

Porting from Linux to Symbian OS

Part II – Guitar Tuner Example

Mark Wilcox

Published by the Symbian Developer Network

Version: 1.0 – August 2008

| | | |
|----------|-----------------------------------|----------|
| 1 | INTRODUCTION | 2 |
| 2 | GUITAR TUNER EXAMPLE | 2 |
| | 2.1 PROJECT SELECTION..... | 2 |
| | 2.2 CODE ANALYSIS | 3 |
| | 2.3 ARCHITECTURE CHANGES | 3 |
| | 2.4 DEVELOPMENT ENVIRONMENT | 4 |
| | 2.5 CREATING THE PROJECT | 4 |
| | 2.6 GETTING IT TO COMPILE | 4 |
| | 2.7 GETTING IT WORKING..... | 7 |
| 3 | SUMMARY | 9 |
| 4 | AUTHOR PROFILE | 9 |

1 Introduction

This paper describes how I ported a simple guitar tuner application from Linux to Symbian OS. It is designed as a companion to my previous paper,¹ which describes the porting process. This paper provides a practical example where the previous one gives the theory and structure. As such, you should read 'Part I – The Porting Process' first; the port in this paper follows the structure described there.

2 Guitar Tuner Example

I'd been thinking about getting an electronic tuner for my (acoustic) guitar for a while because, frankly, I'm unable to tune it properly by ear. The result is that my guitar is usually out of tune and makes my playing sound even worse than it really is. However, I didn't want to pay for one when my smartphones should all be able to do a perfectly good job. There are some commercially available Symbian applications for this purpose but no information on how they work, so I didn't know if they'd be any good. A paper on porting seemed like a great opportunity to test that assumption. The other reason for choosing a guitar tuner is the desire to focus on hybrid code, that is, mixing Symbian code with portable code. As mentioned in Part I, there are a number of porting examples and papers already available that take a portable engine and give it a new UI. However, that isn't always possible. In reality, there will often be parts of the project that don't separate so nicely but you'll still want to re-use as much of the code as possible.

2.1 Project Selection

A quick search reveals four Linux desktop-based tuner projects written in C or C++: Gstring, GuiTuner, KGuitune and G-tune. Looking at the project websites reveals that Gstring and GuiTuner both use the FFTW library, a fairly heavyweight, optimized, floating point Fast Fourier Transform (FFT) library. KGuitune uses a simple, integer-based, Schmitt-triggering algorithm (it basically counts the biggest oscillations) and G-tune uses a much simpler FFT algorithm to determine the sound frequency, but is still floating point. The absence of intensive floating point calculations makes KGuitune the top candidate, enhanced by the fact that there are already three versions, one for KDE, another purely using Qt for the UI and the third for gtkmm (the C++ bindings for the GTK+ libraries). All of the projects are fairly small one- or two-person efforts (as would be expected for a simple utility) and none appear to be abandoned, nor particularly actively maintained. Since Qt is likely to be available on Symbian in the future,² porting from it will become much simpler and any porting guide written now will soon be obsolete. For this reason I selected gtkGuitune, the gtkmm version of KGuitune. Hopefully that will mean that this paper is still useful for anyone porting from GTK+ code in the future.

¹ 'Porting from Linux to Symbian OS: Part I – The Porting Process' is available from developer.symbian.com/main/documentation/porting.

² This strategy document was published as a background to Nokia's acquisition of Trolltech: www.nokia.com/NOKIA_COM_1/Press/Press_Events/Nokia_to_acquire_Trolltech_to_accelerate_software_strategy/Nokia_Software_strategy.pdf.

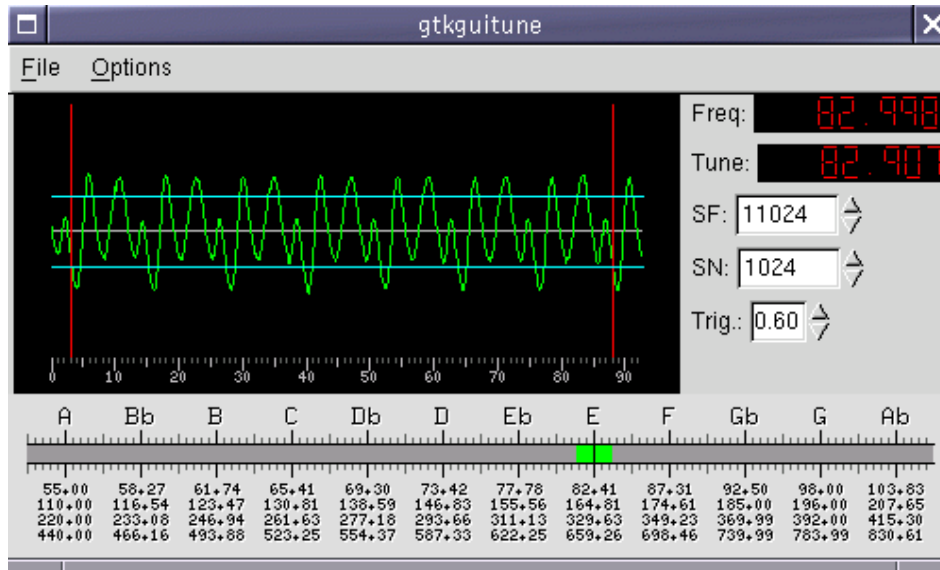


Figure 1: gtkGuitune screenshot.

2.2 Code Analysis

Scanning through the source files shows that there's a main container which hosts the various UI widgets, audio initialization and the main application logic. There are also three custom UI widgets, an oscilloscope-style view of the data, a logarithmic scale showing the note played and an LCD-style box for displaying frequencies. The settings boxes shown in Figure 1 are standard GTK+ widgets. The other points worth noting were that the audio is obtained by reading from `/dev/dsp` directly and that there is code to handle both unsigned 8-bit and signed 16-bit PCM (Pulse Code Modulation) data, although the 16-bit option is always used by default. There is very little use of other external libraries, only some simple mathematics and string formatting. The major negatives were that:

- it appeared to be basically C code in class wrappers
- some fundamental calculations are mixed in with drawing functions
- there are practically no comments.

This might prompt a re-visit of the project selection but in this case the other options seemed far less suitable.

2.3 Architecture Changes

The main change required is to shift from polling the audio via a callback from the GTK+ main loop to driving the application via the callbacks from a Symbian C++ `CMdaAudioInputStream` object. The audio input stream allows you to read buffers from the audio device in a very similar way to reading the raw data from `/dev/dsp` on Linux. The custom widgets are all derived from `Gtk::DrawingArea`, which is equivalent to a custom control in Symbian OS. Deriving these and the main container which holds them from `CCoeControl` on Symbian should make the bulk of the code portable between UI platforms, although I've chosen to target only S60 with this example.

The main widgets (controls) can be re-positioned to fit on the mobile screen, although the table of frequencies at the bottom of the logarithmic control will not fit and is not really necessary anyway. The settings boxes should be re-implemented as a native Symbian OS settings list in a separate view, if required.

2.4 Development Environment

For this project I'm using Carbide.c++ v1.3 Developer Edition under Windows XP with the S60 3rd Edition (Maintenance Release) SDK and a Nokia N95 (although this example should work with minimal or no changes on any S60 3rd Edition device). It is possible to use the free Express Edition of Carbide.c++ for development (commercial and freeware) but I prefer to use the Developer Edition, available from Forum Nokia (www.forum.nokia.com/carbide), which you have to pay for, but is the cheapest of those that are sold at the time of writing. Carbide.c++ Developer Edition gives me access to on-device debugging (which I do via Bluetooth) and a graphical UI designer tool. Both of these easily pay for themselves in terms of productivity gains, and experience working with multimedia on Symbian OS tells me I'll probably want to use the on-device debugging. The 'Express' version of Carbide.c++ is available for free download from the same site without these features.

2.5 Creating the Project

Since I wanted a standard S60 application, I created one from a template using the new project wizard in Carbide.c++ (in fact, I used the UI designer to create a blank container application, in case I needed to add a settings list later, but the result is very similar). This generates a number of source, header and resource files. The MMP and PKG files are also generated automatically. With a few small exceptions that I'll discuss later (in section 2.7) there was no need to edit any of these generated files apart from the application's main container. The container starts out as a blank control which fills the area between the status bar at the top of the screen and the soft keys at the bottom. I then created three copies of the container's header and source files, renaming them and doing a 'find and replace' on the class name internally, to produce the skeletons for the oscilloscope, logarithmic scale and LCD controls. Adding these files to the project in Carbide.c++ automatically adds them to the MMP file. Additional libraries, such as `mediaclientaudiointputstream.lib` (for `CMdaAudioInputStream`) have to be added to the MMP file manually, although there is a graphical editor with a drop-down selection box for this purpose.

My strategy was then to port the functionality across (mostly via cut, paste, find and replace) one control at a time, starting with the main container and the oscilloscope control which represent the core of the application. I debugged these before adding the other controls which are basically just additional (although more useful) views of the data. In this case, one control at a time equates to one source file at a time; it may be possible to use a similar incremental strategy with less UI-centric ports if the various classes or modules are not too closely coupled.

2.6 Getting it to Compile

The first major change was to re-write the audio handling. I based my implementation on the example in the Forum Nokia wiki.³ Typing 'recording audio' into the search box produced three article title matches. The second one, 'Recording audio with stream' was exactly what I wanted. I did simplify the wiki example slightly as I used it (replacing an array of heap-allocated `TBuf8` objects with a single `RBuf8` for the audio buffer⁴); although that wasn't essential, it made it easier to adapt to the ported code. Note that in order to use the native Symbian APIs you'll have to learn about descriptors, which Symbian OS uses for all string and binary data handling. However, once

³ See wiki.forum.nokia.com/index.php/Recording_audio_with_stream.

⁴ A good place to get started with descriptors, with references to more background material for those that want it, is:

developer.symbian.com/main/downloads/papers/RBuf/Introduction_to_RBuf_v1.0.pdf.

you've got the data from the API you can access it as an array from ported code by using the `Ptr()` method to get a pointer to the start of the data.

At this point, I must also confess to using some detailed knowledge of the target device to select a buffer size (4096 bytes) to match that used internally by the API. A novice would likely arrive at a working value by trial and error after discovering some quirks of the interface.⁵ Some slight modifications were needed to the function which processes the audio, as the original design read small blocks (256 bytes) in until it had accumulated enough to do the frequency calculations. Using the new system, we just return from the function and wait for the next large buffer from the audio device if we can't find a suitable section of data to process. If you examine the `CGui tuneContainer::proc_audio()` method in the example code,⁶ then you'll find that many more characters are required to describe the changes than to make them! In the `gtkmm` version, `proc_audio()` is the key function that gets called from the GTK+ main loop. In the Symbian OS port it is called whenever a new buffer of audio data is available, as indicated by a `MiscBufferCopied()` callback from the audio input stream. This should be far more power-efficient than the original polling design.

The other major difference is the way UI layout is done. The `gtkmm` code is using a number of nested horizontal and vertical boxes to arrange the controls. In Symbian OS you simply specify the rectangle that you want the control to occupy when you create it. There are two complicating factors to be aware of here. First, the container is what's known as a compound control, as it owns component controls. The class header contains an enumeration of the component controls, and the methods `CountComponentControls()` and `GetComponentControl(TInt alndex)` are used by the UI framework to determine how many controls there are and then to access each one individually in order to draw it. When you add a control you must include an entry for it in the enumeration and return a pointer to it when that value is passed as `alndex`. The second thing that's important to know is that Symbian OS supports multiple screen sizes and dynamic scalable UIs where the display can switch orientation from portrait to landscape. This means that you need to re-do the layout of your component controls when `SizeChanged()` is called and that the layout needs to work at different resolutions. You can test with the various screen sizes available on the emulator, even if you don't have all of the appropriate devices. The good news is that well written GTK+ widgets will handle re-scaling internally anyway since the windows they occupy on the desktop can be re-sized. The component controls share their parent's window (they could create their own windows but it's not very efficient) and all drawing is done relative to the origin of the parent window. This means you have to add offsets to the co-ordinates used for drawing whenever the layout changes. You can get the co-ordinates for the top left corner of a control by calling its `Position()` method and the size of the control's rectangle with `Size()`.

The minor differences are changing `Gdk_Color` to `TRgb` and switching the graphics context from `Gdk_GC` to `CWindowGc`. In the original version, the widget classes each own a graphics context. On Symbian OS, the standard model is to get the currently-active graphics context from the system when asked to draw. To minimize changes to the function internals I've modified all methods that draw in the original code to take a reference to a graphics context and passed it in as an argument. There are a number of system events in the `gtkmm` version which cause a redraw: `expose_event_impl()`, `draw_default_impl()` and `draw_impl()`. With Symbian OS, this is handled by the framework and anything the system does that requires a control to redraw itself will cause a call to its `Draw()` method with the rectangle that needs to be redrawn supplied as an

⁵ wiki.forum.nokia.com/index.php/TSS000318_-_Customizing_the_buffer_size_of_CMdaAudioInputStream

⁶ Example code, including the original application, is available for download from developer.symbian.com/main/documentation/porting

argument. As a result, these methods from the original version are simply not required. Other drawing-related differences are that functions for drawing lines and rectangles belong to the window class and take a graphics context as an argument in gtkmm, while they belong to the graphics context in Symbian OS. Additionally Symbian C++ has a more object-oriented style so it will take two pairs of co-ordinates, each encapsulated by a `TPoint` object, to draw a line or a point and a size to draw a rectangle, rather than a list of integer arguments. For example:

```
get_window().draw_line(gc, x1, y1, x2, y2);
```

becomes:

```
gc.DrawLine(TPoint(x1, y1), TPoint(x2, y2));
```

and:

```
get_window().draw_rectangle(gc, true, x1, y1, width, height);
```

becomes:

```
gc.DrawRect(TRect(TPoint(x1, y1), TSize(width, height)));
```

Symbian OS uses a pen color and a brush color rather than foreground and background colors. The difference is subtle but if you draw a rectangle it will be drawn in the pen color and filled with the brush color. If all you want is a solid rectangle of one color, then it is easiest to set the brush color and use the `Clear()` method rather than `DrawRect()`, like this:

```
gc.SetBrushColor(KRgbBlack);
gc.Clear(TRect(TPoint(x1, y1), TSize(width, height)));
```

One further drawing related difference highlighted by this port is for application-initiated drawing. In GTK+, you can draw wherever and whenever you like. On Symbian OS, you must invalidate the rectangle you want to redraw and activate the graphics context before drawing. When the system initiates a redraw it takes care of all this for you. For application-initiated drawing you need to use the following incantation:

```
Window().Invalidate(invalidRect);
ActivateGc();
Window().BeginRedraw(invalidRect);
<call drawing functions here>
Window().EndRedraw();
DeactivateGc();
```

The example code uses this pattern to update the oscilloscope and logarithmic scale controls for every new audio sample received. This is required because the existing drawing code for these only updates parts of the control in each case. The LCD controls need a complete update so they simply call `DrawNow()`, which requests an immediate redraw from the framework.

Text handling requires very similar changes to drawing⁷ but is slightly more complex due to the requirement to use 16-bit wide (Unicode) descriptors. It's simple enough to initialize an 8-bit wide descriptor with a zero terminated C-style string, or to use a `TPtr` descriptor type to point to the contents of one, but you then have to widen it to 16 bits. Fortunately, there is a `Copy()` method in the 16-bit descriptor base class (`TDes16`) which takes an 8-bit descriptor as an argument and does the right thing for ASCII text. If your string is UTF-8 encoded, then you'll have to use a text conversion utility.

⁷ Another quirk discovered during this port relates to fonts. `CWScreenDevice::GetNearestFontToMaxHeightInTwips()` returns a font that is missing most of the ASCII characters on S60 3rd Edition but is fine on S60 3rd Edition FP1. `GetNearestFontToDesi gnHeightInTwips()` appears to work on both SDKs, though.

A couple of final points relate to mathematical and string manipulation functions. The application uses `log()` and `fabs()` from `math.h` as well as `sprintf()` from `stdio.h`. These functions are available as part of P.I.P.S.⁸ but since they were only used a couple of times and there were simple alternatives, I've replaced them with native Symbian C++ operations (and a simple macro in the case of `fabs()`) to avoid an unnecessary dependency. In future, P.I.P.S. support is likely to ship as part of the device firmware but current devices require the user to install it as a separate SIS file. In this case, the P.I.P.S. install would be much larger than the application it is supporting. This wouldn't be an issue with a larger project and for one that makes extensive use of P.I.P.S. or Open C/C++ functionality, it wouldn't be at all advisable to switch to native calls.

2.7 Getting it Working

As there is no test harness with the original project, I decided that some very simple manual tests would suffice. I used a signal generator program to produce sine waves across a range of frequencies and ensured that the application showed a sine wave on the oscilloscope and identified the correct note and frequency. After that, the final and most crucial test: could I use it to tune my guitar?

The first serious bug I came across was a blank container after starting the application, when I should have been seeing my custom controls! I spent a frustrated couple of hours checking that the drawing code was running correctly and trying to find out what was drawing over my controls or whether I'd made a mistake with the co-ordinates. The error was one that you could easily repeat if you're in a hurry so I'll share my embarrassment here. I'd copied RGB color values from the GTK+ code into ARGB values without realizing. The result is that the alpha value was set to zero for all the colors I was using. All of the drawing code was working the whole time; I was just drawing in invisible ink!

The next issue was that the oscilloscope control was being drawn too large. This was simply because, while copying the skeleton code, I'd forgotten to remove the following line:

```
SetRect( static_cast< CEikAppUi * >( iCoeEnv->AppUi () )->ClientRect() );
```

from the `HandleResourceChange()` method of the control (which gets called when the control is created and whenever it changes size or position). The control was setting its own rectangle to fill the entire window.

With that solved, the application was basically working in the emulator, so it was time to move over to a real device (although note that at this point I'd only ported the oscilloscope control). In order to use the `CMdaAudioInputStream` API to capture audio, the application needs to have the `UserEnvironment` capability. This just needs to be added to the MMP file. If you miss a capability then your project will still work in the emulator, where platform security enforcement is turned off by default, but not on a real device.

Unfortunately, as soon as the application started in the device it exited with a 'Feature not supported' dialog. Experience (and common sense) tells me that the main difference between the device and the emulator for this application is the access to the audio hardware. So, I fired up my on-device debugging from `Carbide.c++` and set a breakpoint in the audio initialization code. Sure enough, one of the set-up functions for `CMdaAudioInputStream` was leaving:

```
iAudioInput->SetAudioPropertiesL(iAudioSettings, iSampleRate,
                                iAudioSettings, iChannels);
```

I initially thought that this was because of an unsupported sample rate but found that it doesn't work with any sensible value, so it seems that the method is not supported on the Nokia N95 (although it has been on earlier models). Differences between devices like this are not that

⁸ P.I.P.S. Is POSIX on Symbian OS. A developer booklet is available from developer.symbian.com/main/documentation/books/books_files/pdf/P.I.P.S..pdf.

uncommon and it's good practice to TRAP the leave and deal with the failure at runtime. However, in this case the call is redundant because the audio properties are supplied previously in the call to `Open()` the audio input stream, so I simply commented it out. In theory the `MailscOpenComplete()` callback should give you an error code other than `KErrNone` if the settings are not supported. In practice, I've found that this doesn't occur and you get a sample rate of 8000Hz, no matter what value you request, and no error. I hard-coded the sample rate accordingly, so that value doesn't require a setting item. Also, the number of samples displayed is somewhat constrained by the width of the screen so I decided that didn't require a setting either. Eventually, I found that the default value for the trigger level (0.6) worked very well for identifying frequencies and so I decided not to have any settings at all, making the application a simple single view.

Having decided that no user options were required, I changed the resource file that defines the soft keys (this is the application's main resource file, in this case `guitune.rss`) so that it used `R_AVKON_SOFTKEYS_EXIT` rather than `R_AVKON_SOFTKEYS_OPTIONS_EXIT`.

This had the desired effect of producing only a single 'Exit' soft key but now I couldn't exit the application with it, on the emulator or the target. `HandleCommandL()` in the `ApplicationUI` class, in this case derived from `CAknAppUi`, is where the keypad input goes by default, so I set a breakpoint in the debugger to look for the appropriate command. In this case, it seems that `EAKnSoftkeyBack` is generated rather than `EAKnSoftkeyExit`, so I added that value to the ones that result in the application exiting. These were the only changes I had to make to the generated application skeleton. If I were polishing the application for release I might want to change the application icon too.

I now had a working guitar tuner that passed all of my tests. One final change was required to get a satisfactory result on the device. After a short period of use the backlight on the display would go out, making the screen almost impossible to see. This is a standard power saving feature during periods of user inactivity on Symbian smartphones and in most cases is what you want to happen. However, in this case we are using the device but it has no way of knowing that because we aren't pressing any keys. Guitar tuning is a two handed operation, so stopping to press a key to get the backlight back on isn't at all user friendly. The solution here is to call `User::ResetInactivityTime()` at regular intervals, in this case in the callback from the audio input stream. With that in place we have a really usable little application.

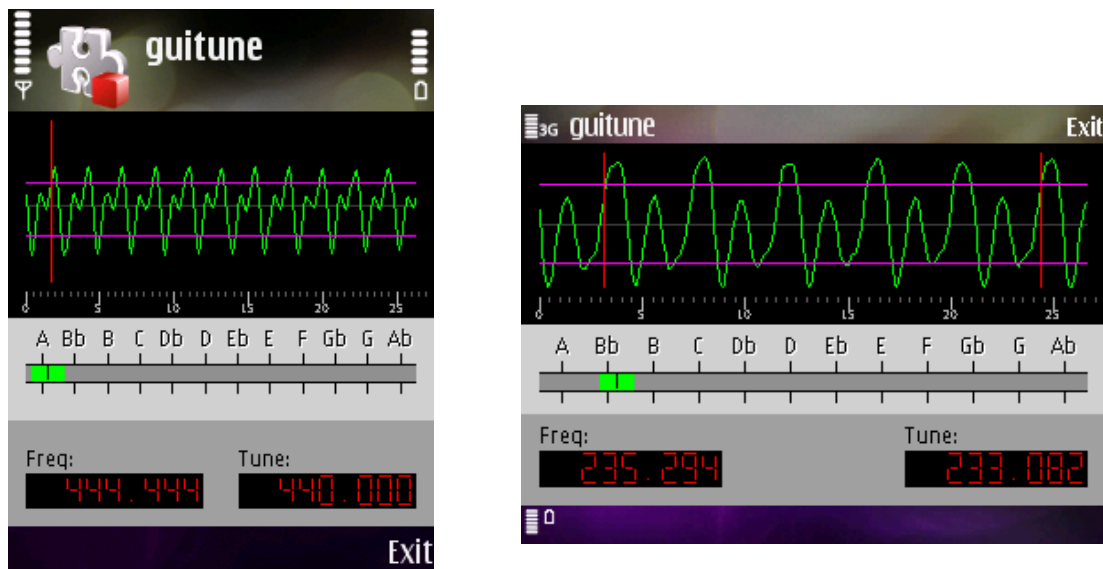


Figure 2: Screenshots of the tuner running on a Nokia N95.

To be a good smartphone citizen, we should also implement our own power saving timeout of suitable length and stop the audio input stream (and hence the inactivity resets). You'd probably then want to add a soft key to re-start it at a later time. The same stop / start mechanism could also be used if the application was moved to the background (currently it will just keep running). However, this is a porting example and not a finished product; these features are left as an exercise for the reader.

Note that the code changes from the original here are too significant to make any re-integration feasible. The ported application would have to be maintained separately from the original but due to the GPLv2 licensing, full source code must be released along with a copy of the license if the project is distributed at all.

3 Summary

In this paper, I've discussed the issues involved in porting a simple application that uses both UI and multimedia functionality from Linux to Symbian OS. Using this guitar tuner example I've shown that even where standard APIs from Linux are not available, porting to Symbian OS is not too daunting a task. The example also highlights some common issues that need to be considered when porting to mobile platforms in general.

Porting is a huge topic and we have only scratched the surface with this example. What do you do if there isn't an equivalent control or widget in your UI toolkit? How do you mix standard C libraries with Symbian C++? Or Symbian leaves with standard C++ exceptions? If you explore the resources listed at the end of Part I you should be able to find most of the answers. If not, you should find someone to help from the fantastic Symbian developer community on one of the discussion boards. Good luck and happy porting!

4 Author Profile



Mark Wilcox freelances as a software architect, developer and technical author, with a strong interest in multimedia technologies, mobile platforms and Symbian OS in particular. He is an Accredited Symbian Developer and a Forum Nokia Champion.

Mark co-authored the recently announced [Multimedia on Symbian OS: Inside the Convergence Device](#) for Symbian Press and is currently working on a second book about porting to Symbian OS.