```

In the Preferences dialog box, we'll need to write a bit of code to load in our templates. For now we'll define the function but complete the code later. Also, when the user clicks the “OK” button, the program should apply all changes (so we'll use the 'on_pref_applybutton_clicked' function for this) and then hide the dialog.

```
### PREFERENCES DIALOG ITEMS ######################

###
### TODO: handle the on_pref_applybutton_clicked function
###
def on_pref_applybutton_clicked(*args):
   pass

def on_pref_cancelbutton_clicked(*args):
   pref.hide()

def on_pref_okbutton_clicked(*args):
   on_pref_applybutton_clicked()
   pref.hide()

def on_pref_helpbutton_clicked(*args):
   help.show()

def on_prefdialog_destroy(*args):
   pref.hide()
```

Another section we'll come back to is the transmitter treeview listing on our main window.

```
###
### TODO:  handle the transmitter treeview section
###
```

Save and run the program:

```
/usr/local/python/bin/python rwt.py
```

All of your menu items and buttons should do things except for the Apply button on the main window and the Apply button on the Preferences dialog box.

Build the Transmitter Listing

Replace our “TODO” comment “### TODO: handle the transmitter treeview section” section with the following code:

```
####################################################
### CONNECT TO OUR TREEVIEW:
####################################################

treeview = wTree.get_widget('treeview1')
```

This is where we get into GTK type thinking. In our GUI, we want to list out our transmitter ID, description and an icon. We' ll ned what GTK refers to as a “liststore model”. A model simply describes the types of fields in the list . This list must match the description of the treeview (the next step in our program).

```
# CREATE A LISTSTORE MODEL TO USE WITH THE TREEVIEW:
# PATTERN IS: ICON IMAGE, ID TEXT, DESCRIPTION TEXT
treelist = gtk.ListStore(gtk.gdk.Pixbuf, str, str)
```

Now that we have our model that describes how information will be presented, we need to attach it to our treeview.

```
# ATTACH THE LISTSTORE MODEL TO THE TREEVIEW:
treeview.set_model(treelist)
```

Now that our treeview is set, we can describe the columns. The first column will contain pixmaps and thus show an icon that represents the transmitter, the next will be a text based column showing the SID of the transmitter and the last column will be text that describes the transmitter.

```
# FILL IN COLUMN HEADERS:
column = gtk.TreeViewColumn('', gtk.CellRendererPixbuf(),
pixbuf=0)
treeview.append_column(column)
column = gtk.TreeViewColumn('SID', gtk.CellRendererText(),
text=1)
treeview.append_column(column)
column = gtk.TreeViewColumn('NAME', gtk.CellRendererText(),
text=2)
treeview.append_column(column)
```

We'll use two icons, one to represent our home base transmitter and the other a general sort of icon to be used as filler. 'gtk-home' is the icon we'll use for the

SDF transmitter and the 'gtk-yes' icon will be used for the others. These are stock icons that glade provides. You can examine the listing of icons by opening glade and clicking one on of your buttons and using the drop down menu to see a full list of available stock buttons (and icons).

```
# GET SOME STOCK ICONS:
homeicon  = treeview.render_icon('gtk-home',
gtk.ICON_SIZE_SMALL_TOOLBAR)
othericon = treeview.render_icon('gtk-yes',
gtk.ICON_SIZE_SMALL_TOOLBAR)
```

Now we are ready to fill in our treeview with data. Remember we defined transmitters list in our global section? It consists of the SID and Description. We'll run through this list to check the SID and assign our icon based upon the SID.

```
# FINALLY ... FILL IN THE TREEVIEW USING OUR LISTSTORE
MODEL:
for i in range(len(transmitters)):
   xmtr = transmitters[i][0]
   description = transmitters[i][1]
   if (xmtr == 'SDF'):
      icon = homeicon
   else:
      icon = othericon
   treelist.append([icon , xmtr, description])
```

Lastly, we'll assign an item to be selected as default.

```
# BE NICE ... PROVIDE A DEFAULT SELECTION:
treeselection = treeview.get_selection()
treeselection.select_path(0)
```

Save and run the program:

```
/usr/local/python/bin/python rwt.py
```

You now see transmitters that you can select listed out on the main application window. SDF is preselected for you.

EXTRA

For debugging, you can change the "on_applybutton_clicked" function to print out the name of the transmitter you have selected. Change the following function:

```
def on_applybutton_clicked(*args):
    pass
```

... to:

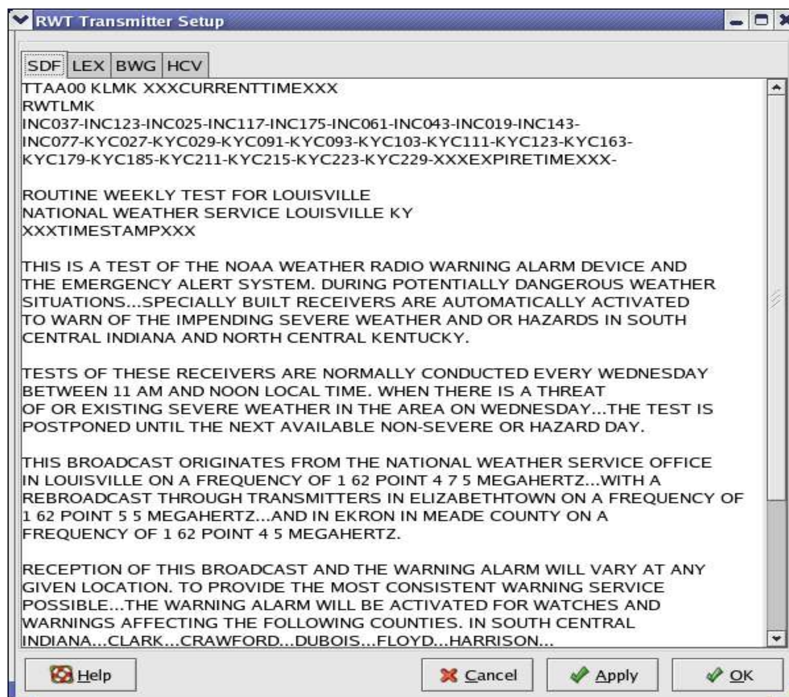
```
def on_applybutton_clicked(*args):
   # FIGURE OUT WHICH ITEM WAS SELECTED:
   (model, iter) = treeselection.get_selected()
   path = model.get_path(iter)
   index = path[0]
   selected = transmitters[index][0]
   print selected
```

Fill in the Notebook pages.

Place this code just above the "### CONNECT SIGNALS TO OUR HANDLERS:" section.

```
#####################################################
### FILL IN NOTEBOOK PAGES:
#####################################################

for i in range(len(transmitters)):
   xmtr_textview = transmitters[i][0] + "textview"
   xmtr_template = transmitters[i][0] + ".template"
   textview = wTree.get_widget(xmtr_textview)
   textbuffer = textview.get_buffer()
   if os.path.isfile(xmtr_template):
      infile = open(xmtr_template, "r")
      string = infile.read()
      textbuffer.set_text(string)
      infile.close()
   else:
      infile = open(xmtr_template, "w")
      infile.close()
```

This will read in our template file called SDF.template (for instance) and fill the textbuffer with it's contents. If the file does not exist, then a blank .template file is created for the user to configure.



An example .template file is in Appendix A

You should create a .template file for each of your transmitters and store those .
template files in the same directory as this rwt.py file.

Preferences Apply Button Code

The preferences dialog has a notebook for the user to tab between transmitter template files. If the user changes anything we need to save those changes to file. The user saves those changes by clicking the “apply” button or “OK” button.

Replace the following code:

```
###
### TODO: handle the on_pref_applybutton_clicked function
###
def on_pref_applybutton_clicked(*args):
   pass
```

... with the following code:

```
def on_pref_applybutton_clicked(*args):
   index = notebook.get_current_page()
   xmtr_textview = transmitters[index][0] + "textview"
   xmtr_template = transmitters[index][0] + ".template"
   textview = wTree.get_widget(xmtr_textview)
   textbuffer = textview.get_buffer()
   if (textbuffer.get_modified()):
      start, end = textbuffer.get_bounds()
      text = textbuffer.get_text(start, end)
      outputfile = open(xmtr_template, "w")
      outputfile.writelines(text)
      outputfile.close()
      textbuffer.set_modified(False)
```

Once the user is finished making changes and clicks the 'Apply” button (or 'OK' button) the code above figures out what tab the user is working on and stores that in a index variable. It then pulls the buffered data into the text variable and writes that to disk (replacing the current .template file for that transmitter). The last step is to reset the 'modified' flag of the buffer to false.

The 'if' statement above tests to see if the user has actually modified anything. If not, then clicking the .template file is left unchanged.

Save and run the program:

```
/usr/local/python/bin/python rwt.py
```

Test each one of the .template files by opening the 'preferences' dialog and

making changes to one of the transmitter's templates and clicking the 'Apply' button. Make sure that your changes are actually being written to disk, etc.

Main Window Apply Button Code

On the main window, when the user clicks the "Apply" button, we would like for our test message for the selected transmitter be sent to the pending side of NWRWAVES.

To do this, our program will pull in the transmitter's template file and substitute current time and expire times via regular expression and write the result to a temporary .dat file. This .dat file is sent to the textdb as a local product ... in my case I use SDFCRSLMK as the local product. Text triggers specify NWRWAVES to handle the SDFCRSLMK trigger. NWRWAVES creates the necessary file/files and is configured to place it/them in the pending directory with the RWT Issuance header and the RWT followup header. See Appendix B for a screenshot of the NWRWAVES setup for SDFCRSLMK. Once NWRWAVES is done, our program goes into the pending directory and removes the extra unneeded RWT files - leaving the RWT test message for our selected transmitter. An Alert box tells the user that the RWT is waiting in the NWRWAVES pending directory for review.

To do all of the above there are three functions:

1. on_applybutton_clicked
2. create_dat_file
3. clean_pending_directory

Replace this code (or whatever code you have for the on_applybutton_clicked function):

```
###
### TODO:  handle the apply button function
###
def on_applybutton_clicked(*args):
   # FIGURE OUT WHICH ITEM WAS SELECTED:
   (model, iter) = treeselection.get_selected()
   path = model.get_path(iter)
   index = path[0]
   selected = transmitters[index][0]
   print selected
```

... with the following code:

```
def on_applybutton_clicked(*args):
   # UPDATE THE TIME STUFF:
```

```
    global tmpl8_current, tmpl8_expire, tmpl8_stamp
    os.environ['TZ'] = 'EST5EDT'
    current_time  = time.localtime()
    current_time_gmt = time.gmtime()
    expire_time   = time.gmtime(time.time() + (7 * 60))
    tmpl8_current = time.strftime("%d%H%M", current_time_gmt)
    tmpl8_expire  = time.strftime("%d%H%M", expire_time)
    tmpl8_stamp   = time.strftime("%I%M %p %Z %a %b %d %Y",
current_time).upper()
    if tmpl8_stamp[0] == '0':
       tmpl8_stamp = tmpl8_stamp[1:]
    # FIGURE OUT WHICH TRANSMITTER IS SELECTED:
    (model, iter) = treeselection.get_selected()
    path = model.get_path(iter)
    index = path[0]
    # CREATE A DATA FILE AND SEND TO TEXTDB:
    datfile = create_dat_file(index)
    cmd = '/awips/fxa/bin/textdb -w SDFCRSLMK < ' + datfile
    os.popen(cmd)
    time.sleep(4)
    clean_pending_directory(index)
    alert.show()
```

The above function updates the global time variables tmpl8_current, tmpl8_expire and tmpl8_stamp. A .dat file is created via a new function we will write later called 'create_dat_file'. This 'create_dat_file' function creates a .dat file and returns the name of the .dat file so that it can be stored in the 'datfile' variable. We then send the information in the .dat file to the textdb program, sleep for 4 seconds and clean out the pendiing directory. Lastly the alert dialog box is shown.

Place this function above the 'on_applybutton_clicked' function:

```
def clean_pending_directory(index):
    rex1 = re.compile('LMKRWT')
    rex2 = re.compile(transmitters[index][0])
    files = os.listdir(pendingdir)
    for i in files:
       if re.search(rex1, i):
          if re.search(rex2, i):
             continue
          else:
             path = pendingdir + i
             os.unlink(path)
```

NWRWAVES saves all the RWT files with the WFO ID, RWT and Transmitter ID.

So our regular expression will be based upon “LMKRWT”. Replace this to reflect your local information. It then goes into the pending directory and removes all the other RWT products except for the transmitter that we have selected. Existing non LMKRWT products are left alone.

Place this function above the 'clean_pending_directory' function:

```
def create_dat_file(index):
   global tmpl8_current, tmpl8_expire, tmpl8_stamp
   xmtr_template = transmitters[index][0] + ".template"
   xmtr_output = transmitters[index][0] + ".dat"
   inputfile  = open(xmtr_template, "r")
   lines = inputfile.readlines()
   inputfile.close()
   new_lines = []
   tmp_lines = []
   ### TODO: CREATE A SINGLE FUNCTION TO HANDLE
   ### TODO: THESE THREE FOR-LOOPS:
   for l in lines:
      nl = l.replace('XXXCURRENTTIMEXXX', tmpl8_current)
      new_lines.append(nl.upper())
   for l in new_lines:
      nl = l.replace('XXXEXPIRETIMEXXX', tmpl8_expire)
      tmp_lines.append(nl.upper())
   new_lines = []
   for l in tmp_lines:
      nl = l.replace('XXXTIMESTAMPXXX', tmpl8_stamp)
      new_lines.append(nl.upper())
   outputfile = open(xmtr_output, "w")
   outputfile.writelines(new_lines)
   outputfile.close()
   return xmtr_output
```

This function replaces the XXX-------XXX placeholders in our template file with the appropriate time stamp information and saves this to a temporary .dat file. The function also returns the name of the .dat file that it created.

Please see Appendix C for a complete code listing.

Save and run the program:

```
/usr/local/python/bin/python rwt.py
```

You should now have a working program to use for operations. Test it out and make changes to the code as needed. There is a lot of room for improvement

such as making it Object Oriented, gathering data from other existing sources rather than hard coding them into the program, etc.

Have fun and good luck.

tony.freeman@noaa.gov

Appendix A
Example .template File

Filename: SDF.template

```
TTAA00 KLMK XXXCURRENTTIMEXXX
RWTLMK
INC037-INC123-INC025-INC117-INC175-INC061-INC043-INC019-INC143-
INC077-KYC027-KYC029-KYC091-KYC093-KYC103-KYC111-KYC123-KYC163-
KYC179-KYC185-KYC211-KYC215-KYC223-KYC229-XXXEXPIRETIMEXXX-

ROUTINE WEEKLY TEST FOR LOUISVILLE
NATIONAL WEATHER SERVICE LOUISVILLE KY
XXXTIMESTAMPXXX

THIS IS A TEST OF THE NOAA WEATHER RADIO WARNING ALARM DEVICE AND
THE EMERGENCY ALERT SYSTEM. DURING POTENTIALLY DANGEROUS WEATHER
SITUATIONS...SPECIALLY BUILT RECEIVERS ARE AUTOMATICALLY ACTIVATED
TO WARN OF THE IMPENDING SEVERE WEATHER AND OR HAZARDS IN SOUTH
CENTRAL INDIANA AND NORTH CENTRAL KENTUCKY.

TESTS OF THESE RECEIVERS ARE NORMALLY CONDUCTED EVERY WEDNESDAY
BETWEEN 11 AM AND NOON LOCAL TIME. WHEN THERE IS A THREAT
OF OR EXISTING SEVERE WEATHER IN THE AREA ON WEDNESDAY...THE TEST IS
POSTPONED UNTIL THE NEXT AVAILABLE NON-SEVERE OR HAZARD DAY.

THIS BROADCAST ORIGINATES FROM THE NATIONAL WEATHER SERVICE OFFICE
IN LOUISVILLE ON A FREQUENCY OF 1 62 POINT 4 7 5 MEGAHERTZ...WITH A
REBROADCAST THROUGH TRANSMITTERS IN ELIZABETHTOWN ON A FREQUENCY OF
1 62 POINT 5 5 MEGAHERTZ...AND IN EKRON IN MEADE COUNTY ON A
FREQUENCY OF 1 62 POINT 4 5 MEGAHERTZ.

RECEPTION OF THIS BROADCAST AND THE WARNING ALARM WILL VARY AT ANY
GIVEN LOCATION. TO PROVIDE THE MOST CONSISTENT WARNING SERVICE
POSSIBLE...THE WARNING ALARM WILL BE ACTIVATED FOR WATCHES AND
WARNINGS AFFECTING THE FOLLOWING COUNTIES. IN SOUTH CENTRAL
INDIANA...CLARK...CRAWFORD...DUBOIS...FLOYD...HARRISON...
JEFFERSON...ORANGE...PERRY...SCOTT...AND WASHINGTON. IN NORTH
CENTRAL KENTUCKY...BRECKINRIDGE...BULLITT...HANCOCK...HARDIN...
HENRY...JEFFERSON...LARUE...MEADE...NELSON...OLDHAM...SHELBY...
SPENCER...TRIMBLE...AND WASHINGTON.

THIS CONCLUDES THE TEST OF THE NOAA WEATHER RADIO WARNING ALARM AND
EMERGENCY ALERT SYSTEM. WE NOW RETURN YOU TO REGULAR PROGRAMMING.

$$
```

Appendix B

SDFCRSLMK Setup in NWRWAVES

NWRWAVES Setup Utility

File Help

Transmitter Configuration | Product Configuration | Summary Message/Misc Settings | Marine/Tropical Product Configuration

Select A Hazard Below Sort By: VTEC Product Issuance Header Followup Header Add Product

CRS - Local routine weekly test RWT RWT Save Edits ?

Message Properties

Process as Generic Message Type? Yes No ?

Use MRD Replace on CRS?: Yes No ?

Process ONLY for Core County/Zone?: Yes No ?

Process for Non-Routine Broadcast Service Area(s)?: Yes No ?

Intro: ?

Include Preamble? Yes No ?

Include County/Zone List? Yes No ?

Include Issue Time? Yes No ?

Include Headlines? Yes No ?

Generate Overview Product? Yes No ?

Include Supplemental Text? Yes No ?

Repeat Headline? Yes No ?

In Summary Message? Yes No ?

Transmission Properties

Select Broadcast Area: Routine Non-Routine

Interrupt Status: On Off ?

Alert Tones: On NWR Only Off ?

COR: On NWR Only Off

CAN: On NWR Only Off

CON: On NWR Only Off

EXA: On NWR Only Off

EXT: On NWR Only Off

EXB: On NWR Only Off

EXP: On NWR Only Off

Silence Period On Begin Time End Time

Local Time:

Storage: VIP (Default) ?

Transmission Status: CRS Pending ?

Periodicity: Minutes ?

CRS Effective Time: minutes after the hour ?

Default Duration: 5 Minutes ?

Use if UGC expiration missing Use in lieu of UGC

Appendix C

Complete Code Listing

```
#!/usr/local/python/bin/python

####################################################
## SETUP THE ENVIRONMENT:
####################################################

import sys
import os
import time
import re

import pygtk
pygtk.require("2.0")

import gtk
import gtk.glade


####################################################
### PULL IN THE GUI:
####################################################

wTree = gtk.glade.XML('rwt.glade')


####################################################
### GLOBAL DEFINITIONS:
####################################################

pref = wTree.get_widget('prefdialog')
about = wTree.get_widget('aboutdialog')
help = wTree.get_widget('helpdialog')
notebook = wTree.get_widget('notebook1')
alert = wTree.get_widget('alertdialog')

### TODO: FIGURE OUT HOW TO USE NWRWAVES CONFIG
### TODO: FILES TO GET TRANSMITTER INFORMATION.
### TODO: HARD-CODING VALUES FOR NOW:
transmitters = (
     ['SDF', 'Standiford Field'],
     ['LEX', 'Lexington'],
```

```
     ['BWG', 'Bowling Green'],
     ['HCV', 'Horse Cave']
)

current_time = time.localtime()
expire_time = time.localtime(time.time() + (5 * 60))
tmpl8_current = time.strftime("%d%H%M", current_time)
tmpl8_expire = time.strftime("%d%H%M", expire_time)
tmpl8_stamp = time.strftime("%H%M %p %Z %a %b %d %Y",
current_time)
pendingdir = '/data/fxa/workFiles/nwr/pending/'


####################################################
### CREATE SIGNAL HANDLERS:
####################################################

### MAIN WINDOW ITEMS: #############################

def on_window1_destroy(*args)     : gtk.main_quit()
def on_closebutton_clicked(*args) : gtk.main_quit()
def on_quit1_activate(*args)      : gtk.main_quit()
def on_about1_activate(*args)     : about.show()


def on_properties1_activate(*args):
   notebook.set_current_page(0)
   pref.show()



def create_dat_file(index):
   global tmpl8_current, tmpl8_expire, tmpl8_stamp
   xmtr_template = transmitters[index][0] + ".template"
   xmtr_output = transmitters[index][0] + ".dat"
   inputfile  = open(xmtr_template, "r")
   lines = inputfile.readlines()
   inputfile.close()
   new_lines = []
   tmp_lines = []
   ### TODO: CREATE A SINGLE FUNCTION TO HANDLE THESE THREE
FOR-LOOPS:
   for l in lines:
      nl = l.replace('XXXCURRENTTIMEXXX', tmpl8_current)
      new_lines.append(nl.upper())
   for l in new_lines:
      nl = l.replace('XXXEXPIRETIMEXXX', tmpl8_expire)
```

```
      tmp_lines.append(nl.upper())
   new_lines = []
   for l in tmp_lines:
      nl = l.replace('XXXTIMESTAMPXXX', tmpl8_stamp)
      new_lines.append(nl.upper())
   outputfile = open(xmtr_output, "w")
   outputfile.writelines(new_lines)
   outputfile.close()
   return xmtr_output



def clean_pending_directory(index):
   rex1 = re.compile('LMKRWT')
   rex2 = re.compile(transmitters[index][0])
   files = os.listdir(pendingdir)
   for i in files:
      if re.search(rex1, i):
         if re.search(rex2, i):
            continue
         else:
            path = pendingdir + i
            os.unlink(path)



def on_applybutton_clicked(*args):
   # UPDATE THE TIME STUFF:
   global tmpl8_current, tmpl8_expire, tmpl8_stamp
   os.environ['TZ'] = 'EST5EDT'
   current_time  = time.localtime()
   current_time_gmt = time.gmtime()
   expire_time   = time.gmtime(time.time() + (7 * 60))
   tmpl8_current = time.strftime("%d%H%M",
current_time_gmt)
   tmpl8_expire  = time.strftime("%d%H%M", expire_time)
   tmpl8_stamp   = time.strftime("%I%M %p %Z %a %b %d %Y",
current_time).upper()
   if tmpl8_stamp[0] == '0':
      tmpl8_stamp = tmpl8_stamp[1:]
   # FIGURE OUT WHICH ITEM WAS SELECTED:
   (model, iter) = treeselection.get_selected()
   path = model.get_path(iter)
   index = path[0]
   # CREATE A DATA FILE AND SEND TO TEXTDB:
   datfile = create_dat_file(index)
   cmd = '/awips/fxa/bin/textdb -w SDFCRSLMK < ' + datfile
```

```
   os.popen(cmd)
   time.sleep(4)
   clean_pending_directory(index)
   alert.show()



def add_image_list_column(title, columnid):
   column = gtk.TreeViewColumn(title,
gtk.CellRendererPixbuf(), pixbuf=columnid)
   treeview.append_column(column)



def add_text_list_column(title, columnid):
   column = gtk.TreeViewColumn(title, gtk.CellRendererText
(), text=columnid)
   treeview.append_column(column)



### PREFERENCES DIALOG ITEMS ######################

def on_pref_applybutton_clicked(*args):
   index = notebook.get_current_page()
   xmtr_textview = transmitters[index][0] + "textview"
   xmtr_template = transmitters[index][0] + ".template"
   textview = wTree.get_widget(xmtr_textview)
   textbuffer = textview.get_buffer()
   if (textbuffer.get_modified()):
      start, end = textbuffer.get_bounds()
      text = textbuffer.get_text(start, end)
      outputfile = open(xmtr_template, "w")
      outputfile.writelines(text)
      outputfile.close()
      textbuffer.set_modified(False)



def on_pref_cancelbutton_clicked(*args):
   pref.hide()



def on_pref_okbutton_clicked(*args):
   on_pref_applybutton_clicked()
   pref.hide()
```

```
def on_pref_helpbutton_clicked(*args):
   help.show()



def on_prefdialog_destroy(*args):
   pref.hide()



### ABOUT DIALOG ITEMS ##############################

def on_about_closebutton_clicked(*args):
   about.hide()


def on_aboutdialog_destroy(*args):
   about.hide()



### HELP DIALOG ITEMS ###############################

def on_help_closebutton_clicked(*args):
   help.hide()


def on_helpdialog_destroy(*args):
   help.hide()



### ALERT DIALOG ITEMS ##############################

def on_alert_closebutton_clicked(*args):
   alert.hide()


def on_alertdialog_destroy(*args):
   alert.hide()



####################################################
```

```
### CREATE A GTKTREEVIEW TO DISPLAY IN OUR GUI:
####################################################

# CONNECT TO THE TREEVIEW:
treeview = wTree.get_widget('treeview1')

# CREATE A LISTSTORE MODEL TO USE WITH THE TREEVIEW:
# PATTERN IS: ICON IMAGE, ID TEXT, DESCRIPTION TEXT
treelist = gtk.ListStore(gtk.gdk.Pixbuf, str, str)

# ATTACH THE LISTSTORE MODEL TO THE TREEVIEW:
treeview.set_model(treelist)

# FILL IN COLUMN HEADERS:
add_image_list_column('', 0)
add_text_list_column('SID', 1)
add_text_list_column('Name', 2)

# GET SOME STOCK ICONS:
homeicon  = treeview.render_icon('gtk-home',
gtk.ICON_SIZE_SMALL_TOOLBAR)
othericon = treeview.render_icon('gtk-yes',
gtk.ICON_SIZE_SMALL_TOOLBAR)

# FINALLY ... FILL IN THE TREEVIEW USING OUR LISTSTORE
MODEL:
for i in range(len(transmitters)):
   xmtr = transmitters[i][0]
   description = transmitters[i][1]
   if (xmtr == 'SDF'):
      icon = homeicon
   else:
      icon = othericon
   treelist.append([icon , xmtr, description])

# BE NICE ... PROVIDE A DEFAULT SELECTION:
treeselection = treeview.get_selection()
treeselection.select_path(0)


####################################################
### FILL IN NOTEBOOK PAGES:
####################################################

for i in range(len(transmitters)):
   xmtr_textview = transmitters[i][0] + "textview"
   xmtr_template = transmitters[i][0] + ".template"
```

```
   textview = wTree.get_widget(xmtr_textview)
   textbuffer = textview.get_buffer()
   if os.path.isfile(xmtr_template):
      infile = open(xmtr_template, "r")
      string = infile.read()
      textbuffer.set_text(string)
      infile.close()
   else:
      infile = open(xmtr_template, "w")
      infile.close()



####################################################
### CONNECT SIGNALS TO OUR HANDLERS:
####################################################

wTree.signal_autoconnect(locals())


####################################################
### START UP THE PROGRAM:
####################################################

gtk.main()
```