Kinoma Porting Layer

Version 1.2

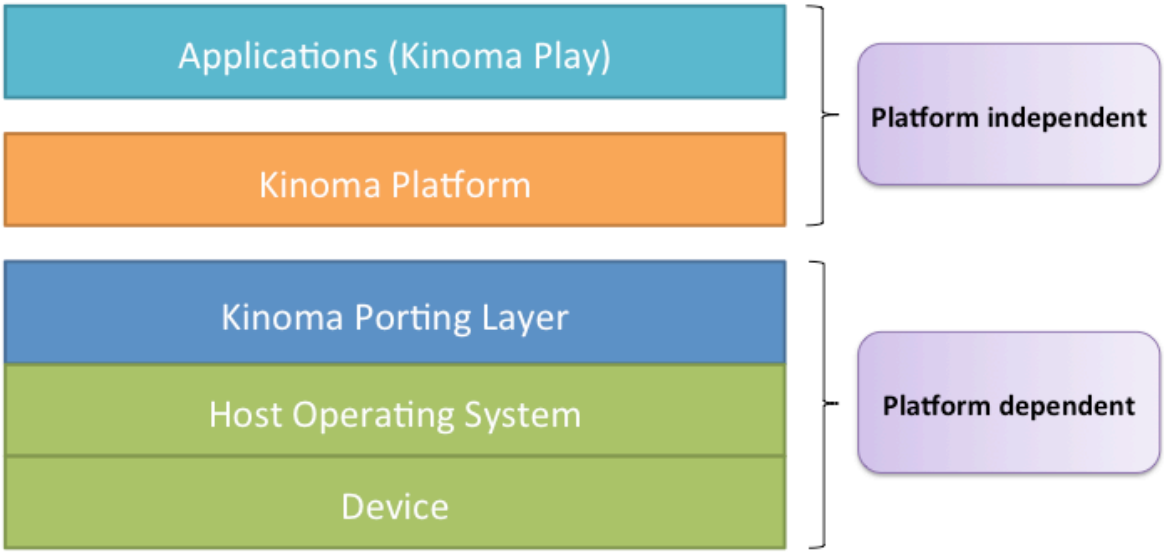
August 6, 2014

Contents

1 Introduction

As shown in Figure 1, Kinoma Play is implemented using the Kinoma Platform, a modern application framework designed for optimal performance on mobile devices, and the Kinoma Platform is in turn built on the Kinoma Porting Layer (KPL).

Figure 1. Kinoma Platform Architecture



The Kinoma Platform, Kinoma Play, and all the bundled applications (Facebook, Maps, Pictures, Box.net, and so on) are built on standard, platform-independent Web development technologies, most significantly ECMAScript and XML. KPL is a simple, low-level API that provides access to essential functionality from the host operating system, enabling the Kinoma Platform to run on a wide range of embedded and mobile operating systems (including multiple mobile Linux distributions, ThreadX, and QNX). You reimplement the KPL API functions to enable the Kinoma Platform to run on your target platform.

This document describes the KPL API along with the KPL Kit, which (for both Linux and Windows) provides a reference implementation of the KPL API plus the components you need to build the Kinoma Platform and an executable for testing your KPL implementation on your target platform.

The KPL API is built with the ANSI C programming language, so readers of this document should be familiar with ANSI C programming. Familiarity with the POSIX standards will also be helpful. In addition, you should understand how to build software on Linux or Windows.

This document is organized as follows:

- Section 2, "The KPL Kit," offers guidance for using the KPL Kit.
- Section 3, "The KPL API," describes the KPL API in detail.
- Section 4, "Glossary," is a glossary of terms defined and used in this document.

2 The KPL Kit

The KPL Kit (shortened in this document to "the kit") provides the following:

- A complete and working reference implementation of the KPL API for both Linux and Windows, providing sample code to help guide the host porting effort.
- For Linux and Windows, source code (for the Kinoma Platform and KPL), interface definition files, and platform-native build scripts, enabling you to build the Kinoma Platform and an executable file that runs on your target platform. You can run the executable (essentially an application to test your KPL implementation), set breakpoints, and trace through the code.
- A set of scripts that you can use to test the various KPL host functions that you implement.

This section describes some required one-time setup, followed by details about building and running the executable, using the `manifest.xml` file in configuring test scripts, and using `xsbug` to debug test scripts.

2.1 One-Time Setup

2.1.1 One-Time Linux Setup

On a Linux system, you must specify and set the following system environment variables:

```
export F_HOME=/home/user-name/kinoma/
export XS_HOME=${F_HOME}xs/
export XS_SDK=0
export FSK_APPLICATION=PLAY
export FSK_EMBED=1
export FSK_ZIP=0
```

The `F_HOME` and `XS_HOME` variables refer to the location where the source tree is installed. In the example above, it is installed in the `kinoma` directory located in the user's home directory.

Note: In this document, entities shown in italics in code, such as *`user-name`* in the code above, represent values that vary; enter whatever is appropriate in your case.

To use the `xsbug` script debugger, the Linux host must create a `debug.txt` configuration file in the `~/.kinoma` directory. The configuration file contains the IP address of the PC host running `xsbug`. Create the `debug.txt` file with the following contents:

```
your-xsbug-host-pc-ip-address:5002
1
```

2.1.2 One-Time Windows and Visual Studio Setup

On a Windows system, you must specify and set the following system environment variables:

```
F_HOME=c:\kinoma
XS_HOME=c:\kinoma\xs
XS_SDK=0
FSK_APPLICATION=PLAY
FSK_EMBED=1
FSK_ZIP=0
```

The F_HOME and XS_HOME variables refer to the location where the source tree is installed. In the preceding example, it is installed in a directory named kinoma at the root of the c: drive.

The kit is designed to be compatible with Visual Studio 2008 but you can also use more recent versions. The fsk.sln solution can be found in the build\windows directory; open that solution in Visual Studio. (If you are using a version of Visual Studio newer than 2008, Visual Studio will prompt you to update the projects; this is expected.) Then, in the Solution Explorer pane, right-click the fsk project and select **Set as Startup Project** from the context menu. Save and exit the solution.

2.2 Building and Running the Executable

2.2.1 Building and Running on Linux

The code of the Linux KPL port can be found in build/linux/kpl. The port is further divided into two distinct hardware variations:

- The generic variation provides no hardware-specific features.
- The i386-alsa-directfb port provides KPL audio support via ALSA audio and KPL screen support via DirectFB on i386 platforms.

The specific port files can be found in the build/linux/hw/generic and build/linux/hw/i386-alsa-directfb directories, respectively. Each directory includes FskPlatform.Kpl.h and linux.make files, which configure the specific hardware characteristics.

The port is selected at build time using the HARDWARE environment variable, which defaults to generic. To select the other hardware variation, define the HARDWARE variable as follows before building the application:

```
export HARDWARE=i386-alsa-directfb
```

You can build the application (debug version) from the command line as follows:

```
cd ${F_HOME}build/linux
make all
```

Then, to execute the built application, do this:

```
cd ${F_HOME}bin/linux/debug
./Kpl
```

2.2.2 Building and Running with Visual Studio

The fsk.sln code is built with the Debug-KPL solution configuration for the Win32 platform. Make sure that **Debug-KPL** and **Win32 Platform** are selected in their associated pop-up menus, which are located in the Visual Studio toolbar.

To build the code, right-click the fsk project in the Solution Explorer and select **Build**. The solution should build each of the subprojects. The final part of the build log should look like this:

```
Build log was saved at "file://c:\kinoma\tmp\win32\fsk\debug\BuildLog.htm"
fsk - 0 error(s), 1 warning(s)
========== Build: 6 succeeded, 0 failed, 0 up-to-date, 1 skipped ==========
```

To run the built code, select **Start Debugging** from the **Debug** menu. The first time, Visual Studio will prompt for the executable file to be used for the debug session; select **Browse** from the pull-down menu, navigate to the bin\win32\debug directory, and select the kinoma.exe file.

The code will launch the executable, which in turn loads and runs the test script. The default test script displays a "frame buffer" window filled with red, and tracks mouse clicks in the window by drawing a smiley-face icon at each mouse location clicked.

You can use the Visual Studio debugger to set breakpoints, trace through code, and inspect runtime data.

2.3 Additional Host Requirements

The kit provides reference source code for the Linux and Win32 implementations of the required host KPL functions (in the `build\linux\kpl` and `build\windows\kpl` directories, respectively) and platform-independent KPL header files (in the `core\kpl` directory). In addition to reimplementing the required host KPL functions for the target platform, the KPL host must edit the following header and source code files to be compatible with the host platform:

- `Kpl.h`, which provides portable type definitions and `#define` values corresponding to platform data structure alignment techniques, CPU characteristics, and the platform endian format.
- `FskPlatform.Kpl.h`, which defines display pixel formats supported by the platform and other definitions relating to text rendering and build options. To reduce the platform memory footprint size, you should define only pixel formats supported by your platform.
- `xs_kpl.h`, which defines the platform endian format for the ECMAScript runtime and is included by the ECMAScript-based build tools.
- `xs_fsk_kpl.h`, which defines platform-specific data structures, functions, and values required by the ECMAScript runtime.
- `xs_fsk_kpl.c`, which defines implementation code and data for the corresponding header file.

2.4 Configuring Test Scripts with manifest.xml

Portions of the build and runtime are configured by the `manifest.xml` file located in the `kinoma\kpl` directory. The `manifest.xml` file specifies characteristics of the root virtual machine, including additional extensions built and loaded at runtime. The `manifest.xml` file also specifies a second `kpl` virtual machine, which hosts the test scripts. For example:

```
<vm name="kpl">
   <environment>
      <variable name="homePath" value="[kinomaSrc]kpl/"/>
      <variable name="mediaPath" value="[homePath]media/"/>
   </environment>
   <script href="[kinomaSrc]kpl/hello.js"/>
  .<!--
...<script href="[kinomaSrc]kpl/audio.js"/>
   <script href="[kinomaSrc]kpl/files.js"/>
   <script href="[kinomaSrc]kpl/notifiers.js"/>
   <script href="[kinomaSrc]kpl/device.js"/>
   <script href="[kinomaSrc]kpl/timers.js"/>
   <script href="[kinomaSrc]kpl/window.js"/>
    -->
</vm>
```

This `vm` element from the provided `manifest.xml` file causes the `hello.js` test script to be loaded at runtime. The other test scripts are commented out; to run one of them,

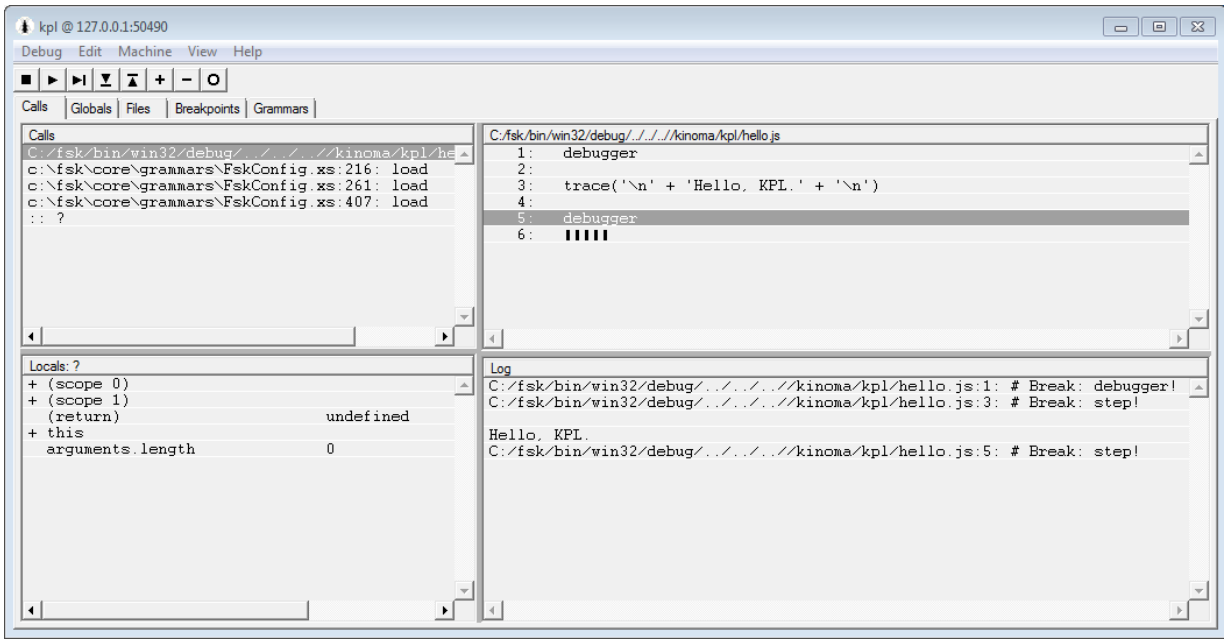
simply “uncomment” that test script, rebuild the port code, and run. The scripts are simple JavaScript code and are located in the same directory as the `manifest.xml` file.

2.5 Debugging Scripts with xsbug

You can use the `xsbug` debugger to trace through and debug test scripts and other ECMAScript code.

The `xsbug` debugger is built for the Windows desktop as part of the Visual Studio kit build and can be found in the `xs\bin\win32` directory. The debugger supports inspecting variables, tracing through code, and setting breakpoints (see Figure 2).

Figure 2. The `xsbug` debugger in Visual Studio



To use `xsbug`, simply launch `xsbug.exe` before launching the native application. After `kinome.exe` launches, the `xsbug` application will connect to the virtual machine (or machines) and stop at `debugger` statements in the test script source. The `xsbug` window in Figure 2 corresponds to a run of the `hello.js` test script. Use the menus or toolbar buttons to control the debugger.

You can also use `xsbug` to debug ECMAScript code built on Linux, once you complete the one-time Linux setup described in Section 2.1.1. The `xsbug` debugger uses a Wi-Fi connection to debug code running on mobile or embedded Linux platforms.

3 The KPL API

This section provides details on the KPL functions you must implement, or optionally may implement, to run the Kinoma Platform on a host operating system. The functions, which are divided into modules (such as memory management and threads modules), are defined using the ANSI C standard. C++ is not used in the interface definition; however, C++ may be used in the implementation of the KPL functions, if necessary or convenient for interacting with the host operating system.

If KPL functions that are optional are not provided, there may be some implications. The two possibilities when a function is unavailable in KPL are as follows:

- The missing function may be provided instead by the Kinoma Platform. In this case, the size of the code deployed will be increased. In addition, the version of the function in KPL may not be as efficient as that provided by the host operating system, resulting in some loss of performance.
- The feature corresponding to the missing function may be unavailable in Kinoma Play. For example, if KPL cannot report the battery level, Kinoma Play cannot display the battery level.

Note: Because POSIX is a well-known, publicly available API, this section refers to functions from POSIX to help define expected behavior.

The Kinoma Platform makes use of the following functions that are part of the ANSI C standard runtime libraries, so the KPL host must also provide these functions:

- Memory: `memmove`, `memcpy`, `memset`
- Print: `sprintf`, `snprintf`
- Random: `rand`, `srand`
- Sorting: `bsearch`, `qsort`
- Strings: `strlen`, `strcpy`, `strncpy`, `strcmp`, `strncmp`, `strcasecmp`, `strncasecmp`, `strchr`, `strstr`, `atoi`, `strtol`, `strtoul`, `strtod`, `strftime`
- Time: `gmtime`, `localtime`, `mktime`, `time`
- Variable arguments: `va_arg`, `va_start`, `va_end`

3.1 KPL Conventions

All functions and data types defined in KPL have a `Kpl` prefix, and all constants have a `kKpl` prefix. These prefixes are used to avoid naming collisions and to make it easy to identify interactions with KPL within source code.

Unless otherwise noted in this section, the objects created by KPL are opaque (that is, private) data structures. These objects will be operated on only by calls to KPL; the Kinoma Platform will never inspect or modify the objects directly.

All strings used by KPL, including file names, are encoded using the UTF-8 standard.

KPL error codes indicate the reason a function failed. The error codes use the `FskErr` enumerated type. Error codes are globally defined. An error code of `kFskErrNone` indicates success.

Functions in KPL are expected to be thread-safe unless otherwise noted. This means, for example, that two threads should be able to safely make requests to allocate memory at the same time.

3.2 Summary of KPL API

Table 1 summarizes the functions in the KPL API, categorizing them according to the subsection in which they are described in detail. Notice that the function prefix is shown separately from the rest of the function name; for example, the full name of the first memory management function is `KplMemPtrNew`.

Table 1. Summary of KPL API

Subsection	Function Name Prefix…	…Rest of Function Name
3.3 Memory Management	`KplMemPtr`	`New`, `Dispose` `Realloc`, `NewClear`
3.4 Files and Directories	`KplFile`	`Initialize`, `Terminate`, `Open`, `Close`, `Read`, `Write`, `Flush`, `GetFileInfo`, `SetFileInfo`, `GetPosition`, `SetPosition`, `GetSize`, `SetSize`, `Create`, `CreateDirectory`, `Delete`, `DeleteDirectory`, `Rename`, `RenameDirectory`, `PathToNative` *(Optional)* `ResolveLink`, `GetThumbnail`, `Map`, `DisposeMap`
	`KplDirectory`	*(Optional)* `GetSpecialPath`
	`KplDirectoryIterator`	`New`, `Dispose`, `GetNext`
	`KplDirectoryChangeNotifier`	*(Optional)* `New`, `Dispose`
3.5 Volumes	`KplVolumeIterator`	`New`, `Dispose`, `GetNext`
	`KplVolume`	`GetID`, `GetInfo`, `GetInfoFromPath`
	`KplVolumeNotifier`	*(Optional)* `New`, `Dispose`
3.6 Threads	`KplThread`	`Create`, `Join`, `GetCurrent`, `Yield`, `Wake`, `CreateMain`, `TerminateMain`, `PostEvent`, `RunloopCycle`, `PostCallback`, `NotifyClientComplete` *(Optional)* `NotifyPendingSocketData`
3.7 Synchronization	`KplMutex`	`New`, `Dispose`, `Acquire`, `Release`
	`KplSemaphore`	`New`, `Dispose`, `Acquire`, `Release`
	`KplCondition`	*(Optional)* `New`, `Dispose`, `Wait`, `Signal`

Subsection	Function Name Prefix...	...Rest of Function Name
3.8 Time and Timers	KplTime	Initialize, Terminate, GetNow, MakeDate, CallbackNew, CallbackDispose, CallbackSet, CallbackCancel *(Optional)* Strftime, Strptime, Gmtime, LocalTime, Mktime, GetZone, GetOSTime
3.9 Sockets	KplSocket	Initialize, Terminate, NewUDP, NewTCP, Close, MakeNonblocking, SetOption, Bind, IsReadable, IsWritable, RecvTCP, SendTCP, RecvUDP, SendUDP, Connect, Listen, AcceptConnection, GetLocalAddress, GetRemoteAddress
	KplSocketMulticast	*(Optional)* Join, SetOutgoingInterface
	FskKplSocket	*(Optional)* Join, SetOutgoingInterface
3.10 Network	KplNet	HostnameResolve, ServerSelection
3.11 Network Interfaces	KplNetInterface	*(Optional)* Initialize, Terminate, Enumerate, SetChangedCallback
3.12 Screen	KplScreen	GetBitmap, DisposeBitmap *(Optional)* LockBitmap, UnlockBitmap, DrawBitmap, GetAuxInfo
3.13 UI Events	KplUIEvent	SetCallback *(Optional)* New, Dispose, Send
3.14 Device	KplDevice	GetProperty, SetProperty, NotifyPropertyChangedNew, NotifyPropertyChangedDispose
3.15 Audio Output	KplAudio	*(Optional)* New, Dispose, SetFormat, GetFormat, Write, Start, Stop, GetProperty, SetProperty, SetDoneCallback, SetMoreCallback, GetSamplesQueued
3.16 OpenGL	KplGL	nitialize, Terminate

Subsection	Function Name Prefix...	...Rest of Function Name
3.17 Utilities	`KplUtilities`	`Initialize`, `Terminate`, `HardwareInitialize`, `HardwareTerminate`, `RandomSeedInit`, `GetApplicationPath`, `Delay`
3.18 Main	`KplMain`	
3.19 Environment	`KplEnvironment`	`Initialize`
3.20 Extensions	`KplLibrary`	*(Optional)* `Load`, `Unload`, `GetSymbolAddress`
3.21 Miscellaneous	`KplBrowser`	*(Optional)* `OpenURI`
	`KplECMAScript`	`GetPlatform` *(Optional)* `GetExtension`

3.3 Memory Management

The KPL memory allocators must return memory properly aligned for storage of data structures on the host processor. On ARM CPUs, for example, memory must be aligned on 4-byte boundaries.

The KPL memory allocators should be called only by the Kinoma Platform and native-code Kinoma Platform extensions.

```
typedef unsigned char *KplMemPtr;
```

```
FskErr KplMemPtrNew(UInt32 size, KplMemPtr *newMemory);
```

Allocates a memory block from the application heap, returning a pointer to the new memory block; equivalent to `malloc` in POSIX. If memory is not available, an error of `kFskErrMemFull` is returned.

```
FskErr KplMemPtrDispose(void *ptr);
```

Returns a memory block to the application heap; similar to `free` in POSIX.

```
FskErr KplMemPtrRealloc(UInt32 size, KplMemPtr *newMemory);
```

Resizes a memory block allocated with `KplMemPtrNew` or with `KplMemPtrRealloc` itself; equivalent to `realloc` in POSIX. The new memory block may have a different address.

```
FskErr KplMemPtrNewClear(UInt32 size, KplMemPtr *newMemory);
```

Same as `KplMemPtrNew` but also clears the memory block to 0; equivalent to `calloc` in POSIX.

3.4 Files and Directories

For fundamental operations on files and directories, KPL uses functions based on the well-known `stdio` library from ANSI C. If the host operating system or runtime provides the `stdio` library, the Kinoma Platform can use that library directly; if not, it uses the KPL functions described here instead.

All file paths in these functions are absolute paths; relative paths are never used. The path separator is always a slash (`/`). A path that refers to a directory always ends with a slash.

3.4.1 Files

```
FskErr KplFileInitialize(void);
```

Initializes the module; called once at startup.

```
FskErr KplFileTerminate(void);
```

Terminates the module; called once at shutdown.

```
FskErr KplFileOpen(const char *fullPath, UInt32 permissions, KplFile
   *fref);
```

Opens the file specified by `fullPath` with the indicated permissions and returns a reference to the file in `fref`.

Values for `permissions`:

```
enum {
    kKplFilePermissionReadOnly = 0,
    kKplFilePermissionReadWrite = 1 << 0,
}
```

```
FskErr KplFileClose(KplFile fref);
```

Closes the specified file and releases all associated resources.

```
FskErr KplFileRead(KplFile fref, UInt32 bytesToRead, void *buffer, UInt32
   *bytesRead);
```

Reads up to `bytesToRead` bytes into `buffer`. The actual number of bytes read is returned in `bytesRead`; it should be equal to `bytesToRead` unless attempting to read past the end of the file.

```
FskErr KplFileWrite(KplFile fref, UInt32 bytesToWrite, const void *buffer,
   UInt32 *bytesWritten);
```

Writes up to `bytesToWrite` bytes pointed to by `buffer` to the file. The actual number of bytes written is returned in `bytesWritten`; it should be equal to `bytesToWrite` unless there is no more storage space on the volume.

```
FskErr KplFileFlush(KplFile fref);
```

Immediately writes out any unwritten data in the file's I/O buffers.

```
FskErr KplFileGetFileInfo(const char *fullpath, KplFileInfo itemInfo);
```

Returns information about the file specified by `fullpath`.

`KplFileInfoRecord` structure:

```
typedef struct KplFileInfoRecord {
    KplInt64   filesize;
    UInt32     filetype;
    UInt32     fileCreationDate;
    UInt32     fileModificationDate;
    UInt32     flags;
} KplFileInfoRecord, *KplFileInfo;
```

The `fileCreationDate` and `fileModificationDate` fields are UTC time values, with the UNIX epoch. Values for the `filetype` and `flags` fields are taken from the following enumerations (respectively):

```
enum {
    kKplDirectoryItemIsFile = 1,
    kKplDirectoryItemIsDirectory = 2,
    kKplDirectoryItemIsLink = 3
};

enum {
    kKplFileFlagFileLocked = 1L << 1,
    kKplFileFlagFileHidden = 1L << 2
};
```

```
FskErr KplFileSetFileInfo(const char *fullpath, const KplFileInfo
   itemInfo);
```

Modifies file information of the file specified by `fullpath`.

```
FskErr KplFileGetPosition(KplFile fref, KplInt64 *position);
```

Returns the current read/write position of the file.

```
FskErr KplFileSetPosition(KplFile fref, const KplInt64 *position);
```

Sets the read/write position for the file.

```
FskErr KplFileGetSize(KplFile fref, KplInt64 *size);
```

Returns the current size of the file.

```
FskErr KplFileSetSize(KplFile fref, const KplInt64 *size);
```

Sets the current size of the file.

```
FskErr KplFileCreate(const char *fullPath);
```

Creates a file at the specified path.

```
FskErr KplFileCreateDirectory(const char *fullPath);
```

Creates a directory at the specified path.

```
FskErr KplFileDelete(const char *fullPath);
```

Deletes the file at the specified path.

```
FskErr KplFileDeleteDirectory(const char *fullPath);
```

Deletes the directory at the specified path.

```
FskErr KplFileRename(const char *fullPath, const char *newName);
```

Renames the file specified by `fullPath` to `newName`. This function is never used to move a file between directories or volumes.

```
FskErr KplFileRenameDirectory(const char *fullPath, const char *newName);
```

Renames the directory specified by `fullPath` to `newName`. This function is never used to move a directory between directories or volumes.

```
FskErr KplFilePathToNative(const char *kplPath, char **nativePath);
```

Converts the path specified by `kplPath` to the host operating system's path format. `*nativePath` will be disposed of using `KplMemPtrDispose`.

```
FskErr KplFileResolveLink(const char *linkPath, char **resolvedPath);
```

(Optional) Resolves the link specified by `linkPath` and returns the result in `resolvedPath`. `*resolvedPath` will be disposed of using `KplMemPtrDispose`. This function should be called only on files of type `kKplDirectoryItemIsLink`.

```
FskErr KplFileGetThumbnail(const char *fullPath, UInt32 width, UInt32
   height, KplBitmap *thumbnail);
```

(Optional) Returns the thumbnail bitmap image associated with the file specified by `fullPath`. The `width` and `height` arguments specify the preferred dimensions of the thumbnail; the bitmap returned may be a different size. The `KplBitmapRecord` structure is shown in Section 3.12, "Screen."

```
FskErr KplFileMap(const char *fullPath, unsigned char **data, KplInt64
   *dataSize, KplFileMapping *map);
```

(Optional) Maps the entire file specified by `fullPath` into memory; returns the size, a pointer to the memory, and a file-mapping instance. The contents of the file may be access starting at address `*data` and ending at address `(*data + *dataSize)`.

```
FskErr KplFileDisposeMap(KplFileMapping map);
```

(Optional) Unmaps the specified file mapping and releases all associated resources. After this function is called, the `*data` address returned by `KplFileMap` for this instance will no longer reference the contents of the file.

```
FskErr KplDirectoryGetSpecialPath(UInt32 type, const Boolean create, const
   char *volumeName, char **fullPath);
```

(Optional) Provides access to predefined directories on the host operating system. Returns the full path to the specified directory; optionally creates the directory if it does not already exist. If `volumeName` is non-`NULL`, the special directory must be on the specified volume. `*fullPath` will be disposed of by the caller.

Values for `type`:

```
enum {
   kKplDirectorySpecialTypeDocument,
   kKplDirectorySpecialTypePhoto,
   kKplDirectorySpecialTypeMusic,
   kKplDirectorySpecialTypeVideo,
   kKplDirectorySpecialTypeTV,
   kKplDirectorySpecialTypeApplicationPreference,
   kKplDirectorySpecialTypeApplicationPreferenceRoot,
   kKplDirectorySpecialTypeTemporary,

   kKplDirectorySpecialTypeSharedFlag = 0x80000000
};
```

`kKplDirectorySpecialTypeSharedFlag` is a mask that can be OR'ed with `type` to indicate that the requested directory is shared across all users.

3.4.2 Directory Iterator

The directory iterator functions enable the Kinoma Platform to enumerate all the files, directories, and links contained with a directory.

```
FskErr KplDirectoryIteratorNew(const char *directoryPath,
   KplDirectoryIterator *dirIt);
```

Creates a directory iterator instance for the directory specified by `directoryPath`.

```
FskErr KplDirectoryIteratorDispose(KplDirectoryIterator dirIt);
```

Releases all resources associated with the specified directory iterator.

```
FskErr KplDirectoryIteratorGetNext(KplDirectoryIterator dirIt, char **name,
   UInt32 *itemType);
```

Returns the name and type of the next item in the directory, or `kFskErrIteratorComplete` if no items remain. The Kinoma Platform may pass `NULL` for `name` or `itemType` if that information is not needed. `*name` will be disposed of by the caller.

3.4.3 Directory Change Notifier

These optional functions enable the Kinoma Platform to monitor a directory for changes.

```
FskErr KplDirectoryChangeNotifierNew(const char *path, UInt32 flags,
   KplDirectoryChangeNotifierCallbackProc callback, void *refCon,
   KplDirectoryChangeNotifier *dirChangeNtf);
```

(Optional) Creates a directory change notifier instance for the directory specified by `path`. The callback function will be invoked whenever a change occurs in the specified directory. The changes monitored include a new file or directory being created, a file or directory being deleted, and a file or directory being changed.

The callback function is invoked in the thread in which the associated directory change notifier was created.

Callback function prototype:

```
typedef FskErr (*KplDirectoryChangeNotifierCallbackProc)(UInt32
whatChanged, const char *path, void *refCon);
```

Values for `whatChanged` argument passed to callback function:

```
enum {
   kKplDirectoryChangeFileUnknown = 0,
   kKplDirectoryChangeFileCreated = 1,
   kKplDirectoryChangeFileDeleted = 2,
   kKplDirectoryChangeFileChanged = 3
};
```

```
FskErr KplDirectoryChangeNotifierDispose(KplDirectoryChangeNotifier
   dirChangeNtf);
```

(Optional) Cancels the specified directory change notifier and releases all associated resources.

3.5 Volumes

The functions for performing operations on volumes (disks, memory cards, and so on) on the device fall into three categories: volume iterator functions, volume information functions, and volume notifier functions.

3.5.1 Volume Iterator

The volume iterator functions enable the Kinoma Platform to discover all the volumes on the device.

```
FskErr KplVolumeIteratorNew(KplVolumeIterator *volIt);
```

Creates a volume iterator instance.

```
FskErr KplVolumeIteratorDispose(KplVolumeIterator volIt);
```

Releases all resources associated with the specified volume iterator.

```
FskErr KplVolumeIteratorGetNext(KplVolumeIterator volIt, UInt32 *id, char
   **path, char **name);
```

Returns the ID, path, and name of the next volume, or `kFskErrIteratorComplete` if no items remain. The Kinoma Platform may pass `NULL` for `id` or `path` if that information is not needed. `*path` and `*name` will be disposed of by the caller.

3.5.2 Volume Information

```
FskErr KplVolumeGetID(const char *fullPath, UInt32 *volumeID);
```

Returns the ID of the volume specified by `fullPath`.

```
FskErr KplVolumeGetInfo(UInt32 volumeID, char **path, char **name, UInt32
   *volumeType, Boolean *isRemovable, KplInt64 *capacity, KplInt64
   *freeSpace);
```

Returns detailed information for the volume specified by `volumeID`.

Values returned for `volumeType`:

```
enum {
   kKplVolumeTypeNone,
   kKplVolumeTypeUnknown,
   kKplVolumeTypeFixed,
   kKplVolumeTypeFD,
```

```
            kKplVolumeTypeOptical,
            kKplVolumeTypeCD,
            kKplVolumeTypeDVD,
            kKplVolumeTypeNetwork,
            kKplVolumeTypeMemoryStick,
            kKplVolumeTypeMMC,
            kKplVolumeTypeSDMemory,
            kKplVolumeTypeCompactFlash,
            kKplVolumeTypeSmartMedia,
            kKplVolumeTypeDirectory
         };
FskErr KplVolumeGetInfoFromPath(const char *path, char **pathOut, char
   **name, UInt32 *volumeType, Boolean *isRemovable, KplInt64 *capacity,
   KplInt64 *freeSpace);
```

Returns detailed information for the volume specified by `path`.

3.5.3 Volume Notifier

These optional functions enable the Kinoma Platform to monitor added or removed volumes.

```
FskErr KplVolumeNotifierNew(KplVolumeNotifierCallbackProc callback, void
   *refCon, KplVolumeNotifier *volNtf);
```

(Optional) Creates a volume notifier. The callback function will be invoked whenever a volume is added or removed. It is invoked in the thread in which the associated volume notifier was created. Note that the Kinoma Platform may create multiple volume notifiers.

Callback function prototype:

```
typedef FskErr (*KplVolumeNotifierCallbackProc)(UInt32
whatChanged, UInt32 volumeID, void *refCon);
```

Values for `whatChanged` argument passed to callback function:

```
enum {
   kKplVolumeAdded = 1,
   kKplVolumeRemoved = 2
};
```

```
FskErr KplVolumeNotifierDispose(KplVolumeNotifier volNtf);
```

(Optional) Cancels the specified volume notifier and releases all associated resources.

3.6 Threads

```
FskErr KplThreadCreate(KplThread *thread, KplThreadProc procedure, void
   *refcon, UInt32 flags);
```

Creates a thread, returning a reference to the thread in `*thread`; equivalent to `pthread_create` in POSIX. After creating the thread, `KplThreadCreate` begins running the thread function indicated by the `procedure` parameter.

Thread function prototype:

```
typedef void (*KplThreadProc)(void *refcon);
```

Values for `flags` parameter to `KplThreadCreate`:

```
enum {
   kFskThreadFlagsDefault         = 0x00000000,

   kFskThreadFlagsTransientWorker = 0x00000001,
   kFskThreadFlagsJoinable        = 0x00000002,
   kFskThreadFlagsIsMain          = 0x00000004,
```

```
            kFskThreadFlagsWaitForInit     = 0x00000008,
            kFskThreadFlagsHighPriority    = 0x10000000,
            kFskThreadFlagsLowPriority     = 0x20000000
         };
```

```
FskErr KplThreadJoin(KplThread thread);
```

Blocks the calling thread until the indicated thread terminates; equivalent to `pthread_join` in POSIX.

```
KplThread KplThreadGetCurrent(void);
```

Returns the `KplThread` instance corresponding to the currently running thread; may be implemented using `pthread_getspecific` in POSIX.

```
void KplThreadYield(void);
```

Enables the current thread to give up the remainder of the current time slice for another thread to run; equivalent to `sched_yield` in POSIX.

```
void KplThreadWake(KplThread thread);
```

Requests that the indicated thread run immediately.

```
FskErr KplThreadCreateMain(KplThread *thread);
```

Creates a thread wrapper for the default/main thread; called once at startup. The main thread corresponds to the host thread that owns the application entry point.

```
FskErr KplThreadTerminateMain(void);
```

Terminates the thread wrapper for the default/main thread; called once at shutdown.

```
FskErr KplThreadPostEvent(KplThread thread, void *event);
```

Posts an event to the specified thread.

```
FskErr KplThreadRunloopCycle(SInt32 msTimeout);
```

Services the host event loop, providing time to the host operating system to perform event processing and housekeeping; typically called in a tight loop from the KPL host's main function.

```
FskErr FskKplThreadPostCallback(KplThread thread, KplThreadCallback
   function, void *arg1, void *arg2, void *arg3, void *arg4);
```

Requests that the Kinoma Platform issue the specified callback function within the context of the specified thread.

Callback function prototype:

```
typedef void (*KplThreadCallback)(void *arg1, void *arg2, void
*arg3, void *arg4);
```

```
FskErr KplThreadNotifyClientComplete(KplThread thread);
```

Enables the KPL host to perform cleanup housekeeping on the thread; called when the specified thread's procedure has completed.

```
FskErr KplThreadNotifyPendingSocketData(KplThread thread, Boolean
   pendingReadable, Boolean pendingWritable);
```

(Optional) Notifies the KPL host that a socket created by the specified thread has pending I/O. Implementation typically is not required by POSIX-compliant hosts.

3.7 Synchronization

KPL has two required synchronization objects: mutexes and semaphores. In addition, there is one optional synchronization object, conditions.

3.7.1 Mutexes

```
FskErr KplMutexNew(KplMutex *mutex);
```

Creates a mutual exclusion variable; equivalent to `pthread_mutex_init` in POSIX.

Mutexes in KPL are recursive. Therefore, if a mutex is acquired by the same thread, its use count is increased. The mutex must be released the same number of times as it was acquired to be available to another thread.

```
FskErr KplMutexDispose(KplMutex mutex);
```

Releases all resources associated with the indicated mutex; equivalent to `pthread_mutex_destroy` in POSIX.

```
FskErr KplMutexAcquire(KplMutex mutex);
```

Attempts to lock the indicated mutex; equivalent to `pthread_mutex_lock` in POSIX. If the mutex is already locked by another thread, the calling thread blocks until the mutex is released by the owning thread.

```
FskErr KplMutexRelease(KplMutex mutex);
```

Unlocks the indicated mutex; equivalent to `pthread_mutex_unlock`.

3.7.2 Semaphores

```
FskErr KplSemaphoreNew(KplSemaphore *sem, UInt32 value);
```

Creates a semaphore instance with the initial value indicated by the `value` argument; equivalent to `sem_init` in POSIX.

```
FskErr KplSemaphoreDispose(KplSemaphore sem);
```

Releases all resource associated with the indicated semaphore; equivalent to `sem_destroy` in POSIX.

```
FskErr KplSemaphoreAcquire(KplSemaphore sem);
```

Decrements the semaphore value; equivalent to `sem_wait` in POSIX. If the semaphore currently has a value of 0, the calling thread blocks until the value is increased by another thread.

```
FskErr KplSemaphoreRelease(KplSemaphore sem);
```

Increments the semaphore value; equivalent to `sem_post` in POSIX. If the semaphore changes from 0 to positive, a thread waiting to acquire the semaphore will be awakened.

3.7.3 Conditions

If these optional functions are not provided, they will be emulated by the Kinoma Platform using the mutex and semaphore functions.

```
FskErr KplConditionNew(KplCondition *condition);
```

(Optional) Creates a condition instance; equivalent to `pthread_cond_init` in POSIX.

```
FskErr KplConditionDispose(KplCondition condition);
```

(Optional) Releases all resource associated with the indicated condition; equivalent to `pthread_cond_destroy` in POSIX.

```
FskErr KplConditionWait(KplCondition condition, KplMutex mutex);
```

(Optional) Releases the mutex specified by the `mutex` argument and blocks the calling thread until the condition is signaled; equivalent to `pthread_cond_wait` in

POSIX. When the condition is signaled, `KplConditionWait` returns and the specified mutex will be locked.

```
FskErr KplConditionSignal(KplCondition condition);
```

(Optional) Unblocks at least one thread that is waiting for this condition; equivalent to `ptread_cond_signal` in POSIX.

3.8 Time and Timers

```
typedef struct {
   SInt32   seconds;
   UInt32   useconds;
 } KplTimeRecord, *KplTime;
```

The `KplTimeRecord` structure contains a time value, in seconds since January 1, 1970. The `useconds` field indicates the fraction of a second in microsecond resolution. The actual resolution of the `useconds` field depends on the timing accuracy of the host operating system. A resolution of no less than milliseconds is recommended.

```
FskErr KplTimeInitialize(void);
```

Initializes the module; called once at startup.

```
void KplTimeTerminate(void);
```

Terminates the module; called once at shutdown.

```
void KplTimeGetNow(KplTime t);
```

Returns the current time.

```
void KplTimeMakeDate(char *dateString, int dateStringSize);
```

Returns a formatted GMT time string; equivalent to `strftime` using `gmtime` in POSIX.

```
void KplTimeCallbackNew(KplTimeCallback *callback);
```

Creates a time callback instance.

```
void KplTimeCallbackDispose(KplTimeCallback callback);
```

Releases all resources associated with the specified time callback instance.

```
void KplTimeCallbackSet(KplTimeCallback callback, const KplTime when,
   KplTimeCallbackProc callbackProc, void *param);
```

Registers a callback function to be called at the time indicated by the `when` argument. The callback function must be called in the same thread in which `KplTimeCallbackSet` was invoked.

Callback function prototype:

```
typedef void (*KplTimeCallback)(KplTimeCallBack callback, const
KplTime time, void *param);
```

```
void KplTimeCallbackCancel(KplTimeCallback callback);
```

Unregisters the function associated with the specified `callback` instance.

```
UInt32 KplTimeStrftime(char *s, UInt32 max, const char *format, const
   KplTimeElements kpltm);
```

(Optional) Formats the time `kpltm` according to the format specification `format`, returning the result in `s`; equivalent to calling `strftime` using `struct tm*` in POSIX.

```
char *KplTimeStrptime(const char *s, const char *format, KplTimeElements
   kpltm);
```

(Optional) Formats the time `kpltm` according to the format specification `format` and returns the result in the character array `s` of size `max`; equivalent to calling `strfpime` in POSIX.

```
FskErr KplTimeGmtime(const KplTime t, KplTimeElements kpltm);
```

(Optional) Converts the calendar time `t` into the broken-down time structure `kpltm`; equivalent to calling `gmtime_r` in POSIX.

```
FskErr KplTimeLocaltime(const KplTime t, KplTimeElements kpltm);
```

(Optional) Converts the local time `t` into the broken-down time structure `kpltm`; equivalent to calling `localtime_r` in POSIX.

```
void KplTimeMktime(const KplTimeElements kpltm, KplTime t);
```

(Optional) Converts the broken-down time structure `kpltm`, expressed as local time, to calendar time `t`; equivalent to calling `mktime` in POSIX.

```
FskErr KplTimeGetZone(const KplTime t, SInt32 *tzOffset, SInt32 *dest,
   const char **tzName);
```

(Optional) Returns the time zone offset `tzOffset`, daylight savings time status `dst` and time zone name `tzName` corresponding to the provided calendar time `t`; equivalent to calling `localtime_r` in POSIX.

```
void KplTimeGetOSTime(KplTime t);
```

(Optional) Returns the number of seconds and microseconds since the Epoch; equivalent to calling `gettimeofday` in POSIX.

3.9 Sockets

For network access, KPL uses functions based on the well-known Berkeley Sockets API. If the host operating system implements the Berkeley Sockets API, the Kinoma Platform can use that API directly; if not, it uses the KPL socket functions instead.

```
FskErr KplSocketInitialize(void);
```

Initializes the module; called once at startup.

```
FskErr KplSocketTerminate(void);
```

Terminates the module; called once at shutdown.

```
FskErr KplSocketNewUDP(KplSocket *newSocket);
```

Creates a socket to use with UDP transport.

```
FskErr KplSocketNewTCP(KplSocket *newSocket, Boolean listener);
```

Creates a socket to use with TCP transport.

```
FskErr KplSocketClose(KplSocket skt);
```

Releases all the resources associated with the specified socket.

```
FskErr KplSocketMakeNonblocking(KplSocket skt);
```

Makes the specified socket nonblocking. By default, all sockets are blocking.

```
FskErr KplSocketSetOption(KplSocket skt, int sktLevel, int sktOption, int
   val);
```

Equivalent to `setsockopt` in POSIX. KPL uses `SO_REUSEADDR`, `SO_DONTROUTE`, `SO_BROADCAST`, `SO_RECVBUF`, and `SO_KEEPALIVE`.

Values for `sktOption`:

```
enum {
   kKplSocketOptionReuseAddr = 1,
   kKplSocketOptionDontRoute,
   kKplSocketOptionBroadcast,
   kKplSocketOptionRcvBuf,
   kKplSocketOptionKeepAlive
};
```

```
FskErr KplSocketBind(KplSocket skt, int addr, int port);
```

Assigns an address and port to the specified socket; equivalent to `bind` in POSIX.

```
Boolean KplSocketIsReadable(KplSocket skt);
```

Indicates whether data is available to read on the specified socket.

```
Boolean KplSocketIsWritable(KplSocket skt);
```

Indicates whether data can be written to the specified socket.

```
FskErr KplSocketRecvTCP(KplSocket skt, void *buf, const int bufSize, int
   *amt);
```

Reads up to `bufSize` bytes from the specified socket into the preallocated buffer. The actual number of bytes read is returned in `amt`.

```
FskErr KplSocketSendTCP(KplSocket skt, void *buf, const int bufSize, int
   *amt);
```

Writes up to `bufSize` bytes from the buffer `buf` to the specified socket. The actual number of bytes sent is returned in `amt`.

```
FskErr KplSocketRecvUDP(KplSocket skt, void *buf, const int bufSize, int
   *amt, int *fromIP, int *fromPort);
```

Reads up to `bufSize` bytes from the specified socket into the preallocated buffer. The actual number of bytes read is returned in `amt`, the source IP address in `fromIP`, and the port in `fromPort`.

```
FskErr KplSocketSendUDP(KplSocket skt, void *buf, const int bufSize, int
   *amt, int toIP, int toPort);
```

Writes up to `bufSize` bytes from the buffer `buf` to the address `toIP` and port `toPort`. The actual number of bytes sent is returned in `amt`.

```
FskErr KplSocketConnect(KplSocket skt, int toIP, int port);
```

Connects the specified TCP socket to the indicated IP address and port; equivalent to `connect` in POSIX.

```
FskErr KplSocketListen(KplSocket skt);
```

Listens for connections on the specified TCP socket; similar to `listen` in POSIX. The socket must have already been bound to an address and port.

```
FskErr KplSocketAcceptConnection(KplSocket listeningSkt, KplSocket
   *createdSocket, int fromIP, int fromPort);
```

Takes a connection pending to the specified listening socket and returns a new socket connected to the remote socket from address `fromIP` and port `fromPort`; equivalent to `accept` in POSIX.

```
FskErr KplSocketGetLocalAddress(KplSocket skt, UInt32 *fromIP, int
   *fromPort);
```

Returns the local address and port of the specified socket; equivalent to `getsockname` in POSIX.

```
FskErr KplSocketGetRemoteAddress(KplSocket skt, UInt32 *fromIP, int
   *fromPort);
```

Returns the remote address and port of the specified socket; equivalent to `getpeername` in POSIX.

```
FskErr KplSocketMulticastJoin(KplSocket skt, int multicastAddr, int
   interfaceAddr, int ttl);
```

(Optional) Joins a multicast session; equivalent to calling `setsockopt` in POSIX with the `IP_ADD_MEMBERSHIP` and `IP_MULTICAST_TTL` options.

```
FskErr KplSocketMulticastSetOutgoingInterface(KplSocket skt, int
   interfaceAddr, int ttl);
```

(Optional) Sets the multicast outgoing interface; equivalent to calling `setsockopt` in POSIX with the `IP_MULTICAST_IF` and `IP_MULTICAST_TTL` options.

```
void FskKplSocketHostEvent(KplSocket skt, UInt32 eventType);
```

Indicates that the specified socket has been signaled with a socket event; a helper function provided by the Kinoma Platform, not typically required by POSIX-compliant hosts. The socket events are equivalent to `FD_READ`, `FD_ACCEPT`, `FD_WRITE`, `FD_CONNECT`, and `FD_CLOSE`.

Values for `eventType`:

```
enum {
   kKplSocketHostEventRead = 0,
   kKplSocketHostEventAccept,
   kKplSocketHostEventWrite,
   kKplSocketHostEventConnect,
   kKplSocketHostEventClose
};
```

3.10 Network

The network functions support host name and DNS resolution services.

```
FskErr KplNetHostnameResolve(char *hostname, int *addr);
```

Returns the IPv4 host address corresponding to the specified host name. The `hostname` string may already be in dotted decimal notation, in which case the function converts it into the numeric IPv4 `addr`.

```
FskErr KplNetServerSelection(const char *dname, char **srvName, UInt16
   *targetPort);
```

Returns the IP address and port for a given service (DNS SRV). The function performs a DNS query to find the host name that provides the service for the domain.

3.11 Network Interfaces

These optional functions enable the Kinoma Platform to monitor added or removed network interfaces.

```
FskErr KplNetInterfaceInitialize(void);
```

(Optional) Initializes the module; called once at startup.

```
FskErr KplNetInterfaceTerminate(void);
```

(Optional) Terminates the module; called once at shutdown.

```
FskErr KplNetInterfaceEnumerate(KplNetInterfaceRecord **interfaceList);
```

(Optional) Enumerates and returns a linked list of available network interfaces.

`KplNetInterfaceRecord` structure:

```
typedef struct KplNetInterfaceRecord {
   struct   KplNetInterfaceRecord *next;
   char     *name;
   int      ip;
   int      netmask;
   char     MAC[6];
   int      status;   // 1 if interface is "up"
} KplNetInterfaceRecord, *KplNetInterface;
```

```
FskErr KplNetInterfaceSetChangedCallback(KplNetInterfaceChangedCallback
   callback, void *refCon);
```

(Optional) Creates a network interface notifier. The callback function will be invoked whenever a network interface is added or removed. The callback function is invoked in the thread in which the associated network interface notifier was created.

Callback function prototype:

```
typedef void (*KplNetInterfaceChangedCallback)(void *refCon);
```

3.12 Screen

The Kinoma Platform has two different ways of accessing pixels on the screen:

- It can access them directly. In this case, KPL implements `KplScreenGetBitmap` to provide the bitmap corresponding to the screen, and `KplScreenLockBitmap` and `KplScreenUnlockBitmap` to provide access to the bitmap. Also, KPL hosts must implement a Kinoma Platform frame buffer extension that uses the KPL screen functions to access the pixels on the screen.
- The alternative method is for KPL to implement the `KplScreenDrawBitmap` function to transfer pixels from an offscreen bitmap to the screen; however, at this time the Kinoma Platform does not support the use of `KplScreenDrawBitmap`.

```
typedef struct {
   void     *baseAddress;
   SInt32   rowBytes;
   UInt32   depth;
   UInt32   pixelFormat;
   UInt32   width;
   UInt32   height;
} KplBitmapRecord, *KplBitmap;
```

The `KplBitmapRecord` structure describes a bitmap.

The `rowBytes` field defines the distance between scan lines in bytes. On some operating systems, this distance is called *stride.* The value of `rowBytes` may be positive or negative.

The `depth` field defines the number of bits per pixel.

The `baseAddress` value must be 4-byte-aligned.

```
FskErr KplScreenGetBitmap(KplBitmap *bitmap);
```

Returns a bitmap instance corresponding to the screen. The `rowBytes` and `baseAddr` fields do not need to be filled in until `KplScreenLockBitmap` is called.

```
FskErr KplScreenDisposeBitmap(KplBitmap bitmap);
```

Releases all resources associated with the bitmap instance.

```
FskErr KplScreenLockBitmap(KplBitmap bitmap);
```

(Optional) Locks the indicated bitmap and ensures that the `baseAddress` and `rowBytes` fields are up to date. If this function is not implemented, `KplScreenDrawBitmap` must be implemented.

```
FskErr KplScreenUnlockBitmap(KplBitmap bitmap);
```

(Optional) Unlocks the indicated bitmap. After calling `KplUnlockBitmap`, the Kinoma Platform will not access the memory pointed to by `baseAddress` until `KplLockBitmap` is called again.

```
FskErr KplScreenDrawBitmap(KplBitmap src, KplRectangle srcRect, KplBitmap
   screen, KplRectangle dstRect);
```

(Optional) Copies the pixels from the `src` bitmap contained in the `srcRect` rectangle to the `screen` bitmap, offsetting and scaling to fit the `dstRect` rectangle. If this function is not implemented, `KplScreenLockBitmap` and `KplScreenUnlockBitmap` must be implemented.

`KplRectangleRecord` structure:

```
typedef struct KplRectangle {
   SInt32   x;
   SInt32   y;
   SInt32   width;
   SInt32   height;
} KplRectangleRecord, *KplRectangle;
```

```
FskErr KplScreenGetAuxInfo(unsigned char **auxInfo, UInt32 auxInfoSize);
```

(Optional) Returns auxiliary information related to the screen. This function is not currently called by the Kinoma Platform but is provided for KPL hosts that may need additional screen hardware details.

3.13 UI Events

KPL user interface events provide a way for the host operating system to provide notifications of user events and UI-related changes.

```
FskErr KplUIEventSetCallback(KplUIEventCallback proc, void *refcon);
```

Sets the callback function for KPL to call to provide UI-related events. The callback function must be called in the same thread in which `KplUIEventSetCallback` was invoked.

Callback function prototype:

```
typedef FskErr (*KplUIEventCallback)(KplUIEvent event, void
*refcon);
```

`KplUIEventRecord` structure:

```
typedef struct {
   UInt32          eventID;
   KplTimeRecord   eventTime;
   union {
      struct {
         SInt32   keyCode;
         UInt32   modifiers;
      } key;
      struct {
         SInt32   x;
         SInt32   y;
         UInt32   index;
      } mouse;
      struct {
         SInt32   width;
```

```
            SInt32   height;
         } resize;
         struct {
            KplRectangleRecord   area;
         } update;
      };
   } KplUIEventRecord, *KplUIEvent;
```

Values for `eventID`:

```
   enum {
      kKplUIEventKeyDown = 1,
      kKplUIEventKeyRepeat,
      kKplUIEventKeyUp,

      kKplUIEventMouseHover,
      kKplUIEventMouseDown,
      kKplUIEventMouseDrag,
      kKplUIEventMouseUp,

      kKplUIEventScreenActive,
      kKplUIEventScreenInactive,
      kKplUIEventScreenUpdate,
      kKplUIEventScreenResize
   };
```

```
FskErr KplUIEventNew(KplUIEvent *event, UInt32 eventID, KplTime eventTime);
```

(Optional) Creates a KPL UI event with the specified `eventID` and `eventTime`. If the `eventTime` value is `NULL`, the `eventTime` field in the new UI event is set to the current time using the `KplTimeGetNow` function.

```
FskErr KplUIEventDispose(KplUIEvent event);
```

(Optional) Disposes of the specified UI event.

```
FskErr KplUIEventSend(KplUIEvent event);
```

(Optional) Sends the indicated Ui event to the callback function specified in the `KplUIEventSetCallback` function.

3.14 Device

To build the user interface and properly integrate with device features, the Kinoma Platform needs up-to-date information about some device properties. All device properties are optional but are recommended where possible.

```
FskErr KplDeviceGetProperty(UInt32 propertyID, KplProperty value);
```

Retrieves the current value of the indicated device property. The device `propertyID` values are listed at the end of this section. If the requested property is not supported on the device, `KplDeviceGetProperty` fails with error code `kFskErrUnimplemented`.

`KplPropertyRecord` structure:

```
   typedef struct {
      UInt32   propertyType;
      union {
         SInt32   integer;
         double   number;
         char     *string;
         struct {
            UInt32   count;
            UInt32   *integer;
         } integers;
```

```
            struct {
               UInt32   count;
               double   *number;
            } numbers;
            Boolean   b;
         };
      } KplPropertyRecord, *KplProperty;
```

Values for `propertyType`:

```
      enum {
         kKplPropertyTypeInteger = 1,
         kKplPropertyTypeDouble,
         kKplPropertyTypeString,
         kKplPropertyTypeIntegers,
         kKplPropertyTypeDoubles,
         kKplPropertyTypeBoolean
      };
```

```
FskErr KplDeviceSetProperty(UInt32 propertyID, KplProperty value);
```

Sets the current value of the indicated device property. If the requested property is not supported on the device, or if the property is read-only, `KplDeviceSetProperty` fails with error code `kFskErrUnimplemented`.

```
FskErr KplDeviceNotifyPropertyChangedNew(UInt32 propertyID,
   KplPropertyChangedCallback proc, void *refcon);
```

Registers a callback function to be invoked whenever the value of the indicated device property changes. The callback function must be called in the thread in which it was created. If the requested property is not supported on the device, `KplDeviceNotifyPropertyChangedNew` fails with error code `kFskErrUnimplemented`.

Callback function prototype:

```
      typedef FskErr (*KplPropertyChangedCallback) (UInt32 propertyID,
      void *refcon);
```

```
FskErr KplDeviceNotifyPropertyChangedDispose(UInt32 propertyID,
   KplPropertyChangedCallback proc, void *recon);
```

Unregisters the specified callback function registered with `KplDeviceNotifyPropertyChangedNew`.

Following are the device `propertyID` values used when calling the KPL device functions and the callback function registered with `KplDeviceNotifyPropertyChangedNew`.

`kKplPropertyDeviceBatteryLevel`

The percentage charge level for the battery, from 0 to 100.

Data type: `kKplPropertyTypeInteger`

Read-only

`kKplPropertyDeviceCharging`

True if the device is plugged in or charging.

Data type: `kKplPropertyTypeBoolean`

Read-only

`kKplPropertyDeviceTouch`

True if the device has a touch screen.

Data type: `kKplPropertyTypeBoolean`

Read-only

`kKplPropertyDeviceOS`

The operating system name and version, appropriate for inclusion in an HTTP User-Agent header.

Data type: `kKplPropertyTypeString`

Read-only

`kKplPropertyDeviceLanguage`

The preferred language (human) for the user interface, formatted according to the ISO 639 standard, with subtags as per RFC 1766 for use in an HTTP Language header.

Data type: `kKplPropertyTypeString`

Read-only

`kKplPropertyDeviceLocation`

The location of the device, as up to date as is feasible. The property consists of an array of doubles: the first value is the latitude, the second is the longitude, and the third is the UTC time that the location was determined. Because determining the location may take some time or require additional hardware to be powered up, the device location may be available only if a property change notifier for this property (registered with `KplDeviceNotifyPropertyChangedNew`) is active.

Data type: `kKplPropertyTypeNumbers`

Read-only

`kKplPropertyDeviceOrientation`

The current physical orientation of the device as determined by a gyroscope-type sensor. Values are 1 for normal, 2 for 90 degrees rotation, 3 for 180 degrees, 4 for 270 degrees, 5 for screen-up, and 6 for screen-down. If the value is unknown now but may be known later, the value is 0. Because determining the orientation may require additional hardware to be powered up, the device orientation may be available only if a property change notifier for this property (registered with `KplDeviceNotifyPropertyChangedNew`) is active.

Data type: `kKplPropertyTypeInteger`

Read-only

`kKplPropertyDevicePowerFlags`

Bit-mask flags related to power management. The defined flag values are:

`kKplPropertyDevicePowerFlagDisableScreenDimming`

When this flag is set, the screen should remain on and should not dim. Typically this value is used when a video or a photo slideshow is being played.

`kKplPropertyDevicePowerFlagDisableSleep`

When this flag is set, the device should not turn off, but the screen may turn off or dim. This flag is set when operations are being performed that should not be interrupted, such as playing music.

Data type: `kKplPropertyTypeUInt32`

Read and write

`kKplPropertyPhoneSignalStrength`

The signal strength of the phone's cellular radio, from 0 to 100.

Data type: `kKplPropertyTypeInteger`

Read-only

`kKplPropertyPhoneCallStatus`

The state of the phone line, as `dial`, `idle`, `offer`, `connect`, or `disconnect`.

Data type: `kKplPropertyTypeString`

Read-only

3.15 Audio Output

```
FskErr KplAudioNew(KplAudio *audio);
```

(Optional) Creates an audio playback instance. Use `KplAudioSetFormat` to configure the audio format.

```
FskErr KplAudioDispose(KplAudio audio);
```

(Optional) Releases all resources associated with the specified audio playback instance. If audio is playing when `KplAudioDispose` is called, the playback will be stopped.

```
FskErr KplAudioSetFormat(KplAudio audio, const char *format, UInt32
   channels, double sampleRate, const unsigned char *formatInfo, UInt32
   formatInfoSize);
```

(Optional) Defines the format of audio to be played. This function must be called before (and may not be called after) `KplAudioStart`.

The `format` argument defines the format of the audio to be played. Some common audio format strings are `x-audio-codec/pcm-16-le` (16-bit PCM data, little-endian), `x-audio-codec/mp3` (MP3 audio), and `x-audio-codec/aac` (AAC audio). If the audio format is not supported—for example, if the audio output does not support MP3 decoding—`KplAudioSetFormat` should return `kFskErrCodecNotFound`. In that case, the Kinoma Platform will try to use a software decompressor instead.

The value of `channels` is 1 for mono and 2 for stereo. The `sampleRate` argument indicates the sample frequency of the audio—for example, 44100 for CD audio. If the channel count or sample rate is not supported, `kFskErrOperationFailed` is returned.

Some compressed audio formats require additional initialization to decode (so-called "out of band" data). The details of this additional information depend on the format. This additional information data is pointed to by the `formatInfo` argument and has a length indicated by the `formatInfoSize` argument.

```
FskErr KplAudioGetFormat(KplAudio audio, const char **format, UInt32
   *channels, double *sampleRate, const unsigned char **formatInfo, UInt32
   *formatInfoSize);
```

(Optional) Returns the currently configured audio format for the audio playback instance. Pass `NULL` for any argument that is not needed.

```
FskErr KplAudioWrite(KplAudio audio, const char *data, UInt32 dataSize,
   void *dataRefCon, UInt32 frameCount, const UInt32 *frameSizes);
```

(Optional) Enqueues audio for playback on the specified audio playback instance. The audio to play is pointed to by `data` with a total byte count of `dataSize`. The number of frames of audio is indicated by the `frameCount` parameter. If the frames are all the same size, the frame size can be calculated by dividing `dataSize` by `frameCount`; if frames are of different sizes (as in variable-bit rate audio encoding), the `frameSizes` array provides the size of each audio frame.

```
FskErr KplAudioStart(KplAudio audio);
```

(Optional) Starts playing the audio previously queued using `KplAudioWrite`.

```
FskErr KplAudioStop(KplAudio audio);
```

(Optional) Stops audio playback on the specified audio playback instance.

```
FskErr KplAudioGetProperty(KplAudio audio, UInt32 propertyID, KplProperty
   value);
```

(Optional) Returns information about various audio properties, specified by one of the following `propertyID` values. Also see `KplDeviceGetProperty` in Section 3.14, "Device," for details on using property values.

`kKplPropertyAudioPreferredSampleRate`

The default sample rate for the device, typically 44100 or 48000.

Data type: `kKplPropertyTypeDouble`

Read-only

`kKplPropertyAudioPreferredUncompressedFormat`

The audio data format preferred to play uncompressed audio, typically `x-audio-codec/pcm-16-le`.

Data type: `kKplPropertyTypeString`

Read-only

`kKplPropertyAudioPreferredBufferSize`

The number of audio samples preferred in each call to `KplAudioWrite`. The value of this property may depend on the audio format settings made with `KplAudioSetFormat`.

Data type: `kKplPropertyTypeInteger`

Read-only

`kKplPropertyAudioSamplePosition`

The number of audio samples played through the speaker. This value is used as a clock, to synchronize video, to provide the user with feedback on playback progress, and to determine when to write additional audio to the audio playback instance. This property is valid only between calls to `KplAudioStart` and `KplAudioStop`.

Data type: `kKplPropertyTypeDouble`

Read-only

`kKplPropertyAudioVolume`

The current audio volume level for each channel. The volume for each channel is a 16.16 fixed-point integer, where 1.0 (that is, 65336) corresponds to full volume.

Data type: `kKplPropertyTypeIntegers`

Read and write

`kKplPropertyAudioSingleThreadedClient`

True if the KPL audio client calls back from a single thread.

Data type: `kKplPropertyTypeBoolean`

Read-only

```
FskErr KplAudioSetProperty(KplAudio audio, UInt32 propertyID, KplProperty
   value);
```

(Optional) Sets the audio property specified by `propertyID`. See `KplAudioGetProperty` for descriptions of audio properties. (Only properties specified as "write" there may be passed to `KplAudioSetProperty`.)

```
FskErr KplAudioSetDoneCallback(KplAudio audio, KplAudioDoneCallback
   callback, void *refcon);
```

(Optional) Registers a callback function to be called after an audio block has been played.

Callback function prototype:

```
typedef void (*KplAudioDoneCallback)(KplAudio audio, void *refcon,
void *dataRefCon, Boolean done);
```

where the `dataRefCon` parameter is provided by the `KplAudioWrite` call.

```
FskErr KplAudioSetMoreCallback(KplAudio audio, KplAudioMoreCallback
   callback, void *refcon);
```

Registers a callback function to be called when more audio samples need to be enqueued for playback.

Callback function prototype:

```
typedef FskErr (*KplAudioMoreCallback)(KplAudio audio, void
*refcon, SInt32 requestedSamples);
```

where the `requestedSamples` parameter indicates the number of samples to be delivered. The callback function returns `kFskErrNone` if it was able to satisfy the request, or a nonzero error code on failure.

```
FskErr KplAudioGetSamplesQueued(KplAudio audio, UInt32 *samplesQueued,
   UInt32 *targetQueueLength);
```

Returns the number of samples enqueued for playback and the suggested queue length. Pass `NULL` for any argument that is not needed.

3.16 OpenGL

The Kinoma Platform supports OpenGL ES 2.0 for acceleration of 2D graphics operations on supporting platforms. The OpenGL KPL functions provide the interface between the device EGL rendering context and the Kinoma Platform.

```
FskErr KplGLInitialize(EGLDisplay *display, EGLSurface *surface, EGLContext
   *context, void *nativeWindow);
```

Initializes the module; called once at startup. Returns the OpenGL ES display, surface, context, and native window. The Kinoma Platform will release these resources at shutdown.

```
void KplGLTerminate(void);
```

Terminates the module; called once at shutdown.

3.17 Utilities

```
FskErr KplUtilitiesInitialize(void);
```

Initializes the module; called once at startup.

```
void KplUtilitiesTerminate(void);
```

Terminates the module; called once at shutdown.

```
FskErr KplUtilitiesHardwareInitialize(void);
```

Performs any required one-time initialization of hardware resources; called once at startup.

```
FskErr KplUtilitiesHardwareTerminate(void);
```

Performs any required one-time termination of hardware resources; called once at shutdown.

```
UInt32 KplUtilitiesRandomSeedInit(void);
```

Returns a 32-bit seed for the Kinoma Platform random number generator. The seed is used to initialize the POSIX `srand` function.

```
char *KplUtilitiesGetApplicationPath(void);
```

Returns the absolute path to the directory containing the application executable.

```
void KplUtilitiesDelay(UInt32 milliseconds);
```

Delays the active thread by the amount of time specified by `milliseconds`; equivalent to the `sleep` or `usleep` POSIX function.

3.18 Main

The KPL host owns the "main" application entry point function—typically the classic C `main()` function with the familiar `argc` and `argv` parameters. The KPL host also owns the platform "event loop" or similar native event handling/processing mechanism.

From the native main application entry point function, the KPL host calls `KplMain` to initialize the Kinoma Platform, passing a pointer to a native event loop processing function. After initialization, the Kinoma Platform calls and transfers control to the native event loop processing function.

```
int KplMain(UInt32 flags, int argc, char **argv, KplMainProc mainProc, void
   *refcon);
```

Initializes the Kinoma Platform and provides optional command-line arguments; called from the main application entry point. Each string in the `argv` list must be encoded in UTF-8. After initialization, the Kinoma Platform calls the `mainProc` function for native event loop processing. When the `mainProc` function returns, the Kinoma Platform runs the shutdown sequence and terminates. The prototype for the `mainProc` function is as follows:

```
typedef int (*KplMainProc)(void *refcon);
```

The `flags` parameter of `KplMain` allows for custom configuration of the initialization. The bit-mask values of `flags` are as follows:

```
enum {
    kKplMainNetwork = (1L << 0),
    kKplMainServer = (1L << 1),
    kKplMainNoECMAScript = (1L << 2)
};
```

Set the `kKplMainNetwork` flag to enable networking, the `kKplMainServer` flag to enable the Kinoma Platform HTTP server, and the `kKplMainNoECMAScript` flag to disable the ECMAScript runtime and virtual machine services.

3.19 Environment

The Kinoma Platform can store environment variables that can subsequently be accessed from native code and ECMAScript applications.

```
FskErr KplEnvironmentInitialize(FskAssociativeArray environment);
```

(Optional) Gives the KPL host an opportunity to register environment variables; called once at startup. Typically, the KPL host provides environment variables for the host operating system name and version.

3.20 Extensions

KPL extensions are optional components used to augment and extend the core Kinoma Platform feature set. Extensions can be written in native C code, ECMAScript, or a combination of both.

Extensions can be loaded dynamically from libraries by the Kinoma Platform or embedded into the single executable binary. KPL hosts that are running on embedded operating system targets typically embed extension code. The KPL extension functions are required only when the extensions are loaded dynamically as shared object files.

```
FskErr KplLibraryLoad(FskLibrary *libraryOut, const char *path);
```

(Optional) Loads the library specified by `path` into memory.

```
FskErr KplLibraryUnload(FskLibrary library);
```

(Optional) Unloads the specified library from memory.

```
FskErr KplLibraryGetSymbolAddress(FskLibrary library, const char *symbol,
   void **address);
```

(Optional) Returns the address of the function named `symbol`.

3.21 Miscellaneous

```
FskErr KplBrowserOpenURI(const char *uri);
```

(Optional) Opens the specified URI in a web-page viewer.

```
const char *KplECMAScriptGetPlatform(void);
```

Returns a string corresponding to the platform name. For the desktop platforms, the Kinoma Platform currently recognizes `win` for Windows and `mac` for Mac OS. Linux platforms are recognized as `linux`. The returned string is made accessible to application script code.

```
const char *KplECMAScriptGetExtension(void);
```

(Optional) Returns a string corresponding to the platform's shared object file extension—for example, `.so` on Linux.

4 Glossary

API

Application programming interface.

Kinoma Platform

A modern application framework designed for optimal performance on mobile devices. Kinoma Play is implemented using this framework.

Kinoma Play Script (KPS)

The API of Kinoma Play, enabling the development and the deployment of media and network applications that fit the user experience of KP.

Kinoma Porting Layer (KPL)

A simple, low-level API that provides access to essential functionality from the host operating system, enabling the Kinoma Platform (which is built on KPL) to run on a wide range of mobile operating systems.

kit

Short for "KPL Kit" in this document.

KPL

See Kinoma Porting Layer (KPL).

KPL Kit

For both Linux and Windows, a reference implementation of the KPL API plus the components you need to build the Kinoma Platform and an executable for testing your KPL implementation on your target platform; shortened to "kit" in this document.

KPS

See Kinoma Play Script (KPS).

POSIX

The Portable Operating System Interface, which defines an API providing software compatibility across UNIX-like operating systems.