Porting JavaFX to additional embedded Linux devices

Build for existing devices

JavaFX on ARM runs on Linux/ARM devices. The first step to creating a port is to be able to build for existing devices. Instructions for building for ARMv6 hard-float devices on a Linux/86 desktop host are at Cross Building for ARM Hard Float. Once you have followed the instructions there to install development tools and libraries and run a build, you should have binaries for ARMv6 with the hard float ABI. (The “ABI” determines what processor registers are used to pass arguments to function calls and return results. With the hard float ABI the floating point registers are used as well as the integer registers. Hard float ABI is a common configuration of Linux distributions for ARM development boards today.)

The instructions below are for ARMv6 hard float platforms, but can be followed with minor differences in the build scripts for configurations using the soft float ABI or other versions of the ARM architecture.

Create a new Monocle Java implementation

JavaFX ports to Linux/ARM devices are implementations of a subsystem in JavaFX called Monocle. Monocle is in turn an implementation of the Glass subsystem that provides JavaFX with windowing, screen access, cursors and input. Monocle needs the following components to make up a port:

- NativeScreen
- AcceleratedScreen
- NativeCursor
- InputDeviceRegistry
- NativePlatform
- NativePlatformFactory

These classes and interfaces are in the JavaFX sources in the directory modules/graphics/src/main/java/com/sun/glass/ui/monocle. Their functions are described below.

NativeScreen

A `NativeScreen` provides a way for JavaFX to get information about the screen it is using and to put pixels on the screen using software rendering. Most implementations in Monocle of `NativeScreen` are based on Linux frame buffers, but other implementations are possible. For example, the `X11Screen` class creates a single window using the X API and renders all content into that.

Note that there is no class "NativeWindow" in Monocle. This is because Monocle does not make use of any native window system on the devices it uses, and does not require such a window system. In general, Monocle assumes that it owns the entire screen and makes no attempt to cooperate with any window system that may be present.

Windows and Stages in JavaFX on embedded Linux devices are concepts internal to JavaFX that do not correspond directly to any native window system. In practice window content is stored in JavaFX and the window stack is composed onto the screen surface whenever JavaFX enders a frame. When using the software rendering pipeline, window content is stored in a `Pixels` object that wraps a `ByteBuffer`. When using the accelerated OpenGL ES 2.0 pipeline, window content is stored internally in an OpenGL Texture.

AcceleratedScreen

An `AcceleratedScreen` provides a way to get an OpenGL ES context on the screen. This typically uses the native EGL API to create a drawing surface. While the EGL API used to create the surface is standard (`eglCreateWindowSurface`), it requires a `NativeWindowType` parameter whose meaning varies from platform to platform. It is the call to the native function `eglCreateWindowSurface` that typically needs to be customized for a new port. This function on some platforms takes 0 as an argument to indicate that the screen's framebuffer is to be used for output. On other platforms a pointer to some data structure is required. For example,

- When using the OMAP3 graphics drivers for the BeagleBoard xM, a value of 0 for the `NativeWindowType` indicates that the screen's framebuffer is to be used for output.
- On the Raspberry Pi's Broadcom chipset, `NativeWindowType` is a pointer to a surface created with calls to the dispmanx API.

- On Mali platforms that support accelerated framebuffer rendering, a pointer to a structure containing the width and height of the screen is used as the `NativeWindowType`.
- With Freescale i.MX6 graphics drivers, the result of calling a function `fbGetDisplayByIndex` is used as the `NativeWindowType`.
- When rendering to an X11 window, the window XID is used.

NativeCursor

If you are using a mouse or other relative pointing device with your platform then you will need a cursor. If your platform is using software rendering then you don't need to do anything; you can use the existing `SoftwareCursor` implementation. For hardware accelerated rendering you will need to create a hardware cursor on a separate rendering layer. How you do this is highly platform dependent, but many Linux/ARM devices support multiple graphics layers that are alpha-blended by the display hardware. For example

- On TI OMAP3 devices, a framebuffer overlay `/dev/fb1` can be configured. JavaFX sets the size of this framebuffer to match the cursor size, and updates the cursor frame buffer's X and Y offsets whenever the cursor is moved.
- On Freescale i.MX6 devices the same concept is used, but the system calls to set the cursor frame buffer's transparency and location are different.
- On the Raspberry Pi a second display surface is created with the `dispmanx` API on a graphics layer on top of the layer used for rendering window content.

A concept frequently used in framebuffer overlays is color keys. Color keys let you designate a particular pixel value as representing the transparent color. This lets you use transparency in the cursor even on screen formats that use otherwise opaque pixel formats such as 16-bit RGB 5/6/5.

Another issue to be aware of when creating a new cursor port using a framebuffer overlay is that not all platforms allow an overlay to extend beyond the screen bounds. This can cause problems when rendering a cursor at the extreme right or bottom of the screen.

The class `NativeCursors` provides methods for handling color keys and adjusting cursor images so that they fit within the screen bounds.

InputDeviceRegistry

An `InputDeviceRegistry` handles discovery of input devices and feeding the events from these devices into the JavaFX event queue. The `LinuxInputDeviceRegistry` included in JavaFX uses the `udev` Linux system to discover devices and reads its input from device nodes in `/dev/input`. Other implementations are also possible, such as `X11InputDeviceRegistry` which takes its input from the X event queue.

NativePlatform

A `NativePlatform` bundles together a `NativeScreen`, `NativeCursor` and `InputDeviceRegistry`. `NativePlatform` is a singleton, as are the classes it contains.

NativePlatformFactory

A `NativePlatformFactory` creates a `NativePlatform` implementation. At startup a list of NativePlatformFactories is instantiated and they are queried in turn to see if they match the platform on which JavaFX is running. Once a match is found, a `NativePlatform` is instantiated.

Adding a platform

Monocle platforms use a naming convention of `com.sun.glass.ui.monocle.<name>PlatformFactory` for a platform called `<name>`. For example, the platform factory for TI OMAP3 devices is `OMAPPlatformFactory` and can be explicitly selected on the command line with `-Dmonocle.platform=OMAP`.

The platform factory should provide an implementation of the `matches()` methods that performs a quick check to see JavaFX is running on a platform that matches the target hardware. For example,

- Check for the existence of virtual file created by a Linux kernel module specific to the platform
- Check for the existence of a shared object unique to the platform
- Check for an environment variable that is set on the platform

The platform factory check does not have to be perfect; it is allowed to respond with a false positive result. However in this case the call to its `createNativePlatform()` method should perform a more rigorous check and fail if necessary.

`LinuxPlatform` is typically a good parent class for a new `NativePlatform` implementation. `LinuxPlatform` provides standard Linux device input, a software cursor and access to `/dev/fb0` for software rendering. Each of these components can be overridden separately.

By convention the classes for each platform are called `<name>Platform`, `<name>Screen`, `<name>AcceleratedScreen`, `<name>Cursor` and `<name>InputDeviceRegistry`. Not all these components need to be implemented for every port.

Implement an accelerated screen

An `AcceleratedScreen` implementation needs to provide a handle for the native display (used in a call to `eglGetDisplay`) and the `NativeWindowType` parameter (used in a call to `eglCreateWindowSurface`). Once the `AcceleratedScreen` is implemented you should be able to run applications and see output on the screen.

A platform that does not support OpenGL ES 2.0 does not need to provide an AcceleratedScreen.

Implement a hardware cursor

A hardware cursor is the part of a JavaFX port that is most dependent on the specific graphics hardware and drivers used. The `NativeCursor` abstract class defines methods for setting cursor image, location and visibility. These methods need to be implemented for each platform that provides a hardware cursor. The `NativeCursors` class provides utility methods for generating framebuffer pixel buffers that use color keys for transparency. `NativeCursors` also provides methods for generating cursor images that are shifted down or to the right; this is needed to render cursors at the right or bottom edges of the screen.

A platform that will only be used with touch or key input does not need to provide a `NativeCursor`.

Implement input

The class `LinuxInputDeviceRegistry` handles discovery of devices using the Linux `udev` system and creation of InputDevice instances based on those devices. For each `InputDevice` a thread is created to read data from its device node in `/dev/input`. This thread reads data into a `LinuxEventBuffer`. On the JavaFX application thread this data is read out of the `LinuxEventBuffer` by a `LinuxInputProcessor` that updates the input state.

None of the above needs to be changed for a system that uses Linux device input. For a platform that takes its input from other sources, a modified `InputDeviceRegistry` might be needed. For example, if a system reads key input from an infrared remote though a receiver connected to a GPIO port, it would need to:

- Report that a key device is connected
- Listen to data on the GPIO port
- Translate that data into calls to `KeyInput` with an updated `KeyState`

The three classes used to process input are: `KeyInput`, `MouseInput` and `TouchInput`. Each has an associated state object that it receives. These classes are not thread-safe and may only be used on the JavaFX application thread.

There is a simple example of an input system in `X11InputDeviceRegistry`. This example takes mouse motion and button events and calls the `MouseInput` class to report changes in mouse state.

To add support for an input device that uses the same Linux device node input mechanism as the currently supported devices, you will need to implement a new `LinuxInputProcessor` subclass and instantiate it in `LinuxInputDeviceRegistry.createInputProcessor`.

Create a new Monocle native implementation

Most calls to native code in Monocle are done through Java classes that wrap a single native system or library call in a Java method. For example, the classes `LinuxSystem`, `C`, `X11` and `EGL` have minimal logic in their C code.
For some platforms additional classes that call platform-specific code might need to be added. In this case a new shared object should be defined in `buildSrc/armv6hf.gradle` in the property `ARMV6HF.glass.variants`. The variant will define a shared object with its own compiler and linker flags. If additional header files and library files for the target platform need to be used for the native library, this is the place to add the compile flags to use them. An example of a monocle native library is in the `monocle_x11` variant, which compiles the Monocle file `X11.c` and links it against X11 native libraries.
If additional compiler and linker flags are needed for all native code in JavaFX for this platform, you will need to change the properties `extraCFlags` and `extraLFlags` in the same file.
If you are making significant changes to the build script then you will probably want to create a new build configuration. To do this you can copy `buildSrc/armv6hf.gradle` to a new file and use its name as a compile target. For example, you could create `buildSrc/armv8hf.gradle` and build with `gradle -PCOMPILE_TARGETS=armv8hf`.

Build

Build as usual, following the instructions at Cross Building for ARM Hard Float