OpenKODE 1.0 Provisional Specification

revision 1

Edited by Tim Renouf

OpenKODE 1.0 Provisional Specification: revision 1

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, or any of its Promoters, Contributors or Members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Table of Contents

1. Introduction

1.1. Specification conventions

1.1.1. Non-normative text

Certain subsections and paragraphs of this specification are descriptive notes to aid understanding or to provide rationale. Such subsections do not form part of the OpenKODE specification, and are marked as such by the text being in a shaded box, like the next section.

1.2. Overview

OpenKODE® is a royalty-free, cross-platform standard that combines a set of native APIs into a comprehensive media stack specification for accelerating rich media and graphics applications. OpenKODE aims to make advanced media capabilities consistently available across multiple devices for increased native source portability and reduced mobile platform fragmentation. OpenKODE 1.0 brings together the OpenGL ES and OpenVG Khronos media APIs to provide state-of-the-art acceleration for vector 2D and 3D graphics and provides the new OpenKODE Core API that abstracts operating system resources to minimize source changes when porting games and applications between Linux, Brew, Symbian, Windows Mobile, WIPI and RTOS-based platforms. Subsequent versions of OpenKODE will add the OpenSL ES and OpenMAX media APIs to provide accelerated video and audio that is fully integrated with graphics processing

1.2.1. OpenKODE and OpenKODE Core

OpenKODE brings together:

- Khronos media APIs (OpenGL ES and OpenVG, with OpenSL ES and OpenMAX AL to be added soon);
- EGL, which acts as a “hub” for the media APIs;
- OpenKODE Core, an API providing an abstraction of operating system functions and libraries such as the event system, file access and memory allocation. OpenKODE Core is part of this specification.

Part I. OpenKODE 1.0 Provisional

2. OpenKODE conformance

2.1. Conformant OpenKODE implementation

A conformant OpenKODE 1.0 Provisional implementation consists of the following components, each of which must be conformant within itself, and must interact with the others in a conformant way.

- OpenKODE Core 1.0 Provisional;
- zero or more of the following media APIs:
  - OpenGL ES 1.1, with, if OpenVG is also present, the GL_OES_egl_image extension;
  - OpenVG 1.0.1, with, if OpenGL ES is also present, the VG_KHR_egl_image extension;
- EGL 1.3.

2.1.1. EGL

When in an OpenKODE 1.0 Provisional implementation, EGL has the following requirements:

EGLImage-related extensions

If both OpenGL ES and OpenVG are present, then the EGL_KHR_image, EGL_KHR_gl_image and EGL_KHR_vg_parent_image extensions must be present.

Lock surface extension

It is intended that the final version of OpenKODE 1.0 will mandate EGL_KHR_lock_surface as in the following text. It cannot mandate that extension yet as it had not been defined at the time that OpenKODE 1.0 Provisional was ratified.

The EGL_KHR_lock_surface extension must be present. EGL must expose a config with `EGL_LOCK_SURFACE_BIT` and `EGL_WINDOW_BIT` bits set in the `EGL_SURFACE_TYPE` attribute of EGLConfigs (so the config allows a window surface which can be locked), and where the following attributes queried by `eglQuerySurface` have the stated values.

`EGL_BITMAP_PIXEL_SIZE`	2
`EGL_BITMAP_PIXEL_RED_BITS`	5
`EGL_BITMAP_PIXEL_GREEN_BITS`	6
`EGL_BITMAP_PIXEL_BLUE_BITS`	5
`EGL_BITMAP_PIXEL_ALPHA_BITS`	0
`EGL_BITMAP_PIXEL_LUMINANCE_BITS`	0
`EGL_BITMAP_PIXEL_RED_OFFSET`	11
`EGL_BITMAP_PIXEL_GREEN_OFFSET`	5
`EGL_BITMAP_PIXEL_BLUE_OFFSET`	0

Lock surface extension rationale

The lock surface extension provides the limited direct blitting to the screen functionality that is required if an application wants to perform its own rendering. On a platform with no hardware acceleration, an application can usually perform its own rendering faster than using a software OpenGL ES imeplementation, since the renderer can be tuned for the application.

The extension is mandated, even on platforms that do have hardware OpenGL ES support, so that applications written for low-end platforms continue to be easily portable to high-end platforms. This may incur a performance penalty on high-end platforms when using the feature, for example it may be that the only way to blit a bitmap onto the screen is as a texture, but, since applications using it will be low-end, this is seen as less important than maximizing performance on the targeted low-end platforms.

It is mandated that EGL must expose a config usable with the lock surface API with a particular fixed pixel format (RGB565), even if this does not reflect the underlying hardware and thus requires a color conversion in the implementation. Without this, it is possible that an application (whose renderer knows about some set of pixel formats) will not work on some platform (which has a pixel format not in that set).

A future version of OpenKODE could conceivably allow for an implementation with audio (OpenSL ES) but no graphics support. If this is the case, then it will state that EGL is optional (except perhaps for `eglGetProcAddress`), and the lock surface extension is required only if (the rest of) EGL is present.

EGL entry points

Certain EGL entry points may be meaningless depending on which of its client APIs are included in the implementation. Such functions are present, but may do nothing (perhaps returning an error code).

The following EGL functions must always be implemented: `eglCopyBuffers`; `eglDestroySurface`; `eglGetConfigAttrib`; `eglGetConfigs` and `eglChooseConfig` (note that an implementation has enormous flexibility in the range of EGLConfigs supported); `eglGetCurrentDisplay`; `eglGetDisplay` (using `EGL_DEFAULT_DISPLAY` must return a default display); `eglGetError`; `eglGetProcAddress`; `eglInitialize`; `eglQueryAPI`; `eglQueryString`; `eglQuerySurface`; `eglReleaseThread`; `eglSurfaceAttrib`; `eglSwapBuffers`; `eglSwapInterval`; `eglTerminate`.

The following EGL calls need not be fully implemented in some circumstances. Where applicable, these calls can be implemented to simply return a failure code (`EGL_FALSE`, `EGL_NO_SURFACE`, `EGL_NO_CONTEXT`, etc.), and possibly raise an EGL error as defined in the EGL specification.

Client API management:

- `eglBindAPI`; `eglWaitClient` (if neither client API is supported, need not be fully implemented)

Surface management:

- `eglCreateWindowSurface`, `eglCreatePbufferSurface`, `eglCreatePixmapSurface` (if window, pbuffer, or native pixmap rendering respectively is not supported by any EGLConfig, then the corresponding create-surface call need not be implemented)
- `eglCreatePbufferFromClientBuffer` (if OpenVG is not supported, need not be fully implemented)

Context management:

- `eglCreateContext`, `eglDestroyContext`, `eglGetCurrentContext`, `eglGetCurrentSurface`, `eglMakeCurrent`, `eglQueryContext` (if no client API using a “current context” is supported, need not be fully implemented)

Client API specific:

- `eglBindTexImage`, `eglReleaseTexImage`, `eglWaitGL` (if OpenGL ES is not supported, need not be fully implemented)
- `eglWaitNative` (if no “native rendering API” is supported, can be stubbed out)

2.1.2. Future directions

A future version of OpenKODE will include OpenMAX AL 1.0 for multimedia functionality and OpenSL ES 1.0 for audio functionality.

2.2. Conformance tests

At the time of ratifying this OpenKODE 1.0 Provisional specification, the conformance tests had not been completed. The intention is that the final OpenKODE 1.0 specification will have a suite of conformance tests, and this section will specify that they must pass, and give some detail of what they test.

Part II. OpenKODE Core 1.0 Provisional

3. Overview

3.1. OpenKODE Core

OpenKODE Core is the part of OpenKODE which specifies an API to provide source-level abstraction of common operating system services in an event-driven environment, such that, combined with the Khronos media APIs into a complete OpenKODE solution, it is possible to create source-portable media and graphics applications.

3.1.1. OpenKODE Core programming environment

OpenKODE Core assumes a C programming environment (although some implementations may provide C++ as well), but none of the C library is assumed. Much of the functionality of the library is instead provided by OpenKODE Core functions.

Some of the OpenKODE Core functions are based on equivalent functions in [C89], [C99] or [POSIX], with the same parameter specification, providing either equivalent or subset functionality. These functions generally have very similar names to the C or [POSIX] equivalents, but with a `kd` prefix and with some capitalization (so the names fit the OpenKODE Core conventions). For example, the OpenKODE Core function `kdMemcpy` is equivalent to the [C89] function `memcpy`.

Some OpenKODE Core functions are based on [POSIX] functions, but with some changes. In these cases, the names are changed more such that a developer does not expect the same parameter specification and functionality. An example is `kdSocketRecv`, which is based on the BSD/[POSIX] socket function `recv`, but with fewer parameters (OpenKODE Core does not support socket flags) and with different semantics (OpenKODE Core sockets are always non-blocking, and interact with the event system).

Other OpenKODE Core functions are unique to OpenKODE Core, in particular the event system and the input/output functions.

OpenKODE Core functions include the following major areas:

Attributes and extensions

These functions allow the application to query attributes of the implementation, such as the version number of OpenKODE Core supported, and to determine the presence of extensions.

Event system

OpenKODE Core provides an event system which abstracts the event system of the platform's OS. Examples of events generated by OpenKODE Core are quit, pause and resume, window resize, input change, timer, and socket ready to read or write.

An OpenKODE application may be written as either loop-in-application, where it contains the top-level loop processing an event each iteration, or loop-in-framework, where the framework calls an event handler for each event.

Application startup and exit

An application has a single entry point called `kdMain`. OpenKODE Core provides an analog of the C standard function `exit`.

Utility functions

There are utility functions including conversions from string to number and vice versa, random number generation, memory allocation, memory and string copying, comparison and scanning, and assertions and logging.

Math

The OpenKODE Core programming environment supports 32-bit floats, and analogs of many of the C standard math library functions.

Time and timers

There are functions which are analogs of C standard time functions, as well as OpenKODE-specific functions for more accurate timekeeping, and for timers which generate events.

File system

The platform’s file system is abstracted to a *virtual file system*, allowing an application which accesses only certain well-known locations (such as “the files that came with the application”) to be written portably. The file functions are analogs of familiar C and [POSIX] functions.

Networking

OpenKODE Core provides an API similar to BSD/[POSIX] sockets, but with different API semantics such that the event system is used to notify when a socket is ready to send to or receive from.

Input/output

The input/output API provides functions to access inputs (such as buttons) and outputs (such as vibrate) in an extensible way, while specifying a small range of inputs and outputs that are likely to be present, such as game keys.

Windowing

OpenKODE Core allows an implementation to support just one full-screen window, but allows support for multiple non-full-screen windows. Simple manipulation of such windows (for example resizing and maximizing) is supported.

3.1.2. API conventions (KD and kd prefixes)

All functions, types and constants defined in OpenKODE Core have a prefix of `KD` or `kd`. Many of these functions, types and constants mirror ones that are part of various ANSI C and [POSIX] standards, and therefore already exist on some platforms. Using the prefix consistently allows for a platform with a faulty implementation of a standard C or [POSIX] type or function to have an OpenKODE implementation which provides a KD-prefixed version of the type or function which works as specified.

The prefix `KD` is used for types and constants. The prefix `kd` is used for functions.

4. Programming environment

4.1. Header file

To use OpenKODE Core functionality, a C source program includes the OpenKODE Core header file:

```
#include <KD/kd.h>
```

`<KD/kd.h>` includes `<EGL/egl.h>`, so an application may use EGL 1.3 facilities without having to include that file itself.

4.1.1. Note for implementers

Implementers are encouraged to code `KD/kd.h` such that it includes as few as possible of the platform's include files, and if possible to avoid declaring C and [POSIX] standard functions. This will ease the creation of portable OpenKODE applications, and help stop non-portable code being added accidentally.

4.1.2. Future directions

A future version of OpenKODE will include OpenSL ES as a client API. Then, the statements above will change to allow for an application and implementation using OpenKODE with OpenSL ES, without EGL or any EGL client API. An implementation supporting only that will be allowed not to include `<EGL/egl.h>` from `<KD/kd.h>`, since the application will not be using any EGL facilities, and EGL may be not present anyway.

4.2. C subset

An OpenKODE Core application is programmed to an environment which supports a subset of [C89], except that, in any case where a later C standard is incompatible with [C89], it is undefined which standard is followed.

- The language is supported, but the library is not. None of the standard header files is supported.
- Non-automatic (i.e. static, global and file scope) variables are not supported.

Rationale

An OpenKODE Core implementation that does support non-automatic variables states support for the KD_KHR_staticdata extension in its conformance statement, such that a programmer knows s/he can use such variables.

It is expected that most platforms will support non-automatic variables, the only exceptions being when an OpenKODE Core application is embedded in the ROM of certain types of low-end platforms. Thus, a programmer may use non-automatic variables safe in the knowledge that only these embedded ROM platforms will be excluded.

- Memory is an array of 8-bit bytes.
- No statement is made about the size, range, alignment requirements or behavior of the C standard intrinsic types over and above what the C standard specifies.

- The variable argument facilities normally provided by `<stdarg.h>` in [C89] are supported, and are provided by `<KD/kd.h>`.

4.2.1. Rationale

Although the C library is not supported, a subset of its functionality is provided by OpenKODE functions.

4.3. OpenKODE Core structures

Except where individually noted, structure types in this specification are defined only in the sense that an implementation must have the stated fields with the stated types. The implementation is free to:

- have the stated fields in any order;
- have padding gaps between fields;
- add extra fields not mentioned in this specification, as long as the name of each one begins with an underscore.

Note that a source portable OpenKODE application must not reference any extra fields added by an implementation to a structure type.

4.4. OpenKODE Core functions

Except where individually noted, OpenKODE Core functions behave as functions in the following respects:

- When calling a function, each argument is evaluated exactly once (although the order of evaluation is undefined).
- It is possible to take the address of an OpenKODE Core function.

However, undefining a macro of the same name as an OpenKODE Core function like this:

```
#undef funcname
```

causes undefined behavior (including the possibility of a compile or link error) when the function *`funcname`* is called.

4.4.1. Note for implementers

The implementation is free to use a macro, an inline function, a statically linked external function, or a dynamically linked external function for each OpenKODE Core function. OpenKODE Core does not impose any restrictions such as name of functions exported from a shared object.

4.5. Threading

OpenKODE is not a multi-threaded environment. No means is provided to create threads, and, if an implementation-dependent means is used to create multiple threads within the process, and two threads both attempt to call the OpenKODE API at the same time, then undefined behavior results, except as otherwise noted.

4.5.1. Future directions

Multi-threading support may be added to a future version of OpenKODE.

4.6. Types

OpenKODE defines a number of types, which are intrinsic (i.e. they participate in C's casting and promotion rules):

type	description
`KDchar`	8-bit binary integer of unspecified signedness (two's complement if signed)
`KDint32`	32-bit binary two's complement signed integer
`KDuint32`	32-bit binary unsigned integer
`KDint64`	64-bit binary two's complement signed integer
`KDuint64`	64-bit binary unsigned integer
`KDint16`	16-bit binary signed integer
`KDuint16`	16-bit binary unsigned integer
`KDint8`	8-bit binary signed integer
`KDuint8`	8-bit binary unsigned integer
`KDint`	binary two's complement signed integer of *at least* 32 bits
`KDuint`	binary unsigned integer of *at least* 32 bits
`KDuintptr`	unsigned binary integer that is large enough to contain a pointer value
`KDsize`	unsigned binary integer that is large enough to be used as the size of any object in memory
`KDssize`	signed binary integer the same size as `KDsize`
`KDsocklen`	unsigned 32-bit binary integer used as the size of a `KDSockaddr` structure
`KDfloat32`	floating point value with [IEEE 754] format and behavior
`KDboolean`	same type as `KDint`, but used for a boolean true (non-zero) or false (zero) value
`KDtime`	as KDint64, but used for time in seconds
`KDust`	as `KDint64`, but used for time in nanoseconds
`KDoff`	as `KDint64`, but used as an offset into or size of a file

type	description
KDmode	as KDuint32, used for the *st_mode* field in a KDStat structure

4.6.1. Rationale

OpenKODE Core provides only 32-bit floats, not 64-bit doubles. It was judged that developers of games and other demanding interactive applications use 32-bit floats in preference for increased performance, especially when no floating point hardware is available.

It is likely that many implementations will provide a “compliant” mode which meets this specification with regard to [IEEE 754] floating point behavior, and a “fast” mode which does not meet this specification but yields improved performance for floating point operations. It is recommended that application developers test applications with both settings of an implementation; the “compliant” mode is more likely to reveal otherwise hidden portability problems with floating point computations such as overflow or underflow, or generating a not-a-number.

4.7. Constants

Related to these types, OpenKODE defines the following:

constant	value	description
KDINT_MIN	no greater than -0x80000000	minimum value of KDint
KDINT_MAX	no less than 0x7fffffff	maximum value of KDint
KDUINT_MAX	no less than 0xffffffff	maximum value of KDuint
KDINT32_MIN	-0x80000000	minimum value of KDint32
KDINT32_MAX	0x7fffffff	maximum value of KDint32
KDUINT32_MAX	0xffffffff	maximum value of KDuint32
KDINT64_MIN	-0x8000000000000000	minimum value of KDint64
KDINT64_MAX	0x7fffffffffffffff	maximum value of KDint64
KDUINT64_MAX	0xffffffffffffffff	maximum value of KDuint64
KD_TRUE	1	canonical true value of a KDboolean
KD_FALSE	0	false value of a KDboolean

In addition, OpenKODE defines this constant:

constant	defined as
KD_NULL	((void *)0)

constant	**defined as**

5. Errors

5.1. Introduction

Many OpenKODE Core functions signal an error by returning some special error value (usually -1 for a function that returns an integer or `KD_NULL` for a function that returns a pointer), and setting the OpenKODE Core *error indicator*. The application inspects the error indicator by calling `kdGetError`. The error codes, and the concept of an error indicator, are based on [C89]'s `errno` and [POSIX]'s error list.

5.2. Constants

`KD_EACCES` (13)	Permission denied.
`KD_EADDRINUSE` (98)	Address in use.
`KD_EADDRNOTAVAIL` (99)	Address not available on the local platform.
`KD_EAFNOSUPPORT` (97)	Address family not supported.
`KD_EAGAIN` (11)	Resource unavailable, try again.
`KD_EALREADY` (114)	A connection attempt is already in progress for this socket.
`KD_EBADF` (9)	File not opened in the appropriate mode for the operation.
`KD_EBUSY` (16)	Device or resource busy.
`KD_ECONNREFUSED` (111)	Connection refused.
`KD_ECONNRESET` (104)	Connection reset.
`KD_EDESTADDRREQ` (89)	Destination address required.
`KD_EDOM` (33)	Mathematics argument out of domain of function.
`KD_ERANGE` (34)	Mathematics argument out of range.
`KD_EEXIST` (17)	File exists.
`KD_EFBIG` (27)	File too large.
`KD_EHOSTUNREACH` (113)	Host is unreachable.
`KD_EINVAL` (22)	Invalid argument.
`KD_EIO` (5)	I/O error.
`KD_EILSEQ` (84)	Illegal byte sequence.
`KD_EISCONN` (106)	Socket is connected.
`KD_EISDIR` (21)	Is a directory.
`KD_EMFILE` (24)	Too many open files.

KD_ENAMETOOLONG (36)	Filename too long.
KD_ENOENT (2)	No such file or directory.
KD_ENOMEM (12)	Not enough space.
KD_ENOSPC (28)	No space left on device.
KD_ENOSYS (38)	Function not supported.
KD_ENOTCONN (107)	The socket is not connected.
KD_EOPNOTSUPP (95)	Operation not supported.
KD_EOVERFLOW (75)	Overflow.
KD_EPERM (1)	Operation not permitted.
KD_EPIPE (32)	Socket is no longer connected.
KD_ERANGE (34)	Math library range error.
KD_ETIMEDOUT (110)	Connection timed out.

5.3. Functions

5.3.1. kdGetError

Get last error indication.

Synopsis

```
KDint kdGetError(void);
```

Description

OpenKODE Core maintains a last error indication, which is set by certain functions to indicate an error, as specified with each such function. No OpenKODE Core function sets the last error indication to a value other than as specified for that function. No OpenKODE Core function resets the last error indication to 0 on success.

This function retrieves the last error indication. It does not reset the last error indication back to 0.

If the application uses an implementation-defined method to create one or more further threads, whether the OpenKODE Core last error indication is global or per-thread is implementation-defined.

Return value

The function returns the last error code set by an OpenKODE Core function.

Future Directions

If a future version of OpenKODE includes support for threads, the last error indication will be per-thread.

5.3.2. kdSetError

Set last error indication.

Synopsis

```
void kdSetError(KDint error);
```

Description

This function sets the last error indication, as retrieved by kdGetError. Any KDint32 is allowed and, after setting, is returned unchanged by `kdGetError` (until the last error indication is otherwise set). A value which does not fit in KDint32 is cast to fit.

6. Versioning and attribute queries

6.1. Introduction

OpenKODE Core provides these functions to query attributes of the implementation, such as the version number, but principally the presence of extensions. Khronos has a well-defined mechanism for defining extensions to its APIs, and it is expected that extensions to OpenKODE Core will be defined. An extension is typically either a “staging extension”, which covers an area of functionality which could be expected to join the main OpenKODE Core specification in a future version, or an “optional extension”, which covers an area of functionality which will never be implemented on all platforms.

6.2. Functions

6.2.1. kdQueryAttribi

Obtain the value of a numeric OpenKODE Core attribute.

Synopsis

```
KDint kdQueryAttribi(KDint attribute, KDint *value);
```

Description

This function is used to obtain the value of a numeric OpenKODE Core attribute.

The value of *attribute* for the OpenKODE Core implementation is returned in the KDint pointed to by *value*. *attribute* may be one of the following values:

KD_ATTRIB_NUM_EXTENSIONS (0)	Querying KD_ATTRIB_NUM_EXTENSIONS returns the number of OpenKODE extensions supported. The name of each extension may be determined using kdQueryIndexedAttribcv.

Return value

On success, the function returns 0 and stores the requested value into the location pointed to by *value*. On failure, the function returns -1 and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_EINVAL	*attribute* is not a valid OpenKODE Core numeric attribute name.

6.2.2. kdQueryAttribcv

Obtain the value of a string OpenKODE Core attribute.

Synopsis

```
const KDchar *kdQueryAttribcv(KDint attribute);
```

Description

This function is used to obtain the value of a string OpenKODE attribute.

The value of *`attribute`* for the OpenKODE Core implementation is returned as a pointer to a static, null-terminated UTF-8 string. *`attribute`* may be one of the following:

`KD_ATTRIB_VENDOR (0)`	The format and contents of the returned string are implementation dependent, but typically include the name of the supplier of the OpenKODE Core implementation and the name of the platform on which it is running.
`KD_ATTRIB_VERSION (1)`	The format of the returned string is: major version number; period; minor version number; space; vendor-specific information. Both the major and minor portions of the version are integers of arbitrary length, corresponding to the major and minor version numbers of OpenKODE Core supported by the implementation. The vendor-specific information is optional; if present, its format and contents are undefined.

> Typically, the vendor-specific information identifies a platform-specific release number, which is unrelated to the version number of OpenKODE Core supported.

Return value

On success, the function returns a pointer to the string value, which remains valid for the life of the application. On failure, the function returns `KD_NULL` and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EINVAL`	*`attribute`* is not a valid OpenKODE string attribute name.

6.2.3. kdQueryIndexedAttribcv

Obtain the value of an indexed string OpenKODE Core attribute.

Synopsis

```
const KDchar *kdQueryIndexedAttribcv(KDint attribute, KDint index);
```

Description

This function is used to obtain the value of an indexed string OpenKODE Core attribute.

The value of the *`index`*’th *`attribute`* for the OpenKODE implementation is returned as a pointer to a static, null-terminated UTF-8 string. *`attribute`* must be one of the values shown below. The valid range of *`index`* depends on *`attribute`* as described below.

`KD_ATTRIB_EXTENSIONS (0)`	*`index`* may range from 0 to the value of the `KD_ATTRIB_NUM_EXTENSIONS` attribute (obtained with `kdQueryAttribi`) minus one. The returned string is the name of an OpenKODE Core *`extension`* supported by the implementation. Each extension name returned indicates that the corresponding OpenKODE functionality for that extension is supported by the implementation. Function pointers to entry points defined by an extension may be obtained using `kdGetProcAddress`. Whether an application can statically link to functions in an extension is defined within the extension.

Return value

On failure, the function returns `NULL` and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EINVAL`	*`attribute`* is not a valid OpenKODE Core indexed string attribute name, or *`index`* is not a valid index for *`attribute`*.

6.2.4. kdGetProcAddress

Get the address of an extension function.

Synopsis

```
void *kdGetProcAddress(const KDchar *name);
```

Description

This function obtains the function pointer for the function *`name`* defined by an OpenKODE Core extension.

If *`name`* is not the name of a function defined by an OpenKODE Core extension which is shown to be supported by the `KD_ATTRIB_EXTENSIONS` indexed attribute in `kdQueryIndexedAttribcv`, then the return value is undefined.

If *`name`* is not a readable null-terminated string, then undefined behavior results.

Return value

The function returns a function pointer for the named function.

Rationale

EGL and EGL's client APIs all use EGL's `eglGetProcAddress` to obtain a pointer to an extension function. OpenKODE Core uses its own function because it is probable that some implementations of OpenKODE will consist of OpenKODE Core from one provider (perhaps the OS or middleware vendor) and EGL and client APIs from another provider (the graphics hardware vendor). In this scenario, forcing EGL to be aware of how to find OpenKODE Core extension functions adds an unnecessary complication to the implementation.

Note the condition that the return value is undefined if *`name`* is not the name of a function in an extension that the OpenKODE implementation claims to support. This means that `kdGetProcAddress` cannot be used to determine whether an extension is present; `KD_ATTRIB_EXTENSIONS` in `kdQueryIndexedAttribcv` must be used first. Also note that `kdGetProcAddress` has an undefined return value when *`name`* is the name of a function in OpenKODE Core itself (not an extension).

7. Events

7.1. Introduction

OpenKODE Core provides an abstraction of the underlying OS’s event system.

7.1.1. Event model

An *event* is a notification of some event occurring delivered by one piece of software (the OpenKODE layer, another Khronos API, or the application) and delivered to and processed by another piece of software (the application).

Loop-in-application versus callbacks

OpenKODE Core presents a *loop-in-application* model, in which the application has a single entry point, its `kdMain` function, and that (or a subroutine called from it) contains the top level event loop.

This is in contrast to the *callback* model presented by some embedded operating systems, in which the application registers callback functions to handle various events, and the operating system itself calls those callbacks.

OpenKODE Core uses the loop-in-application model because it is recognized that programmers coming from a PC and console game environment will be expecting it, and imposing a callback model could hurt adoption of OpenKODE by reducing the amount of content ported and created for it.

OpenKODE Core does provide a callback mechanism for the application programmer who would prefer to use that model. After initializing and registering callbacks, an application can have the following code:

```
    KDEvent *event;
    while ((event = kdWaitEvent(-1)) != 0)
        kdDefaultEvent(event);
    kdExit(1);
```

Here, the application loops processing events, using callbacks that have been registered. The callbacks are called by OpenKODE from inside `kdWaitEvent`. The loop exits only on an error from `kdWaitEvent`; otherwise, the application exits by a callback using `kdExit`.

Note for implementers: loop-in-application

It is recognized that OpenKODE Core mandating a loop-in-application model may cause extra complexity in an OpenKODE implementation, where the underlying operating system uses a callback model. Suggested implementations are:

- For an operating system which has threads, use one thread (the main thread) to receive events as operating system callbacks, and use a second thread to run the OpenKODE application. The first thread passes an event to the second thread, and, when the OpenKODE application asks for another event (by a callback returning, or by the main loop calling `kdWaitEvent`), the first thread’s callback is allowed to return.
- For an operating system with no threads, the above can be used with a co-operative threading system created for the purpose. This would typically involve switching stacks in a platform-dependent way.

Event contents

An event contains the following:

- a *timestamp* giving the time that the event occurred (or was noted by the OpenKODE Core implementation);
- an *event type*;
- a *user pointer*, which is set by the application when calling some OpenKODE Core function which causes the creation of events, and can thus differ between two sources of the same event type (e.g. two sockets);
- *event data*, whose meaning differs for each event type.

Event delivery

Events are *queued* until the application is ready to receive them in its own context. There are two ways for the application to receive events, and it chooses which to use for each event type/user pointer combination:

- The event can be delivered via a callback when the event queue is processed by the application calling one of several functions that do this. Thus the callback executes in the application context, as a callback from the function.
- The event can be returned by the application calling `kdWaitEvent`.

An event enabled for delivery by callback is prioritized over a non-callback enabled event, in that both `kdPumpEvents` and `kdWaitEvent` process a callback enabled event first. This is to allow `kdPumpEvents` to be used in the middle of an application’s render loop, to ensure that events which need fast processing are processed, without such events getting stuck behind lower priority non-callback enabled events.

Some event types merge, such that the queuing of a new event of that type causes an older event of the same type already in the queue to be removed. Where this occurs, it is documented with the event type. This only occurs if both the old and new events were generated by the OpenKODE implementation, rather than being posted by `kdPostEvent`.

7.2. Types

7.2.1. KDEvent

Struct type containing an event.

Synopsis

```
typedef struct KDEvent KDEvent;
#define KD_EVENT_USER 0x100000


struct KDEvent {
    KDust timestamp;
    KDint32 type;
    void *userptr;
    union KDEventData {
        KDEventInput input;
        KDEventInputPointer inputpointer;
        KDEventInputStick inputstick;
        KDEventSocketReadable socketreadable;
        KDEventSocketWritable socketwritable;
        KDEventSocketError socketerror;
        KDEventSocketConnect socketconnect;
        KDEventSocketIncoming socketincoming;
        KDEventNameLookup namelookup;
        KDEventWindowFocus windowfocus;
        KDEventUser user;
    } data;
};
```

Description

`KDEvent` is the struct type of an event. It may contain some implementation-defined fields not shown above, and the fields defined here may appear in a different order.

The *`timestamp`* field contains a time as Unadjusted System Time (as reported by `kdGetTimeUST`) no earlier than the time the event actually occurred, and no later than the first occasion on which `kdWaitEvent` returns after that, and no later than the time the callback (if any) for the event is called.

The *`type`* field contains the type of the event, one of the `KD_EVENT_*` constants. Values in the range `KD_EVENT_USER` to `KDINT32_MAX` inclusive may be used by user code for its own private events, and are guaranteed not to be generated by OpenKODE Core.

The *`userptr`* field contains a pointer provided by the application to the API that generates the event. Each event type documents where its *`userptr`* value comes from.

The *`data`* field contains the data provided with the event. It is a union, and the event type determines which element of the union is applicable. The alignment and size of the event data union are determined by the maximum alignment and size of the *`generic`* element.

7.3. Functions

7.3.1. kdWaitEvent

Get next event from queue.

Synopsis

```
const KDEvent *kdWaitEvent(KDust timeout);
```

Description

This function is used in the application's event loop to get the next event in the queue whose event type and user pointer combination is not covered by an installed callback (see `kdInstallCallback`).

The function times out after no less than *`timeout`* nanoseconds, as soon as the queue is empty. The function may in fact take longer than the requested timeout because of the implementation-dependent timer resolution, and because of event callbacks taking non-zero time. The function never times out if *`timeout`* is `-1`.

The function effectively consists of a loop which performs the following processing:

- If an error has occurred, the function returns.
- If any event in the queue is covered by an installed callback, the function removes the first such event from the queue and calls the callback for it (in the same context as the caller of `kdWaitEvent`), then jumps back to the top of the loop.
- Each event remaining in the queue has an event type and user pointer combination not covered by an installed

callback. If any such event remains, the function removes the first such event from the queue and returns with it.

- If the queue is empty, the function waits until an event arrives, jumping back to the top of the loop when one has arrived. If the timeout expires during that wait, the function returns with a timeout error.

Return value

If a non-callback enabled event becomes available, the function returns a pointer to its KDEvent. This pointer remains valid until the next time kdWaitEvent is called.

If no event is available, the function returns KD_NULL and stores one of the error codes listed below into the error indicator returned by kdGetError. This includes the case where the timeout expired.

If the caller does not recognize the event returned, or does not want to process it, it must call kdDefaultEvent with it before calling kdWaitEvent again, otherwise an action requested by the user (by pressing a key or clicking an on-screen button or other UI interaction) may be lost if the action is handled by the underlying OS when the application has decided not to handle it.

Error codes

KD_EAGAIN

The timeout expired while the event queue was empty.

KD_ENOMEM

OpenKODE Core ran out of resources when queuing events. This error need not be fatal; once it is able to allocate memory again, the event system will continue to function normally. However, one or more events may have been lost.

Rationale

The rules above mean that callback enabled events are delivered first, before this function returns a non-callback enabled event. This is for consistency with kdPumpEvents.

Future directions: expansion of KDEvent

This function returns a pointer to a KDEvent allocated by the OpenKODE implementation. The intention is to allow two directions for adding new fields to an event:

- A later version of the OpenKODE specification might add a new mandatory field to events. An implementation can implement this by extending KDEvent without breaking binary compatibility with applications compiled and linked with an earlier version of the same implementation.
- An optional or vendor extension to OpenKODE can add a new optional field by adding an “accessor” function to read it (and one to write it when posting an event) using the KDEvent pointer as the handle.

Future directions: thread support

If a future version of OpenKODE Core has support for threads, then the KDEvent pointer returned by this function will remain valid until the next call to kdWaitEvent *in the same thread*.

7.3.2. kdSetEventUserptr

Set the *userptr* for global events.

Synopsis

```
void kdSetEventUserptr(void *userptr);
```

Description

Certain events generated by OpenKODE core are *global*; they are not associated with any part of the API such as input/output or sockets that could provide a *userptr* field. This function sets the value to use for the *userptr* field in such events.

A global event has its *userptr* field set to the value supplied to the most recent call to this function at the time the event is generated (which is earlier than the time at which the event is processed by the application). If there has not been any call to this function, the value `KD_NULL` is used.

7.3.3. kdDefaultEvent

Perform default processing on an unrecognized event.

Synopsis

```
void kdDefaultEvent(const KDEvent *event);
```

Description

This function is used to perform default processing on an event returned by `kdWaitEvent` or passed to a callback installed with `kdInstallCallback` that the caller does not recognize or does not want to process.

If the event is `KD_EVENT_QUIT`, then `kdDefaultEvent` has the same effect as a call to `kdExit` with a parameter of `0`.

7.3.4. kdPumpEvents

Pump the event queue, performing callbacks.

Synopsis

```
KDint kdPumpEvents(void);
```

Description

This function performs an *event pump*. Each event in order in the queue whose type and userptr combination is callback enabled is delivered by the applicable installed callback. `kdDefaultEvent`). The callback is in the same context as the caller of `kdPumpEvents`.

Any non-callback enabled event is left in the queue.

The function returns after processing all callback enabled pending events, or immediately if there is none.

Return value

The function returns 0 on success, otherwise, it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM`	OpenKODE Core ran out of resources when queuing events. This error need not be fatal; once it is able to allocate memory again, the event system will continue to function normally. However, one or more events may have been lost.

Rationale

The prioritizing of callback enabled events over non-callback enabled events (the latter are left in the queue by this function) is to allow for `kdPumpEvents` being used in the middle of an application's render loop in order to process events that need attention in a shorter time than the render loop's execution time. Assuming such events are callback enabled, this prioritization ensures that they do not get "stuck" behind non-callback enabled events in the queue.

7.3.5. kdInstallCallback

Install or remove a callback function for event processing.

Synopsis

```
typedef void (KDCallbackFunc)(const KDEvent *event);
```

```
KDint kdInstallCallback(KDCallbackFunc *func, KDint eventtype, void
*eventuserptr);
```

Description

This function installs or removes a callback function for a particular set of event type and user pointer combinations, as specified by the *`eventtype`* and *`eventuserptr`* parameters. Setting *`eventtype`* to `0` matches any event type. Setting *`eventuserptr`* to `NULL` matches any user pointer.

Where a particular event type and user pointer combination would be covered by more than one of the calls made to `kdInstallCallback`, the information from the most recent call is the one which is used.

The *`func`* parameter specifies the callback function to use for the set of event type and user pointer combinations. A value of `KD_NULL` specifies that the event type and user pointer combination is enabled for reporting by `kdWaitEvent`, and not handled by a callback at all. This is the initial state of all event type and user pointer combinations. Any other value stops events of the specified type and user pointer combination being returned by `kdWaitEvent`.

The specification of a callback function is described by the typedef above. Thus the callback function is passed a pointer to a `KDEvent` struct that describes the event.

The callback is called whenever an event pump occurs, which is in the `kdWaitEvent` and `kdPumpEvents` functions. The callback is made in the same context as the caller of whichever of these two functions caused the event pump.

If the callback function does not recognize the event passed to it, or does not want to process it, it must call `kdDefaultEvent` with it before returning, otherwise an action requested by the user (by pressing a key or clicking an on-screen button or other UI interaction) may be lost if the action is handled by the underlying OS when the application has decided not to handle it.

If *`func`* is not `KD_NULL` or a pointer to a function whose type matches the ty[edef above, then undefined behavior results when the event system attempts to handle an event of type and userptr combination covered by the newly installed callback.

Return value

The function returns 0 on success, otherwise, it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`. On failure, the function has not changed the callback state of any event type and user pointer combination.

Error codes

`KD_ENOMEM`	Out of memory.

7.3.6. kdCreateEvent

Create an event for posting.

Synopsis

```
KDEvent *kdCreateEvent(void);
```

Description

To post an event, the caller uses this `kdCreateEvent` function, sets the fields of the returned `KDEvent` appropriately, and then either calls `kdPostEvent`, or calls `kdFreeEvent` to abandon the newly constructed event.

Return value

On success, the function returns the pointer to a new `KDEvent`, which remains valid until it is used in a `kdPostEvent` call or a `kdFreeEvent` call. The new event has its *`timestamp`* field set to 0; other fields have undefined values. On failure, the function returns `KD_NULL` and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM`	Out of memory.

Future directions

In any update to OpenKODE on release of OpenSL ES 1.0 and OpenMAX AL 1.0, the specification of the functions `kdCreateEvent`, `kdPostEvent` and `kdFreeEvent` will be updated to allow them to be used from the context in which an OpenSL ES or OpenMAX AL callback occurs. That context is implementation dependent and may be different from the OpenKODE single threaded application context.

7.3.7. kdPostEvent

Post an event into the queue.

Synopsis

```
KDint kdPostEvent(KDEvent *event);
```

Description

This function posts the event pointed to by the *event* parameter, which is one returned by `kdCreateEvent`, although the fields in the KDEvent structure can have been altered to any values. If the *timestamp* field is 0, `kdPostEvent` stores the current time at some point during this `kdPostEvent` call (as returned by `kdGetTimeUST`) into that field. The event is otherwise unaltered by `kdPostEvent`.

Any event type may be posted. The event may have any *userptr* value, even if the event is of a type defined in this specification. Thus care must be taken to set the *userptr* field and event data to values that are expected by the application code that handles the event type being posted.

Specification of each event type and its *userptr* and event data elsewhere in this document refers to events generated by OpenKODE Core; this specifically excludes events posted to `kdPostEvent` by the application.

The event data may be in any of the event data structures detailed elsewhere in this specification, or in the *user* element of the event data, which has this type:

```
typedef struct KDEventUser {
    union {
        KDint64 i64;
        void *p;
        struct {
            KDint32 a;
            KDint32 b;
        } i32pair;
    } value1;
    union {
        KDint64 i64;
        struct {
            union {
                KDint32 i32;
                void *p;
            } value2;
            union {
                KDint32 i32;
                void *p;
            } value3;
        } i32orp;
    } value23;
} KDEventUser;
```

Once the event has been passed to `kdPostEvent`, it is “owned” by the OpenKODE Core event system. If the application attempts to access or free it after the call to `kdPostEvent`, undefined behavior results. This is the case even if `kdPostEvent` failed.

If *event* is not an event struct pointer returned by an earlier `kdCreateEvent`, or it has already been passed to `kdPostEvent` or `kdFreeEvent`, then undefined behavior results.

Return value

The function returns 0 on success, otherwise, it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM`	Out of memory.

Future directions

In any update to OpenKODE on release of OpenSL ES 1.0 and OpenMAX AL 1.0, the specification of the functions `kdCreateEvent`, `kdPostEvent` and `kdFreeEvent` will be updated to allow them to be used from the context in which an OpenSL ES or OpenMAX AL callback occurs. That context is implementation dependent and may be different from the OpenKODE single threaded application context.

7.3.8. kdFreeEvent

Abandon an event instead of posting it.

Synopsis

```
void kdFreeEvent(KDEvent *event);
```

Description

This function frees an event in the case that the caller decides not to post it. The event to be freed is pointed to by the *`event`* parameter, which was returned by `kdCreateEvent`.

If *`event`* is not an event struct pointer returned by an earlier `kdCreateEvent`, or it has already been passed to `kdPostEvent` or `kdFreeEvent`, then undefined behavior results.

Future directions

In any update to OpenKODE on release of OpenSL ES 1.0 and OpenMAX AL 1.0, the specification of the functions `kdCreateEvent`, `kdPostEvent` and `kdFreeEvent` will be updated to allow them to be used from the context in which an OpenSL ES or OpenMAX AL callback occurs. That context is implementation dependent and may be different from the OpenKODE single threaded application context.

8. System events

8.1. Introduction

OpenKODE Core exposes certain system events to the application programmer, and these events are documented here.

Some events are exposed as normal OpenKODE Core events, documented below. Others are exposed as inputs in a `KD_IOGROUP_EVENT` I/O group, also documented below.

8.2. Events

8.2.1. KD_EVENT_QUIT

Event to request to quit application.

Synopsis

```
#define KD_EVENT_QUIT 1
```

Description

This event type is generated by OpenKODE Core (typically as the result of a request from the underlying OS) to signal that the application should quit. The event has no associated data, but the event's *`userptr`* field is set to the value supplied to the most recent call to `kdSetEventUserptr`, or `KD_NULL` if none.

Application asking to close itself

An application can post this event to itself using `kdPostEvent`. It is up to the application to ensure that the event's *`userptr`* field is set to a value that the event's handler code is expecting (if any).

8.2.2. KD_EVENT_PAUSE

Application pause event.

Synopsis

```
#define KD_EVENT_PAUSE 3
```

Description

This event type, which has no associated data, is generated by OpenKODE (typically as the result of a request from the underlying OS) to signal that the application should pause, releasing any resources that can be reasonably released. After this event has been processed, no further event will arrive for the application to process until either a `KD_EVENT_QUIT` tells the application to quit, or a `KD_EVENT_RESUME` tells the application to resume execution. This event is a global event, and as such its *`userptr`* field is set to the value supplied to the most recent call to `kdSetEventUserptr` at the time the event was generated.

8.2.3. KD_EVENT_RESUME

Application resume event.

Synopsis

```
#define KD_EVENT_RESUME 4
```

Description

This event type, which has no associated data, is generated by OpenKODE (typically as the result of a request from the underlying OS) to signal that the application should resume execution after an earlier `KD_EVENT_PAUSE` event. It is possible to receive a `KD_EVENT_RESUME` without an earlier `KD_EVENT_PAUSE`. This event is a global event, and as such its *`userptr`* field is set to the value supplied to the most recent call to `kdSetEventUserptr` at the time the event was generated.

8.3. I/O groups and items

8.3.1. KD_IOGROUP_EVENT

I/O group for OpenKODE Core system events implemented as inputs.

Synopsis

```
#define KD_IOGROUP_EVENT 0x100
#define KD_IO_EVENT_USING_BATTERY       (KD_IOGROUP_EVENT + 0)
#define KD_IO_EVENT_LOW_BATTERY         (KD_IOGROUP_EVENT + 1)
```

Description

This I/O group defines inputs which implement several OpenKODE Core system events, allowing the application to poll the state using `kdInputPoll*` functions, as well as or instead of enabling the events using `kdInputEventEnable`.

All inputs are mandatory, however it is undefined for each one whether it gives useful information, or is always set to the same value.

I/O items

index	**type**	**range**	**usage**
`KD_IO_EVENT_USING_BATTERY`	mandatory binary input	0..1	1 if using battery, 0 if using mains power. Where an implementation is not able to give this information, this input is always 0.
`KD_IO_EVENT_LOW_BATTERY`	mandatory binary input	0..1	1 if battery is low, 0 otherwise. It is undefined what the threshold is for “low”. Where an implementation is not able to give this information, this input is always 0.

9. Application startup and exit.

9.1. Introduction

Like a standard [C89] program, an OpenKODE Core application has a single top-level function which, in OpenKODE Core's case, is called `kdMain`. A library function `kdExit` is provided to exit from the application.

9.2. Functions

9.2.1. kdMain

The application-defined main function.

Synopsis

```
KDint kdMain(KDint argc, const KDchar **argv);
```

Description

This function is implemented *by the application*, and is not provided by the OpenKODE implementation. It is the application's single entry point.

`argv` is an array of size *`argc`*+1, containing pointers to *`argc`* program arguments, plus a terminating `KD_NULL` (in `argv[argc]`).

It is undefined whether and how arguments can be passed to an OpenKODE program, but it is defined that, if *`argc`* is not 0, then *`argv`*`[0]` is some form of the program name (or an empty string), and further elements of *`argv`* are program parameters.

If the application attempts to modify the *`argv`* array or the strings it points to, undefined behavior results.

Return value

`kdMain` returning is equivalent to calling `kdExit`, using `kdMain`'s return value as its parameter.

Rationale

`kdMain` is based on [C89] `main`. [C89] allows the application to modify the *`argv`* array and the strings it points to.

9.2.2. kdExit

Exit the application.

Synopsis

```
void kdExit(KDint status);
```

Description

This function causes the application to exit immediately with exit status *`status`*. It is undefined what semantics if any the OpenKODE Core implementation attaches to the exit status, except that 0 signifies success.

Rationale

`kdExit` is based on [C89] `exit`, but with the additional [POSIX] semantics of an exit status of 0 meaning success.

When using C++ with OpenKODE Core, it is possible that automatic variables are not destroyed, the same as `exit`.

10. Utility library functions

10.1. Introduction

The functions in this section are miscellaneous library functions, primarily for number-to-string and string-to-number conversion, but also including an integer absolute number function, and a function that returns (non-pseudo-) random data.

OpenKODE Core does not provide the [C89] `sprintf` or [C99] `snprintf` functions, as it was judged that the implementation burden would be too great where the operating system's C library does not already provide a conformant implementation.

Instead, `kdLtostr`, `kdUltostr` and `kdFtostr` provide limited subsets of `snprintf`'s functionality regarding integer, unsigned integer and float conversion respectively.

10.2. Functions

10.2.1. kdAbs

Compute the absolute value of an integer.

Synopsis

```
KDint kdAbs(KDint i);
```

Description

This function computes the absolute value of the integer parameter *`i`*.

Return value

The function returns the absolute value of *`i`*. If *`i`* is `KDINT_MIN`, the returned value is undefined.

Rationale

`kdAbs` is based on the [C89] function `abs`.

10.2.2. kdStrtof

Convert a string to a floating point number.

Synopsis

```
KDfloat32 kdStrtof(const KDchar *s, KDchar **endptr);
```

Description

This function converts the initial part of the string *`s`* to a KDfloat32 number.

The string starts with an arbitrary amount of whitespace (space, form-feed ('`\f`'), newline ('`\n`'), carriage return

\r’), horizontal tab (‘\t’), and vertical tab (‘\v’) characters), then has an optional minus sign (which causes negation of the converted number) or plus sign, then one of:

- a decimal number
- a hexadecimal number
- an infinity
- a NaN.

A *decimal number* consists of one or more decimal digits, possibly including a decimal point character ‘.’, optionally followed by an exponent. An exponent consists of the exponent character ‘e’ or ‘E’, then an optional plus or minus sign, then one or more decimal digits. The exponent indicates multiplication by that power of ten.

A *hexadecimal number* consists of the hexadecimal prefix “0x” or “0X”, then one or more hexadecimal digits, possibly including a hexadecimal point character ‘.’, optionally followed by a binary exponent. A binary exponent consists of the binary exponent character ‘p’ or ‘P’, then an optional plus or minus sign, then one or more decimal digits. The exponent indicates multiplication by that power of two. At least one of the hexadecimal point and the binary exponent must be present.

An *infinity* is either "INF" or "INFINITY", case insensitive.

A *NaN* is "NAN" (case insensitive), optionally followed by an arbitrary sequence of characters enclosed in parentheses. It is not defined which exact NaN representation results, or how the representation relates to the arbitrary sequence of characters when present.

If *endptr* is not KD_NULL, then the function determines a pointer to the first character of the string not included in the conversion, and stores that pointer into the location referenced by *endptr*.

If *s* does not point to a readable string, or *endptr* is not KD_NULL and does not point to a writable pointer location, then undefined behavior results.

Return value

The function returns the converted number.

If the converted value would cause overflow, the function returns plus or minus KD_HUGE_VALF (according to the sign of the converted value) and gives an KD_ERANGE error. If the converted value would cause underflow, the function returns 0, and gives an KD_ERANGE error. In either case, the function stores KD_ERANGE into the error indicator returned by kdGetError.

If conversion fails completely, in that the initial part of the string does not have any of the expected forms above, then *s* is used as the end of conversion pointer (stored into the location referenced by *endptr*), and the function returns 0.

Rationale

kdStrtof is based on the [C99] function strtof, assuming C locale. [C89] has strtod. The C standards do not allow for errno being set when the conversion has failed completely; [POSIX] states that errno *may* be set to EINVAL in that case.

10.2.3. kdStrtol, kdStrtoul

Convert a string to an integer.

Synopsis

```
KDint kdStrtol(const KDchar *s, KDchar **endptr, KDint base);
KDuint kdStrtoul(const KDchar *s, KDchar **endptr, KDint base);
```

Description

This function converts the initial part of the string *s* to an integer.

The string starts with an arbitrary amount of whitespace (space, form-feed (‘\f’), newline (‘\n’), carriage return (‘\r’), horizontal tab (‘\t’), and vertical tab (‘\v’) characters), then has an optional minus sign (which causes negation of the converted number) or plus sign. If *base* is 0 or 16, there may then be a “0x” or “0X” prefix, which indicates that the base used for the conversion is 16. Otherwise, if *base* is 0 and the next character is ‘0’, then the base used for conversion is 8. Otherwise, if *base* is 0, the base used is 10.

The remainder of the string is converted to an integer using the specified *base* (or, if that is 0, the base as specified above), stopping at the first character which is not a valid digit in the base.

If *endptr* is not KD_NULL, then the function determines a pointer to the first character of the string not included in the conversion, and stores that pointer into the location referenced by *endptr*.

If *base* is not 0, 8, 10 or 16, then the value returned by this function, and whether and to what the error indicator returned by kdGetError is set, are undefined. If *s* does not point to a readable string, or *endptr* is not KD_NULL and does not point to a writable pointer location, then undefined behavior results.

Return value

The function returns the converted number.

For kdStrtol, if the converted value is less than KDINT_MIN or greater than KDINT_MAX, then the function returns KDINT_MIN or KDINT_MAX (respectively), and stores KD_ERANGE into the error indicator returned by kdGetError.

For kdStrtoul, if the converted value before any negation (caused by the string having a minus sign in the appropriate place) is greater than KDUINT_MAX, then the function returns KDUINT_MAX and stores KD_ERANGE into the error indicator returned by kdGetError.

If the initial part of the string after any leading whitespace does not contain a valid number, then the function returns 0, and *s* is used as the end of conversion pointer (stored into the location referenced by *endptr*). It is undefined whether it also sets the error indicator returned by kdGetError to some error code.

Rationale

kdStrtol and kdStrtoul are based on the [C89] functions strtol and strtoul (assuming C locale). However the return types for the OpenKODE Core functions are KDint and KDuint rather than long and unsigned long. There are no KDlong and KDulong types. KDint64 and KDuint64 are not used because in many implementations they are longer than the native long type.

The C standards allow for *base* being 0 or any number from 2 to 36. OpenKODE Core allows only 0, 8, 10 or 16.

The C standards do not allow for errno being set when the conversion has failed completely; [POSIX] states that errno *may* be set to EINVAL in that case.

10.2.4. kdLtostr, kdUltostr

Convert an integer to a string.

Synopsis

```
#define KD_LTOSTR_MAXLEN ((sizeof(KDint)*8*3+6)/10+2)
#define KD_ULTOSTR_MAXLEN ((sizeof(KDint)*8*3+9)/10+1)
```

```
KDssize kdLtostr(KDchar *buffer, KDsize buflen, KDint number);

KDssize kdUltostr(KDchar *buffer, KDsize buflen, KDuint number, KDint radix);
```

Description

These functions convert *number* into a string. Each stores the null-terminated string representation of the number into *buffer*, which has maximum length *buflen*. This string representation has no leading 0 characters, except that if *number* is 0, then the textual representation is a single 0 character.

The buffer length given by *buflen* is the maximum number of characters that can be stored in the buffer.

`kdLtostr` treats *number* as signed, and always perform a decimal conversion. If it is negative, the string representation starts with a minus sign, which is followed by the decimal textual representation of the absolute value.

`kdUltostr` always treats *number* as unsigned. *base* specifies the radix to use for the conversion. A value of 0 or 10 specifies decimal, 8 specifies octal, and 16 specifies hexadecimal (with digits a-f in lower case).

The maximum length of the result, including its null termination character, is `KD_LTOSTR_MAXLEN` for `kdLtostr` or `KD_ULTOSTR_MAXLEN` for `kdUltostr`.

If *buflen* is 0, then the functions do nothing other than return -1.

If *buflen* is not 0 and *buffer* does not point to an area of writable memory *buflen* characters long, then undefined behavior results. If *base* is not one of the values specified above, then the function returns an undefined value, and it is undefined what if anything is written into the buffer.

On success, the functions return the length of the stored string, which is strictly less than *buflen* since the returned length does not include the terminating null character. The function fails when the string representation of the number and the terminating null character do not fit into *buflen* characters; in this case the function returns -1, and only *buflen* - 1 characters of the textual representation of the number are written, followed by a terminating null character.

Rationale

OpenKODE Core does not provide the [C89] `sprintf` or [C99] `snprintf` functions, as it was judged that the implementation burden would be too great where the operating system's C library does not already provide a conformant implementation.

`kdLtostr` and `kdUltostr` are intended to provide a subset of `snprintf`'s functionality regarding integer conversion, where `kdLtostr` is analogous to `snprintf` with a format of `"%i"`. and `kdUltostr` with a *base* of 8, 10 or 16 is analogous to `snprintf` with a format of `"%o"`, `"%u"` or `"%x"` respectively. In all cases, C locale is assumed.

The use of "l" rather than "i" in the function names is to provide symmetry with `kdStrtol` and `kdStrtoul`.

Note the difference in return value when the buffer is not large enough; [C99] `snprintf` returns the length the string would have been if the buffer had been long enough, whereas `kdLtostr` and `kdUltostr` return -1.

10.2.5. kdFtostr

Convert a float to a string.

Synopsis

```
#define KD_FTOSTR_MAXLEN 16


KDssize kdFtostr(KDchar *buffer, KDsize buflen, KDfloat32 number);
```

Description

These functions convert *`number`* into a string. Each stores the null-terminated string representation of the number in decimal notation into *`buffer`*, which has maximum length *`buflen`*.

The string representation of *`number`* is calculated as follows:

- If *`number`* is a NaN, the string representation starts with `nan`, and is undefined thereafter (except the overall length limit below).
- If *`number`* is plus infinity, the string representation is either `inf` or `infinity`. If *`number`* is minus infinity, the string representation is either `-inf` or `-infinity`.
- If the absolute value of *`number`* is between 1e9 (exclusive) and 1e-4 (inclusive), or if *`number`* is plus or minus zero, then the representation is a - sign if the number is negative, then digits with no leading zeroes except that there must be at least one digit, then a decimal point character and zero or more digits. Nine significant digits are used, but trailing zeroes after the decimal point are omitted, and if no digits remain after the decimal point, it too is omitted.
- Otherwise, the representation is a - sign if the number is negative, then exactly one digit which is not zero, then a decimal point character, then eight digits (but with trailing zeroes omitted, and if that removes all eight then the decimal point is omitted too), then the character `e`, then the exponent as a plus or minus sign and exactly two digits.

The maximum length of the result, including its null termination character, is `KD_FTOSTR_MAXLEN`.

For a non-zero finite number, the “correct” value of the nine significant mantissa digits is determined by successively multiplying or dividing the number by 10 until (ignoring the sign) it is in the range [1e9,1e10), and then rounding to an integer. However it is permitted for the output of the function to have mantissa digits one out from this “correct” value.

The buffer length given by *`buflen`* is the maximum number of characters that can be stored in the buffer.

If *`buflen`* is 0, then the functions do nothing other than return -1.

If *`buflen`* is not 0 and *`buffer`* does not point to an area of writable memory *`buflen`* characters long, then undefined behavior results. If *`radix`* is not one of the values specified above, then the function returns an undefined value, and it is undefined what if anything is written into the buffer.

On success, the function returns the length of the stored string, which is strictly less than *`buflen`* since the returned length does not include the terminating null character. The function fails when the string representation of the number and the terminating null character do not fit into *`buflen`* characters; in this case the function returns -1, and only *buflen* - 1 characters of the textual representation of the number are written, followed by a terminating null character.

> **Rationale**
>
> OpenKODE Core does not provide the [C89] `sprintf` or [C99] `snprintf` functions, as it was judged that the implementation burden would be too great where the operating system's C library does not already provide a conformant implementation.
>
> `kdFtostr` is intended to provide a subset of `snprintf`'s functionality regarding float conversion (assuming C locale). The conversion rules above are intended to be equivalent to the `snprintf` format `"%.9g"`. Note the difference in return value when the buffer is not large enough; [C99] `snprintf` returns the length the string would have been if the buffer had been long enough, whereas `kdFtostr` returns -1.
>
> Nine significant digits are specified because this is the minimum that guarantees that the original value can be recovered by a conversion with `kdStrtof` or `kdWcstof`.

10.2.6. kdCryptoRandom

Return random data.

Synopsis

```
KDint kdCryptoRandom(KDuint8 *buf, KDsize buflen);
```

Description

This function fills the buffer pointed to by *buf*, of length *buflen* bytes, with random valued bytes.

The random number generator exposed by this function is expected to be initialized from a source of entropy, or otherwise guaranteed to be genuinely unpredictible. However, the function does not block if entropy is exhausted.

On success, the function returns 0. On failure, it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM`	Out of memory or other resource.

> **Rationale**
>
> `kdCryptoRandom` provides a secure random number generator suitable for most cryptographic use. However, since it does not block waiting for entropy, it should not be used for tasks such as generating high-value keys.

11. Locale specific functions

11.1. Introduction

OpenKODE Core does not provide any locale support; where an OpenKODE Core function is based on a C or [POSIX] standard function, it acts like that function in the default C locale.

The functions here allow an application to tailor itself to the (platform's idea of the) language, locale and timezone in which it is running.

11.2. Functions

11.2.1. kdGetLocale

Determine the current language and locale.

Synopsis

```
const KDchar *kdGetLocale(void);
```

Description

This function is used to determine the current language and locale as set on the platform on which the OpenKODE implementation is running.

Return value

On success, the function returns a pointer to a static string which consists of the ISO 639-1 language code, then an underscore, then an ISO 3166-1 alpha-2 location code. The string is null terminated. If the information is not available, an the function returns a pointer to an empty string.

On failure, the function returns `KD_NULL` and stores the error code listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM`	Out of memory or other resource.

Rationale

Despite OpenKODE Core's lack of support for locale, `kdGetLocale` allows an application to determine which language and locale it is running in, so it can tailor its own language- and territory-dependent features.

Example

The string `"en_US"` indicates the English language and USA locale.

11.2.2. kdGetTzOffset

Return the timezone offset.

Synopsis

```
KDint kdGetTzOffset(void);
```

Description

This function is used to determine the difference between the timezone and UTC.

Return value

The function returns the number of seconds that the current timezone is west of (behind) UTC, or 0 if the implementation does not have a concept of timezone.

Rationale

This function is based on the [POSIX] external variable `timezone`, as if already set up by a call to `tzset` or a time conversion function.

12. Memory allocation

12.1. Introduction

The functions here allow an application to allocate and free memory, and are based on [C89] library functions.

12.2. Functions

12.2.1. kdMalloc

Allocate memory.

Synopsis

```
void *kdMalloc(KDsize size);
```

Description

This function allocates a block of memory of at least *size* bytes. The allocated block is suitable to store any C type (including array) that is no longer than *size* bytes.

Unfreed memory is automatically freed when the application exits.

Return value

On success, the function returns a pointer to the allocated memory block. The block contains undefined values. On failure, it returns `KD_NULL` and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

If *size* is 0, either a unique pointer is returned which cannot be used to store any data but can be successfully passed to `kdFree` or `kdRealloc`, or `KD_NULL` is returned; it is undefined which.

Error codes

`KD_ENOMEM`	Not enough space.

Rationale

`kdMalloc` is based on the [C89] and [POSIX] function `malloc`. [C89] does not specify setting `errno` on error.

12.2.2. kdFree

Free allocated memory block.

Synopsis

```
void kdFree(void *ptr);
```

Description

This function frees a block of memory allocated by `kdMalloc` or `kdRealloc`. If *`ptr`* is `KD_NULL`, then the function does nothing.

If *`ptr`* is not `KD_NULL` or a pointer returned by `kdMalloc` or `kdRealloc`, which has not since been supplied to `kdFree` or `kdRealloc`, then undefined behavior results.

After this call, *`ptr`* no longer points to valid memory; attempting to dereference it causes undefined behavior.

Rationale

`kdFree` is based on the [C89] and [POSIX] function `free`.

12.2.3. kdRealloc

Resize memory block.

Synopsis

```
void *kdRealloc(void *ptr, KDsize size);
```

Description

This function resizes the block of memory pointed to by *`ptr`* such that it is at least *`size`* bytes long, and suitable to store any C type (including array) that is no longer than *`size`* bytes. If n is the minimum of *`size`* and the requested size at allocation of the old memory block *`ptr`*, then the first n bytes of the new block have the same values as as the first n bytes of the old block, and any remaining bytes of the new block have undefined values.

The returned pointer may or may not differ from *`ptr`*. If it does differ, then *`ptr`* no longer points to valid memory; attempting to dereference it causes undefined behavior.

If *`ptr`* is `KD_NULL`, then this function behaves like `kdMalloc` for the specified size.

Unfreed memory is automatically freed when the application exits.

If *`buffer`* is not `KD_NULL` or a pointer returned by `kdMalloc` or `kdRealloc`, which has not since been supplied to `kdFree` or `kdRealloc`, then undefined behavior results.

Return value

On success, the function returns a pointer to the allocated memory block. On failure, it returns `KD_NULL` and stores one of the error codes listed below into the error indicator returned by `kdGetError`. In this failure case, the old block pointed to by *`ptr`* (if not `KD_NULL`) is not freed.

If *`size`* is 0, either a unique pointer is returned which cannot be used to store any data but can be successfully passed to `kdFree` or `kdRealloc`, or `KD_NULL` is returned; it is undefined which.

Error codes

`KD_ENOMEM`	Not enough space.

Rationale

`kdRealloc` is based on the [C89] and [POSIX] function `realloc`. [C89] does not specify setting `errno` on error.

13. Thread-local storage.

13.1. Introduction

The functions here provide a facility to get and set a pointer which can be used to locate the application's per-thread data. OpenKODE Core does not yet support threading, however this facility is still useful for the case where the OpenKODE Core implementation does not support non-automatic (i.e. static, global and file scope) variables. (It is expected that most OpenKODE Core implementations will support non-automatic variables, which is indicated by stating support for the KD_KHR_staticdata extension.)

13.2. Functions

13.2.1. kdGetTLS

Get the thread-local storage pointer.

Synopsis

```
void *kdGetTLS(void);
```

Description

This function gets the pointer passed to the most recent call to `kdSetTLS` in the same thread, or `KD_NULL` if that function has not yet been called.

If an implementation-defined method is used to create additional threads, it is undefined whether the thread-local storage pointer is thread-local, or is in fact global across the application process.

Return value

The function returns the thread-local storage pointer, and cannot fail.

Future directions

If a future version of OpenKODE Core supports threads, then the thread-local storage pointer will be per-thread.

13.2.2. kdSetTLS

Set the thread-local storage pointer.

Synopsis

```
void kdSetTLS(void *ptr);
```

Description

This function sets the thread-local storage pointer, such that it is returned by any call to `kdGetTLS` in the same thread, until it is changed again.

If an implementation-defined method is used to create additional threads, it is undefined whether the thread-local storage pointer is thread-local, or is in fact global across the application process.

Future directions

If a future version of OpenKODE Core supports threads, then the thread-local storage pointer will be per-thread.

14. Mathematical functions

14.1. Introduction

The OpenKODE Core mathematical functions provide mathematical operations which, where applicable, give results as specified by [IEEE 754].

Almost all of these functions are based on [C99] equivalents, which in turn are generally float versions of functions in the original [C89] standard. Some of the behavior from [POSIX]'s MX extension is mandated, regarding NANs (not a number values) and infinite values.

See the rationale in Programming Environment for a short discussion of non-compliant but higher performance implementations.

14.2. Constants

`KD_E_F`	The constant e.
`KD_PI_F`	The constant pi.
`KD_PI_2_F`	pi/2
`KD_2PI_F`	2 times pi
`KD_LOG2E_F`	Value of log2e.
`KD_LOG10E_F`	Value of log10e.
`KD_LN2_F`	Value of loge2.
`KD_LN10_F`	Value of loge10.
`KD_PI_4_F`	Value of PI/4.
`KD_1_PI_F`	Value of 1/PI.
`KD_2_PI_F`	Value of 2/PI.
`KD_2_SQRTPI_F`	Value of 2/sqrt(PI).
`KD_SQRT2_F`	Value of sqrt(2).
`KD_SQRT1_2_F`	Value of sqrt(1/2).
`KD_MAXFLOAT`	The largest possible finite float.
`KD_INFINITY`	Positive infinity.
`KD_NAN`	A NAN value.
`KD_HUGE_VALF`	Used as an error value by certain functions; equivalent to `KD_INFINITY`.
`KD_DEG_TO_RAD_F`	Multiply by this number to convert degrees to radians.

KD_RAD_TO_DEG_F	Multiply by this number to convert radians to degrees.

14.2.1. Rationale

The constants above appear in [C99] (without the KD_ prefix and _F suffix), except for KD_2PI_F, KD_DEG_TO_RAD_F and KD_RAD_TO_DEG_F.

OpenKODE Core adds a _F suffix to indicate that the constant is a float constant, in case any future version of OpenKODE Core were to add double support. Unlike in [C99], it is necessary for the constants to be float constants to avoid warnings in the Microsoft C compiler when the warning level is turned up. (The Microsoft C compiler does not have the [C99] constants, so does not normally encounter the problem.)

14.3. Functions

14.3.1. kdAcosf

Arc cosine function.

Synopsis

```
KDfloat32 kdAcosf(KDfloat32 x);
```

Description

This function calculates the arc cosine in radians of *x*, that is the angle whose cosine is *x*.

Return value

On success, the function returns the principal value of the arc cosine of *x*, in the range 0 to PI (inclusive).

If *x* is a NAN, the function returns a NAN.

If *x* falls outside the range -1 to 1, the function returns a NAN value, but it is undefined whether or not it additionally stores KD_EDOM into the error indicator returned by kdGetError.

Error codes

KD_EDOM	Input out of range.

Rationale

kdAcosf is based on the [C99]/[POSIX] function acosf. [C89] has acos. The C standards do not specify any NAN, infinity or domain or range error behavior; [POSIX] and/or [POSIX]'s MX extension specify those issues a little more tightly than OpenKODE Core.

14.3.2. kdAsinf

Arc sine function.

Synopsis

```
KDfloat32 kdAsinf(KDfloat32 x);
```

Description

This function calculates the principal value of the arc sine in radians of `x`, that is the angle whose sine is `x`.

Return value

On success, the function returns the arc sine of `x`, in the range -PI/2 to PI/2 (inclusive).

If `x` is a NAN, the function returns a NAN.

If `x` falls outside the range -1 to 1, the function returns a NAN value, but it is undefined whether it additionally stores `KD_EDOM` into the error indicator returned by `kdGetError`.

Error codes

`KD_EDOM`	Input out of range.

Rationale

`kdAsinf` is based on the [C99]/[POSIX] function `asinf`. [C89] has `asin`. The C standards do not specify any NAN, infinity or domain or range error behavior; [POSIX] and/or [POSIX]'s MX extension specify those issues a little more tightly than OpenKODE Core.

14.3.3. kdAtanf

Arc tangent function.

Synopsis

```
KDfloat32 kdAtanf(KDfloat32 x);
```

Description

This function calculates the principal value of the arc tangent in radians of `x`, that is the angle whose tangent is `x`.

Return value

On success, the function returns the arc tangent of `x`, in the range -PI/2 to PI/2 (inclusive).

If `x` is a NAN, the function returns a NAN.

If `x` is subnormal, it is undefined whether the function succeeds or returns `x`, and it is undefined whether the function stores `KD_ERANGE` into the error indicator returned by `kdGetError`.

Error codes

`KD_ERANGE`	Input is subnormal.

Rationale

`kdAtanf` is based on the [C99]/[POSIX] function `atanf`. [C89] has `atan`. The C standards do not specify any NAN, infinity or domain or range error behavior; [POSIX] and/or [POSIX]'s MX extension specify those issues a little more tightly than OpenKODE Core.

14.3.4. kdAtan2f

Arc tangent function.

Synopsis

```
KDfloat32 kdAtan2f(KDfloat32 y, KDfloat32 x);
```

Description

This function calculates the principal value of the arc tangent in radians of *y*/*x*, that is the angle whose tangent is *y*/*x*, using the signs of both inputs to determine the quadrant of the result.

Return value

On success, the function returns the arc tangent of *y*/*x*, in the range -PI to PI (inclusive). If either input is a NAN, the function returns a NAN. If *y*/*x* is subnormal, it is undefined whether the function succeeds or returns *y*/*x*, and in the latter case it is undefined whether the function stores `KD_ERANGE` into the error indicator returned by `kdGetError`.

Error codes

`KD_ERANGE`	Input is subnormal.

Rationale

`kdAtan2f` is based on the [C99]/[POSIX] function `atan2f`. [C89] has `atan2`. The C standards do not specify any NAN, infinity or domain or range error behavior; [POSIX] and/or [POSIX]'s MX extension specify those issues a little more tightly than OpenKODE Core.

14.3.5. kdCosf

Cosine function.

Synopsis

```
KDfloat32 kdCosf(KDfloat32 x);
```

Description

This function calculates the cosine of *x*.

Return value

On success, the function returns the cosine of *x*, in the range -1 to +1.

If *x* is a NAN, the function returns a NAN.

If *x* is infinite, the return value is undefined, and it is undefined whether the function stores `KD_EDOM` into the error indicator returned by `kdGetError`.

Error codes

`KD_EDOM`	Input is infinite.

Rationale

`kdCosf` is based on the [C99]/[POSIX] function `cosf`. [C89] has `cos`. The C standards do not specify any NAN, infinity or domain or range error behavior; [POSIX] and/or [POSIX]'s MX extension specify those issues a little more tightly than OpenKODE Core.

14.3.6. kdSinf

Sine function.

Synopsis

```
KDfloat32 kdSinf(KDfloat32 x);
```

Description

This function calculates the sine of *x*.

Return value

On success, the function returns the sine of *x*, in the range -1 to +1.

If *x* is a NAN, the function returns a NAN.

If *x* is infinite, the return value is undefined, and it is undefined whether the function stores `KD_EDOM` into the error indicator returned by `kdGetError`.

Error codes

`KD_EDOM`	Input is infinite.

Rationale

`kdSinf` is based on the [C99]/[POSIX] function `sinf`. [C89] has `sin`. The C standards do not specify any NAN, infinity or domain or range error behavior; [POSIX] and/or [POSIX]'s MX extension specify those issues a little more tightly than OpenKODE Core.

14.3.7. kdTanf

Tangent function.

Synopsis

```
KDfloat32 kdTanf(KDfloat32 x);
```

Description

This function calculates the tangent of *x*.

Return value

On success, the function returns the tangent of *x*.

If *x* is a NAN, the function returns a NAN.

If *x* is infinite, the function returns an undefined value, and it is undefined whether it stores `KD_EDOM` into the error indicator returned by `kdGetError`.

If the calculation causes overflow, the function returns plus or minus `KD_HUGE_VALF` (with the same sign as the correct value of the function). It is undefined whether it stores `KD_ERANGE` into the error indicator.

If the correct value would cause underflow, the function returns a representation of the correct result, (either a denormal or 0), but it is undefined whether it stores `KD_ERANGE` into the error indicator.

Error codes

`KD_EDOM`	Input is infinite.
`KD_ERANGE`	Result has overflowed or underflowed.

Rationale

`kdTanf` is based on the [C99]/[POSIX] function `tanf`. [C89] has `tan`. The C standards do not specify any NAN, infinity or domain or range error behavior; [POSIX] and/or [POSIX]’s MX extension specify those issues a little more tightly than OpenKODE Core.

14.3.8. kdExpf

Exponential function.

Synopsis

```
KDfloat32 kdExpf(KDfloat32 x);
```

Description

This function calculates e raised to the power of x.

Return value

On success, the function returns the result of the exponential function. This includes the cases of *x* being infinite of either sign.

If *x* is a NAN, the function returns a NAN.

If *x* is finite and the correct value would cause overflow, the function returns `KD_HUGE_VALF` and stores `KD_ERANGE` the error indicator returned by `kdGetError`.

If *x* is finite and the correct value would cause underflow, the function returns a representation of the correct result, (either a denormal or 0), but it is undefined whether it stores `KD_ERANGE` into the error indicator.

Error codes

`KD_ERANGE`	Result has overflowed or underflowed.

Rationale

`kdExpf` is based on the [C99]/[POSIX] function `expf`. [C89] has `exp`. The C standards specify the range error behavior; [POSIX] and/or [POSIX]'s MX extension specify NAN and infinity behavior.

14.3.9. kdLogf

Natural logarithm function.

Synopsis

```
KDfloat32 kdLogf(KDfloat32 x);
```

Description

This function computes the natural logarithm of *x*.

Return value

On success, the function returns the result of the natural logarithm function. This includes the case of *x* being +inf.

If *x* is a NAN, the function returns a NAN.

If *x* is 0, the function returns `-KD_HUGE_VALF` and stores `KD_ERANGE` into the error indicator returned by `kdGetError`.

If *x* is less than 0 (including the case of -inf), the function returns a NAN value and stores `KD_EDOM` into the error indicator.

Error codes

`KD_EDOM`	Input is negative.
`KD_ERANGE`	Input is 0.

Rationale

`kdLogf` is based on the [C99]/[POSIX] function `logf`. [C89] has `log`. The C standards specify the range error behavior; [POSIX] and/or [POSIX]'s MX extension specify NAN and infinity behavior.

14.3.10. kdFabsf

Absolute value.

Synopsis

```
KDfloat32 kdFabsf(KDfloat32 x);
```

Description

This function calculates the absolute value of its input floating point number.

Return value

The function returns the absolute value of `x`, that is, the same value with the sign changed (if necessary) to positive. This includes the case of `x` being infinite.

If `x` is a NAN value, the function returns a NAN value.

14.3.11. kdPowf

Power function.

Synopsis

```
KDfloat32 kdPowf(KDfloat32 x, KDfloat32 y);
```

Description

This function computes the value of `x` raised to the power of `y`.

Return value

On success, the function returns the value of `x` raised to the power of `y`.

If `x` is finite and negative and `y` is finite and non-integer, the function returns a NAN value, and stores `KD_EDOM` into the error indicator returned by `kdGetError`.

If the correct result would cause overflow, the function returns `KD_HUGE_VALF` with the same sign as the correct result, and stores `KD_ERANGE` into the error indicator.

If the correct result would cause underflow, the function returns a representation of the correct result (either 0 or a denormal), but it is undefined whether it stores `KD_ERANGE` into the error indicator.

If `x` or `y` is a NAN value, the function returns a NAN value, except as allowed below.

If `x` is 0 and `y` is negative, the function returns `KD_HUGE_VALF` (of undefined sign) and stores `KD_ERANGE` into the error indicator.

Rationale

`kdPowf` is based on the [C99] and [POSIX] function `powf`. [C89] has `pow`.

[C99] specifies less than above about edge cases; [POSIX]'s MX extension specifies more.

14.3.12. kdSqrtf

Square root function.

Synopsis

```
KDfloat32 kdSqrtf(KDfloat32 x);
```

Description

This function computes the square root of its input.

Return value

On success, the function returns the square root of *x*. This includes the case of *x* being +inf.

If *x* is a NAN value, the function returns a NAN value.

If *x* is negative (including -inf), the function returns a NAN value, and stores `KD_EDOM` into the error indicator returned by `kdGetError`.

Return value

The square root of x

Rationale

`kdSqrtf` is based on the [C99] and [POSIX] function `sqrtf`. [C89] has `sqrt`.

[C99] does not specify the NAN conditions.

14.3.13. kdCeilf

Return ceiling value.

Synopsis

```
KDfloat32 kdCeilf(KDfloat32 x);
```

Description

This function computes the smallest integer (i.e. nearest to -inf) that is not less than the argument, thus rounding it up.

Return value

On success, the function returns the computed ceiling value.

If *x* is a NAN value, the function returns a NAN value.

If *x* is ±inf, the function returns its input.

Rationale

`kdCeilf` is based on the [C99] and [POSIX] function `ceilf`. [C89] has `ceil`.

[C99] does not specify the NAN conditions. [POSIX] additionally specifies overflow behavior, however that only applies if the largest possible float is not integral, which is not the case for [IEEE 754] floats.

14.3.14. kdFloorf

Return floor value.

Synopsis

```
KDfloat32 kdFloorf(KDfloat32 x);
```

Description

This function computes the largest integer (i.e. nearest to +inf) that is not greater than the argument, thus rounding it down.

Return value

On success, the function returns the computed floor value.

If *x* is a NAN value, the function returns a NAN value.

If *x* is ±inf, the function returns its input.

Rationale

`kdFloorf` is based on the [C99] and [POSIX] function `floorf`. [C89] has `floor`.

[C99] does not specify the NAN conditions. [POSIX] additionally specifies overflow behavior, however that only applies if the largest possible float is not integral, which is not the case for [IEEE 754] floats.

14.3.15. kdRoundf

Round value to nearest integer.

Synopsis

```
KDfloat32 kdRoundf(KDfloat32 x);
```

Description

This function computes the integer closest in value to the argument. If the argument is exactly half way between two integers, the one furthest away from 0 is selected.

Return value

On success, the function returns the computed rounded value.

If *x* is a NAN value, the function returns a NAN value.

If *x* is ±inf, the function returns its input.

Rationale

`kdRoundf` is based on the [C99] and [POSIX] function `roundf`. [C89] has `round`.

[C99] does not specify the NAN conditions. [POSIX] additionally specifies overflow behavior, however that only applies if the largest possible float is not integral, which is not the case for [IEEE 754] floats.

14.3.16. kdInvsqrtf

Inverse square root function.

Synopsis

```
KDfloat32 kdInvsqrtf(KDfloat32 x);
```

Description

This function computes the inverse square root of its input, that is, 1 divided by the square root.

Return value

On success, the function returns the inverse square root of *x*. This includes the case of *x* being +inf (whose inverse square root is 0).

If *x* is a NAN value, the function returns a NAN value.

If *x* is negative (including -inf), the function returns a NAN value, and stores `KD_EDOM` into the error indicator returned by `kdGetError`.

If the result would cause overflow, the function returns +inf. It does not set the error indicator returned by `kdGetError`.

Rationale

`kdInvsqrt` is not based on any C or [POSIX] function, but is provided since some platforms make it easy to accelerate in hardware, and it is a computation often used in games and other graphical applications.

14.3.17. kdFmodf

Calculate floating point remainder.

Synopsis

```
KDfloat32 kdFmodf(KDfloat32 x, KDfloat32 y);
```

Description

This function computes the floating point remainder of *x* divided by *y*. For finite non-zero *y*, the value is the difference between *x* and the closest integral multiple of *y* that has the same sign as and is no greater in magnitude than *x*. Thus the result has the same sign as *x*, and its magnitude is less than *y*'s.

Return value

On success, the return value is the floating point remainder as described above.

If *y* is 0 or *x* is infinite, the function returns a NAN value and stores `KD_EDOM` into the error indicator returned by `kdGetError`.

If *x* or *y* is a NAN value, the function returns a NAN value.

If the correct result would cause underflow, the function returns a representation of the correct result (0 or a denormal), but it is undefined whether it stores `KD_ERANGE` into the error indicator.

Rationale

`kdFmodf` is based on the [C99] and [POSIX] function `fmodf`. [C89] has `fmod`.

[C99] does not specify the NAN conditions.

15. String and memory functions

15.1. Introduction

The functions here copy, scan and compare memory buffers or null-terminated strings. They are based on a subset of the functions found in [C89]'s `<string.h>`, but some functions have been replaced with equivalents of Microsoft functions for added safety.

15.2. Functions

15.2.1. kdMemchr

Scan memory for a byte value.

Synopsis

```
void *kdMemchr(const void *src, KDint byte, KDsize len);
```

Description

This function scans up to *len* bytes of the buffer pointed to by *src* to find the first occurrence of *byte*. Each byte is treated as KDuint8, therefore *byte* must be in the range 0..255 to match anything at all.

If *src* is not a readable buffer of *len* bytes, or up to and including the first byte of value *byte* if shorter, then undefined behavior results.

Return value

The function returns a pointer to the first occurrence of *byte*. If none was found, the function returns `KD_NULL`.

Rationale

`kdMemchr` is based on the [C89] function `memchr`.

15.2.2. kdMemcmp

Compare two memory regions.

Synopsis

```
KDint kdMemcmp(const void *src1, const void *src2, KDsize len);
```

Description

This function compares the two memory buffers *src1* and *src2* up to length *len* bytes.

If either *src1* or *src2* is not a readable buffer of *len* bytes, or up to and including the first mismatching byte if shorter, then undefined behavior results.

Return value

If no differing byte is found in the first *len* bytes of the two memory regions, then the function returns 0.

If the first differing byte has a smaller value in *src1* than in *src2* (considering bytes as unsigned, i.e. type KDuint8), then the function returns a negative value.

If the first differing byte has a larger value in *src1* than in *src2* (considering bytes as unsigned, i.e. type KDuint8), then the function returns a non-zero positive value.

Rationale

`kdMemcmp` is based on the [C89] function `memcmp`.

15.2.3. kdMemcpy

Copy a memory region, no overlapping.

Synopsis

```
void *kdMemcpy(void *buf, const void *src, KDsize len);
```

Description

This function copies *len* bytes from the memory pointed to by *src* into the buffer pointed to by *buf*.

If the two areas overlap, or if *buf* is not a writable buffer of *len* bytes, or if *src* is not a readable buffer of *len* bytes, then undefined behavior results.

Return value

The function returns *buf*.

Rationale

`kdMemcpy` has undefined behavior when the two memory areas overlap. Use `kdMemmove` for this case.

`kdMemcpy` is based on the [C89] function `memcpy`.

15.2.4. kdMemmove

Copy a memory region, overlapping allowed.

Synopsis

```
void *kdMemmove(void *buf, const void *src, KDsize len);
```

Description

This function copies *len* bytes from the memory pointed to by *src* into the buffer pointed to by *buf*. The memory areas are allowed to overlap.

If *buf* is not a writable buffer of *len* bytes, or if *src* is not a readable buffer of *len* bytes, then undefined behavior results.

Return value

The function returns *buf*.

Rationale

kdMemmove behaves correctly when the two memory areas overlap, however this means that it may be slower than kdMemcpy.

kdMemmove is based on the [C89] function memmove.

15.2.5. kdMemset

Set bytes in memory to a value.

Synopsis

```
void *kdMemset(void *buf, KDint byte, KDsize len);
```

Description

This function stores the value *byte* into each of the first *len* bytes of the buffer pointed to by *buf*.

If *buf* is not a writable buffer of *len* bytes, then undefined behavior results.

Return value

The function returns *buf*.

Rationale

kdMemset is based on the [C89] function memset.

15.2.6. kdStrchr

Scan string for a byte value.

Synopsis

```
KDchar *kdStrchr(const KDchar *str, KDint ch);
```

Description

This function scans the null-terminated string *str* to find the first byte which, when considered as a KDchar, matches *ch*. No match is found if *ch* is outside the range -128..+127 if KDchar is signed, or 0..255 if KDchar is unsigned. The null termination byte is included in this scan, and thus matches if *ch* is 0.

If *str* is not a readable buffer up to and including the first match, or up to and including the null termination if no match, then undefined behavior results.

Return value

If a match is found, the function returns a pointer to it. Otherwise, the function returns KD_NULL.

Rationale

`kdStrchr` is based on the [C89] function `strchr`.

15.2.7. kdStrcmp

Compares two strings.

Synopsis

```
KDint kdStrcmp(const KDchar *str1, const KDchar *str2);
```

Description

This function compares two strings byte by byte, until either a mismatch is found, or both strings terminate at the same length.

If *`str1`* and *`str2`* are not both readable buffers up to and including the first mismatched byte, or up to and including the null termination bytes if sooner, then undefined behavior results.

Return value

If no differing byte is found in the strings up to and including their null termination bytes (thus they are exactly the same), then the function returns 0.

If the first differing byte has a smaller value in *`str1`* than in *`str2`* (considering bytes as unsigned, i.e. type KDuint8) (including the case that *`src1`* is shorter than *`str2`*), then the function returns a negative value.

If the first differing byte has a larger value in *`str1`* than in *`str2`* (considering bytes as unsigned, i.e. type KDuint8) (including the case that *`src1`* is longer than *`str2`*), then the function returns a non-zero positive value.

Rationale

`kdStrcmp` is based on the [C89] function `strcmp`.

If there is any danger that in some circumstances one of the strings might not be null terminated, then `kdStrncmp` should be used instead, as this provides a length limit.

15.2.8. kdStrlen

Determine the length of a string.

Synopsis

```
KDsize kdStrlen(const KDchar *str);
```

Description

This function scans the string *`str`* to find its null termination and determine its length.

If *`str`* is not a readable buffer up to and including the null termination byte, then undefined behavior results.

Return value

The function returns the length of the string in bytes, not including the null termination byte.

Rationale

`kdStrlen` is based on the [C89] function `strlen`.

If there is any danger that in some circumstances the string might not be null terminated, then `kdStrnlen` should be used instead, as this provides a length limit.

15.2.9. kdStrnlen

Determine the length of a string.

Synopsis

```
KDsize kdStrnlen(const KDchar *str, KDsize maxlen);
```

Description

This function scans the string *`str`* to find its null termination and determine its length, up to a maximum of *`maxlen`*.

If *`str`* is not a readable buffer of *`maxlen`* bytes, or up to and including the null termination byte if sooner, then undefined behavior results.

Return value

The function returns the length of the string in bytes, not including the null termination byte, or *`maxlen`* if no greater.

Rationale

There is no analog of `kdStrnlen` in any C standard or [POSIX]; it is based on `strnlen`, a GNU extension.

15.2.10. kdStrncat_s

Concatenate two strings.

Synopsis

```
KDint kdStrncat_s(KDchar *buf, KDsize buflen, const KDchar *src, KDsize
srcmaxlen);
```

Description

This function appends at the first *`srclen`* characters of the null-terminated string *`src`* (or the whole string without the null termination if no longer) onto the string already in *`buf`*, null terminating the resulting string in *`buf`*.

If *`buf`* is not a readable and writable buffer of at least *`buflen`* bytes, or it does not contain a null termination character in those *`buflen`* bytes, or *`src`* is not a readable buffer up to the first of a null termination character or *srclen* bytes, or the buffers overlap, then undefined behavior results.

Return value

On success, the function returns 0.

If the resulting string, including the terminating null character, would not fit in *buflen* bytes, then memory is left unchanged and the function fails, returning `KD_ERANGE`.

Rationale

`kdStrncat_s` is based on the Microsoft function `strncat_s`. The Microsoft function has additional null pointer checks.

Like the Microsoft function, and unlike other OpenKODE Core functions, `kdStrncat_s` actually returns its error code of `KD_ERANGE`, rather than storing it in the error indicator returned by `kdGetError`.

OpenKODE Core does not have any analogs of the C functions `strcat` or `strncat`. OpenKODE Core's `kdStrncat_s` is considered safer, as it allows the programmer to specify limits for both the overall buffer length and the length of source string to read.

15.2.11. kdStrncmp

Compares two strings with length limit.

Synopsis

```
KDint kdStrncmp(const KDchar *str1, const KDchar *str2, KDsize maxlen);
```

Description

This function compares two strings byte by byte, until a mismatch is found, or both strings terminate at the same length, or *maxlen* bytes have been compared.

If *str1* and *str2* are not both readable buffers to the earliest of up to and including the first mismatched byte, or up to and including the null termination bytes, or *maxlen* bytes, then undefined behavior results.

Return value

If no differing byte is found in the first *maxlen* bytes of the strings up to and including their null termination bytes (thus they are exactly the same, or the same in the first *maxlen* bytes if at least as long as that), then the function returns 0.

If the first differing byte has a smaller value in *str1* than in *str2* (considering bytes as unsigned, i.e. type KDuint8) (including the case that *src1* is shorter than *str2*), then the function returns a negative value.

If the first differing byte has a larger value in *str1* than in *str2* (considering bytes as unsigned, i.e. type KDuint8) (including the case that *src1* is longer than *str2*), then the function returns a non-zero positive value.

Rationale

`kdStrncmp` is based on the [C89] function `strncmp`.

15.2.12. kdStrcpy_s

Copy a string with an overrun check.

Synopsis

```
KDint kdStrcpy_s(KDchar *buf, KDsize buflen, const KDchar *s);
```

Description

This function copies the null-terminated string *src* into *buf*, but does not write more than *buflen* bytes of *buf*.

If *buf* is not a writable buffer of *buflen* bytes, or *src* is not a readable null-terminated string, or the two buffers overlap, then undefined behavior results.

Return value

On success, the function returns 0.

If *buflen* is 0, the function does not write to memory, and returns `KD_EINVAL`.

If the bytes to copy, including the null termination, would not fit in *buflen* bytes, then *buf*`[0]` is set to 0, the rest of the buffer has undefined values, and the function returns `KD_EINVAL`.

Rationale

`kdStrcpy_s` is based on the Microsoft function `strcpy_s`. The Microsoft function has additional null pointer checks.

Like the Microsoft function, and unlike other OpenKODE Core functions, `kdStrcpy_s` actually returns its error code of `KD_ERANGE`, rather than storing it in the error indicator returned by `kdGetError`.

OpenKODE Core does not have any analogs of the C functions `strcpy` and `strncpy`. OpenKODE Core's `kdStrcpy_s` functions and `kdStrncpy_s` are considered safer, as they allow the programmer to specify a limit for the buffer length.

15.2.13. kdStrncpy_s

Copy a string with an overrun check.

Synopsis

```
KDint kdStrncpy_s(KDchar *buf, KDsize buflen, const KDchar *src, KDsize
srclen);
```

Description

This function copies the first *srclen* bytes of null-terminated string *src* (or the whole string if no longer) into *buf*.

If *buf* is not a writable buffer of *buflen* bytes, or *src* is not a readable buffer up to the first of a null termination character or *srclen* bytes, or the two buffers overlap, then undefined behavior results.

Return value

On success, the function returns 0.

If *buflen* is 0, the function does not write to memory, and returns `KD_EINVAL`.

If the bytes to copy plus the null termination would not fit in *buflen* bytes, then *buf*`[0]` is set to 0, the rest of the buffer has undefined values, and the function returns `KD_EINVAL`.

Rationale

`kdStrncpy_s` is based on the Microsoft function `strncpy_s`. The Microsoft function has additional null pointer checks.

Like the Microsoft function, and unlike other OpenKODE Core functions, `kdStrncpy_s` actually returns its error code of `KD_ERANGE`, rather than storing it in the error indicator returned by `kdGetError`.

OpenKODE Core does not have any analogs of the C functions `strcpy` and `strncpy`. OpenKODE Core’s `kdStrcpy_s` and `kdStrncpy_s` functions are considered safer, as they allow the programmer to specify a limit for the buffer length.

16. Time functions

16.1. Introduction

Here, OpenKODE Core provides functions based on [C89]'s `<time.h>`, where time in seconds since epoch (midnight UST, January 1st 1970) can be obtained and converted into human-readable date and time.

In addition, OpenKODE Core provides unadjusted system time (UST), measured in nanoseconds since some arbitrary datum, which, as well as being fine-grained enough to expose whatever accuracy the platform allows, is defined never to decrease so it could be useful in timestamping multimedia objects and events.

16.2. Functions

16.2.1. kdGetTimeUST

Get the current unadjusted system time.

Synopsis

```
KDust kdGetTimeUST(void);
```

Description

This function returns the current unadjusted system time.

Unadjusted system time measures time in nanoseconds since a datum (for example since the platform was powered up). It is guaranteed to be monotonically increasing, and is not adjusted even if the device's wall clock time is adjusted in some way. UST may or may not stand still while the platform is suspended, but it will not decrease or be reset back as a result of the suspension.

Return value

The function returns the current UST.

16.2.2. kdTime

Get the current wall clock time.

Synopsis

```
KDtime kdTime(KDtime *timep);
```

Description

This function gets the current wall clock time in seconds since midnight UTC, January 1st 1970 (the *epoch*).

If *`timep`* is not `KD_NULL`, then the time is also stored in the location pointed to by *`timep`*, as well as being returned by the function.

No guarantee can be made about the accuracy of the wall clock time returned by this function. In particular, the user may be able to change it to the wrong value, the platform may change it in response to some external time signal, and the platform may have no concept of time zones and thus will return the local time rather than UTC.

If *`timep`* is not `KD_NULL` and does not point to a writable KDtime location, then undefined behavior results.

Return value

The function returns (the platform’s idea of) wall clock time in seconds since midnight UTC, January 1st 1970.

Rationale

`kdTime` is based on the [C89] function `time`. [C89] does not define that time_t (its analog of KDtime) needs to be an arithmetic type; [POSIX] does.

16.2.3. kdGmtime_r, kdLocaltime_r

Convert a seconds-since-epoch time into broken-down time.

Synopsis

```
typedef struct KDtm {
    KDint32 tm_sec;
    KDint32 tm_min;
    KDint32 tm_hour;
    KDint32 tm_mday;
    KDint32 tm_mon;
    KDint32 tm_year;
    KDint32 tm_wday;
    KDint32 tm_yday;
    KDint32 tm_isdst;
} KDtm;
```

```
KDtm *kdGmtime_r(const KDtime *timep, KDtm *result);

KDtm *kdLocaltime_r(const KDtime *timep, KDtm *result);
```

Description

These functions convert the seconds-since-epoch time (as returned by `kdTime`) in the location pointed to by *`timep`* into broken-down time, which it stores in the KDtm structure pointed to by *`result`*.

`kdGmtime_r` writes UTC broken-down time, whereas `kdLocaltime_r` writes local broken-down time. It is undefined whether the platform understands time zones; if not, `kdTime` returns local time and the two functions here produce the same results.

The fields of *`result`* are written as follows:

`tm_sec`	seconds
`tm_min`	minutes
`tm_hour`	hours
`tm_mday`	day of the month
`tm_mon`	month

`tm_year`	year
`tm_wday`	day of the week
`tm_yday`	day in the year
`tm_isdst`	daylight saving time

If *`result`* does not point to a writable KDtm structure, then undefined behavior results.

Return value

The functions return *`result`*.

Rationale

`kdGmtime_r` and `kdLocaltime_r` are based on the [POSIX] functions `gmtime_r` and `localtime_r`. The more familiar functions `gmtime` and `localtime` appear in [C89]; these use a static buffer instead of a buffer supplied by the caller. The KDtm structure type used here is based on [C89]'s struct tm. KDtm may be used as either a struct name or a typedef name.

16.2.4. kdUSTAtEpoch

Get the UST corresponding to KDtime 0.

Synopsis

```
KDust kdUSTAtEpoch(void);
```

Description

This function determines the unadjusted system time (UST) (as returned by `kdGetTimeUST`) at the time that seconds-since-epoch time (as returned by `kdTime`) was 0, by extrapolating back from the current correspondence between the two types of time value.

The relationship between the two types of time value specified by the return from this function only applies between the most recent point at which either was adjusted, through now, up to the next point at which either will be adjusted.

Return value

The function returns UST at KDtime 0, determined by extrapolating back from the current correspondence between the two types of time value. This value can be negative.

Rationale

This function is provided so that an application may convert between the two types of time with some simple arithmetic. However, this conversion will work only for times as far back into the past and as far into the future that neither type of time has been adjusted. KDtime time can be adjusted by the user, or by an automatic adjustment based on an external time signal. KDust time does not get adjusted as such, but is valid only while the platform is considered powered up, and may remain constant while it is suspended, which for the purpose of conversion counts as an adjustment.

17. Timer functions

17.1. Introduction

OpenKODE Core allows the setting of multiple timers, each one-shot or periodic, and each generating an event when the timer expires.

17.2. Functions

17.2.1. kdSetTimer

Set timer.

Synopsis

```
#define KD_TIMER_ONESHOT 0
#define KD_TIMER_PERIODIC_AVERAGE 1
#define KD_TIMER_PERIODIC_MINIMUM 2


typedef struct KDTimer KDTimer;



KDTimer *kdSetTimer(KDint64 interval, KDint periodic, void *eventuserptr);
```

Description

This function creates and sets a timer.

If *periodic* is KD_TIMER_ONESHOT, then the timer fires once, at a time which is as close as possible to and no less than *interval* nanoseconds after the time of this function call. After that, the timer does not fire again.

If *periodic* is KD_TIMER_PERIODIC_AVERAGE, then the timer fires repeatedly with an interval which is as close as possible to the requested *interval* in nanoseconds, such that the average approaches this value.

If *periodic* is KD_TIMER_PERIODIC_MINIMUM, then the timer fires repeatedly with an interval which is as close as possible to the requested *interval* in nanoseconds, but never less than that value.

No limit is defined on how much the actual interval is permitted to differ from the requested interval. But it is expected that an implementation will make a timer as accurate as the underlying operating system’s limitations allow.

In any case, when the timer fires, it generates a KD_EVENT_TIMER event, with its *userptr* field set to the *eventuserptr* passed into this function.

If *periodic* takes any other value, then it is undefined whether the function fails or succeeds, and, if it succeeds, it is undefined whether or when any KD_EVENT_TIMER events are generated.

On success, the function returns a KDTimer* handle for use in a call to kdCancelTimer. On failure, it returns KD_NULL and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

`KD_EIO` General I/O or device failure.

`KD_ENOMEM` Out of memory or other resource, including the case that any implementation-defined limit (at least 16) of set timers has been reached.

17.2.2. kdCancelTimer

Cancel and free a timer.

Synopsis

```
void kdCancelTimer(KDTimer *timer);
```

Description

This function cancels and frees the timer with handle *`timer`*, invalidating the handle, stopping it generating events, and removing any outstanding event generated by this timer from the event queue.

Even for a one-shot timer which has already fired, it is necessary to call this function to ensure that all resources associated with it are freed.

`timer` must be a timer handle returned by an earlier call to `kdSetTimer` and not since freed by a call to `kdCancelTimer`, otherwise undefined behavior results.

17.3. Events

17.3.1. KD_EVENT_TIMER

Timer fire event.

Synopsis

```
#define KD_EVENT_TIMER 0x300
```

Description

When a timer (as configured by `kdSetTimer`) fires, it generates a `KD_EVENT_TIMER` event. No more than one event from the same timer is left outstanding; if one is outstanding when a second one is generated, the first one is removed from the event queue.

The *`userptr`* field in the event is as supplied to `kdSetTimer` when the timer was created.

No data is supplied with this event.

18. File system

18.1. Introduction

OpenKODE Core provides functions to access an abstraction of the platform's file system.

File paths are in a *virtual file system*, which allows portable access to several defined areas, while also allowing non-portable access to the platform's real file system. The virtual file system has its root at `/`, and then has subdirectories such as `/res` and `/data` to allow portable access, and the subdirectory `/native` to allow non-portable access to the platform's real file system if the implementation chooses to allow that.

To ensure portability, as well as only using the defined areas, an application must be constrained by the OpenKODE Core defined limits on path length and characters that may appear in the file paths, when creating a file (including the case of creating a file for delivery along with the application during application development). A portable application reading or otherwise accessing already-present files, and a non-portable application accessing the platform's real file system, do not need to observe these constraints; they are constrained only by the platform's limits.

Functions that open, read, write and close a file are based on the [C89] (and [POSIX]) "stdio" functions, in which the handle to an open file is a FILE*. No analogs of the [POSIX]-only file functions (where the handle to an open file is an integer file descriptor) are provided. No analogs of the [C89] "stdio" formatted reading and writing functions (e.g. `fprintf` and `fscanf`) are provided, as it was judged that they are of limited use in a typical OpenKODE application, and the implementation burden is too great on a platform which does not have conformant implementations of those functions.

Of the other file functions, some are based on [C89] functions, and some are based on [POSIX] functions. `kdGetFree` is not based on either [C89] or [POSIX].

18.2. File path

A file or directory has a name, known as its *file path*. These file paths exist in a virtual file system which has four top-level directories:

`/res`
: Resources: Where the read-only data files that came installed along with the application are stored. This is read only; it is an error to attempt to write to a file accessed via this path.

  This is not necessarily the same location as where the application itself is stored.

`/data`
: A suitable location to store the application's persistent state. Each installed OpenKODE application has its own `/data` area. It is undefined whether `/data` and `/res` are the same location; if they are, then files from each are visible in the other.

`/tmp`
: A suitable location for temporary files. It is undefined whether files stored here are deleted by the platform in between application runs. It is undefined whether this is the same location as `/data`. It is undefined whether multiple applications share the same `/tmp` area.

`/removable`
: The location of any removable media devices on the device. This directory will contain 0 or more subdirectories, each corresponding to a particular removable media that is currently present. They may be named after the media itself, or after the slot.

  It is permitted for implementations to ignore certain removable media if it is not expected that OpenKODE applications will want to access them. For instance, a PC may well want to ignore the floppy drive, so directory listings in /removable are much faster.

/removable is optional. If it is not present, this indicates that the platform has no removable media exposed to OpenKODE Core applications.

/native The contents of /native are undefined by OpenKODE Core. It is intended to allow an implementation to map some non-portable file area if it so chooses. Rules below on the limits and semantics of file and directory names do not apply in /native.

The OpenKODE Core implementation can map anything it likes in /native. One implementation might map the platform's native file system, a second might map nothing at all leaving /native always empty and not able to accept new files or directories, and a third might map some subset of the native file system.

Each of these locations already exists when the OpenKODE application starts (except for /removable where it is not present at all). Subdirectories are supported within each of these locations.

Filenames are defined to be UTF-8, but the only characters defined to be usable within filenames are the letters A-Z and a-z, the digits 0-9, and the characters '.' (period), '_' (underscore) and '-' (hyphen-minus). It is undefined whether other characters are allowed. It is undefined whether filenames are case sensitive.

Forward slash characters are used as the directory separator. Directory separators separate a file path into *components*. Where a file path has adjacent multiple directory separators, it is undefined what it actually refers to.

A file path specified to an OpenKODE Core function is either relative or absolute:

- A relative file path starts with a character other than the directory separator, and is considered relative to the OpenKODE Core current directory as set by kdChdir.

  A relative file path may start with one or more components with a name consisting of two periods ".."; this carries the conventional meaning of going up a level in the directory tree. Where this is used to attempt to go up a level from one of the top-level directories listed above, it is undefined what the file path actually refers to.

- An absolute file path starts with one of the top-level directories listed above.

If any component of a file path is two periods ".." other than as specified above, then it is undefined what the file path refers to.

Any component of a file path may be a single period ".", which refers to the directory specified so far to the left of that. It is undefined whether such a component is ignored completely, or whether it causes an error if what is specified so far to the left of that is not in fact a directory.

A file path is allowed to be up to 48 bytes long, not including the initial top-level directory component (but including the directory separator just after it). This limit applies to an absolute file path; the limit of a relative file path is 47 minus the length (not including the initial top-level directory component) of the current directory name. Where a file path exceeds the limit, it is undefined what it refers to or whether it causes an error on any attempt to use it.

18.2.1. File path limits

It is expected that most implementations will allow a file path considerably more than 48 bytes long, however an application which takes advantage of this will lose some portability.

Note that this set of characters specified above are the minimum requirements for OpenKODE compliance. Implementations should expose the full capabilities of their native file systems in terms of what characters are permitted, however portable applications should be written so that their files with fixed names are named according to these rules. A portable application can, of course, still access files with names outside these rules, and can allow the user to create files with names outside these rules. A portable application should ensure that there are no files whose names only differ by case, and should access those files using the cannonical case, in order to ensure it works on both case-sensitive and case-insensitive file systems.

18.3. Constants

`KD_EOF (-1)`	Used to indicate end-of-file or error conditions.

18.4. Functions

18.4.1. kdFopen

Open a file from the file system.

Synopsis

```
typedef struct KDFile KDFile;


KDFile *kdFopen(const KDchar *pathname, const KDchar *mode);
```

Description

This function opens, and possibly creates, a file in the file system of name *`pathname`*.

`mode` is a pointer to a string whose value determines the mode in which the file is opened, and is one of the following:

`"r"` or `"rb"`	Read: file is opened for reading only
`"w"` or `"wb"`	Write: file is created if necessary, otherwise truncated to 0 length, and opened for writing only
`"a"` or `"ab"`	Append: file is created if necessary, and opened for writing only, positioned at the end of the file
`"r+"` or `"rb+"` or `"r+b"`	Update: file is opened for reading and writing positioned at the start of the file
`"w+"` or `"wb+"` or `"w+b"`	Update with create/truncate: file is created if necessary, otherwise truncated to 0 length, and opened for reading and writing
`"a+"` or `"ab+"` or `"a+b"`	Append: file is created if necessary, and opened for reading and writing positioned at the end of the file

Normally, there is an automatic conversion between the platform specific end-of-line encoding used in files in the file system and a single linefeed character as file data appears to the application. When the *`mode`* string contains the character ‘`b`’, the file is opened in “binary” mode, meaning that this automatic conversion is suppressed.

If the string pointed to by *mode* does not have one of the above values, it is undefined whether the open succeeds, and, if so, what changes are made to the file and whether reading, writing or both are permitted.

Any files left open are automatically flushed and closed at application exit.

If *pathname* and *mode* are not both readable null-terminated strings, then undefined behavior results.

Return value

On success, the function returns a handle to the open file. On failure it returns KD_NULL and stores one of the error codes below into the error indicator returned by kdGetError.

Error codes

KD_EACCES	Permission denied.
KD_EINVAL	The specified mode is invalid.
KD_EISDIR	The specified file path is a directory.
KD_EMFILE	Too many open files.
KD_ENAMETOOLONG	Path name is longer than the implementation-defined limit.
KD_ENOENT	File or directory not found.
KD_ENOMEM	Out of memory or other resource.
KD_ENOSPC	Out of filesystem space.

Rationale

kdFopen is based on the [C89] function fopen. [POSIX] adds the setting of errno on error.

[POSIX] defines additional error codes, some of which are Unix specific and so not applicable to OpenKODE Core, but also including:

- ENFILE (global file table full): folded into KD_EMFILE by OpenKODE Core.
- ENOTDIR (a file path component other than the last is not a directory): folded into KD_ENOENT by OpenKODE Core.
- EROFS (attempt to write on a read-only file system): folded into KD_EACCES by OpenKODE Core.

18.4.2. kdFclose

Close an open file.

Synopsis

```
KDint kdFclose(KDFile *file);
```

Description

This function closes an open file. Whether this function succeeds or not, *file* is no longer valid for use after the call.

If the file is open for writing, any buffered data is written during kdFclose. If the write fails, the function gives an error.

If *file* is not an open file, undefined behavior results.

Return value

On success, the function returns 0, otherwise it returns KD_EOF and stores one of the error codes below into the error indicator returned by kdGetError .

Error codes

KD_EFBIG	File too large.
KD_EIO	I/O error.
KD_ENOMEM	Out of memory or other resource.
KD_ENOSPC	Out of filesystem space.

Rationale

kdFclose is based on the [C89] function fclose. [POSIX] adds the setting of errno on error.

18.4.3. kdFflush

Flush an open file.

Synopsis

```
KDint kdFflush(KDFile *file);
```

Description

This function flushes any buffered written data to the file system for *file*.

If *file* is KD_NULL, then a flush is performed for each open file.

If *file* is not KD_NULL and is not an open file, then undefined behavior results.

Return value

On success, the function returns 0, otherwise it returns KD_EOF and stores one of the error codes below into the error indicator returned by kdGetError .

Error codes

KD_EFBIG	File too large.

`KD_EIO`	I/O error.
`KD_ENOMEM`	Out of memory or other resource.
`KD_ENOSPC`	Out of filesystem space.

Rationale

`kdFflush` is based on the [C89] function `fflush`. [POSIX] adds the setting of `errno` on error.

18.4.4. kdFread

Read from a file.

Synopsis

```
KDsize kdFread(void *buffer, KDsize size, KDsize count, KDFile *file);
```

Description

This function reads data from the open file *`file`*, starting at the file’s position indicator. It reads up to *`count`* multiplied by *`size`* bytes, and stores them into the buffer pointed to by *`buffer`*. It advances the file’s position indicator by the number of bytes actually read. If an error occurs, the file’s position indicator is left in an undefined state.

If either of *`size`* or *`count`* is zero, then this function does nothing (as long as none of the conditions below causes undefined behavior) and returns 0.

If *`file`* is not an open file, or *`buffer`* is not a writable buffer of *`count`* multiplied by *`size`* bytes, then undefined behavior results.

Return value

This function returns the number of complete items (each containing *`size`* bytes) that were read. If that is less than *`count`*, then either the end-of-file has been reached, in which case the function sets the file’s end-of-file indicator (as returned by `kdFEOF`), or an error has occurred, in which case the function sets the file’s error indicator (as returned by `kdFerror`) and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

Refer to `kdGetc`.

Rationale

`kdFread` is based on the [C89] function `fread`. [POSIX] adds the setting of `errno` on error.

18.4.5. kdFwrite

Write to a file.

Synopsis

```
KDsize kdFwrite(const void *buffer, KDsize size, KDsize count, KDFile *file);
```

Description

This function writes data to the open file *`file`*, starting at the file's position indicator. It writes up to *`count`* multiplied by *`size`* bytes, reading them from the buffer pointed to by *`buffer`*. It advances the file's position indicator by the number of bytes actually written. If *`file`* was opened in append mode, then the no position indicator is used, and the data is simply appended to the file. If an error occurs, the file's position indicator is left in an undefined state.

If either of *`size`* or *`count`* is zero, then this function does nothing (as long as none of the conditions below cause undefined behavior) and returns 0.

If *`file`* is not an open file, or *`buffer`* is not a readable buffer of *`count`* multiplied by *`size`* bytes, then undefined behavior results.

Return value

The function returns the number of complete items (each containing *`size`* bytes) that were written. If that is less than *`count`*, then an error has occurred, in which case the function sets the file's error indicator (as returned by `kdFerror`) and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

Refer to `kdPutc`.

Rationale

`kdFwrite` is based on the [C89] function `fwrite`. [POSIX] adds the setting of `errno` on error.

18.4.6. kdGetc

Read next byte from an open file.

Synopsis

```
KDint kdGetc(KDFile *file);
```

Description

This function reads the byte from an open file at the file's position indicator. If successful, it then advances the position indicator.

If *`file`* is not an open file, then undefined behavior results.

Return value

On success, the function returns value of the read byte, as a KDuint8 promoted to KDint (therefore zero extended). Otherwise, it returns `KD_EOF`, and either sets the file's end-of-file indicator (as returned by `kdFEOF`) to indicate that end-of-file has been reached, or it sets the file's error indicator (as returned by `kdFerror`) and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

KD_EBADF	*file* is not open for reading.
KD_EIO	I/O error.
KD_ENOMEM	Out of memory or other resource.

Rationale

kdGetc is based on the [C89] function getc. [POSIX] adds the setting of errno on error.

18.4.7. kdPutc

Write a byte to an open file.

Synopsis

```
KDint kdPutc(KDchar c, KDFile *file);
```

Description

This function writes the byte *c* to the open file *file* at the file’s position indicator. If successful, it advances the file’s position indicator by one. If *file* was opened in append mode, then the no position indicator is used, and the byte is simply appended to the file. If an error occurs, the file’s position indicator is left in an undefined state.

If *file* is not an open file, then undefined behavior results.

Return value

On success, the function returns the byte written, as a KDuint8 promoted to a KDint (i.e. zero extended). On failure, the function returns KD_EOF, sets the file’s error indicator (as returned by kdFerror) and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_EBADF	*file* is not open for writing.
KD_EFBIG	File too large.
KD_EIO	I/O error.
KD_ENOMEM	Out of memory or other resource.
KD_ENOSPC	Out of filesystem space.

Rationale

kdPutc is based on the [C89] function putc. [POSIX] adds the setting of errno on error.

18.4.8. kdFgets

Read a line of text from an open file.

Synopsis

```
KDchar *kdFgets(KDchar *buffer, KDsize buflen, KDFile *file);
```

Description

This function reads data from the open file *file*, starting at the file's position indicator. It reads up to and including the next newline character (after any conversion if the file is not open in binary mode), or up to the end of the file, or up to *buflen* minus one bytes, whichever occurs first. It advances the file's position indicator by the number of bytes actually read. If an error occurs, the file's position indicator is left in an undefined state.

If the function succeeds, a terminating null byte is written just after the data that has been read from the file.

If *file* is not an open file, or *buffer* is not a writable buffer of *buflen* bytes, then undefined behavior results.

Return value

On success, the function returns *buffer*. If the file is at end-of-file, the function sets the file's end-of-file indicator (as returned by `kdFEOF`) and returns `KD_NULL`. If an error occurs, the function sets the file's error indicator (as returned by `kdFerror`) and stores one of the error codes listed below into the error indicator returned by `kdGetError`, and returns `KD_NULL`.

Error codes

Refer to `kdGetc`.

Rationale

`kdFgets` is based on the [C89] function `fgets`. [POSIX] adds the setting of `errno` on error.

18.4.9. kdFEOF

Check for end of file.

Synopsis

```
KDint kdFEOF(KDFile *file);
```

Description

This function returns the end-of-file indicator for *file*, which is set by any of `kdFread`, `kdGetc` or `kdFgets` when the end of the file is encountered.

If *file* is not an open file, then undefined behavior results.

Return value

The function returns `KD_EOF` if the file's end-of-file indicator is set, or 0 otherwise.

Rationale

`kdFEOF` is based on the [C89] function `feof`. However its return value is more precisely defined than `feof`; that function is specified to return any non-zero value if the end-of-file indicator is set.

18.4.10. kdFerror

Check for an error condition on an open file.

Synopsis

```
KDint kdFerror(KDFile *file);
```

Description

This function returns the error indicator for *file*. The error indicator is set by any of the file reading and writing functions when an error is encountered, and is cleared by `kdClearerr`.

If *file* is not an open file, then undefined behavior results.

Return value

The function returns `KD_EOF` if the file's error indicator is set, or 0 otherwise.

Rationale

`kdFerror` is based on the [C89] function `ferror`. However its return value is more precisely defined than `ferror`; that function is specified to return any non-zero value if the end-of-file indicator is set.

18.4.11. kdClearerr

Clear a file's error and end-of-file indicators.

Synopsis

```
void kdClearerr(KDFile *file);
```

Description

This function clears the error and end-of-file indicators for *file*.

If *file* is not an open file, then undefined behavior results.

18.4.12. kdFseek

Reposition the file position indicator in a file.

Synopsis

```
typedef enum {
    KD_SEEK_SET =  0,
    KD_SEEK_CUR =  1,
    KD_SEEK_END =  2
} KDfileSeekOrigin;
```

```
KDint kdFseek(KDFile *file, KDoff offset, KDfileSeekOrigin origin);
```

Description

This function moves the file position indicator for *file*, such that subsequent read or write operations on the file will operate starting at the new file position.

If *origin* is KD_SEEK_SET, then the new file position is *offset* bytes from the start of the file. If *origin* is KD_SEEK_CUR, then the new file position is *offset* bytes from the current file position. If *origin* is KD_SEEK_END, then the new file position is *offset* bytes from the end of the file. *offset* is treated as a signed quantity, even though its type is KDoff, which is unsigned. If *origin* has any other value, the function returns an error. If the resulting file position would be negative or out of range of a KDoff, the function returns an error.

On success, the function clears the end-of-file indicator (returned by kdFEOF) and the error indicator (returned by kdFerror) for the file.

If the file is opened in a writable mode, any data written up to the point of the call to kdFseek is flushed as if by kdFflush.

It is permitted to set the file position indicator beyond the end of the file. If data is subsequently written at that position, the intervening empty space is filled with 0 bytes.

If *file* is not an open file, then undefined behavior results.

Return value

The function returns 0 on success, otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_EFBIG	File too large.
KD_EINVAL	*origin* is invalid, or the new file position would be negative.
KD_EIO	I/O error.
KD_ENOMEM	Out of memory or other resource.
KD_ENOSPC	Out of filesystem space.
KD_EOVERFLOW	The new file position would be a number which cannot be represented in a KDoff.

Rationale

kdFseek is based on the [C89] function fseek, but with a offset parameter of type KDoff analogous to the [POSIX] function fseeko.

18.4.13. kdFtell

Get the file position of an open file.

Synopsis

```
KDoff kdFtell(KDFile *file);
```

Description

This function gets the file position indicator for *`file`*.

If *`file`* is not an open file, then undefined behavior results.

Return value

The function returns the current file offset on success, otherwise it returns `(KDoff)-1` and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EOVERFLOW`	The file position is a number which cannot be represented in a KDoff.

Rationale

`kdFtell` is based on the [C89] function `ftell`, but with a return value of type KDoff analogous to the [POSIX] function `ftello`.

18.4.14. kdMkdir

Create new directory.

Synopsis

```
KDint kdMkdir(const KDchar *pathname);
```

Description

This function creates a new directory whose file path is *`pathname`*. Removing the last component from *`pathname`* must yield a path which is an already existing directory for this function to succeed.

If *`pathname`* does not point to a readable null-terminated string, then undefined behavior results.

Return value

On success the function returns 0, otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EACCES`	Permission denied.
`KD_EEXIST`	A file or directory with the given name already exists.
`KD_EIO`	I/O error.
`KD_ENAMETOOLONG`	Path name is longer than the implementation-defined limit.
`KD_ENOENT`	File or directory not found.
`KD_ENOMEM`	Out of memory or other resource.

`KD_ENOSPC`	Out of filesystem space.

Rationale

`kdMkdir` is based on the [POSIX] function `mkdir`. [POSIX] `mkdir` has an additional parameter to specify the access rights of the new directory; OpenKODE Core has no such concept so omits it.

[POSIX] defines additional error codes, some of which are Unix specific and so not applicable to OpenKODE Core, but also including:

- `ENOTDIR` (a file path component other than the last is not a directory): folded into `KD_ENOENT` by OpenKODE Core.
- `EROFS` (attempt to write on a read-only file system): folded into `KD_EACCES` by OpenKODE Core.

18.4.15. kdRmdir

Delete a directory.

Synopsis

```
KDint kdRmdir(const KDchar *pathname);
```

Description

This function deletes the directory whose path name is specified by *`pathname`*. If the directory is not empty, the function fails.

It is undefined whether attempting to remove a directory currently open with `kdOpenDir` succeeds or fails with an error.

If *`pathname`* does not point to a readable null-terminated string, then undefined behavior results.

Return value

On success the function returns 0, otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EACCES`	Permission denied.
`KD_EBUSY`	*`pathname`* is in use in some undefined way which makes the operation impossible.
`KD_EEXIST`	Directory is not empty.
`KD_EINVAL`	*`pathname`*’s final component is a single period “`.`” (it is undefined whether that situation causes this error or not).
`KD_EIO`	I/O error.
`KD_ENAMETOOLONG`	Path name is longer than the implementation-defined limit.

KD_ENOENT	File or directory not found.
KD_ENOMEM	Out of memory or other resource.

Rationale

kdRmdir is based on the [POSIX] function rmdir.

[POSIX] defines additional error codes, some of which are Unix specific and so not applicable to OpenKODE Core, but also including:

- ENOTDIR (a file path component other than the last is not a directory): folded into KD_ENOENT by OpenKODE Core.
- ENOTEMPTY is a [POSIX] alternative to EEXIST, and is mapped by OpenKODE Core to KD_EEXIST.
- EROFS (attempt to write on a read-only file system): folded into KD_EACCES by OpenKODE Core.

18.4.16. kdRename

Rename a file.

Synopsis

```
KDint kdRename(const KDchar *src, const KDchar *dest);
```

Description

This function renames the file with path name *src* such that it has a new name of *dest*. The path name obtained by removing the final component of *dest* must be a directory. If a file of name *dest* already existed, it is deleted as part of the operation.

It is undefined whether attempting to rename an open file succeeds or fails with an error.

If either of *src* or *dest* is the path name of a directory, it is undefined whether the function fails or not.

If the function fails with any error code other than KD_EIO, then any file or directory named by *dest* remains unchanged.

If either of *src* or *dest* does not point to a readable null-terminated string, then undefined behavior results.

Return value

On success the function returns 0, otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_EACCES	Permission denied.
KD_EBUSY	Either *src* or *dest* is in use in some undefined way which makes the operation impossible.

KD_EINVAL	The operation failed for an undefined reason related to the path names *src* and *dest* and whether they are directories.
KD_EIO	I/O error.
KD_ENAMETOOLONG	Path name is longer than the implementation-defined limit.
KD_ENOENT	File or directory not found.
KD_ENOMEM	Out of memory or other resource.
KD_ENOSPC	Out of filesystem space.

Rationale

kdRename is based on the [C89] and [POSIX] function rename.

[POSIX] defines additional error codes, some of which are Unix specific and so not applicable to OpenKODE Core, but also including:

- ENOTDIR (a file path component other than the last is not a directory): folded into KD_ENOENT by OpenKODE Core.
- EROFS (attempt to write on a read-only file system): folded into KD_EACCES by OpenKODE Core.

18.4.17. kdRemove

Delete a file.

Synopsis

```
KDint kdRemove(const KDchar *pathname);
```

Description

This function deletes the file whose path name is specified by *pathname*.

It is undefined whether this function succeeds when *pathname* specifies a directory.

It is undefined whether attempting to remove an open file succeeds or fails with an error.

If *pathname* does not point to a readable null-terminated string, then undefined behavior results.

Return value

On success the function returns 0, otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_EACCES	Permission denied.

KD_EBUSY	*pathname* is in use in some undefined way which makes the operation impossible.
KD_EIO	I/O error.
KD_ENAMETOOLONG	Path name is longer than the implementation-defined limit.
KD_ENOENT	File or directory not found.
KD_ENOMEM	Out of memory or other resource.

Rationale

kdRemove is based on the [C89] function remove. [POSIX] defines that the function can work on a directory; OpenKODE Core leaves it undefined whether this is the case.

[POSIX] defines additional error codes, some of which are Unix specific and so not applicable to OpenKODE Core, but also including:

- ENOTDIR (a file path component other than the last is not a directory): folded into KD_ENOENT by OpenKODE Core.
- EROFS (attempt to write on a read-only file system): folded into KD_EACCES by OpenKODE Core.

18.4.18. kdTruncate

Truncate or extend a file.

Synopsis

```
KDint kdTruncate(const KDchar *pathname, KDoff length);
```

Description

This function sets the length of the file of name *pathname* to be *length* bytes. If the file was longer than this, it is truncated and the data after that point is discarded. If the file was shorter than this, it is padded with zero bytes.

If *pathname* does not point to a readable null-terminated string, then undefined behavior results.

Return value

On success the function returns 0, otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_EACCES	Permission denied.
KD_EINVAL	The size of the file would be negative or greater tham the maximum file size.
KD_EIO	I/O error.
KD_ENAMETOOLONG	Path name is longer than the implementation-defined limit.

`KD_ENOENT` File or directory not found.

`KD_ENOMEM` Out of memory or other resource.

Rationale

`kdTruncate` is based on the [POSIX] function `truncate`.

[POSIX] defines additional error codes, some of which are Unix specific and so not applicable to OpenKODE Core, but also including:

- `EFBIG` (file would be too big) is listed by [POSIX] as an alternative to `EINVAL` for this particular error condition. OpenKODE Core folds it into `KD_EINVAL`.
- `EISDIR` (named file is a directory): folded into `KD_EACCES` by OpenKODE Core.
- `ENOTDIR` (a file path component other than the last is not a directory): folded into `KD_ENOENT` by OpenKODE Core.
- `EROFS` (attempt to write on a read-only file system): folded into `KD_EACCES` by OpenKODE Core.

18.4.19. kdStat, kdFstat

Return information about a file.

Synopsis

```
typedef struct KDStat {
    KDmode st_mode;
    KDoff st_size;
    KDtime st_mtime;
} KDStat;
```

```
KDint kdStat(const KDchar *pathname, struct KDStat *buf);

KDint kdFstat(const KDFile *file, struct KDStat *buf);
```

Description

This function retrieves information about the specified file or directory. `kdStat` is passed a file path, and retrieves information about the named file or directory. `kdFstat` is passed a KDFile* handle to an open file, and retrieves information about that file.

The filled in KDStat structure contains the following fields:

`st_size` Size of file in bytes. For something that is not a file, the value is undefined.

`st_mtime` Time of last modification as a KDtime time (as returned by `kdTime`).

`st_mode` This field provides information about whether the described file system entity is a file or a directory, and whether it is readable or writable by the application. The value of this field is undefined, but the following macros are provided to interpret it. Each of these macros takes a *`st_mode`* value as its only

argument.

- KD_ISREG returns non-zero if the file system entity is a regular file.
- KD_ISDIR returns non-zero if the file system entity is a directory.
- KD_READABLE returns non-zero if the permissions of the file or directory are such that it can be read by the application.
- KD_WRITABLE returns non-zero if the permissions of the file or directory are such that it can be written by the application.

Note that it is possible for a file system entity to be something other than a regular file or a directory. Attempting to use such a non-file non-directory entity in any OpenKODE Core function other than kdStat has undefined semantics regarding whether the function fails or succeeds and what information is returned.

If *pathname* does not point to a readable null-terminated string, or *file* is not an open file, or *buf* does not point to a writable KDStat structure, then undefined behavior results.

Return value

On success the function returns 0, otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_EACCES	Permission denied.
KD_EIO	I/O error.
KD_ENAMETOOLONG	Path name is longer than the implementation-defined limit.
KD_ENOENT	File or directory not found.
KD_ENOMEM	Out of memory or other resource.
KD_EOVERFLOW	The file size in bytes cannot be represented by a KDoff.

Rationale

kdStat is based on the [POSIX] function stat. kdFstat is inspired by the [POSIX] function fstat, but it uses a KDFile* rather than an integer file descriptor as the handle to the file, since the latter does not exist in OpenKODE Core.

[POSIX] defines additional error codes, some of which are Unix specific and so not applicable to OpenKODE Core, but also including:

- ENOTDIR (a file path component other than the last is not a directory): folded into KD_ENOENT by OpenKODE Core.

OpenKODE Core's KDStat is analogous to [POSIX]'s struct stat, but [POSIX] defines additional fields which are not applicable to OpenKODE Core. [POSIX] also defines more information which can be obtained from the *st_mode* field.

Because of the general rule that an implementation may change the order of fields and add extra ones in an OpenKODE Core structure, an implementation may choose to make KDStat the same as the OS's struct stat, such that the OS's `stat` and `fstat` can be used directly, as long as the types KDmode, KDoff and KDtime match the OS's corresponding struct stat types.

18.4.20. kdOpenDir

Open a directory ready for listing.

Synopsis

```
typedef struct KDDir KDDir;


KDDir *kdOpenDir(const KDchar *pathname);
```

Description

This function opens a KDDir* handle for the directory of path name *pathname*, and positions it at the first entry.

Any directory left open is automatically closed at application exit.

The function fails with `KD_ENOENT` if the specified directory is `/` (or resolves to that after `..` removal).

If *pathname* does not point to a readable null-terminated string, undefined behavior results.

Return value

On success the function returns the directory handle, otherwise it returns `KD_NULL` and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EACCES`	Permission denied.
`KD_EIO`	I/O error.
`KD_ENAMETOOLONG`	Path name is longer than the implementation-defined limit.
`KD_ENOENT`	File or directory not found.
`KD_ENOMEM`	Out of memory or other resource.

Rationale

`kdOpenDir` is based on the [POSIX] function `opendir`.

[POSIX] defines additional error codes, some of which are Unix specific and so not applicable to OpenKODE Core, but also including:

- `ENOTDIR` (a file path component is not a directory): folded into `KD_ENOENT` by OpenKODE Core.

18.4.21. kdReadDir

Return the next file in a directory.

Synopsis

```
typedef struct KDDirent {
    KDchar d_name[256];
} KDDirent;
```

```
KDDirent *kdReadDir(KDDir *dir);
```

Description

This function reads the next entry in the specified directory, and advances the position. It returns a pointer to a KDDirent structure describing the directory entry; the pointer remains valid until the next call to `kdReadDir` or `kdCloseDir` with the same *`dir`* parameter.

It is undefined whether entries are returned for `.` and `..` by this function.

If a file or subdirectory is created or deleted in the directory subsequent to the `kdOpenDir` call which created *`dir`*, then it is undefined whether an entry is returned for that file/subdirectory.

The returned KDDirent contains a field *`d_name`*, which contains the null-terminated name of the directory entity, relative to the directory being listed (thus no path separator characters in the name). The 256 length of this field in the synopsis above is arbitrarily chosen and does not reflect any length limit to this field.

If *`dir`* is not an open directory, then undefined behavior results.

Return value

On success the function returns a KDDirent pointer. If the end of the directory listing has been reached, the function returns `KD_NULL`. On other failure, it returns `KD_NULL` and stores one of the error codes listed below into the error indicator returned by `kdGetError`. To tell the difference between end of directory and error, the application must use `kdSetError` to zero the error indicator first.

Error codes

`KD_EIO`	I/O error.
`KD_ENOMEM`	Out of memory or other resource.

Rationale

`kdReadDir` is based on the [POSIX] function `readdir`.

18.4.22. kdCloseDir

Close a directory.

Synopsis

```
KDint kdCloseDir(KDDir *dir);
```

Description

This function closes the directory handle `dir` that was opened by `kdOpenDir`.

If `dir` is not an open directory, then undefined behavior results.

Return value

On success, this function returns 0. It cannot fail.

Rationale

`kdCloseDir` is based on the [POSIX] function `closedir`.

[POSIX] defines some error codes for ways in which the function can fail, but these are all inapplicable to OpenKODE Core.

18.4.23. kdGetFree

Get free space on a drive.

Synopsis

```
KDoff kdGetFree(const KDchar *pathname);
```

Description

This function retrieves the free space (in bytes) on the file system containing the file path `pathname`. How the virtual filesystem tree is split into different physical file systems is not defined.

If `pathname` is not a pointer to a readable null-terminated string, then undefined behavior results.

Return value

On success, the function returns the number of bytes of free space. Otherwise, it returns `(KDoff)-1` and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EACCES`	Permission denied.
`KD_EIO`	I/O error.
`KD_ENAMETOOLONG`	Path name is longer than the implementation-defined limit.
`KD_ENOENT`	File or directory not found.
`KD_ENOMEM`	Out of memory or other resource.
`KD_EOVERFLOW`	The free space size cannot be represented by a KDoff.

18.4.24. kdChdir

Change the current directory.

Synopsis

```
KDint kdChdir(const KDchar *pathname);
```

Description

This function changes OpenKODE Core’s current directory, which is used as the base for a relative path used in any subsequent function which takes a path name, including this one. If a relative path is supplied, then the current directory is set to the resolved absolute path after processing any leading “`..`” components in the relative path name. In both the absolute and relative cases, the current directory is set to the resolved absolute path after removing any “`.`” components.

If the function fails, it leaves the current directory unchanged.

On application startup, the OpenKODE Core current directory is `/res`.

If *`pathname`* does not point to a readable null-terminated string, then undefined behavior results.

Return value

On success the function returns 0, otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EACCES`	Permission denied.
`KD_EIO`	I/O error.
`KD_ENAMETOOLONG`	Path name is longer than the implementation-defined limit.
`KD_ENOENT`	Directory not found.

Rationale

`kdChdir` is based on the [POSIX] function `chdir`.

[POSIX] defines additional error codes, some of which are Unix specific and so not applicable to OpenKODE Core, but also including:

- `ENOTDIR` (a file path component is not a directory): folded into `KD_ENOENT` by OpenKODE Core.

18.4.25. kdGetCwd

Get the current directory.

Synopsis

```
KDchar *kdGetCwd(KDchar *buf, KDsize buflen);
```

Description

This function retrieves the absolute path name of the OpenKODE Core current directory, as stored by `kdChdir`. The current directory is `/res` before the first successful call to that function. If successful, it stores the null-terminated path name of the current directory in the buffer pointed to by *`buf`*.

If *`buflen`* is less than the length of the current directory plus one (for the terminating null character), then the function fails with an error. If *`buf`* does not point to a writable buffer of *`buflen`* bytes, then undefined behavior results.

Return value

On success the function returns *`buf`*, otherwise it returns `KD_NULL` and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EINVAL`	*`buflen`* is 0.
`KD_ENOMEM`	Out of memory.
`KD_ERANGE`	*`buflen`* is not 0, but is less than the number of bytes required to store the path name, including its terminating null character.

Rationale

`kdGetCwd` is based on the [POSIX] function `getcwd`.

[POSIX] defines an additional error code `EACCES`. The absence of this from OpenKODE Core implies that, on an OS where these errors could occur because `getcwd` (or equivalent) actually accesses the file system, the OpenKODE Core implementation must either keep its own record of the current directory, or it must turn those errors into some default current directory such as `/res`.

The limits on what processing is performed on the path name by `kdChdir` before storing it as the current directory are intended to prevent an implementation for example returning a current directory within `/data` when it was set within `/res` if they happen to be the same part of the OS’s file system. This is another reason why an OpenKODE implementation is likely to need to keep its own record of the current directory.

19. Network sockets

19.1. Introduction

OpenKODE Core provides a network sockets API based on BSD/[POSIX] sockets. Notable differences are:

- OpenKODE Core is event based. Thus, blocking mode sockets are not feasible, and non blocking mode sockets signal the completion of an action or availability of data or buffer space by posting an OpenKODE event. This takes the place of using `select` or `poll` to wait for the completion of an action or availability of data or buffer space in a BSD/[POSIX] non blocking socket.
- The BSD/[POSIX] concepts of address family (domain), type and protocol are combined into a single parameter when creating a socket. Currently only TCP over IPv4 and UDP over IPv4 are supported.

A TCP socket is a *reliable*, *connection-based* socket. This means that data sent is reliably delivered, otherwise an error is generated, and that no data can be sent until the socket is connected. A connection is established when the “server” end is listening (it has successfully called `kdSocketListen`), the “client” end attempts to connect (it calls `kdSocketConnect`), and the server end accepts the connection (it calls `kdSocketAccept`).

A UDP socket is a *unreliable*, *connectionless* socket. This means that no guarantees are made about data delivery, and no connection is necessary before sending or receiving data. Before a connectionless socket can receive any data, it must be bound to a local address using `kdSocketBind`. Once bound, `kdSocketSendTo` is used to send data and specify the host and port to send it to; `kdSocketRecvFrom` is used to receive data together with an indication of the sending host and port; `kdSocketRecv` is used to receive data when the sending host and port are of no interest. `kdSocketConnect` may be used on a connectionless socket, just to associate a sending address with the socket so that `kdSocketSend` may be used instead of `kdSocketSendTo`.

19.1.1. Event generation

Where it is stated that an event is generated when a particular condition is met, the implementation is in fact free to delay generating the event until the next call to `kdWaitEvent` or `kdPumpEvents` (if not already in such a call). The application cannot tell the difference between this and the event being generated immediately.

19.2. Types

19.2.1. KDSockaddr_ structures

Struct types for socket addresses.

Description

The general form of `KDSockaddr` is:

```
typedef struct KDSockaddr {
    KDint16 sa_family;
    KDuint8 sa_data[14];
} KDSockaddr;
```

This struct is used simply to obtain the address family, which indicates which specialized address struct the address of a KDSockaddr needs to be cast to. *`sa_family`* is the address family of the address. OpenKODE supports only `KD_AF_INET`, which indicates that the address is stored in a KDSockaddr_in struct, specified below.

The size of this struct bears no relation to the size of the specialized address struct which is actually used for the address.

KDSockaddr_in struct

This struct stores an address for use with IPv4 based protocols, such as TCP and UDP over IPv4. Here an address consists of an IP address and a port number.

```
#define KD_AF_INET 2
typedef struct KDSockaddr_in {
    KDint16 sin_family;
    KDuint32 sin_address;
    KDuint16 sin_port;
} KDSockaddr_in;
```

sin_family Set to KD_AF_INET. This is the same field as *sa_family* when a pointer to KDSockaddr_in is cast to a pointer to KDSockaddr.

sin_address IP address in network byte order

sin_port Port number in network byte order

19.3. Functions

19.3.1. kdNameLookup

Look up a hostname.

Synopsis

```
KDint kdNameLookup(KDint type, const KDchar *hostname, void *eventuserptr);
```

Description

This function initiates the retrieval of the network address of the given *hostname*, for example using DNS.

type is the type of socket that will use the returned address, from which the function deduces the address family in which to look for the name. It is one of the following values:

KD_SOCK_TCP or KD_SOCK_UDP Search for the name in the IPv4 address family. These two values for *type* can be used interchangeably. For IPv4, *hostname* may be an IP address in textual “dotted quad” notation instead of a name.

If the function does not fail immediately, results are returned by one or more KD_EVENT_NAME_LOOKUP_COMPLETE events; the *userptr* value of each event is as supplied to this function in the *eventuserptr* parameter.

The limit on simultaneous lookups in progress is undefined. An attempt to exceed the implementation-defined maximum results in this function failing immediately with an error as below.

If *type* is not a socket type that supports name lookup, then the function fails with an error as below. If *name* does not point to a readable null-terminated string, then undefined behavior results.

Return value

On immediate failure, `kdNameLookup` returns -1 and stores one of the error codes below into the error indicator returned by `kdGetError`. In particular, if the implementation does not support networking at all, the function fails with error `KD_ENOSYS`. Otherwise, the function returns 0 to indicate that it has successfully initiated the lookup operation.

Error codes

`KD_EBUSY`	The maximum number of simultaneous lookups are already in progress.
`KD_EINVAL`	Socket type unknown or does not support name lookup.
`KD_ENOMEM`	Not enough space.
`KD_ENOSYS`	Networking not supported at all.

Rationale

`kdNameLookup` is based on the functionality of BSD/[POSIX] `gethostbyname`, but with different semantics such that the address family is specified, and the results are returned asynchronously so the application is not stalled indefinitely.

19.3.2. kdNameLookupCancel

Selectively cancels ongoing kdNameLookup operations.

Synopsis

```
void kdNameLookupCancel(void *eventuserptr);
```

Description

This function cancels any outstanding lookup operations initiated by calls to `kdNameLookup` whose *`eventuserptr`* values match the *`eventuserptr`* supplied to this function. If this function's *`eventuserptr`* is `KD_NULL`, then all outstanding lookup operations are cancelled. This includes removing any pending events from a completed `kdNameLookup` matching this criterion.

The function does nothing and succeeds if *`eventuserptr`* does not match any outstanding lookup operation.

19.3.3. kdSocketCreate

Creates a socket.

Synopsis

```
typedef struct KDSocket KDSocket;


KDSocket *kdSocketCreate(KDint type, void *eventuserptr);
```

Description

This function creates a socket.

`type` specifies the type of the socket, and is one of the following values:

`KD_SOCK_TCP` (0) TCP over IPv4. The socket is connection-based.

`KD_SOCK_UDP` (1) UDP over IPv4. The socket is connectionless.

If *`type`* is not one of the above values, or is one that is not supported on this implementation, the function fails with the error specified below.

The socket is created in an unbound and unconnected state. Data can be sent on a connectionless socket with no further preparation; a `KD_EVENT_SOCKET_WRITABLE` event is generated as soon as a send operation would not block, and a `KD_EVENT_SOCKET_ERROR` event is generated when there is an error.

`eventuserptr` is the value that will be used for the *`userptr`* field in any event associated with the socket. If *`eventuserptr`* is `KD_NULL`, then the function fails with a `KD_EINVAL` error.

Any socket left open at application termination is automatically closed.

Return value

`kdSocketCreate` returns the created socket on success. On failure, the function returns `KD_NULL` and stores one of the error codes listed below into the error indicator returned by `kdGetError`. In particular, if the implementation does not support networking at all, the function fails with error `KD_ENOSYS`.

Error codes

`KD_EACCES` Permission to create a socket of the specified type is denied.

`KD_EINVAL` Unknown socket type, or socket type not supported, or *`eventuserptr`* is `KD_NULL`.

`KD_EIO` General I/O or network error.

`KD_EMFILE` Too many open sockets.

`KD_ENOMEM` Out of memory or buffers.

`KD_ENOSYS` Networking not supported at all.

Rationale

This function is based on the `socket` function in BSD and [POSIX], but with the following differences:

The OpenKODE Core socket API is based around a `KDSocket *` handle, rather than an integer file descriptor.

`kdSocket` combines the BSD/[POSIX] notions of domain (address family), type and protocol into one parameter *`type`*. If a future version of OpenKODE Core was to support other protocols or domains, extra values of *`type`* would be defined.

OpenKODE Core sockets are non-blocking, and use the event system to notify completion of or readiness for an operation. *`eventuserptr`* is an OpenKODE Core addition.

[POSIX] additionally specifies these error codes:

- `ENFILE` (which OpenKODE Core folds into `KD_EMFILE`);
- `EAFNOSUPPORT`, `EPROTONOSUPPORT` and `EPROTOTYPE` (all of which OpenKODE Core folds into `KD_EINVAL`; `EINVAL` is not mentioned in [POSIX]);
- `ENOBUFS` (which OpenKODE Core folds into `KD_ENOMEM`).

Future directions

If a future version of OpenKODE Core supports threading, it may also support blocking sockets, enabled by *`eventuserptr`* being `KD_NULL`.

19.3.4. kdSocketClose

Closes a socket.

Synopsis

```
KDint kdSocketClose(KDSocket *socket);
```

Description

This function closes *`socket`* and frees all resources associated with it.

Any event still in the event queue that was generated by the socket is removed.

If *`socket`* is not a socket, or has already been closed, then undefined behavior results.

Return value

This function returns 0 on success. On failure, it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`. Note that, even on failure, the socket is considered closed.

Error codes

`KD_EIO`	I/O error.

19.3.5. kdSocketBind

Bind a socket.

Synopsis

```
KDint kdSocketBind(KDSocket *socket, const struct KDSockaddr *addr, KDsocklen
addrlen, KDboolean reuse);
```

Description

This function binds *`socket`* to the local address specified in the location pointed to by *`addr`* (which has one of the `KDSockaddr_` types). *`addrlen`* is no less than the length in bytes of the address information.

If *`addr`*`->sa_family` is `KD_AF_INET`, then *`addr`* specifies an IPv4 address, the only address family supported by OpenKODE Core. Then, *`addr`* is considered a pointer to a `KDSockaddr_in` structure, which specifies the local IP address and port number to bind to. If the *`sin_address`* field is `KD_INADDR_ANY (0)`, then the socket is bound to all local IP addresses.

The *`reuse`* parameter determines whether address reuse is to be enabled. If it is 0, there may be a delay between closing a TCP socket and its IP address and port combination becoming available for reuse. If *`reuse`* is non-zero, the IP address and port combination becomes available immediately on a close, but some implementations warn that this could be at the expense of making TCP less reliable.

A successful call of this function leaves the socket in the bound state. For a connectionless socket, this means that the socket can now receive data, and a `KD_EVENT_SOCKET_READABLE` event is generated as soon as data arrives.

If *`socket`* is not a socket, or has already been closed, or *`addr`* is not a readable location of at least *`addrlen`* bytes containing one of the `KDSockaddr_` types, then undefined behavior results.

Return value

This function returns 0 on success. On failure, it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EADDRINUSE`	Address in use.
`KD_EADDRNOTAVAIL`	Address not available on the local platform.
`KD_EAFNOSUPPORT`	*`sin_family`* is not `KD_AF_INET`
`KD_EINVAL`	Socket is already bound to an address, or *`addrlen`* is wrong
`KD_EIO`	General I/O or network error.
`KD_EISCONN`	Socket is already connected
`KD_ENOMEM`	Out of memory or other resource

Rationale

`kdSocketBind` is based on the [POSIX] function `bind`.

[POSIX] defines some additional error codes which are not applicable to the subset of socket functionality that OpenKODE Core provides.

19.3.6. kdSocketGetName

Get the local address of a socket.

Synopsis

```
KDint kdSocketGetName(KDSocket *socket, struct KDSockaddr *addr, KDsocklen
*addrlen);
```

Description

This function stores the local address that *socket* is bound to into the location pointed to by *addr* (which has one of the KDSockaddr_ types), truncated if necessary to fit in **addrlen* bytes. The number of bytes actually written there is then stored into **addrlen*.

If the socket is not bound to a local address, then the function writes undefined data.

OpenKODE Core supports only IPv4, thus the location is filled in as a KDSockaddr_in structure with the local IP address and port that the socket is bound to.

If *socket* is not a socket, or has already been closed, then undefined behavior results. If *addrlen* is not a pointer to a readable and writable KDsockaddr location, or if *addr* does not point to a writable buffer of **addrlen* bytes where one of the KDSockaddr_ structures can be stored, then undefined behavior results.

Return value

The function returns 0 on success. On failure, it returns -1 and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_EIO	General I/O or network error.
KD_ENOMEM	Out of memory or other resource
KD_EOPNOTSUPP	The socket is of a type for which this operation is not supported.

Rationale

kdSocketGetName is based on the [POSIX] function getsockname.

[POSIX] defines some additional error codes which are not applicable to the subset of socket functionality that OpenKODE Core provides.

19.3.7. kdSocketConnect

Connects a socket.

Synopsis

```
KDint kdSocketConnect(KDSocket *socket, const KDSockaddr *addr, KDsocklen
addrlen);
```

Description

This function initiates an operation to connect *socket* to the remote address specified in the location pointed to by *addr* (which has one of the KDSockaddr_ types). *addrlen* is no less than the length in bytes of the address information.

If *addr*->sa_family is KD_AF_INET, then *addr* specifies an IPv4 address, the only address family supported by OpenKODE Core. Then, *addr* is considered a pointer to a KDSockaddr_in structure, which specifies the remote IP address and port number to connect to.

For a connection-based socket, connecting involves communicating with the remote host to establish a connection. For a connectionless (UDP) socket, no network traffic results from this call, but a remote endpoint is associated with the socket so that `kdSocketSend` (or `kdSocketSendTo` with no remote address specified) may be used.

If the socket is already connected, or a connection is already in progress, then the connect operation fails.

If *`socket`* is not a socket, or has already been closed, or *`addr`* does not point to a readable location of at least *`addrlen`* bytes containing one of the `KDSockaddr_` types, then undefined behavior results.

Return value

On success, this function returns 0 and initiates the connect operation, which causes a `KD_EVENT_SOCKET_CONNECT_COMPLETE` when it has finished or failed. Otherwise, on immediate failure, the function returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`. In this failure case, the socket is left in an undefined state; the application should close it and create a new one.

Error codes

`KD_EADDRINUSE`	Address in use.
`KD_EAFNOSUPPORT`	*`sin_family`* is not `KD_AF_INET`
`KD_EALREADY`	A connection attempt is already in progress for this socket.
`KD_ECONNREFUSED`	The remote host was not listening or refused the connection.
`KD_ECONNRESET`	The remote host reset the connection.
`KD_EHOSTUNREACH`	Remote host cannot be reached.
`KD_EINVAL`	*`addrlen`* is wrong, or the socket is listening.
`KD_EIO`	General I/O or network error.
`KD_EISCONN`	Socket is connection-based and already connected.
`KD_ETIMEDOUT`	Connection attempt timed out.

Rationale

`kdSocketConnect` is based on the [POSIX] function `connect`. `kdSocketConnect` is always non-blocking, generating an event when the operation has completed.

Some BSD/[POSIX] socket implementations support using `connect` with an address family of `AF_UNSPEC` in order to “unconnect” a connectionless socket, i.e. to remove an earlier remote address association. This is not supported by OpenKODE Core.

[POSIX] defines some additional errors, some of which are not applicable to the subset of socket functionality which OpenKODE Core provides, but notably including:

- `ENETDOWN` and `ENETUNREACH` are folded into the catch-all `KD_EIO` by OpenKODE Core;
- `EOPNOTSUPP` for when the socket is listening so cannot connect. OpenKODE Core folds this into `KD_EINVAL`.

19.3.8. kdSocketListen

Listen on a socket.

Synopsis

```
KDint kdSocketListen(KDSocket *socket, KDint backlog);
```

Description

This function puts *socket*, a connection-based socket, into listen mode, so it listens for incoming connections. The socket must have already been bound but not connected.

Once a socket is in listen mode, a KD_EVENT_SOCKET_INCOMING event is generated each time a new connection arrives, or when an error occurs on the socket.

backlog is the maximum length of the queue of pending connections. It is undefined whether the actual limit is this number or lower. It is undefined whether the limit refers to the number of completed connections or the total number of in progress and completed connections. If *backlog* is negative or zero, then it is undefined whether the limit is zero (thus not allowing any connections) or some greater value.

It is allowed for an OpenKODE Core implementation to support the rest of the socket API but not kdSocketListen. In that case, kdSocketListen always fails with an error of KD_ENOSYS.

If *socket* is not a socket, or has already been closed, then undefined behavior results.

Return value

On success, the function returns 0, otherwise it returns -1 and stores one of the error codes below into the error indicator returned by kdGetError.

Error codes

KD_EACCES	The application does not have the appropriate privileges.
KD_EADDRINUSE	Another socket (possibly in a different application) is already listening on the same port.
KD_EINVAL	The socket is already connected, or is not bound.
KD_EIO	General I/O or network error.
KD_ENOMEM	Out of memory or other resources.
KD_ENOSYS	Implementation does not support the function at all.
KD_EOPNOTSUPP	The socket is not of a type that supports listening.

Rationale

kdSocketListen is based on the BSD and [POSIX] function listen.

[POSIX] defines some additional errors which are not applicable to the subset of socket functionality provided by OpenKODE Core. It also defines EDESTADDRREQ for when the socket is not bound; OpenKODE Core folds this into KD_EINVAL.

[POSIX] does not specify the error EADDRINUSE.

19.3.9. kdSocketAccept

Accept an incoming connection.

Synopsis

```
KDSocket *kdSocketAccept(KDSocket *socket, KDSockaddr *addr, KDsocklen
*addrlen, void *eventuserptr);
```

Description

This function accepts a waiting connection from *socket*, a socket in listen mode, returning a new socket of the same type as the listening one, but in a connected state.

Since the new socket is in a connected state, KD_EVENT_SOCKET_READABLE, KD_EVENT_SOCKET_WRITABLE and KD_EVENT_SOCKET_ERROR events are generated as soon as the socket is readable, writable or has an error respectively.

The original (listening) socket continues to listen, and thus generates a further KD_EVENT_SOCKET_INCOMING event as soon as another connection is available to accept (or has an error), which may be immediately.

If the function successfully returns a new connected socket, and *addr* is not KD_NULL, then the function stores the address of the remote end of the connection into the location pointed to by *addr* (which has one of the KDSockaddr_ types), truncated if necessary to fit in **addrlen* bytes. The number of bytes actually written there is then stored into **addrlen*.

eventuserptr is the value to use for the *userptr* field of any event generated by the new, connected, socket.

If *socket* is not a socket, or has already been closed, then undefined behavior results. If *addr* is not KD_NULL and *addrlen* is not a pointer to a readable and writable KDsockaddr location, then undefined behavior results. If *addr* is not KD_NULL and does not point to a writable buffer of **addrlen* bytes where one of the KDSockaddr_ structures can be stored, then undefined behavior results.

Return value

On success the function returns the new, connected, socket. On failure it returns KD_NULL and stores one of the error codes below into the error indicator returned by kdGetError.

Error codes

KD_EAGAIN No connection ready to accept.

KD_EINVAL The socket is not in listening mode (including the case where the implementation does not support kdSocketListen at all), or *eventuserptr* is KD_NULL.

KD_EIO General I/O or network error.

KD_EMFILE Too many open sockets.

KD_ENOMEM Out of memory or other resource.

Rationale

kdSocketAccept is based on the BSD/[POSIX] function accept, but always non-blocking, and with the addition of the *eventuserptr* to supply a userptr for any events generated by the newly-created socket.

[POSIX] defines additional error codes, some of which are not applicable to the subset of socket functionality provided by OpenKODE Core, but also including:

- ECONNABORTED when a connection has been aborted; OpenKODE Core simply ignores the aborted connection and returns the next connection in the queue, or gives an error of KD_EAGAIN if none is available.
- ENFILE is folded into KD_EMFILE by OpenKODE Core.
- EOPNOTSUPP when the socket is not of a type to accept connections; OpenKODE Core has kdSocketListen returning that error (as [POSIX] also does).

19.3.10. kdSocketSend, kdSocketSendTo

Send data to a socket.

Synopsis

```
KDint kdSocketSend(KDSocket *socket, const void *buf, KDint len);

KDint kdSocketSendTo(KDSocket *socket, const void *buf, KDint len, const
KDSockaddr *addr, KDsocklen addrlen);
```

Description

These functions send data to a socket. A call to kdSocketSend is equivalent to a call to kdSocketSendTo with *addr* set to KD_NULL and *addrlen* set to 0.

In kdSocketSendTo, if *addr* is not KD_NULL, it points to a location (which has one of the KDSockaddr_ types) which specifies the remote address to send to. *addrlen* is no less than the length in bytes of the address information.

Since OpenKODE Core supports only IPv4, this address is in fact a KDSockaddr_in structure specifying the remote IP address and port.

If kdSocketSendTo is used on a connection-based socket with *addr* and *addrlen* set to values other than KD_NULL and 0 respectively, it is undefined whether the values are ignored or whether an error (and which one) is generated.

For a connectionless socket which has not had a remote address associated with it, kdSocketSendTo must be used specifying an address, otherwise the function returns an error.

A connection-based socket can send data only when it is connected.

The functions are non-blocking: if there is no buffer space to write at least some of the data immediately, they return an error.

If the call successfully writes a non-zero number of bytes, and buffer space remains such that further data could be written immediately, then a KD_EVENT_SOCKET_WRITABLE is generated.

Some UDP implementations may use ICMP to generate errors when packets are rejected by the recipient. It is undefined whether an OpenKODE implementation generates errors on the basis of these or other messages when writing to UDP sockets.

If *`socket`* is not a socket, or has already been closed, then undefined behavior results. If *`addr`* is not `KD_NULL` and is not a readable location of at least *`addrlen`* bytes containing one of the `KDSockaddr_` types, then undefined behavior results.

Return value

The functions return the number of bytes sent on success. Success does not imply that the data reached its destination, although a reliable (TCP) socket will give an error at some point if its connection is lost. When an error is detected, the functions return -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EAGAIN`	Buffers full; retry after the next `KD_EVENT_SOCKET_WRITABLE` event on this socket.
`KD_ECONNRESET`	Connection reset by peer.
`KD_EDESTADDRREQ`	Destination address not supplied for a connectionless socket which has not had a remote address associated with it.
`KD_EIO`	General I/O or network error.
`KD_ENOMEM`	Out of memory or other resource.
`KD_ENOTCONN`	The socket is connection-based but is currently not connected.
`KD_EPIPE`	Socket is no longer connected.

Rationale

`kdSocketSend` and `kdSocketSendTo` are based on the BSD and [POSIX] functions `send` and `sendto`.

[POSIX] defines additional error codes, some of which are not applicable to the subset of socket functionality defined by OpenKODE Core, but also including:

- `ENOTCONN` is the error returned when the caller attempts to specify an address for a connection-based socket. OpenKODE leaves it undefined whether such an address specification is ignored or generates some unlisted error.
- `ENETDOWN` and `ENETUNREACH` are folded into the catch-all `KD_EIO` by OpenKODE Core.

19.3.11. kdSocketRecv, kdSocketRecvFrom

Receive data from a socket.

Synopsis

```
KDint kdSocketRecv(KDSocket *socket, void *buf, KDint len);

KDint kdSocketRecvFrom(KDSocket *socket, void *buf, KDint len, KDSockaddr
addr, KDsocklen *addrlen);
```

Description

These functions receive data from a socket. `kdSocketRecv` is equivalent to `kdSocketRecvFrom` with *`addr`* and *`addrlen`* both set to `KD_NULL`.

The call is non-blocking. If no data can be read immediately, the functions return an error.

A connection-based socket can receive data only when it is connected. A connectionless socket can receive data only when it is bound to a local address.

If `kdSocketRecvFrom` successfully reads a non-zero number of bytes, and *`addr`* is not `KD_NULL`, then the function stores the address of the remote sender into the location pointed to by *`addr`* (which has one of the `KDSockaddr_` types), truncated if necessary to fit in *`*addrlen`* bytes. The number of bytes actually written there is then stored into *`*addrlen`*.

If this call successfully reads a non-zero number of bytes, and further unread data remains, then a `KD_EVENT_SOCKET_READABLE` event is generated.

Some UDP implementations may use ICMP to generate errors when packets are rejected by the recipient. It is undefined whether an OpenKODE implementation generates errors on the basis of these or other messages when writing to UDP sockets.

If *`socket`* is not a socket, or has already been closed, then undefined behavior results. If *`addr`* is not `KD_NULL` and *`addrlen`* is not a pointer to a readable and writable KDsockaddr location, then undefined behavior results. If *`addr`* is not `KD_NULL` and does not point to a writable buffer of *`*addrlen`* bytes where one of the `KDSockaddr_` structures can be stored, then undefined behavior results.

Return value

The functions return the number of bytes received on success. When an error is detected, the functions return -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EAGAIN`	Buffers empty; retry after the next `KD_EVENT_SOCKET_READABLE` event on this socket.
`KD_ECONNRESET`	Connection reset by peer.
`KD_EIO`	General I/O or network error.
`KD_ENOMEM`	Out of memory or other resource.
`KD_ENOTCONN`	The socket is connection-based but is currently not connected.
`KD_ETIMEDOUT`	Connection timed out.

Rationale

`kdSocketRecv` and `kdSocketRecvFrom` are based on the BSD and [POSIX] functions `recv` and `recvfrom`.

[POSIX] defines additional error codes, some of which are not applicable to the subset of socket functionality defined by OpenKODE Core, but also including:

- ENOTCONN is the error returned when the caller attempts to specify an address for a connection-based socket. OpenKODE leaves it undefined whether such an address specification is ignored or generates some unlisted error.
- ENETDOWN and ENETUNREACH are folded into the catch-all KD_EIO by OpenKODE Core.

19.3.12. kdHtonl

Convert a 32-bit integer from host to network byte order.

Synopsis

```
KDuint32 kdHtonl(KDuint32 hostlong);
```

Description

This function converts a 32-bit integer from host to network byte order. It involves reversing the bytes within the 32-bit integer if and only if the platform stores integers in little endian order.

Return value

The function returns the converted integer.

Rationale

kdHtonl is based on the BSD/[POSIX] function htonl.

19.3.13. kdHtons

Convert a 16-bit integer from host to network byte order.

Synopsis

```
KDuint16 kdHtons(KDuint16 hostshort);
```

Description

This function converts a 16-bit integer from host to network byte order. It involves reversing the bytes within the 16-bit integer if and only if the platform stores integers in little endian order.

Return value

The function returns the converted integer.

Rationale

kdHtons is based on the BSD/[POSIX] function htons.

19.3.14. kdNtohl

Convert a 32-bit integer from network to host byte order.

Synopsis

```
KDuint32 kdNtohl(KDuint32 netlong);
```

Description

This function converts a 32-bit integer from network to host byte order. It involves reversing the bytes within the 32-bit integer if and only if the platform stores integers in little endian order.

Return value

The function returns the converted integer.

Rationale

`kdNtohl` is based on the BSD/[POSIX] function `ntohl`.

19.3.15. kdNtohs

Convert a 16-bit integer from network to host byte order.

Synopsis

```
KDuint16 kdNtohs(KDuint16 netshort);
```

Description

This function converts a 16-bit integer from network to host byte order. It involves reversing the bytes within the 16-bit integer if and only if the platform stores integers in little endian order.

Return value

The function returns the converted integer.

Rationale

`kdNtohs` is based on the BSD/[POSIX] function `ntohs`.

19.3.16. kdInetAton

Convert a “dotted quad” format address to an integer.

Synopsis

```
KDint kdInetAton(const KDchar *cp, KDuint32 *inp);
```

Description

This function converts an IPv4 address in textual “dotted quad” format, as well as some related formats, into a network order 32-bit integer.

cp points to a string containing one to four numbers, separated by dots. Each number is converted, with 0x or 0X denoting a hexadecimal number, or a leading 0 denoting an octal number. Each but the last number occupies 8 bits in the result integer, with the last occupying the remaining space, between 8 and 32 bits. The first number occupies the topmost space.

Return value

On success, the converted integer is stored in **inp*, and the function returns non-zero. If no valid address is found to convert, the function returns 0.

Rationale

kdInetAton is based on the [POSIX] function inet_aton. Its functionality is similar to the [POSIX] function inet_addr, but that function is considered obsolete because it returns the IP address directly using -1 as an error value, even though -1 is a valid IP address.

19.3.17. kdInetNtoa

Convert an address as a 32-bit integer to dotted quad format

Synopsis

```
const KDchar *kdInetNtoa(KDuint32 in);
```

Description

This function converts the IP address *in* (in network order) to a “dotted quad” format string. The result string always has four components, each in decimal.

Return value

The function returns a pointer to a static buffer containing the null-terminated result string. The string is not overwritten until the next call to kdInetNtoa.

Rationale

kdInetNtoa is based on the [POSIX] function inet_ntoa.

19.4. Events

19.4.1. KD_EVENT_SOCKET_READABLE

Event to indicate that a socket is readable.

Synopsis

```
#define KD_EVENT_SOCKET_READABLE 0x100
```

Description

This event is generated for a connected socket or a bound connectionless socket when it becomes readable, or when it remains readable after a successful call to read data from it (via kdSocketRecv or kdSocketRecvFrom).

Thus, this event indicates that, at the time it was generated, a call to the applicable one of those two functions will return a non-zero amount of data.

KD_EVENT_SOCKET_READABLE events are merged, i.e. if such an event is generated by OpenKODE Core when another event generated by OpenKODE Core for the same socket is already in the event queue, the earlier one is removed.

The event's *userptr* field is set to the value supplied in the *eventuserptr* parameter when the socket was created.

The event data is in event->data.socketreadable element of the event's data union, which has the following type:

```
typedef struct KDEventSocketReadable {
    KDSocket *socket;
} KDEventSocketReadable;
```

socket is the socket which caused the event.

19.4.2. KD_EVENT_SOCKET_WRITABLE

Event to indicate that a socket is writable.

Synopsis

```
#define KD_EVENT_SOCKET_WRITABLE 0x101
```

Description

This event is generated for a connected socket or a bound connectionless socket when it becomes writable, or when it remains writable after a successful call to write data to it (via kdSocketSend or kdSocketSendTo). Thus, this event indicates that, at the time it was generated, a call to the applicable one of those two functions will successfully write a non-zero amount of data.

KD_EVENT_SOCKET_WRITABLE events are merged, i.e. if such an event is generated by OpenKODE Core when another event generated by OpenKODE Core for the same socket is already in the event queue, the earlier one is removed.

The event's *userptr* field is set to the value supplied in the *eventuserptr* parameter when the socket was created.

The event data is in event->data.socketwritable element of the event's data union, which has the following type:

```
typedef struct KDEventSocketWritable {
    KDSocket *socket;
} KDEventSocketWritable;
```

socket is the socket which caused the event.

19.4.3. KD_EVENT_SOCKET_ERROR

Event to indicate that a socket has an error.

Synopsis

```
#define KD_EVENT_SOCKET_ERROR 0x105
```

Description

This event is generated for a connected socket or a bound connectionless socket when it has an error. It is generated only once for a particular socket, when the error condition first arises. The error indicates that calling one of `kdSocketSend` (for a connected or bound connectionless socket), `kdSocketRecv`, `kdSocketSendTo` or `kdSocketRecvFrom` will fail and yield the error code.

The event's *`userptr`* field is set to the value supplied in the *`eventuserptr`* parameter when the socket was created.

The event data is in `event->data.socketerror` element of the event's data union, which has the following type:

```
typedef struct KDEventSocketError {
    KDSocket *socket;
} KDEventSocketError;
```

`socket` is the socket which caused the event.

19.4.4. KD_EVENT_SOCKET_CONNECT_COMPLETE

Event generated when a socket connect is complete

Synopsis

```
#define KD_EVENT_SOCKET_CONNECT_COMPLETE 0x102
```

Description

This event is generated when a socket connect initiated by a call to `kdSocketConnect` completes.

The event's *`userptr`* field is set to the value supplied in the *`eventuserptr`* parameter when the socket was created.

The event data is in `event->data.socketconnect` element of the event's data union, which has the following type:

```
typedef struct KDEventSocketConnect {
    KDSocket *socket;
    KDint32 error;
} KDEventSocketConnect;
```

`socket` is the socket which caused the event. *`error`* is as defined below.

If the connect completed successfully, *`error`* is 0, and the socket is in the connected state. As such, a `KD_EVENT_SOCKET_READABLE` event is generated as soon as the socket is readable or has an error, and a `KD_EVENT_SOCKET_WRITABLE` event is generated as soon as the socket is writable or has an error.

If the connect failed, *`error`* is set to one of the error codes listed in the specification of `kdSocketConnect`. The socket is left in an undefined state; the application should close it and create a new one.

19.4.5. KD_EVENT_SOCKET_INCOMING

Event generated when a listening socket detects an incoming connection or an error.

Synopsis

```
#define KD_EVENT_SOCKET_INCOMING 0x103
```

Description

This event is generated when a listening socket (one on which `kdSocketListen` has been called) detects such a connection, or detects an error. `kdSocketAccept` may then be used to accept the connection or retrieve the error code.

The event's *`userptr`* field is set to the value supplied in the *`eventuserptr`* parameter when the socket was created.

The event data is in `event->data.socketincoming` element of the event's data union, which has the following type:

```
typedef struct KDEventSocketIncoming {
    KDSocket *socket;
} KDEventSocketIncoming;
```

`socket` is the socket which caused the event. *`unused`* is set to an undefined value.

Multiple events of this type referring to the same socket are merged. When an event is generated by the OpenKODE implementation, if another event generated by the OpenKODE implementation of the same type and same socket is already in the queue, the older one is removed.

19.4.6. KD_EVENT_NAME_LOOKUP_COMPLETE

`kdNameLookup` complete event.

Synopsis

```
#define KD_EVENT_NAME_LOOKUP_COMPLETE 0x104
```

Description

This event is generated when a lookup initiated by a call to `kdNameLookup` is complete, either successfully or with an error.

The event's *`userptr`* field is set to the value supplied in the *`eventuserptr`* parameter to `kdNameLookup`.

The event data is in `event->data.namelookup` element of the event's data union, which has the following type:

```
typedef struct KDEventNameLookup {
    KDint32 error;
    KDint32 resultlen;
    const KDSockaddr *result;
    KDboolean more;
} KDEventNameLookup;
```

If the lookup completed successfully, *`error`* is 0 and the result is stored in the location pointed to by *`result`*, which has one of the `KDSockaddr_` types. The length of the returned KDSockaddr is *`resultlen`*. If multiple results are returned, then for all but the last result, *`more`* will be set to 1. Otherwise, *`more`* is set to 0. The information pointed to by *`result`* will remain valid and not be overwritten until the next `KD_EVENT_NAME_LOOKUP` event caused by a `kdNameLookup` completing is delivered to the application, either by a callback for the event being called, or by the event being returned by `kdWaitEvent`.

Error codes

This event uses different error codes from other events and functions in OpenKODE Core.

KD_HOST_NOT_FOUND (1) The specified name is not known.

KD_NO_DATA (4) The specified name is valid but does not have an address.

KD_NO_RECOVERY (3) A non-recoverable error has occurred on the name server.

KD_TRY_AGAIN (2) A temporary error has occurred on an authoratitive name server, and the lookup may succeed if retried later.

20. Input/output

20.1. Introduction

OpenKODE Core input/output allows access to the platform’s inputs and outputs that allow interaction with the human user, such as buttons (input) and vibrate (output). Communications devices and the screen are excluded from OpenKODE Core input/output.

The OpenKODE Core input/output model aims to be simple and extensible so it can represent any new input devices which appear in the future. The model has inputs each of type binary, integer or floating point, and outputs each of type integer or floating point. More complex devices are made up of these simple inputs and outputs, for example a joystick might be two integer inputs for the two axes and a binary input for the fire button.

Inputs and outputs are collectively known as *I/O item*s. Each I/O item has at least one *index*. Indexes are in the range `0`..`KDINT32_MAX`. OpenKODE Core defines indexes in the range `0`..`0x3fffffff`. Indexes in the range `0x40000000`..`0x7fffffff` are reserved for implementation-defined non-portable I/O items. It is expected that an implementation will map all available I/O items that have no portable OpenKODE Core definition into this non-portable space, and may also map OpenKODE Core defined items into it as well (so each has two indexes).

It is possible for an I/O item to have more than one index.

An input can be polled using one of the `kdInputPoll*` functions. In addition, inputs in a particular I/O group can be event enabled, so an event is generated whenever an input changes, using `kdInputEventEnable`. If the same input has multiple indexes in different I/O groups, then an event is generated for each event enabled one when the input changes. Most inputs use the `KD_EVENT_INPUT` event, which carries the new value of the input. However, a mouse/pointer input uses the `KD_EVENT_INPUT_POINTER` event, which carries the whole mouse/pointer state at the time of the event, to allow an application to tell where the pointer was at the time a button was pressed, and a joystick stick uses the `KD_EVENT_INPUT_STICK` event, which carries the state of both (or all three) axes.

An output is set using one of the `kdOutputSet*` functions.

20.1.1. I/O groups

An *I/O group* is a group of I/O items which are specified together. OpenKODE Core defines a number of I/O groups, such as game keys, pointer and vibrate. All the inputs in a particular I/O group are event enabled by a single call to `kdInputEventEnable`.

20.2. Events

20.2.1. KD_EVENT_INPUT

Input changed event.

Synopsis

```
#define KD_EVENT_INPUT 0x200

typedef struct KDEventInput {
    KDint32 index;
    union {
        KDint32 i;
        KDint64 l;
        KDfloat32 f;
    } value;
} KDEventInput;
```

Description

Unless otherwise specified, an input which has been event enabled using `kdInputEventEnable` generates this event whenever its value changes.

The event data is in the *`input`* of the event data union, of type KDEventInput. Within this struct, *`index`* is the index of the input whose change caused the event, and one of *`value.i`*, *`value.l`* or *`value.f`* is the new value of the input, for input type binary/KDint32, KDint64 or KDfloat32 respectively.

For a binary input, the field’s value is either 0 or 1.

When one of application’s windows has input focus, the *`eventuserptr`* parameter supplied when the window was created is used as the value of the *`userptr`* field of an input event. When none of the application’s windows has input focus, the input event’s *`userptr`* field is `KD_NULL`.

20.2.2. KD_EVENT_INPUT_POINTER

Pointer input changed event.

Synopsis

```
#define KD_EVENT_INPUT_POINTER 0x201
typedef struct KDEventInputPointer {
    KDint32 index;
    KDint32 select;
    KDint32 x;
    KDint32 y;
} KDEventInputPointer;
```

Description

When an input in the pointer device changes, this event is generated.

The data is in the *`inputpointer`* element of the event data union, with type KDEventInputPointer, with the following fields:

- *`index`* is the index number of the input that actually changed;
- *`select`* contains the button state, with value 1 if the select button is pressed or 0 if it is not;
- *`x`* and *`y`* contain the X and Y coordinate input values.

The input values reflect the state at the time that the event was generated.

`KD_EVENT_INPUT_POINTER` events are merged as follows: If a new event is created by the OpenKODE implementation, and the previous event created by the OpenKODE implementation of the same type in the queue was for a change to the X or Y coordinate (rather than a change of the button state), then the old event is removed from the queue as the new event is added to the end of the queue. Thus, from the application’s point of view, any `KD_EVENT_INPUT_POINTER` event can reflect a change of either coordinate, whatever the value of *`index`*.

When one of application’s windows has input focus, the *`eventuserptr`* parameter supplied when the window was created is used as the value of the *`userptr`* field of an input event. When none of the application’s windows has input focus, the input event’s *`userptr`* field is `KD_NULL`.

20.2.3. KD_EVENT_INPUT_STICK

Joystick stick changed event.

Synopsis

```
#define KD_EVENT_INPUT_STICK 0x202
typedef struct KDEventInputStick {
    KDint32 index;
    KDint32 x;
    KDint32 y;
    KDint32 z;
} KDEventInputStick;
```

Description

When one of the axes in a joystick stick changes, this event is generated.

The data is in the *`inputstick`* element of the event data union, with type KDEventInputStick, with the following fields:

- *`index`* is the index number of the joystick stick axis input that actually changed;
- *`x`*, *`y`* and *`z`* contain the X, Y and Z axis input values.

The input values reflect the state at the time that the event was generated.

`KD_EVENT_INPUT_STICK` events are merged; on creation of a new event, any event of this type for the same joystick stick already in the queue is removed. Thus, from the application's point of view, any `KD_EVENT_INPUT_STICK` event can reflect a change of either coordinate, whatever the value of *`index`*.

When one of application's windows has input focus, the *`eventuserptr`* parameter supplied when the window was created is used as the value of the *`userptr`* field of an input event. When none of the application's windows has input focus, the input event's *`userptr`* field is `KD_NULL`. (If the event was posted by the application using `kdPostEvent`, *`userptr`* is as set in the event passed to that function.)

20.3. Functions

20.3.1. kdInputEventEnable

Enable events for inputs in an I/O group.

Synopsis

```
KDint kdInputEventEnable(KDint idx, KDint enable);
```

Description

This function enables or disables events for inputs in the I/O group which includes the specified index. Each input in that group becomes event enabled if *`enable`* is non-zero, or event disabled if *`enable`* is zero.

An event enabled input generates the `KD_EVENT_INPUT` event when it changes, unless otherwise specified for that particular input index.

If *`idx`* is not in any I/O group, then the function fails.

Return value

On success, the function returns 0. On failure, it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EINVAL`	*`idx`* is not in any I/O group.
`KD_EIO`	Non-specific error from I/O device.
`KD_ENOMEM`	Out of memory or other resource.

20.3.2. kdInputPollb, kdInputPolli, kdInputPolll, kdInputPollf

poll inputs

Synopsis

```
KDint kdInputPollb(KDint startidx, KDuint numidxs, KDint32 *buffer);
KDint kdInputPolli(KDint startidx, KDuint numidxs, KDint32 *buffer);
KDint kdInputPolll(KDint startidx, KDuint numidxs, KDint64 *buffer);
KDint kdInputPollf(KDint startidx, KDuint numidxs, KDfloat32 *buffer);
```

Description

This function polls the state of *`numidxs`* (zero or more) inputs in a contiguous index range starting at *`startidx`*, all of the same type. A negative *`numidxs`* value is treated as zero.

`kdInputPollb` polls binary inputs. Each group of up to 32 inputs is placed in a single KDint32 word in the buffer, starting at bit 0 in each word. Unused bits in the final written word are set to 0.

`kdInputPolli` polls KDint32 inputs, `kdInputPolll` polls KDint64 inputs, and `kdInputPollf` polls KDfloat32 inputs.

If not all of the indexes in the range are inputs of the applicable type or are not all in the same I/O group, then the range is cut short so all indexes are inputs of the applicable type and are in the same I/O group.

The OpenKODE input state, as reflected by the values written by this function, is updated whenever either of `kdPumpEvents` or `kdWaitEvent` is called. The state may or may not be updated at other times.

If *`buffer`* does not point to a writable array whose length is at least the length of the index range being polled (as modified above, so not necessarily the same as *`numidxs`*), and with entries of the applicable type for the function, then undefined behavior results. For `kdInputPollb`, the array must have a length of at least (rangelength + 31) / 32 and type KDint32.

Return value

On success, the function returns the number of inputs actually read. Otherwise, it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM`	Out of memory or other resource.

`KD_EIO` Non-specific error from I/O device.

20.3.3. kdOutputSeti, kdOutputSetf

set outputs

Synopsis

```
KDint kdOutputSeti(KDint startidx, KDuint numidxs, const KDint32 *buffer);

KDint kdOutputSetf(KDint startidx, KDuint numidxs, const KDfloat32 *buffer);
```

Description

This function sets the values of *`numidxs`* (zero or more) outputs starting at *`startidx`*, all of the same type. A negative *`numidxs`* value is treated as zero.

`kdOutputSeti` sets KDint32 outputs, and `kdOutputSetf` sets KDfloat32 outputs.

If not all of the indexes in the range are outputs of the applicable type all in the same I/O group, the range is cut short so all indexes are outputs of the applicable type all in the same I/O group.

If *`buffer`* does not point to a readable array whose length is at least the length of the index range being read (as modified above, so not necessarily the same as *`numidxs`*), and with entries of the applicable type for the function, then undefined behavior results.

The state of a physical output need not always correspond to that of the OpenKODE Core API output, depending on how the particular output is virtualized by the platform in the presence of concurrent applications. The physical output should correspond to the OpenKODE Core API output when the platform is in a state where the application is selected as the one that the user interacts with.

Return value

On success, the function returns the number of outputs actually set. Otherwise, it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM` Out of memory or other resource.

`KD_EIO` Non-specific error from I/O device.

20.4. I/O groups and items

An *I/O item* is an input or an output. Each I/O item is referenced, in polling and setting functions and in events, using an index. Indexes are in the range 0..`KDINT_MAX`.

Each possible index has one of these states:

- empty;
- binary input;

- KDint32 input;
- KDint64 input;
- KDfloat32 input;
- KDint32 output;
- KDfloat32 output.

An *I/O group* is a group of I/O items which are specified together, and on which `kdInputEventEnable` acts. An I/O group specified below may or may not be present. If not present, all the indexes in it have empty state, which means that attempting to poll an input or set an output fails (in the sense that the applicable function returns 0 to indicate that it had to cut the index range short to 0 length). This can be used to test whether the I/O group is present.

A further means of testing whether an I/O group is present is whether `kdInputEventEnable` fails with `KD_EINVAL`.

Where an I/O group is present, but a particular I/O item within it is specified as optional and is in fact not present, then the index takes the same state as if it was present, but:

- if it is an input, it never generates an event, and it has an undefined value, as seen when polling it and when including its value in another input's event;
- if it is an output, setting it has no effect.

20.4.1. KD_IOGROUP_GAMEKEYS

I/O group for game keys.

Synopsis

```
#define KD_IOGROUP_GAMEKEYS 0x1000
#define KD_IO_GAMEKEYS_AVAILABILITY    (KD_IOGROUP_GAMEKEYS + 0)
#define KD_IO_GAMEKEYS_UP              (KD_IOGROUP_GAMEKEYS + 1)
#define KD_IO_GAMEKEYS_LEFT            (KD_IOGROUP_GAMEKEYS + 2)
#define KD_IO_GAMEKEYS_RIGHT           (KD_IOGROUP_GAMEKEYS + 3)
#define KD_IO_GAMEKEYS_DOWN            (KD_IOGROUP_GAMEKEYS + 4)
#define KD_IO_GAMEKEYS_FIRE            (KD_IOGROUP_GAMEKEYS + 5)
#define KD_IO_GAMEKEYS_A               (KD_IOGROUP_GAMEKEYS + 6)
#define KD_IO_GAMEKEYS_B               (KD_IOGROUP_GAMEKEYS + 7)
#define KD_IO_GAMEKEYS_C               (KD_IOGROUP_GAMEKEYS + 8)
#define KD_IO_GAMEKEYS_D               (KD_IOGROUP_GAMEKEYS + 9)
```

Description

This I/O group defines the keys that are available in Java MIDP2, and are thus likely to be available in handsets. The value of each of these button inputs is 1 when the button is pressed and 0 when it is not. When event enabled, separate events are generated for button press and button release at the appropriate times, even if there is no call to `kdWaitEvent` or `kdPumpEvents` in between the press and release.

The keys in this I/O group are not necessarily dedicated; they may have another function such as in the phone keypad.

If this I/O group is present, then `KD_IOGROUP_GAMEKEYSNC` is also present, and the two groups represent the same keys.

Whether the OpenKODE input state for these inputs reflects the actual state of the keys when no application window has the input focus is undefined.

I/O items

index	type	range	usage
KD_IO_GAMEKEYS_AVAILABILITY	mandatory KDint32 input		availability bitmap
KD_IO_GAMEKEYS_UP	mandatory binary input	0..1	up button
KD_IO_GAMEKEYS_LEFT	mandatory binary input	0..1	left button
KD_IO_GAMEKEYS_RIGHT	mandatory binary input	0..1	right button
KD_IO_GAMEKEYS_DOWN	mandatory binary input	0..1	down button
KD_IO_GAMEKEYS_FIRE	mandatory binary input	0..1	fire button
KD_IO_GAMEKEYS_A	optional binary input	0..1	game_a button
KD_IO_GAMEKEYS_B	optional binary input	0..1	game_b button
KD_IO_GAMEKEYS_C	optional binary input	0..1	game_c button
KD_IO_GAMEKEYS_D	optional binary input	0..1	game_d button

KD_IO_GAMEKEYS_AVAILABILITY is an input that indicates using a bitmap which inputs are present. Bit n represents input KD_IO_GAMEKEYS_UP + n, set to 1 if the input is available and 0 if not, with unused bits set to 0.

Thus the value of the input is 31 if the minimum set (direction keys plus fire) is present, 511 if all keys are present, and other values if some of the game A, B, C or D keys are absent.

The value of this input may change if the user takes some action which causes reconfiguration, for example reorienting the handset.

Like any other input, if the I/O group is event enabled, an event is generated when this input changes, for example because a reconfiguration on reorientation has caused more or fewer game keys to be available.

Simultaneous key presses (chording)

If the user presses two adjacent direction keys plus any one of “fire”, “A”, “B”, “C” or “D”, then OpenKODE Core events and input state accurately reflect the keys pressed. Similarly for any two keys in such a three key combination.

20.4.2. KD_IOGROUP_GAMEKEYSNC

I/O group for game keys, no chording.

Synopsis

```
#define KD_IOGROUP_GAMEKEYSNC 0x1100
#define KD_IO_GAMEKEYSNC_AVAILABILITY  (KD_IOGROUP_GAMEKEYSNC + 0)
#define KD_IO_GAMEKEYSNC_UP            (KD_IOGROUP_GAMEKEYSNC + 1)
#define KD_IO_GAMEKEYSNC_LEFT          (KD_IOGROUP_GAMEKEYSNC + 2)
#define KD_IO_GAMEKEYSNC_RIGHT         (KD_IOGROUP_GAMEKEYSNC + 3)
#define KD_IO_GAMEKEYSNC_DOWN          (KD_IOGROUP_GAMEKEYSNC + 4)
#define KD_IO_GAMEKEYSNC_FIRE          (KD_IOGROUP_GAMEKEYSNC + 5)
#define KD_IO_GAMEKEYSNC_A             (KD_IOGROUP_GAMEKEYSNC + 6)
#define KD_IO_GAMEKEYSNC_B             (KD_IOGROUP_GAMEKEYSNC + 7)
#define KD_IO_GAMEKEYSNC_C             (KD_IOGROUP_GAMEKEYSNC + 8)
#define KD_IO_GAMEKEYSNC_D             (KD_IOGROUP_GAMEKEYSNC + 9)
```

Description

This I/O group defines the same keys as `KD_IOGROUP_GAMEKEYS`, except that this I/O group does not have to meet the simultaneous key presses (chording) requirements of that I/O group. Otherwise, it functions the same, and is subject to the same rules, including which inputs are mandatory.

When both I/O groups are present, both represent the same group of keys.

Rationale

The chording requirements of `KD_IOGROUP_GAMEKEYS` are intended to meet the playability requirements of many games. However it is recognized that some handsets do not meet those requirements, so this `KD_IOGROUP_GAMEKEYSNC` is specified to allow portable access to the game keys by applications which do not have the chording requirements.

A game or other application which requires the chording should use `KD_IOGROUP_GAMEKEYS`, which stops it being used on handsets that do not match the chording requirements.

A game or other application which does not have chording requirements should use `KD_IOGROUP_GAMEKEYSNC`, in order to be portable to as many handsets as possible.

Manufacturers are encouraged to build handsets to implement and meet the chording requirements of `KD_IOGROUP_GAMEKEYS` in order to allow as many applications as possible to be runnable and playable.

20.4.3. KD_IOGROUP_PHONEKEYPAD

I/O group for phone keypad.

Synopsis

```
#define KD_IOGROUP_PHONEKEYPAD 0x2000
#define KD_IO_PHONEKEYPAD_AVAILABILITY  (KD_IOGROUP_PHONEKEYPAD + 0)
#define KD_IO_PHONEKEYPAD_0             (KD_IOGROUP_PHONEKEYPAD + 1)
#define KD_IO_PHONEKEYPAD_1             (KD_IOGROUP_PHONEKEYPAD + 2)
#define KD_IO_PHONEKEYPAD_2             (KD_IOGROUP_PHONEKEYPAD + 3)
#define KD_IO_PHONEKEYPAD_3             (KD_IOGROUP_PHONEKEYPAD + 4)
#define KD_IO_PHONEKEYPAD_4             (KD_IOGROUP_PHONEKEYPAD + 5)
#define KD_IO_PHONEKEYPAD_5             (KD_IOGROUP_PHONEKEYPAD + 6)
#define KD_IO_PHONEKEYPAD_6             (KD_IOGROUP_PHONEKEYPAD + 7)
#define KD_IO_PHONEKEYPAD_7             (KD_IOGROUP_PHONEKEYPAD + 8)
#define KD_IO_PHONEKEYPAD_8             (KD_IOGROUP_PHONEKEYPAD + 9)
#define KD_IO_PHONEKEYPAD_9             (KD_IOGROUP_PHONEKEYPAD + 10)
#define KD_IO_PHONEKEYPAD_STAR          (KD_IOGROUP_PHONEKEYPAD + 11)
#define KD_IO_PHONEKEYPAD_HASH          (KD_IOGROUP_PHONEKEYPAD + 12)
```

```
#define KD_IO_PHONEKEYPAD_LEFTSOFT      (KD_IOGROUP_PHONEKEYPAD + 13)
#define KD_IO_PHONEKEYPAD_RIGHTSOFT     (KD_IOGROUP_PHONEKEYPAD + 14)
```

Description

This I/O group defines the keys in a phone keypad, plus the left and right “soft keys” found just below the screen on many handsets. The value of each of these key inputs is 1 when the key is pressed and 0 when it is not. When a particular input is event enabled (using `kdInputEventEnable`), separate events are generated for button press and button release at the appropriate times, even if there is no call to `kdWaitEvent` or `kdPumpEvents` in between the press and release.

Whether the OpenKODE input state for these inputs reflects the actual state of the keys when no application window has the input focus is undefined.

I/O items

index	**type**	**range**	**usage**
`KD_IO_PHONEKEYPAD_AVAILABILITY`	mandatory KDint32 input		availability bitmap
`KD_IO_PHONEKEYPAD_0`	mandatory binary input	0..1	0 key
`KD_IO_PHONEKEYPAD_1`	mandatory binary input	0..1	1 key
`KD_IO_PHONEKEYPAD_2`	mandatory binary input	0..1	2 key
`KD_IO_PHONEKEYPAD_3`	mandatory binary input	0..1	3 key
`KD_IO_PHONEKEYPAD_4`	mandatory binary input	0..1	4 key
`KD_IO_PHONEKEYPAD_5`	mandatory binary input	0..1	5 key
`KD_IO_PHONEKEYPAD_6`	mandatory binary input	0..1	6 key
`KD_IO_PHONEKEYPAD_7`	mandatory binary input	0..1	7 key
`KD_IO_PHONEKEYPAD_8`	mandatory binary input	0..1	8 key
`KD_IO_PHONEKEYPAD_9`	mandatory binary input	0..1	9 key
`KD_IO_PHONEKEYPAD_STAR`	mandatory binary input	0..1	* key
`KD_IO_PHONEKEYPAD_HASH`	mandatory binary input	0..1	# key
`KD_IO_PHONEKEYPAD_LEFTSOFT`	optional binary input	0..1	left soft key
`KD_IO_PHONEKEYPAD_RIGHTSOFT`	optional binary input	0..1	right soft key

`KD_IO_PHONEKEYPAD_AVAILABILITY` is an input that indicates using a bitmap which inputs are present. Bit n represents input `KD_IO_PHONEKEYPAD_0 + n`, set to 1 if the input is available and 0 if not, with unused bits set to 0.

Thus the value of the input is 0xfff if the minimum set (0-9, *, #) is present, 0x3fff if the two soft keys are additionally present, or 0x1fff or 0x2fff if only the left or right (respectively) softkey is present.

The value of this input may change if the user takes some action which causes reconfiguration, for example reorienting the handset or screen such that the soft keys are no longer in the expected place below the screen as viewed by the user.

Like any other input, if the I/O group is event enabled, an event is generated when this input changes, for example because a reconfiguration on reorientation has caused the soft keys to become available or unavailable.

20.4.4. KD_IOGROUP_VIBRATE

I/O group for vibrate.

Synopsis

```
#define KD_IOGROUP_VIBRATE 0x3000
#define KD_IO_VIBRATE_AVAILABILITY  (KD_IOGROUP_VIBRATE + 0)
#define KD_IO_VIBRATE_MINFREQUENCY  (KD_IOGROUP_VIBRATE + 1)
#define KD_IO_VIBRATE_MAXFREQUENCY  (KD_IOGROUP_VIBRATE + 2)
#define KD_IO_VIBRATE_VOLUME        (KD_IOGROUP_VIBRATE + 3)
#define KD_IO_VIBRATE_FREQUENCY     (KD_IOGROUP_VIBRATE + 4)
```

Description

This I/O group defines the vibrate outputs as might be found in a handset.

I/O items

index	type	range	usage
KD_IO_VIBRATE_AVAILABILITY	mandatory KDint32 input	9, 31	availability bitmap
KD_IO_VIBRATE_MINFREQUENCY	optional KDint32 input		frequency minimum in millihertz (constant)
KD_IO_VIBRATE_MAXFREQUENCY	optional KDint32 input		frequency maximum in millihertz (constant)
KD_IO_VIBRATE_VOLUME	mandatory KDint32 output	0..1000	volume in permilles
KD_IO_VIBRATE_FREQUENCY	optional KDint32 output	see below	frequency in millihertz

Output `KD_IO_VIBRATE_VOLUME` sets the volume level in permilles (i.e. thousandths), and is mandatory. The initial state is 0, and setting to 0 silences the handset’s vibrate. Setting it to a value outside the range 0..1000 is the same as setting it to 1000 (full volume). The resolution of the actual volume may be less than 1 permille, in which case the available volume setting nearest to that requested is selected. In particular, the handset may only allow vibrate settings of 0 (off) and 1000 (on).

Output `KD_IO_VIBRATE_FREQUENCY` sets the frequency in millihertz (e.g. 25000 represents 25Hz), and is optional. The range is determined by the constant values of the inputs `KD_IO_VIBRATE_MINFREQUENCY` and `KD_IO_VIBRATE_MAXFREQUENCY`. Setting the output to a value outside the range leaves the vibrate settings in an undefined state in respect of both its volume and frequency. The resolution of the actual frequency may be less than 1 mHz, in which case the available frequency nearest to that requested is selected.

Inputs KD_IO_VIBRATE_MINFREQUENCY and KD_IO_VIBRATE_MAXFREQUENCY have constant values which indicate the minimum and maximum (respectively) frequencies that the handset's vibrate implements. They are present if and only if output 4 is present.

Input KD_IO_VIBRATE_AVAILABILITY has a constant value which is a bitmap which indicates which I/O items are available, such that bit n is 1 if and only if I/O item index KD_IOGROUP_VIBRATE + n is available. Where an I/O item is mandatory, the corresponding bit is 1. All bits corresponding to I/O item indexes not defined above are 0. Thus, the value of the input is 9 if it is not possible to set the frequency, or 31 if it is possible.

20.4.5. KD_IOGROUP_POINTER

I/O group for pointer.

Synopsis

```
#define KD_IOGROUP_POINTER 0x4000
#define KD_IO_POINTER_X              (KD_IOGROUP_POINTER + 0)
#define KD_IO_POINTER_Y              (KD_IOGROUP_POINTER + 1)
#define KD_IO_POINTER_SELECT         (KD_IOGROUP_POINTER + 2)
```

Description

This I/O group defines the inputs in a pointer device such as a touchscreen pointer, mouse or trackpad.

Rationale

The primary role of this I/O group is for a touchscreen pointer on a handset, hence the limit of one button. If the platform has a mouse, then it is expected that it would be exposed by this I/O group, but if OpenKODE platforms with mice were to become common, the OpenKODE group would consider adding a new I/O group specifically for a mouse.

Inputs and outputs

index	type	range	usage
KD_IO_POINTER_X	mandatory KDint32 input	0..windowwidth-1	X coordinate
KD_IO_POINTER_Y	mandatory KDint32 input	0..windowheight-1	Y coordinate
KD_IO_POINTER_SELECT	mandatory binary input	0..1	select button

The X and Y coordinates use the top left of the current input focus window as the origin, and are mandatory inputs. The select button is also mandatory.

The button has a value of 1 when pressed and 0 when released. Separate events are generated for button press and button release at the appropriate times, even if there is no call to kdWaitEvent or kdPumpEvents in between the press and release.

Whether the OpenKODE input state for these inputs reflects the actual state of the inputs when no application window has the input focus is undefined.

It is undefined whether the OpenKODE input state is updated (and hence an event is generated if enabled) when the pointer is outside any window of the application.

The presence of the pointer I/O items can be tested by attempting to poll any of the inputs; if the poll function returns non-zero to indicate that the item was read, then the pointer I/O items are present.

If any of the inputs in this I/O group changes, the KD_EVENT_INPUT_POINTER event is generated, rather than the normal KD_EVENT_INPUT.

20.4.6. KD_IOGROUP_BACKLIGHT

I/O group for backlight.

Synopsis

```
#define KD_IOGROUP_BACKLIGHT 0x5000
#define KD_IO_BACKLIGHT_FORCE (KD_IOGROUP_BACKLIGHT + 0)
```

Description

This I/O group defines an output to control the handset’s backlight, such that an application can keep the backlight on even when the user is not pressing any keys or using other input.

I/O items

index	**type**	**range**	**usage**
KD_IO_BACKLIGHT_FORCE	mandatory KDint32 output		force backlight: non-zero to force backlight on, 0 to allow platform’s default backlight handling.

The initial value of KD_IO_BACKLIGHT_FORCE is 0.

20.4.7. KD_IOGROUP_JOGDIAL

I/O group for a jog dial.

Synopsis

```
#define KD_IOGROUP_JOGDIAL 0x6000
#define KD_IO_JOGDIAL_AVAILABILITY  (KD_IOGROUP_JOGDIAL + 0)
#define KD_IO_JOGDIAL_UP            (KD_IOGROUP_JOGDIAL + 1)
#define KD_IO_JOGDIAL_LEFT          (KD_IOGROUP_JOGDIAL + 2)
#define KD_IO_JOGDIAL_RIGHT         (KD_IOGROUP_JOGDIAL + 3)
#define KD_IO_JOGDIAL_DOWN          (KD_IOGROUP_JOGDIAL + 4)
#define KD_IO_JOGDIAL_SELECT        (KD_IOGROUP_JOGDIAL + 5)
```

Description

This I/O group defines a jog dial, either a three-way one with up and down movements and a select action, or a five-way one which additionally allows left and right movements.

A direction input has a value of either 0 or 1, and toggles from one to the other when the jog dial clicks one position in that direction. For a dial which can be moved one stop only in each direction, the clicks auto-repeat if the dial is held in that position. For a true wheel, clicks are related to how far the wheel is rotated in that direction.

Because of this toggling between 0 and 1 as clicks arrive, reading the input state for a direction input is generally not useful; the input is typically event enabled.

Events are generated to reflect the number of clicks, even when multiple clicks in a particular direction occur between calls to `kdWaitEvent` or `kdPumpEvents`.

The select button has a value of 1 when pressed and 0 when released. Separate events are generated for button press and button release at the appropriate times, even if there is no call to `kdWaitEvent` or `kdPumpEvents` in between the press and release.

Whether the OpenKODE input state and events reflect the physical inputs when no application window has the input focus is undefined.

I/O items

index	type	range	usage
`KD_IO_JOGDIAL_AVAILABILITY`	mandatory KDint32 input		availability bitmap
`KD_IO_JOGDIAL_UP`	mandatory binary input	0..1	toggled when dial clicked up
`KD_IO_JOGDIAL_LEFT`	optional binary input	0..1	toggled when dial clicked left
`KD_IO_JOGDIAL_RIGHT`	optional binary input	0..1	toggled when clicked right
`KD_IO_JOGDIAL_DOWN`	mandatory binary input	0..1	toggled when clicked down
`KD_IO_JOGDIAL_SELECT`	mandatory binary input	0..1	whether dial is being pushed in (or selected in some other way

`KD_IO_JOGDIAL_AVAILABILITY` is an input with constant value that indicates, using a bitmap, which inputs are present. Bit n represents input `KD_IO_JOGDIAL_UP + n`, set to 1 if the input is available and 0 if not, with unused bits set to 0. Thus the value of the input is 25 for a three-way jog dial, or 31 for a five-way jog dial.

20.4.8. KD_IOGROUP_JOYSTICK

I/O group for joystick.

Synopsis

```
#define KD_IOGROUP_JOYSTICK 0x10000
#define KD_IO_JOYSTICK_NUMSTICKS    (KD_IOGROUP_JOYSTICK + 0)
#define KD_IO_JOYSTICK_NUMBUTTONS   (KD_IOGROUP_JOYSTICK + 1)
#define KD_IO_JOYSTICK_NUMHATS      (KD_IOGROUP_JOYSTICK + 2)
#define KD_IO_JOYSTICK_NUMBALLS     (KD_IOGROUP_JOYSTICK + 3)

#define KD_IO_JOYSTICK_STICK            (KD_IOGROUP_JOYSTICK + 4)
#define KD_IO_JOYSTICK_STICK_NUMAXES    (KD_IO_JOYSTICK_STICK + 0)
#define KD_IO_JOYSTICK_X                (KD_IO_JOYSTICK_STICK + 1)
#define KD_IO_JOYSTICK_Y                (KD_IO_JOYSTICK_STICK + 2)
#define KD_IO_JOYSTICK_Z                (KD_IO_JOYSTICK_STICK + 3)
#define KD_IO_JOYSTICK_HAT              (KD_IOGROUP_JOYSTICK + 8)
#define KD_IO_JOYSTICK_HAT_UP           (KD_IO_JOYSTICK_HAT + 0)
#define KD_IO_JOYSTICK_HAT_LEFT         (KD_IO_JOYSTICK_HAT + 1)
#define KD_IO_JOYSTICK_HAT_RIGHT        (KD_IO_JOYSTICK_HAT + 2)
#define KD_IO_JOYSTICK_HAT_DOWN         (KD_IO_JOYSTICK_HAT + 3)
#define KD_IO_JOYSTICK_BALL             (KD_IOGROUP_JOYSTICK + 12)
#define KD_IO_JOYSTICK_BALL_X           (KD_IO_JOYSTICK_BALL + 0)
#define KD_IO_JOYSTICK_BALL_Y           (KD_IO_JOYSTICK_BALL + 1)
#define KD_IO_JOYSTICK_BUTTON           (KD_IOGROUP_JOYSTICK + 32)
```

```
#define KD_IO_JOYSTICK_STRIDE 64
```

Description

This I/O group defines a joystick.

A joystick contains the following elements:

- One or more *sticks*. Each stick has two or three axes. Each axis is a KDint32 input which takes an analog value from -32768 at one extreme to 0 in the middle to +32767 at the other extreme.
- One or more *buttons*, each of which is a binary input.
- Zero or more *hats*, each of which contains four binary inputs for the four directions.
- Zero or more *balls*, each of which contains two KDint32 inputs (for the two axes). Each input accumulates the deltas received from the associated ball axis, and will wrap when it increases past `KDINT32_MAX` or when it decreases past `KDINT32_MIN`.

Index range 0x10000..0x103ff is reserved for the first joystick, which imposes the following limits on a joystick: 16 sticks, 512 buttons, 16 hats and 16 balls.

Index range 0x10400..0x1ffff is reserved for up to a further 63 joysticks, where each joystick has the same limits as the first.

Whether the OpenKODE input state for these inputs reflects the actual state of the physical inputs when no application window has the input focus is undefined.

Inputs and outputs

index	type	range	usage
`KD_IO_JOYSTICK_NUMSTICKS`	mandatory KDint32 input	1 or more	number of sticks
`KD_IO_JOYSTICK_NUMBUTTONS`	mandatory KDint32 input	1 or more	number of buttons
`KD_IO_JOYSTICK_NUMHATS`	mandatory KDint32 input	0 or more	number of hats
`KD_IO_JOYSTICK_NUMBALLS`	mandatory KDint32 input	0 or more	number of balls
`KD_IO_JOYSTICK_STICK_NUMAXES`	KDint32 input	2 or 3	number of axes on first stick
`KD_IO_JOYSTICK_X`	KDint32 input	-32768..+32767	X axis of first stick
`KD_IO_JOYSTICK_Y`	KDint32 input	-32768..+32767	Y axis of first stick
`KD_IO_JOYSTICK_Z`	KDint32 input	-32768..+32767	Z axis of first stick; only present if the “number of axes” input for the stick has value 3
`KD_IO_JOYSTICK_HAT_UP`	binary input	0..1	whether first hat is being pushed up (or up-left or up-right)
`KD_IO_JOYSTICK_HAT_LEFT`	binary input	0..1	whether first hat is being pushed left (or up-left or down-left)

index	type	range	usage
KD_IO_JOYSTICK_HAT_RIGHT	binary input	0..1	whether first hat is being pushed right (or up-right or down-right)
KD_IO_JOYSTICK_HAT_DOWN	binary input	0..1	whether first hat is being pushed down (or down-left or down-right)
KD_IO_JOYSTICK_BALL_X	KDint32 input	KDINT32_MIN .. KDINT32_MAX	accumulated X deltas of first ball
KD_IO_JOYSTICK_BALL_Y	KDint32 input	KDINT32_MIN .. KDINT32_MAX	accumulated Y deltas of first ball
KD_IO_JOYSTICK_BUTTON	binary input	0..1	whether first button is pressed

For sticks, hats and balls, further instances after the first one within the same joystick are accessed by adding KD_IO_JOYSTICK_STRIDE to the index to access the second one, twice that to access the third one, and so on.

The indexes of buttons are arranged in contiguous ranges of 32. Thus the index of button n (where n=0 is the first one) is KD_IO_JOYSTICK_BUTTON + (n % 32) + (n / 32 * KD_IO_JOYSTICK_STRIDE).

If any stick axis input changes, the KD_EVENT_INPUT_STICK event is generated, rather than the normal KD_EVENT_INPUT.

Further joysticks after the first are accessed by adding (n * 0x400) to the index number, for 0<=n<=63.

20.4.9. KD_IO_UNDEFINED

I/O items reserved for implementation-dependent use.

Synopsis

```
#define KD_IO_UNDEFINED 0x40000000
```

Description

I/O indexes in the range KD_IO_UNDEFINED..KDINT32_MAX are reserved for implementation-dependent I/O items.

The indexes in this range do not form a single I/O group. Instead, the index range contains 0 or more I/O groups; for each one, the types, indexes and semantics of the I/O items contained in it are undefined.

21. Windowing

21.1. Introduction

The windowing API provides an abstraction for managing windows and performing simple operations on them.

An implementation is only required to support one full-screen window. However, an implementation is also allowed to support more windows.

A full-screen window is not necessarily exactly the size of the display, as some platforms may wish to display graphic items such as battery power indicators at the edges of the screen, and only dedicate the area remaining from them for application use.

21.1.1. Future directions

A future version of OpenKODE will include OpenSL ES as a client API. Then, an implementation will be allowed to support *just* OpenKODE and OpenSL ES, without EGL or any EGL client APIs. In such an implementation, the window functions in this section will not be present.

21.2. Types

`KDWindow`	An opaque struct used to represent a window. A pointer to this type is used as a handle to a window.

21.3. Functions

21.3.1. kdCreateFullScreenWindow

Create a full-screen window.

Synopsis

```
KDWindow *kdCreateFullScreenWindow(EGLDisplay display, const void *mode, void
*eventuserptr);
```

Description

This function creates a single “full-screen”window on a display. It is undefined whether such a window is actually full screen, and if not what its position and size are. The window is created visible.

A window created by this function must be the only window owned by the application; any attempt to create another window when the application already owns a full-screen one, or any attempt to create a full-screen window when the application already owns a window of either type, generates an error.

On entry, *`display`* is the EGL display handle of the display on which the window is to appear. This handle is as returned by the EGL function `eglGetDisplay`.

The *`mode`* parameter must be `KD_NULL`.

It is mandatory for an OpenKODE Core implementation to support creating a window via this kdCreateFullScreenWindow function.

eventuserptr is the value to use for the *userptr* of any event associated with the window. If *eventuserptr* is KD_NULL, then the window's KDWindow * is used as the user pointer instead.

A window created with this function can be detroyed using kdDestroyWindow, or will be destroyed automatically (after freeing EGL resources) on application exit by OpenKODE Core.

If *display* is not an EGL display handle then undefined behavior results.

Return value

On success, the function returns the KDWindow * pointer for the newly created window. This window supports all window-capable configs exposed by EGL for the *display*. Otherwise the function returns KD_NULL and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_EINVAL	*mode* is not KD_NULL.
KD_ENOMEM	Out of memory or other resource.
KD_EPERM	Attempt to create a full-screen window when the application already has a window.

Rationale

The *mode* parameter, which must be KD_NULL, may be used in a future version of the specification to allow the application to set the screen mode, on a platform where that is a meaningful concept.

21.3.2. kdCreateWindow

Create a window.

Synopsis

```
KDWindow *kdCreateWindow(EGLDisplay display, void *eventuserptr);
```

Description

This function creates a window on a display in such a way that multiple windows are allowed. It is implementation defined whether this kind of window is supported at all; if not, this function always fails.

The initial size and position of the window are undefined.

If the application already owns a full-screen window, this function generates an error.

On entry, *display* is the EGL display handle of the display on which the window is to appear. This handle is as returned by the EGL function eglGetDisplay.

eventuserptr is the value to use for the *userptr* of any event associated with the window. If *eventuserptr* is KD_NULL, then the window's KDWindow * is used as the user pointer instead.

If *display* is not an EGL display handle then undefined behavior results.

Return value

On success, the function returns the KDWindow * pointer for the newly created window, which is initially in the hidden state. This window supports all window-capable configs exposed by EGL for the *display*. Otherwise the function returns KD_NULL and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_ENOMEM	Out of memory or other resource.
KD_EOPNOTSUPP	Implementation does not support this kind of window.
KD_EPERM	Attempt to create a window when the application already has a full-screen window.

21.3.3. kdDestroyWindow

Destroy a window.

Synopsis

```
void kdDestroyWindow(KDWindow *window);
```

Description

This function destroys *window* and all OpenKODE Core resources related to it. EGL resources associated with the window must be freed before calling this function.

If *window* is not a window, or has already been destroyed, or has EGL resources associated with it, then undefined behavior results.

21.3.4. kdShowWindow

Set window’s visibility status.

Synopsis

```
KDint kdShowWindow(KDWindow *window, KDint status);
```

Description

For the specified *window*, this function sets its visibility status according to the value of *status*:

KD_WINDOWSTATUS_HIDDEN (0)	Window is completely hidden; it is not shown at all.
KD_WINDOWSTATUS_VISIBLE (1)	Window is visible.
KD_WINDOWSTATUS_MINIMIZED (2)	Window is minimized to taskbar or equivalent.

If *window* is a full-screen window (one created with kdCreateFullScreenWindow), then the function fails with an error. For a normal (non-full-screen) window, it is undefined whether KD_WINDOWSTATUS_MINIMIZED is supported; if not, a request to set it causes this function to fail with an error.

If *window* is not a window, or has been destroyed, then undefined behavior results. If *status* is not one of the values listed above, then it is undefined whether the function succeeds or fails with an error, and, if it succeeds, what the resulting change to the window's visibility status is.

Return value

On success, this function returns 0. Otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM`	Out of memory or other resource.
`KD_EINVAL`	*`window`* is a full-screen window.
`KD_EOPNOTSUPP`	The implementation does not support the requested operation.

21.3.5. kdGetWindowNativeType

Get the window handle for passing to EGL

Synopsis

```
void *kdGetWindowNativeType(KDWindow *window);
```

Description

For the specified *`window`*, this function returns the native window handle required by the EGL function `eglCreateWindowSurface`.

If *`window`* is not a window, or has been destroyed, then undefined behavior results.

Return value

On success, the function returns the native window handle. It cannot fail.

21.3.6. kdActivateWindow

Give focus to a window

Synopsis

```
KDint kdActivateWindow(KDWindow *window);
```

Description

This function activates *`window`*, giving it focus. In the case that no window owned by the application had focus, it is undefined whether this function actually gives the window focus, or merely marks that it will have focus next time the user selects the application. It is implementation defined whether this function makes the window visible if it was hidden or minimized before.

If some other window owned by the application has focus, this function causes it to lose focus, so a `KD_EVENT_WINDOW_FOCUS` event for that window is generated.

If this function does cause the specified window to gain focus where it did not have it before, it generates a

`KD_EVENT_WINDOW_FOCUS` event for the window.

If *`window`* is not a window, or has been destroyed, then undefined behavior results.

Return value

On success, this function returns 0. Otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM` Out of memory or other resource.

21.3.7. kdSetWindowCaption

Set window caption

Synopsis

```
KDint kdSetWindowCaption(KDWindow *window, const KDchar *caption);
```

Description

This function sets *`window`*'s caption to the text *`caption`*. It is implementation defined whether and where the caption is displayed, and whether there is a length limit after which the supplied caption text is ignored.

If *`window`* is not a window, or has been destroyed, or *`caption`* does not point to a readable null-terminated UTF-8 string, then undefined behavior results.

Return value

On success, this function returns 0. Otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM` Out of memory or other resource.

21.3.8. kdGetWindowPosition

Get window position

Synopsis

```
KDint kdGetWindowPosition(KDWindow *window, KDint *x, KDint *y);
```

Description

This function retrieves the display coordinates of *`window`*'s top left corner, and stores them in `*`*`x`* and `*`*`y`*. Each of *`x`* and *`y`* is allowed to be `KD_NULL`, in which case the corresponding coordinate is not stored.

If *`window`* is a full-screen window, the returned coordinates are undefined.

If *window* is not a window, or has been destroyed, or either of *x* or *y* is not KD_NULL or a pointer to a writable KDint, then undefined behavior results.

Return value

On success, this function returns 0. Otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_ENOMEM	Out of memory or other resource.

21.3.9. kdSetWindowPosition

Set window position

Synopsis

```
KDint kdSetWindowPosition(KDWindow *window, KDint x, KDint y);
```

Description

This function updates *window*'s position so its top left is at (*x*, *y*) on the display. If called with coordinates that cannot be achieved, or if called on a full-screen window, the request is ignored and the function succeeds.

If *window* is not a window, or has been destroyed, then undefined behavior results.

Return value

On success, this function returns 0. Otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by kdGetError.

Error codes

KD_ENOMEM	Out of memory or other resource.

21.3.10. kdSetWindowSize

Set window size

Synopsis

```
KDint kdSetWindowSize(KDWindow *window, KDint width, KDint height);
```

Description

This function attempts to resize *window* such that its content area has the specified dimensions. (The *content area* of a window is the actual 2D pixel array that is accessible by the application through EGL and its client APIs.)

If the requested dimensions are not achievable, then different dimensions are used instead in an implementation-dependent way, and this may include leaving the dimensions as they are now. The dimensions of a full-screen window always remain unchanged.

If one or both dimensions of the window change, then a `KD_EVENT_WINDOW_RESIZE` event is generated.

If *`window`* is not a window, or has been destroyed, then undefined behavior results.

Return value

On success, this function returns 0. Otherwise it returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_ENOMEM` Out of memory or other resource.

Rationale

Note that there is no OpenKODE Core function to get the window size, since EGL can be used for that once a surface has been created from the window.

21.4. Events

21.4.1. KD_EVENT_WINDOW_CLOSE

Event to request to close window.

Synopsis

```
#define KD_EVENT_WINDOW_CLOSE 2
```

Description

This event type is generated by OpenKODE Core (typically as the result of a request from the underlying OS) to signal that the window should close. The event has no associated data, but the event's *`userptr`* field is set to the *`eventuserptr`* value for the window which is being asked to close. This value was supplied by the application when the window was created with `kdCreateWindow`.

Application asking to close its own window

An application can post this event to itself using `kdPostEvent`. It is up to the application to ensure that the event's *`userptr`* field is set to a value that the event's handler code is expecting (if any).

21.4.2. KD_EVENT_WINDOW_RESIZE

Window resize event.

Synopsis

```
#define KD_EVENT_WINDOW_RESIZE 5
```

Description

This event type signals that a window being used by the application has changed size (including the case where the window resize is caused by the display changing orientation). The event has no associated data, but the event's *userptr* field is set to the *userptr* value for the window which is being resized. This value was supplied by the application when the window was created with kdCreateWindow.

KD_EVENT_WINDOW_RESIZE events merge: if such an event generated by OpenKODE Core is queued when another one generated by OpenKODE Core for the same window is already in the queue, then the earlier one is removed.

It is possible to receive this event even when the application has only a full-screen window, typically when the user reorients the handset such that the screen dimensions are swapped.

21.4.3. KD_EVENT_WINDOW_FOCUS

Window focus gained/lost event.

Synopsis

```
#define KD_EVENT_WINDOW_FOCUS 6
```

Description

The KD_EVENT_WINDOW_FOCUS event type signals that a window being used by the application has gained or lost the focus.

The event data is in event->data.windowfocus element of the event's data union, which has the following type:

```
typedef struct KDEventWindowFocus {
    KDint hasfocus;
} KDEventWindowFocus;
```

The *hasfocus* field contains 0 if focus has been lost, or 1 if focus has been gained.

The event's *userptr* field is set to the *userptr* value for the window which is losing or gaining focus. This value was supplied by the application when the window was created with kdCreateWindow.

22. Assertions and logging

22.1. Introduction

OpenKODE Core provides C standard-like assertions, and in addition specifies the function that is called when an assertion fails, so an application may override it. A a means of sending output to an implementation-defined debug log file or other location is also provided.

These facilities are intended to help the programmer when writing and debugging code. In production code, they are disabled by defining the `KD_NDEBUG` macro.

22.2. Functions

22.2.1. kdAssert

Test assertion and call assertion handler if it is false

Synopsis

```
kdAssert(condition);
```

Description

If the macro `KD_NDEBUG` was defined at the point that `<KD/kd.h>` was first included, then `kdAssert` does nothing, and does not evaluate its argument.

Otherwise, `kdAssert` evaluates its argument exactly once as a condition, and, if it is false, it calls `kdHandleAssertion` to output a message to indicate the assertion failure and then terminate the application.

`kdAssert` is a macro, which means that it is not possible to take the address of it.

22.2.2. kdHandleAssertion

Handle assertion failure.

Synopsis

```
void kdHandleAssertion(const KDchar *condition, const KDchar *filename, KDint
linenumber);
```

Description

This function is the default handler for a failed `kdAssert`. It outputs a message containing *`condition`*, *`filename`* and *`linenumber`* as if by `kdLogMessage`, and then terminates the application.

An application can override this handler by defining its own `kdHandleAssertion` and using an implementation-defined mechanism to ensure it is linked in to the application before the OpenKODE Core provided one.

22.2.3. kdLogMessage

Output a log message.

Synopsis

```
#ifdef KD_NDEBUG
#define kdLogMessage(s)
#else


void kdLogMessage(const KDchar *string);
```

Description

If the macro `KD_NDEBUG` was defined at the point that `<KD/kd.h>` was first included, then `kdLogMessage` does nothing, and does not evaluate its argument.

Otherwise, it evaluates its argument exactly once, and logs it as a message to the usual debug log location on the device. This could be a file, a debugger window or similar. A newline is added unless the string already ends in a newline. Embedded newlines are permitted.

`kdLogMessage` may be a macro, which means that it is undefined whether it is possible to take the address of it.

Appendix A. OpenKODE versions and changes

A.1. OpenKODE 1.0 Provisional

OpenKODE 1.0 Provisional was approved by the Khronos Board of Promoters on February 8th, 2007.

A.1.1. Acknowledgements

OpenKODE 1.0 Provisional is the result of the contributions of many people, representing a cross section of the hand-held and embedded computer industry. Following is a partial list of contributors, including the company they represented at the time of their contributions:

Mikko Strandborg (Acrodea); Keh-Li Sheng (Aplix); Ed Plowman (ARM); Roger Nixon (Broadcom); Paul Novak (Ericsson); Brian Murray (Freescale); Petri Talala (Futuremark); Timo Suoranta (Futuremark); Avi Shapira (Graphic Remedy); Yaki Tebeka (Graphic Remedy); Mark Callow (HI); Hwanyong Lee (Huone); Leon Clarke (Ideaworks3D); Aaron Burton (Imagination); Eero Penttinen (Nokia); Pasi Keranen (Nokia); Neil Trevett (NVIDIA); Petri Kero (Hybrid/NVIDIA); Ville Miettinen (NVIDIA); Aviad Lahav (Samsung); Remi Arnaud (Sony); Gabriele Svelto (STMicroelectronics); Jerry Evans (Sun); Bill Pinnell (Symbian); Robert Palmer (Symbian); Phil Huxley (Tao); Tim Renouf (Tao); Leo Estevez (TI); Marion Lineberry (TI); Tom Olsen (TI); Jon Leech.

A.1.2. Revisions

Revision 1, 2007-03-30

- Added “OpenKODE versions and changes” appendix.
- Stated that the window returned by kdCreateFullScreenWindow or kdCreateWindow supports all window-capable configs exposed by EGL for the display.
- Clarified that a KDEvent cannot be accessed or freed after passing to kdPostEvent, even if kdPostEvent fails.
- Changed KD_MAXFLOATF to KD_MAXFLOAT.
- Clarified that an event posted to kdPostEvent can have any userptr value and event data, even if it is an event type defined by OpenKODE Core, and the event is not altered (except possibly for the timestamp field) by kdPostEvent.
- Added descriptive note that an implementation can delay the generation of a socket event on a particular condition until the next kdWaitEvent/kdPumpEvents (if not already in one).
- Fixed problem where multiple lines before the function prototype in a function's synopsis were jammed together without linebreaks.
- Fixed the description of KD_EVENT_INPUT_STICK, which was incorrectly referring to event data union element inputpointer of type EventInputPointer. It has been changed to element inputstick of type EventInputStick.
- kdNameLookupCancel is now specified to remove any pending events from a completed kdNameLookup matching the removal criterion.
- The value of KD_AF_INET has changed from 0x800 to 2, to match the commonly used value of AF_INET.
- Inconsistencies in the name of the window focus event and its associated event data and type have been removed.

The event is now known as KD_EVENT_WINDOW_FOCUS, and its event data union element is windowfocus of type KDEventWindowFocus.

- Minor clarifications have been made in the I/O index numbering of multiple parts (e.g. sticks) within a joystick, and of multiple joysticks. In addition, a statement that up to 16 axes are supported within one joystick has been fixed to be up to 16 sticks.
- The `/native` file area is now completely undefined, removing any unintended hint that mapping the platform's native file system is mandated.
- The mode parameter to kdCreateFullScreenWindow has been changed so it must be KD_NULL. Thus there is now no way for an application to request a particular screen mode, although the parameter remains for a future extension or change to re-add this functionality.
- Mention has been added to kdOutputSet* functions that a platform may virtualize its outputs.
- The EGL display parameter to kdCreateFullScreenWindow and kdCreateWindow has been changed to type EGLDisplay. To accommodate this, <KD/kd.h> is now defined to include <EGL/egl.h>. Notes have been added that this need not be the case in a future version of OpenKODE where an OpenKODE and OpenSL ES only implementation is supported.
- The definitions of certain I/O groups have been clarified by stating in the table of I/O indexes whether the I/O item is mandatory or optional.

Revision 0, 2007-02-08

Initial revision.

Bibliography

[C89] *ANSI X3.159-1989 “Programming Language C”* .

[C99] *ISO/IEC 9899:TC2 “Programming Language C”* .

[IEEE 754] *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)* .

[POSIX] *IEEE Std 1003.1, 2004 Edition (“Single Unix Specification version 3”)* .

Index