Introduction to Python Project Report

Dave Fugelso Introduction to Python, Fall 2014

# Abstract

This project proves the viability of incorporating PyQt to test application, network interfaces, and native Interfaces. The test module \_HupInterfaces replaces he native layer interface and displays templates directly as part of our HUP executable.

# Introduction

My company offer a product that runs on car stereo head units, referred as just ‘head units (HU)’ or platform that offers an interface to smart phone and acts a proxy to the Internet placing content on the HU from applications such as Google Search or Facebook or plays music from the Internet from application such as Pandora and Spotify.

The program is named “HUP” and acts as a gateway from the handset over Bluetooth or USB connection to the native code running on the HU. Our customers implement the user interface that the end user sees, but we give them a set of ‘templates’ that display data according to our interfaces.

Often system use Qt as the display API. This project provides templates for out use in testing both HUP and the application from the handsets on devices that have Qt, such as Ubuntu flavored embedded Linux, which is the platform used on this project.

Our interface to the native software goes from Python to a C interface. As part of the HUP executable, all the C function are compiled into a shared object named “\_HupInterface.so” and this in Imported in one place to the HupInterface.py. \_HupInterface creates a thread in which asynchronous events, such as user button clicks, can be sent to the HUP executable through a callback mechanism.

In order to short circuit the design, I replaced \_HupInterface.so with a python module which exposes the same interface. This is why this why the interface looks like C implemented in Python! But, the module defines a class (“Template’) to hold UI information and create a form when invoked.

# PyQt

First, a complaint. It took me almost an entire Saturday to get the package installed. Qt is installed on Ubuntu distributions, so that was there. I installed the PyQt using there instructions and the install failed because of mismatched Unicode byte sizes. Installation is two-byte while the PyQt was using four-byte. PyQt does not use a standard installation process, so I had to dig through all sorts of documentation and finally had to get the source and compile locally. There is a write up in the project folder that had the steps written down – mostly for future reference

Since I had so much difficulty, I am thinking that you may not want to install Qt and PyQt on anything other than an Ubuntu Linux system. I have no experience installing this system on any other OS.

Other than installation, to run the project, navigate to Alpine and run “python TestHarness.py.” (Did not make it bash executable.)

The code for the project is in ../PyQt plus the actuall TestHarness.py.

I did modify much of Template.py and HupInterface.py for this project but those were not created for this project.

# Test Harness

TestHarness reads json files form Alpine/testfiles to defines the contents of the template. If you run the program, you can change the behavior by modifying Template1A.json.

# UI Designer

I used PyQt designer to create the form tmeplate1.ui, which is used to create ui\_template.py automatically using:

pyuic4 template1.ui > ui\_template.py

From there I import the form into \_HupInterface.py.

# String Formats

The HUP executable communicates to platform device through a C library and includes many strings. Since we serve an international community all strings are passed as UTF-8, so all strings that are displayed in this project are UTF-8.

However, strings for pathnames are plain Unicode.

# Template1 Design

This is the template design used. This is a reference design we pass to our customer to implement. In this instance Template 1A. There eight templates and another four variants.

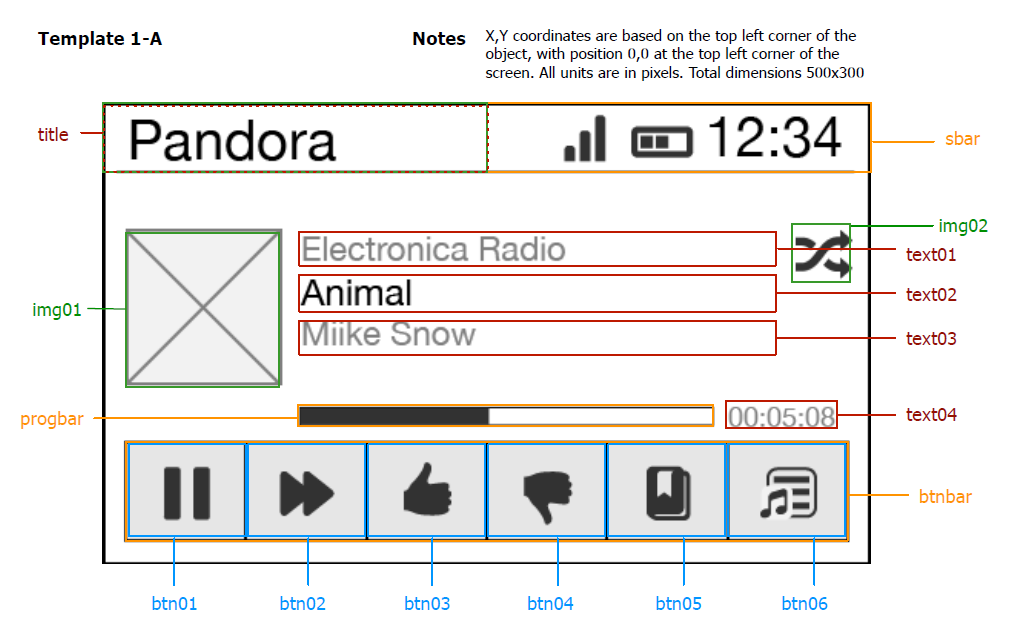


Figure 1 – Template Design

# Image Fetch Mechanism

One of the quirks in the project is how images are fetched. The template is given image IDs for each image in the template and it is up implementation to get these images. The purpose of this mechanism is that some images are stored as part of the executable, some are stored in a file cache, but others are loaded via connection to a smart phone which in turn acts as a proxy to the Internet.

For example, Pandora album artwork is fetched from Pandora and may take a while to load. We don’t stop the user from interacting with the UI while waiting for images to load.

# Button Console Output

The other important aspect of the design we want to test is getting push button events. On head units these are touch screen events. Here they are just normal buttons.

The output of the button are just reported to the console.

# Results

The following figure shows Template 1A shown in Ubuntu Linux.

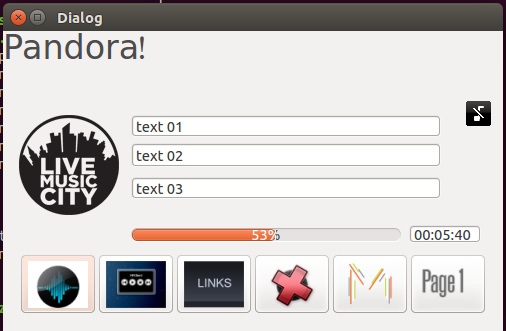


Figure 2 – Actual UI Display

# To Do

Finish the rest of the templates and do a lot more with formatting the UI, especially regarding handling text. There is no error checking here.

Add a hardware button screen to simulate buttons that are available on the HU but are not part of the display i.e. pause, fast forward/skip, etc.

# Interesting Things

The Templates module needs to keep track of the current template being displayed, so I ended up iusing a global variable to manage that. When the UI makes a callback into HUP (in the HUP Interface module), it needs to request the current template object from template.

So,

CurrentTemplate = None

And everywhere CurrentTemplate is set (eg.):

def \_\_init\_\_ (self):

global CurrentTemplate

CurrentTemplate = self

And retrieve using the following:

currentTemplate = None

def CurrentTemplate():

global currentTemplate

return currentTemplate

Another interesting piece of code, learned in the last class, is the use of a partial from functools.

I wanted to have a single callback function for all buttons. As it turns out buttons are differentially determined by index. The handset application just gets a message saying button ‘N’ is pushed and acts appropriately. The PyQt button class QToolButton doesn’t have a place to store user data.

This is what I did.

The callback function with key as a parameter:

def buttonClick (self, key):

'''

Pass a button click back to HUIP for processising.

'''

cbDict = {}

cbDict['Index'] = key

cbDict['templateId'] = self.appId

cbDict['type'] = 0

self.callback('TemplateButtonPress', cbDict)

But, QToolButton.clocked.connect(func) takes a function with only self as an argument.

So I used,

wid.clicked.connect(partial(self.buttonClick, self.button.Key))

Which creates a closure on the key data! Works like a champ.

# Conclusion

Since I had so much difficulty installing PyQt, I may have wanted to use PySide instead, which provides an alternate binding.

This was more work than I anticipated.

Replacing \_HupInterface.so, a C based interface, with a Python based \_HupInterface was much easier than anticipated. All I had to do was define the entry points and redirect where to load the module.

I really like the outcome, and this will be usable for work.

I found several bugs in production code while building this one simple template.