KPR Config

1. Overview

KPR Config is a set of tools, patches and make fragments to build KPR based apps across platforms. Its objectives are:

- To use the same kind of embed build across all platforms.
- To simplify the development of multiple apps in parallel.
- To simplify the development of KPR extensions in XS and C.
- To automate the compilation of programs and modules deployed on devices.
- To eliminate useless virtual machines (root, Kinoma Create) and to alias all virtual machines (shell, application, HTTP, SSL) from the same shared virtual machine.
- To allow KPR shells to be described with KPR Markup or defined with ECMAScript.

Separate efforts are ongoing to provide an Eclipse based development environment for KPR, and to provide a CMake based build system for FSK. Hopefully KPR Config will help such processes, although building Kinoma Play is obviously out of scope.

The following paragraphs contain relevant although sometimes obscure details. You can skip them to go to a platform specific how-to.

2. Embed Build

XS provides a tool, `xsconfig`, to build dynamically loadable binaries (.xsb) and libraries (.so, .dll) from a grammar and a make fragment; and FSK provides a tool, `fskconfig`, to build FSK itself and the relevant extensions from the app manifest. There are platform dependent make files to build the FSK core; `fskconfig` generates a make file that uses `xsconfig` to build each extension.

To optimize apps on devices, `fskconfig` can also generate XS, C and make files to create an embed build: all XS code is linked into one binary and all C code is linked into one executable. It is quite a challenge to combine all make files on every platforms, and the process even requires two passes for KPR based apps.

Now KPR provides a tool, `kprconfig`, which is like a combination of `fskconfig` and `xsconfig`. It generates XS, C and make files from the app manifest and from the extensions grammars and make fragments. The make file always creates an embed build.

To build the FSK core, `kprconfig` uses a grammar and a make fragment similar to the extensions grammars and make fragments. See `kpr/patches/FskCore.xs` and `kpr/patches/FskCore.mk`.

Each make fragment can have platform specific parts. The main platform specific files are `FskPlatform.h` and `FskPlatform.mk` for flags and options; `prefix.make` and `suffix.make` for variables and rules. See for instance such files in `kpr/make/mac/`.

KPR based apps use several programs, modules and assets; the make generated by `kprconfig` contains instruction to compile or copy such files based on the `copy` and `exclude` elements of the manifest.

3. Extensions

KPR extensions are FSK extensions. The manifest of an app lists the extensions used on such or such platforms. Because of the embed build, extensions can always use the KPR C and ECMAScript programming interfaces without intricate code to bind functions across extensions.

Extensions can augment the ECMAScript programming interface of KPR. For instance the Crypt extension provides decryption/encryption properties to apps. KPR executes programs and modules in the XS sandbox so extensions have to use script="true" to make properties available.

Extensions can also provide features thru KPR messages and services, like the WiFi extension for instance. With KPR Config, if an extension does not need an ECMAScript programming interface, it does not need to provide an empty .xs file, just a .mk file and .c files.

4. Compiled Programs and Modules

KPR programs and modules can be described with KPR Markup (.xml files) or defined with ECMAScript (.js files). At runtime, KPR parses .xml files to generate ECMAScript then parses it, or parses .js files. It helps to design and develop apps: reloading the .xml and .js files is quicker than rebuilding the application.

But on devices, the overhead is significant. Loading byte code is always faster than generating and parsing ECMAScript. So KPR Config uses a preprocessor, `kpr2js`, to generate .js files from .xml files and the XS compiler, `xsc`, to generate .jsb files (byte code) from .js files.

The XS compiler needs to be told if a .js file is a program or a module. It is obvious with KPR Markup, since the document is either a `<program>` or a `<module>` element. When programs and modules are defined in ECMAScript they need to begin with the conventional `//@program` or `//@module`.

The compilation has an interesting side effect to debug programs and modules on devices because the path of the source is part of the byte code.

5. Virtual Machines and Symbols

Historically, the XS in C programming interface provided the `xsID` macro to access properties. The `xsID` macro uses a string to lookup the symbol and returns an id. For the sake of performance, FSK cached some ids.

The XS compiler now generates a C header file with macros that return ids from the symbol table of the grammar: `xsID_foo` instead of `xsID("foo")`. The process is now incremental so there is no need to recompile C source files when symbols are added or removed. The XS compiler parses the formerly generated C header file to reuse the formerly generated symbol table.

C functions can be called by various virtual machines. For such macros to work, all virtual machines need to be loaded with the generated symbol table. KPR Config patches load the symbol table in one virtual machine then share and alias it to create all other virtual machines.

6. Shell

Initially, for the sake of integration with the FSK virtual machines, KPR shells used a file format unlike other KPR programs and modules.

```
<shell xmlns="http://www.kinoma.com/kpr/1"
          width="1260" height="840"
          modules="../../simulator/modules/">
<![CDATA[
     // ECMAScript code
]]></shell>
```

Such a format is still supported but the width, height and modules attributes are ignored.

The shell can now be described with KPR Markup or defined with ECMAScript. Like other programs and modules, the shell is also compiled on devices.

7. Manifest

KPR Config works with the usual manifests and make fragments files.

However, two new variables are required in the manifest of your app:

- `modulePath`: the paths to your module directories, separated by semicolons. For the `mac` and `win` platforms, paths are usually relative to the build time environment variable `F_HOME`; for the other platforms, paths are usually relative to the run time environment variable `applicationPath`.

```
<variable name="modulePath"
     value="[F_HOME]/kinoma/kpr/simulator/modules/"
     platform="mac"/>
<variable name="modulePath"
     value="[applicationPath]kinoma/kpr/simulator/modules/"
     platform="iphone,linux"/>
```

- `shellPath`: the path to your app shell. The shell can of course depend on the platform.

```
<variable name="shellPath"
     value="[F_HOME]/kinoma/kpr/applications/k2/shell.xml"
     platform="mac"/>
<variable name="shellPath"
     value="[applicationPath]kinoma/kpr/applications/k2/phoneShell.jsb"
     platform="iphone"/>
<variable name="shellPath"
     value="[applicationPath]kinoma/kpr/applications/k2/switchShell.jsb"
     platform="linux"/>
```

There are also two optional variables to define the initial width and height of the window on for the `mac` and `win` platforms.

```
<variable name="windowWidth" value="1600" platform="mac,win"/>
<variable name="windowHeight" value="900" platform="mac,win"/>
```

8. Platforms

8.1. Mac OS

The only required environment variable is `F_HOME`, the path of the `fsk` directory. For convenience, add the path of the tools directory, `$F_HOME/xs/bin/mac`, to the `PATH` variable.

Firstly, build the `xsc`, `kprconfig` and `kpr2js` tools:

```
cd $F_HOME/xs/makefiles/MacOSX
make kpr
```

Command Line

To build an app, for instance here *k3remote*

```
cd $F_HOME/kinoma/kpr/applications/k3remote
kprconfig -c debug -m
```

The temporary files are in `$F_HOME/tmp/mac/debug/k3remote`, the executable bundle is `$F_HOME/bin/mac/debug/k3remote`.

To delete the temporary files and the executable bundle:

```
kprconfig -c debug -m clean
```

To build outside of the `fsk` directory, use the `-o` option, for instance:

```
kprconfig -c debug -m -o /Users/ps/dev/build
```

Xcode

Create a project based on the *OS X – Other – Empty Project* template. The *Product Name* cannot contain white spaces; I use *apps.* The project can be anywhere; I put it besides the `fsk` directory.

In the project *Build Settings*, add the following user-defined settings:

- ARCHS: `i386`
- F_HOME: the path of the `fsk` directory, for instance `/Users/ps/dev/fsk`
- SDKROOT: `macos10.7`

For every app, add a target based on the *OS X - Other - External Build System* template:

- *Product Name*: the name of the directory that contains the manifest of the app, for instance here `k3remote`.
- *Build Tool*: the path of the `kprconfig` tool, for instance `/Users/ps/dev/fsk/xs/bin/mac/kprconfig`

In the target *Build Settings*, delete the ARCHS and SDKROOT settings to inherit them from the project.

In the target *Info*:

- *Arguments*: `-c $(CONFIGURATION) -m $(ACTION)`
- *Directory*: the path of the directory that contains the manifest, for instance here `$(F_HOME)/kinoma/kpr/applications/k3remote`

To build outside of the `fsk` directory, use the `-o` option, for instance:

- *Arguments*: `-c $(CONFIGURATION) -m $(ACTION) -o /Users/ps/dev/build`

Thanks to the `$(ACTION)` argument, you can build and clean apps with the corresponding Xcode commands.

To debug, set the *Executable* in the *Run* action of the *Scheme* editing dialog (*Product - Scheme - Edit Scheme*).

8.2. Windows

The only required environment variable is `F_HOME`, the path of the `fsk` directory. For convenience, also add the path of the tools directory to the `Path` system environment variable.

Firstly, build the `xsc`, `kprconfig` and `kpr2js` tools:

```
cd %F_HOME%\xs\makefiles\Windows
kprbuild.bat
```

Command Line

To build an app, for instance here *k3remote*

```
cd %F_HOME%\kinoma\kpr\applications\k3remote
kprconfig -c debug -m
```

The temporary files are in `%F_HOME%\tmp\win\debug\k3remote`, the binary files are in `%F_HOME%\bin\win\debug\k3remote`.

To delete the temporary files and the binary files:

```
kprconfig -c debug -m clean
```

To build outside of the `fsk` directory, use the `-o` option, for instance:

```
kprconfig -c debug -m -o Z:\build
```

Visual Studio

Create a *Makefile project*. For the first app, select *Create new solution*; for the next apps, select *Add to solution*. The project name, the project location and the solution name cannot contain white spaces.

I use Visual Studio Express 2012 for Windows Desktop on Windows 8 Pro. My solution is *apps*, my location is *C:* and my projects are the names of the directories that contain the manifest of the apps, for instance here *k3remote*.

In the *Makefile Project Wizard*:

- *Build command line:* `kprconfig -c $(Configuration) -m $(F_HOME)\kinoma\kpr\applications\k3remote\manifest.xml`
- *Clean commands:* `kprconfig -c $(Configuration) -m clean $(F_HOME)\kinoma\kpr\applications\k3remote\manifest.xml`
- *Output (for debugging):* `$(F_HOME)\bin\win\debug\k3remote\k3remote.exe`

To build outside of the `fsk` directory, use the `-o` option, for instance:

- *Build command line:* `kprconfig -c $(Configuration) -m -o Z:\build $(F_HOME)\kinoma\kpr\applications\k3remote\manifest.xml`
- *Output (for debugging):* `Z:\build\bin\win\debug\k3remote\k3remote.exe`

The Debug and Release configurations are the same. The *Build Command Line, Clean Command Line* and *Output* can also be edited in the project *Property Pages - Configuration Properties - NMake*.

8.3. iOS

The iOS platform uses the same `xsc`, `kprconfig` and `kpr2js` tools as the Mac OS platform. See here above.

The iOS platform requires Xcode. Not my taste but it seems necessary to debug and release on iOS. I currently use Xcode 4.6. I tried to simplify the configuration as much as possible.

Create a project based on the *OS X – Other – Empty Project* template. The *Product Name* cannot contain white spaces; I use *iapps.* The project can be anywhere; I put it besides the `fsk` directory.

In the project *Build Settings*, add the following user-defined settings:

- F_HOME: the path of the `fsk` directory, for instance `/Users/ps/dev/fsk`
- KPRCONFIG*:* `$F_HOME/xs/bin/mac/kprconfig -p $PLATFORM_NAME -c $CONFIGURATION -b $TARGET_BUILD_DIR/$EXECUTABLE_FOLDER_PATH -t $TARGET_TEMP_DIR`

For every app, add a target based on the *IOS – Application – Empty Application* template. The *Product Name* is the name of the directory that contains the manifest of the app, for instance here `k3remote`.

After adding the first app, in the project *Build Settings*, edit the following settings:

- *IOS Deployment Target*: `IOS 5.1`
- *Other Linker Flags*: `-ObjC -L$(TARGET_TEMP_DIR) -lfsk -lresolv -lstdc++`

So all targets share such settings. Let us continue now to configure the `k3remote` target…

In the *Project Navigator*, delete the `AppDelegate.h`, `AppDelegate.m` and `app.xcdatamodelId` files in the `k3remote` folder, and delete the `main.m` file in the `Supporting Files` folder. Then add the `kpr/patches/main.m` file to the `k3remote` folder.

In the target *Build Phases*, add two *Run Script* build phases, one at the beginning after the *Target Dependencies* build phase and one at the end after the *Copy Bundle Resources* build phase.

In the first *Run Script* build phase, enter:

```
cd $F_HOME/kinoma/kpr/applications/k3remote

$KPRCONFIG

cd $TARGET_TEMP_DIR

make build
```

In the *Link Binary With Libraries* build phase, delete the frameworks if any then add the following frameworks:

- AddressBook.framework
- AssetsLibrary.framework

- AudioToolbox.framework
- AVFoundation.framework
- CoreAudio.framework
- CoreData.framework
- CoreGraphics.framework
- CoreLocation.framework
- CoreTelephony.framework
- CoreText.framework
- Foundation.framework
- OpenGLES.framework
- QuartzCore.framework
- SystemConfiguration.framework
- UIKit.framework

Xcode also adds the frameworks at the root of the *Project Navigator*! Move them to the `Frameworks` folder. For the next apps, the frameworks can be dragged and dropped in the *Link Binary With Libraries* build phase.

In the second *Run Script* build phase, enter:

```
cd $TARGET_TEMP_DIR

make copy
```

That is all! You can build, clean and debug apps with the corresponding Xcode commands.

8.4. Android

The Android platform uses the same `xsc`, `kprconfig` and `kpr2js` tools as the Mac OS platform. See here above.

Download the latest Android SDK (*ADT Bundle for Mac*) from

`http://developer.android.com/sdk/index.html`

Notice that the NDK r8 is necessary; download it from:

`http://dl.google.com/android/ndk/android-ndk-r8-darwin-x86.tar.bz2`

Besides `F_HOME`, the Android platform requires four environment variables:

- `ANDROID_SDK` = the path of the directory containing your Android SDK, for instance `/Users/asoquet/Kinoma/android/sdk`
- `NDK_DIR` = the path of the directory containing your Android NDK, for instance `/Users/asoquet/Kinoma/android/android-ndk-r8`
- `NDK_TOOLCHAIN_VERSION` = `4.4.3`
- `NDK_TOOLCHAIN_HOME` = `$NDK_DIR/toolchains/arm-linux-androideabi-$NDK_TOOLCHAIN_VERSION/prebuilt/darwin-x86`

Besides the path of the tools directory, `$F_HOME/xs/bin/mac`, the `PATH` variable has to contain three paths into your Android SDK and NDK:

- `$NDK_TOOLCHAIN_HOME/bin`

- `$ANDROID_SDK/tools`
- `$ANDROID_SDK/platform-tools`

To build an app, for instance here *k3remote*:

```
cd $F_HOME/kinoma/kpr/applications/k3remote
kprconfig -c debug -m -p android
```

The temporary files are in `$F_HOME/tmp/android/debug/k3remote`, the `apk` is `$F_HOME/tmp/android/debug/k3remote/ndk/project/bin/k3remote-debug.apk`.

The release build uses the `keystore` located in `$F_HOME/tmp/android/debug/k3remote/k3remote-release.keystore`.

Copy yours there, or `kprconfig` will launch the tool to create it.

To build outside of the `fsk` directory, use the `-o` option, for instance:

```
kprconfig -c debug -m -o /Users/asoquet/Kinoma/build -p android
```

When the build succeeds, instructions are printed to install, uninstall, start, stop and debug the application on the USB connected device.

To customize the resources used by the application, create an `android` directory in your application directory and override the icons and string.xml accordingly. See *k3remote* for an example.

8.5. Linux Aspen

To build for the Linux Aspen platform, i.e. for a K* device, you need a Linux system and cross-platform tools.

Use the distribution of your choice, I tested Debian and Ubuntu in VirtualBox. Install an HTTP server on the Linux system so the K* device can get builds.

Install the cross compiler and libraries as described in the document `gPlugD-fsk-cross-compile-setup-howto`.

Besides `F_HOME`, define two environment variables on the Linux system:

- `ARM_MARVELL_LINUX_GNUEABI` = the path to the `arm-marvell-linux-gnueabi` directory
- `FSK_SYSROOT_LIB` = the path to the `fsk_sysroot_lib.1.1` directory

Firstly, build the `xsc`, `kprconfig` and `kpr2js` tools:

```
cd $F_HOME/xs/makefiles/Linux
make kpr
```

To build an app, for instance here `k3simulator`:

```
cd $F_HOME/kinoma/kpr/applications/k3simulator
kprconfig -c debug -m -p linux/aspen
```

The temporary files are in `$F_HOME/tmp/linux/aspen/debug/k3simulator`, the binary files are in `$F_HOME/bin/linux/aspen/debug/k3simulator`, and the archive is `$F_HOME/bin/linux/aspen/k3simulator.tgz`.

To build outside of the `fsk` directory, use the -o option, for instance:

```
kprconfig -c debug -m -o /some-directory/build -p linux/aspen
```

On the K* device, get the archive from the Linux system HTTP server, expand it and move files around to please the keep alive shell script:

```
mkdir -p /kpr/debug/k3simulator/bin/linux/release
cd kpr
wget http://linux-server/debug/k3simulator.tgz -O k3simulator.tgz
tar -xzf k3simulator.tgz
mv k3simulator/* /kpr/debug/k3simulator/bin/linux/release
rm /KPL
ln -s kpr/debug/k3simulator /KPL
```

9. Tools

9.1. kprconfig

<manifest>
The manifest file. The default is `manifest.xml` in the current directory.

-p <platform>
`android`, `iphone`, `linux`, `mac` or `win`. The platform is case insensitive and supports several synonyms. The default is the platform running `kprconfig`.

-c <configuration>
`debug` or `release`. The configuration is case insensitive. The default is `release`.

-a <application>
The application. The default is the name of the directory that contains the manifest.

-o <directory>
The output directory. The directory where to create binary and temporary files is based on the output directory, the platform, the configuration and the application. The default is `F_HOME`.

-b <directory>
The directory where to create binary files. Overrides the -o option for binary files.

-t <directory>
The directory where to create temporary files. Overrides the -o option for temporary files.

-m <target>
`kprconfig` executes `make` or `nmake`. The target can be `clean` or `all`, if omitted it is `all` by default.

-v
Verbose.

-i
Instrument. Adding/removing that option recompiles all C files.

-l
Log memory usage. Adding/removing that option recompiles all C files.

-x
Create a CMake project.