OpenKODE 1.0.2 Specification

Edited by Tim Renouf

OpenKODE 1.0.2 Specification:

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 an open, royalty-free standard to enable source portability for demanding mixed media applications such as advanced user interfaces, navigation software, media players and games. OpenKODE defines a collection of native APIs to provide comprehensive system, graphics and media functionality that can be made reliably available on diverse platforms such as Linux, Brew, Symbian, Windows Mobile, WIPI and Nucleus.

OpenKODE includes the new OpenKODE Core API that provides a thin, POSIX-like, multi-threaded abstraction into operating system resources such as IO devices, files and networking to minimize source fragmentation when porting applications between devices with different operating systems.

OpenKODE 1.0 also includes the OpenGL ES and OpenVG Khronos media APIs combined with the EGL API that provides abstracted access to native windowing systems and rendering surfaces to enable state-of-the-art mixed-mode acceleration for vector 2D and 3D graphics. Subsequent versions of OpenKODE will add the OpenSL ES and OpenMAX AL media APIs to provide accelerated video and audio that is fully integrated with graphics processing.

OpenKODE defines a full set of conformance tests for OpenKODE Core, together with trans-API tests to ensure that the defined mixed mode media, graphics and operating system functionality interoperate correctly on conformant implementations. Hence, OpenKODE provides a reliable and trusted set of functionality for developers – to enable and encourage the development of portable applications that use advanced mixed media functionality.

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 1.0.2

2. OpenKODE conformance

2.1. Conformance tests

An organization may sign up to be an *OpenKODE adopter* in order to gain access to the OpenKODE adopters' page, allowing download of the OpenKODE conformance tests and upload of an OpenKODE conformance submission. The rules governing this process are specified in the Khronos Group Conformance Process document linked to from that page.

2.2. Conformant OpenKODE implementation

OpenKODE Core's networking is optional only in the sense that an implementation may return the specified error from certain functions to indicate that networking is not supported. Thus the functions and constants in that part of the specification must always be present.

If networking is supported, the implementation may support either or both of UDP and TCP.

In addition, if TCP networking is supported, support for listening and accepting connections is optional, again in the sense that certain functions return a specified error if not supported.

OpenKODE Core's threading is optional only in the sense that an implementation may return the specified error from certain functions to indicate that threading is not supported. Thus the functions and constants in that part of the specification must always be present.

OpenKODE Core optionally supports static data. However this support is mandatory if the implementation supports threading.

A conformant OpenKODE 1.0.2 implementation may be either windowless or windowing:

2.2.1. Windowless OpenKODE

A conformant implementation may consist of OpenKODE Core 1.0.2 without the windowing API (thus all functions and constants in *Windowing* are not present). In this case, EGL may be included, but it must be incapable of creating a window surface.

2.2.2. Windowing OpenKODE

A conformant implementation may consist of OpenKODE Core 1.0.2 with the windowing API. In this case, EGL must be supported, must be capable of creating a window surface, and must include the lock surface extension (below).

In addition, a windowing OpenKODE 1.0.2 implementation may include zero or more of the following media APIs:

- OpenGL ES 1.1, with, if OpenVG is also present, the GL_OES_egl_image extension (version 4);
- OpenVG 1.0.1, with, if OpenGL ES is also present, the VG_KHR_egl_image extension (version 4);

2.3. EGL

When in an OpenKODE 1.0.2 implementation, EGL has the following requirements:

2.3.1. Supported client APIs

EGL is allowed to support more client APIs than are included in the conformant OpenKODE implementation. For example, the conformant OpenKODE implementation might include OpenGL ES 1.1, where EGL also allows OpenVG 1.0 in such a way that it is not OpenKODE conformant.

2.3.2. EGLImage-related extensions

If both OpenGL ES and OpenVG are present, then the EGL_KHR_image (version 7), EGL_KHR_gl_texture_2D_image (version 4), EGL_KHR_gl_texture_cubemap_image (version 4), EGL_KHR_gl_texture_3D_image (version 4), EGL_KHR_gl_renderbuffer_image (version 4) and EGL_KHR_vg_parent_image (version 4) extensions must be present.

2.3.3. Lock surface extension

If an EGL implementation in which it is possible to create a window surface is included in the conformant OpenKODE implementation, then the EGL_KHR_lock_surface extension (version 16) 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 which is returned as a match when the attribute `EGL_MATCH_FORMAT_KHR` has the value `EGL_FORMAT_RGB_565_EXACT_KHR` in a call to `eglChooseConfig`.

Note that, if either OpenGL ES or OpenVG is present, there is no requirement for there to be any config which is both lockable (has `EGL_LOCK_SURFACE_BIT` set) and is capable of OpenGL ES or OpenVG rendering.

2.3.4. 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).

An implementation *can* choose to export configs supporting creation of lockable surfaces which also support rendering by OpenGL ES, OpenVG, or other client APIs (when the surface is not locked). But this specification does not require this, and the motivation for the lock surface extension is simply to support software rendering.

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.

2.3.5. 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)

2.3.6. EGL power management event

EGL allows for a power management event which causes all contexts to be lost, such that the application must recreate any context(s) it was using. It is outside the scope of EGL how such a power management event is caused or how the application might be notified of this.

OpenKODE Core specifically states that, on some implementations, a `KD_EVENT_PAUSE` event may be such an EGL power management event. However, the application is not able to recreate its context(s) until after a `KD_EVENT_RESUME` event has been received.

2.4. Future directions

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

Part II. OpenKODE Core 1.0.2

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.

Threads

OpenKODE Core provides a threading API for thread creation and synchronization. Although the API is mandatory, some implementations support only a single thread and thus give an error code on an attempt to create a new thread.

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>
```

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

The OpenKODE Core header file defines the following macro to aid in compile-time OpenKODE version detection:

```
#define KD_VERSION_1_0 1
```

4.1.1. Note for implementers

Khronos supplies a reference `KD/kd.h` and sample `KD/kdplatform.h` along with this specification, where `KD/kd.h` contains portable definitions, #including `KD/kdplatform.h` for the non-portable defnitions.

The intention is that an implementer does not need to edit `KD/kd.h`; it should be usable as it is supplied. All edits should be applied to `KD/kdplatform.h`.

Implementers are encouraged to code `KD/kdplatform.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.

In an implementation that succeeds in not including any of the platform's include files in `KD/kdplatform.h`, there will be no clashes between OpenKODE-defined symbols and symbols of the same name in the platform's include files, for example where `s_addr` is both a struct member name in OpenKODE and a `#define` preprocessor symbol in `winsock.h` on Windows.

However, such clashes may occur when using `KD/kd.h` and `KD/kdplatform.h` *within the OpenKODE implementation*. Source files within the implementation may need to take avoiding action before #including `KD/kdplatform.h`.

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 mandatorily supported. It is expected that most OpenKODE Core implementations will support them; one which does not has `KD_NO_STATIC_DATA` #defined in `KD/kd.h`, and is then said to implement *OpenKODE Core without static data*. An OpenKODE Core implementation that supports threads (so `kdThreadCreate` does not return `KD_ENOSYS`) must support non-automatic variables (static data).

Rationale

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.

Such low-end platforms without non-automatic variables are unlikely to support threading. Therefore OpenKODE Core mandates that an implementation with threading must always support non-automatic variables.

- Memory consists 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.

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 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 (with undefined order of evaluation, as normal in C).
- 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.

4.3.1. 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.

4.3.2. Error detection

Most functions in this specification have a *Error codes* section. Unless otherwise specified, the condition listed with an error code must be detected and cause that error. If two or more of the listed conditions occur together, it is undefined which condition’s error code results.

4.3.3. Undefined behavior

Within this document, certain circumstances are specified as resulting in undefined behavior. Possible results include but are not limited to:

- the operation succeeds, or appears to succeed;
- the operation fails with an error code that may or may not be specified for that operation;
- the application terminates;
- the application continues but then a later operation fails;
- the application continues but then terminates later.

It is expected, but not mandated, that an application terminating due to undefined behavior will not cause the rest of the platform to crash or terminate.

4.4. Thread safety

Except as otherwise noted in individual function specifications, it is safe to make concurrent OpenKODE function calls from multiple OpenKODE threads, where an OpenKODE thread is either the main thread in which `kdMain` was called, or a thread created using `kdThreadCreate`.

The exceptions noted in individual function specifications are that any function taking a file, directory, socket, window or timer handle cannot overlap a function in another thread using the same handle. The exception to that is that `kdSocketAccept` is thread safe with respect to other function calls using the same socket handle.

Using an OS-specific mechanism to create a new thread then calling an OpenKODE function from that thread results in undefined behavior.

4.4.1. 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`, `kdPostThreadEvent` 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 outside any OpenKODE thread.

4.5. 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 two's complement signed integer

type	description
`KDuint16`	16-bit binary unsigned integer
`KDint8`	8-bit binary two’s complement 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`	two’s complement signed binary integer the same size as `KDsize`
`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
`KDmode`	as `KDuint32`, used for the *`st_mode`* field in a `KDStat` structure

OpenKODE Core does not provide a means for specifying an integer constant bigger than 32 bits. For portability, an application should specify a bigger-than-32-bit integer constant by combining two constants. For example, `(((KDint64)0x12345678 << 32) | 0x9abcdef0U)` to obtain the 64-bit constant `0x123456789abcdef0`.

4.5.1. Rationale

OpenKODE Core provides only 32-bit floats, not 64-bit doubles. It was judged that developers of demanding interactive applications use 32-bit floats in preference for increased performance, especially when no floating point hardware is available. Khronos defines the extension KHR_float64 to allow an OpenKODE implementation to include 64-bit floats, but that is outside the scope of this specification.

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.

4.6. 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`

constant	value	description
`KDINT64_MIN`	`-0x8000000000000000`	minimum value of `KDint64`
`KDINT64_MAX`	`0x7fffffffffffffff`	maximum value of `KDint64`
`KDUINT64_MAX`	`0xffffffffffffffff`	maximum value of `KDuint64`
`KDSSIZE_MIN`		minimum value of `KDssize`
`KDSSIZE_MAX`		maximum value of `KDssize`
`KDSIZE_MAX`		maximum value of `KDsize`
`KDUINTPTR_MAX`		maximum value of `KDuintptr`
`KD_TRUE`	`1`	canonical true value of a `KDboolean`
`KD_FALSE`	`0`	false value of a `KDboolean`

In addition, OpenKODE defines the following:

constant	defined as
`KD_NULL`	`((void *)0)` OpenKODE Core specifies that the programming environment is C, not C++. However, in an implementation that allows C++, when C++ is in use, `KD_NULL` needs to be defined as something like `const int KD_NULL = 0;` to avoid the need for casts when using it.
`KD_NORETURN`	Macro used in a function declaration to declare that a call to the function never returns.

4.7. Main thread stack size

The OpenKODE Core main thread (the thread in which the implementation calls `kdMain` at application startup) is defined to have a stack that is large enough for the application to do the following (not at the same time):

- have a 15000 byte automatic variable;
- include the following code and call `testrecurse(0, 0)` (to make it recurse 625 times):

```
struct recurse {
    struct recurse *next;
    KDint32 value;
};

static KDint32 testrecurse(KDint32 count, struct recurse *lastrecurse)
{
    if (count != 625)
    {
        struct recurse thisrecurse;
        thisrecurse.value = ++count;
        thisrecurse.next = lastrecurse;
        return testrecurse(count, &thisrecurse);
    }
    else
    {
        KDint32 product = 1;
        while (lastrecurse)
        {
            product = product * lastrecurse->value | 1;
            lastrecurse = lastrecurse->next;
        }
        return product;
    }
}
```

4.7.1. Rationale

These criteria are intended to specify that the stack is around 15k or 16k (for a 32-bit platform) in a way that can be tested in the OpenKODE conformance tests.

4.8. Extensions

An extension to OpenKODE Core may be registered at Khronos.

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.

An extension is expected to follow these rules:

- The identifiers added in the extension follow the conventions used in the OpenKODE Core specification, except that a suffix consisting of a small number of upper-case letters identifying the vendor defining the extension is added to every identifier. For a constant, an underscore is used before this suffix. The suffix is `KHR` for a Khronos-defined extension, or `VEN` for an extension agreed between a number of vendors.

- Each extension has its own include file in the `KD` directory.

- An extension supplied with an OpenKODE implementation has a constant of the same name as the extension’s “name string” (for example `KD_KHR_crypto`) #defined in `KD/kd.h`, so that an application may tell at compile time whether the extension is present. This rule does not apply to an extension supplied by a third party, since it is then not possible to add the definition to `KD/kd.h`.

It is recommended that, where applicable, an OpenKODE Core implementation provides a mechanism allowing a third party to add an extension to it. This mechanism would typically be as simple as adding an extra library to the link command of an OpenKODE application wishing to use the extension.

Unlike other Khronos APIs, no mechanism is provided for dynamically determining which extensions are present. This is because it is viewed as likely that any OpenKODE extension would add functionality such that an application needs to know at compile time that it is present, not at run time. Also, not having such a mechanism simplifies the adding of an OpenKODE Core extension by a vendor other than the vendor of the main OpenKODE Core implementation.

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` (1)	Permission denied.
`KD_EADDRINUSE` (2)	Address in use.
`KD_EADDRNOTAVAIL` (3)	Address not available on the local platform.
`KD_EAFNOSUPPORT` (4)	Address family not supported.
`KD_EAGAIN` (5)	Resource unavailable, try again.
`KD_EALREADY` (6)	A connection attempt is already in progress for this socket.
`KD_EBADF` (7)	File not opened in the appropriate mode for the operation.
`KD_EBUSY` (8)	Device or resource busy.
`KD_ECONNREFUSED` (9)	Connection refused.
`KD_ECONNRESET` (10)	Connection reset.
`KD_EDEADLK` (11)	Resource deadlock would occur.
`KD_EDESTADDRREQ` (12)	Destination address required.
`KD_EEXIST` (13)	File exists.
`KD_EFBIG` (14)	File too large.
`KD_EHOSTUNREACH` (15)	Host is unreachable.
`KD_EHOST_NOT_FOUND` (16)	The specified name is not known.
`KD_EINVAL` (17)	Invalid argument.
`KD_EIO` (18)	I/O error.
`KD_EILSEQ` (19)	Illegal byte sequence.
`KD_EISCONN` (20)	Socket is connected.
`KD_EISDIR` (21)	Is a directory.

`KD_EMFILE` (22)	Too many open files.
`KD_ENAMETOOLONG` (23)	Filename too long.
`KD_ENOENT` (24)	No such file or directory.
`KD_ENOMEM` (25)	Not enough space.
`KD_ENOSPC` (26)	No space left on device.
`KD_ENOSYS` (27)	Function not supported.
`KD_ENOTCONN` (28)	The socket is not connected.
`KD_ENO_DATA` (29)	The specified name is valid but does not have an address.
`KD_ENO_RECOVERY` (30)	A non-recoverable error has occurred on the name server.
`KD_EOPNOTSUPP` (31)	Operation not supported.
`KD_EOVERFLOW` (32)	Overflow.
`KD_EPERM` (33)	Operation not permitted.
`KD_ERANGE` (35)	Result out of range.
`KD_ETIMEDOUT` (36)	Connection timed out.
`KD_ETRY_AGAIN` (37)	A temporary error has occurred on an authoratitive name server, and the lookup may succeed if retried later.

5.3. Functions

5.3.1. kdGetError

Get last error indication.

Synopsis

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

Description

OpenKODE Core maintains a per-thread last error indication, which is set by certain functions to indicate an error, as specified with each such function. This last error indication is initially 0. Other than `kdSetError` below, no OpenKODE Core function sets the last error indication to a value other than as specified for that function, and no OpenKODE Core function sets the last error indication unless its return value is one defined to set the last error indication in the specification of the function.

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

Return value

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

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.

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`. Currently no `attribute` is supported.

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 (39)`	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 (40)`	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 major portion must be 1; the minor portion must be no less than the minor portion of the version of this OpenKODE Core specification. 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.
`KD_ATTRIB_PLATFORM (41)`	A free-format string is returned which indicates in some way the exact platform (handset etc) on which the OpenKODE Core implementation is running. The string is recommended to contain at least the platform vendor, to avoid clashes with other vendors' platforms. Beyond that, the string might identify the platform in an OpenKODE implementation-dependent way, possibly including the series name, the model name and the firmware revision. An application may use this, possibly together with `KD_ATTRIB_VENDOR` (the OpenKODE vendor), to tailor itself to the platform, for example relying on the exact form factor and button layout. However a portable application should still be able to function using a default input mapping when it is running on a platform that it does not recognize.

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. Currently no *`attribute`* is supported.

Return value

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

7. Threads and synchronization

7.1. Introduction

OpenKODE Core optionally supports threads. An implementation that does not support threads gives the error `KD_ENOSYS` as specified in certain functions below. An implementation that does support threads supports all of the functionality in this section, and thus does not give `KD_ENOSYS` for any of the functions in this section.

Multiple threads run pre-emptively, that is a thread does not need to take any yield action to ensure that other threads can run.

Further, an implementation that supports threads also supports non-automatic (global, static and file scope) data in the application.

7.2. Overview

7.2.1. Thread handling

The threads and synchronization API supported by OpenKODE Core is based on a subset of [POSIX] threads, plus the non-inter-process functionality of unnamed semaphores.

A new thread is created using `kdThreadCreate`. This function takes a pointer to the function to run in the new thread. The new thread exits either when it returns from that function, or when it calls `kdThreadExit`. A thread ID of type KDThread* is returned by `kdThreadCreate`.

`kdThreadCreate` also optionally takes a *thread attributes object* handle, used to specify whether the thread is created detached and its stack size. The thread attributes object is created with `kdThreadAttrCreate`, modified with `kdThreadAttrSetDetachState` and `kdThreadAttrSetStackSize`, and freed with `kdThreadAttrFree`.

A thread is in one of these two states:

- *Joinable*, which means that another thread can wait for it to finish and collect its return value by calling `kdThreadJoin`. This means that, when the thread exits, resources associated with it are not freed until another thread has called `kdThreadJoin` on it.
- *Detached*, which means that its resources are freed as soon as it exits, but it is not possible for another thread to wait for it to finish and collect its return value.

A newly created thread is by default in the joinable state, although that can be changed using `kdThreadAttrSetDetachState` on the attributes used to create the thread. A thread in the joinable state can be changed to the detached state using `kdThreadDetach`.

7.2.2. Dynamic initialization

`kdThreadOnce` is used to ensure that a function (for example an initialization function) is executed only once, in whichever thread reaches that point first.

`kdThreadOnce` works on an OpenKODE Core implementation that does not support threading, still only allowing the initialization function to be executed once.

7.2.3. Mutexes

A *mutex* is a synchronization primitive which can be locked by at most one thread at a time. It is typically used to protect access to some resource which only one thread can access at a time.

A mutex is created by a call to `kdThreadMutexCreate`, which returns a handle to the new mutex, and freed by a call to `kdThreadMutexFree`.

A thread locks a mutex by calling `kdThreadMutexLock`, which blocks until the mutex is available. The mutex is unlocked by a call to `kdThreadMutexUnlock`; if any other threads are blocked waiting for the mutex in `kdThreadMutexLock`, exactly one of them acquires the lock and is unblocked.

Mutexes work on an OpenKODE Core implementation that does not support threading.

7.2.4. Condition variables

A *condition variable* is a synchronization primitive used to wait for some application-defined condition, typically that a shared resource is available, to become true. The basic operations on a condition variable are to wait for the condition to become true, and to signal that the condition has just become true.

A condition variable has a mutex associated with it, to avoid the race condition where a thread is about to wait on a condition variable just as another thread signals it. Thus a thread locks the mutex before waiting on the condition variable, and returns from a successful wait with the mutex locked, but the wait function atomically unlocks the mutex during the wait.

A condition variable is created by a call to `kdThreadCondCreate`, which returns a handle to the new condition variable, and freed by a call to `kdThreadCondFree`.

A thread waits on a condition variable by calling `kdThreadCondWait`. Another thread signals the condition by calling `kdThreadCondSignal` to release at least one of the waiting threads, or `kdThreadCondBroadcast` to release all waiting threads. The released threads return from their respective `kdThreadCondWait` calls one at a time, since a returning thread has the associated mutex locked.

7.2.5. Semaphores

A semaphore is a synchronization primitive acting as an integer whose value can never fall below zero. A post operation increments the value; a wait operation decrements the value, but, if the value is already zero, it waits until another thread posts.

A semaphore is created by a call to `kdThreadSemCreate`, which takes the initial count of the semaphore and returns a handle to the new semaphore, and freed by a call to `kdThreadSemFree`.

A thread posts a semaphore by calling `kdThreadSemPost`. A thread waits on a semaphore by calling `kdThreadSemWait`.

Semaphores work on an OpenKODE Core implementation that does not support threading.

7.3. Functions

7.3.1. kdThreadAttrCreate

Create a thread attribute object.

Synopsis

```
typedef struct KDThreadAttr KDThreadAttr;
```

```
KDThreadAttr *kdThreadAttrCreate(void);
```

Description

This function creates a thread attributes object, and returns a valid handle to it.

The new thread attributes object contains default attributes. Attributes can then be modified using `kdThreadAttrSetDetachState` or `kdThreadAttrSetStackSize` as required, and then passed to `kdThreadCreate` such that the thread creation uses the supplied attributes. A single thread attributes object can be passed to multiple `kdThreadCreate` calls, even multiple simultaneous calls.

Note that it is possible to create and manipulate a thread attributes object, even when the implementation does not support threading.

Return value

On success, the function returns the new thread attributes object. 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_ENOMEM`	Out of memory.

Rationale

The notion of a thread attributes object is based on [POSIX]. There, a thread attributes object is represented by a struct in user data which is initialized with `pthread_attr_init`.

It is mandated that a thread attributes object can be created and manipulated even when threading is not supported in order to allow for a possible future extension (from a vendor or from Khronos) that allows co-operative, non-timeslicing threads. For backwards compatibility with applications written to assume timeslicing threads, the extension would only allow creation of a thread on an implementation with co-operative threads if the thread attribute object had had a newly defined attribute set which states that the application knows about co-operative threads and has been written to deal with them.

7.3.2. kdThreadAttrFree

Free a thread attribute object.

Synopsis

```
KDint kdThreadAttrFree(KDThreadAttr *attr);
```

Description

This function frees the thread attributes object *`attr`*. Once the function has been entered, *`attr`* is no longer a valid thread attributes object handle.

If *`attr`* is not a valid thread attributes object handle, then undefined behavior results.

Return value

On success, the function returns 0. The function has no defined failure condition.

Rationale

The notion of a thread attributes object is based on [POSIX]. There, a thread attributes object is represented by a struct in user data which is deinitialized with `pthread_attr_destroy`.

7.3.3. kdThreadAttrSetDetachState

Set detachstate attribute.

Synopsis

```
#define KD_THREAD_CREATE_JOINABLE 0
#define KD_THREAD_CREATE_DETACHED 1

KDint kdThreadAttrSetDetachState(KDThreadAttr *attr, KDint detachstate);
```

Description

This function sets the detachstate attribute in the thread attributes object *`attr`* to the value *`detachstate`*.

A value of `KD_THREAD_CREATE_JOINABLE` causes a thread created using this attributes object to be created in the joinable state, such that its ID can be specified to `kdThreadJoin` or `kdThreadDetach`. This is the default setting.

A value of `KD_THREAD_CREATE_DETACHED` causes a thread created using this attributes object to be created in the detached state, such that its ID cannot be specified to the above functions, but resources associated with the thread are freed as soon as the thread ends.

If *`detachstate`* is not one of the above values, an error is returned as described below.

If *`attr`* is not a valid thread attributes object handle, then undefined behavior results.

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`	*`detachstate`* is not one of the values defined above.

Rationale

`kdThreadAttrSetDetachState` is based on [POSIX] `pthread_attr_setdetachstate`. The POSIX function returns any error code, rather than returning -1 and setting the error indicator.

A thread created joinable (by using this attribute’s default value `KD_THREAD_CREATE_JOINABLE`) can still be made detached using `kdThreadDetach`.

7.3.4. kdThreadAttrSetStackSize

Set stacksize attribute.

Synopsis

```
KDint kdThreadAttrSetStackSize(KDThreadAttr *attr, KDsize stacksize);
```

Description

This function sets the stacksize attribute in the thread attributes object `attr` to the value `stacksize`.

When a thread is created with `kdThreadCreate` using this thread attributes object, the size of the new thread’s stack in bytes is at least the value of the stacksize attribute.

The default value is implementation defined.

If the function is used to attempt to set a stack size larger than an implementation-defined maximum, an error is given.

If `attr` is not a valid thread attributes object handle, then undefined behavior results.

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`	Requested stack size is greater than the implementation-defined maximum.

Rationale

`kdThreadAttrSetStackSize` is based on [POSIX] `pthread_attr_setstacksize`. The POSIX function returns any error code, rather than returning -1 and setting the error indicator.

7.3.5. kdThreadCreate

Create a new thread.

Synopsis

```
typedef struct KDThread KDThread;

KDThread *kdThreadCreate(const KDThreadAttr *attr, void *(*start_routine)(void
*), void *arg);
```

Description

This function creates a new thread. The new thread runs the function `start_routine`, passing `arg` as its only parameter. The thread finishes either by calling `kdThreadExit` passing the thread’s return value, or equivalently by returning from the thread’s `start_routine` with the return value.

The `attr` is either a thread attributes object, or is `KD_NULL`. In the former case, attributes set in the thread attributes object are applied to the thread creation. The `KD_NULL` case is equivalent to supplying a thread attributes object which has been created with `kdThreadAttrCreate` and then not modified.

Threads run pre-emptively, that is, execution could switch from one thread to another at any time, or multiple threads can actually run concurrently on multiple CPU cores.

The scheduling algorithm is undefined. No thread priority mechanism exists, which means there is no way of making a thread "high priority" to ensure that it runs in preference to other lower priority threads.

If *`start_routine`* is not a pointer to a function taking a single void* parameter and returning void*, or *`attr`* is not `KD_NULL` and is not a valid thread attributes object handle, then undefined behavior results.

Return value

On success, the function returns the new thread's ID. 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_EAGAIN`	Not enough system resources, or maximum number of threads already active.
`KD_ENOSYS`	Threading not supported.

Rationale

`kdThreadCreate` is based on [POSIX] `pthread_create`. The POSIX function stores the thread ID into a location pointed to by an extra parameter and returns any error code, rather than returning the thread ID or returning `KD_NULL` and setting the error indicator.

7.3.6. kdThreadExit

Terminate this thread.

Synopsis

```
KD_NORETURN void kdThreadExit(void *retval);
```

Description

This function causes the calling thread to exit, with *`retval`* as the return value.

When called from the main thread (the thread in which the application started in `kdMain`), `kdThreadExit` acts as `kdExit` with an exit code of 0.

This function works as specified even when the implementation does not support threads.

If a thread calls `kdThreadExit` while it has a mutex locked, it is undefined whether the mutex remains locked.

Rationale

`kdThreadExit` is based on [POSIX] `pthread_exit`. However `pthread_exit` does not treat the main thread specially.

7.3.7. kdThreadJoin

Wait for termination of another thread.

Synopsis

```
KDint kdThreadJoin(KDThread *thread, void **retval);
```

Description

This function blocks the calling thread until the specified *thread* terminates. If the specified thread has already terminated, blocking does not occur.

If *retval* is not `KD_NULL`, the specified thread's return value is stored in the void* location it points at.

The specified thread must be in the joinable state. On successful return, the thread and any associated resources are freed, and *thread* is no longer a valid thread ID.

The results of multiple simultaneous calls to `kdThreadJoin` specifying the same target thread are undefined.

If *thread* is not a valid thread ID, or is the main thread, or *retval* is not `KD_NULL` and is not a pointer to a writable void* location, then undefined behavior results.

Return value

On success, the function returns 0, stores the specified thread's return value into the location pointed to by *retval* (if *retval* is not `KD_NULL`), and frees all resources associated with the specified thread. 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_EDEADLK`	*thread* is the current thread.
`KD_EINVAL`	*thread* has been detached.

Rationale

`kdThreadJoin` is based on [POSIX] `pthread_join`. The POSIX function returns any error code, rather than returning -1 and setting the error indicator.

However, `kdThreadJoin` has undefined behavior rather than an error when given an invalid thread ID.

7.3.8. kdThreadDetach

Allow resources to be freed as soon as a thread terminates.

Synopsis

```
KDint kdThreadDetach(KDThread *thread);
```

Description

This function puts the specified *thread* into *detached* state, and thus no longer in joinable state. This means that resources associated with the specified thread will be freed as soon as the thread terminates (or immediately if the thread has already terminated), and at that point *thread* will no longer be a valid thread ID.

When detached, the thread cannot be the subject of a call to `kdThreadJoin` to wait for it to terminate and retrieve its return value.

If another thread is already in a `kdThreadJoin` waiting for the specified *`thread`* to terminate, then the call to `kdThreadDetach` causes undefined behavior.

If *`thread`* is not a valid thread ID, or is the main thread, then undefined behavior results.

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`	*`thread`* is already detached.

Rationale

`kdThreadDetach` is based on [POSIX] `pthread_detach`. The POSIX function returns any error code, rather than returning -1 and setting the error indicator.

However, `kdThreadDetach` has undefined behavior rather than an error when given an invalid thread ID.

7.3.9. kdThreadSelf

Return calling thread's ID.

Synopsis

```
KDThread *kdThreadSelf(void);
```

Description

This function simply returns the thread ID of the calling thread.

The application's main thread has a thread ID, even though it was not created by `kdThreadCreate`. Using `kdThreadSelf` from the main thread returns this thread ID.

Return value

On success, the function returns the thread ID of the calling thread. The function never fails. Even on an OpenKODE implementation that does not support threads, this function returns a thread ID that can be used in `kdPostThreadEvent`.

Rationale

`kdThreadSelf` is based on [POSIX] `pthread_self`.

7.3.10. kdThreadOnce

Wrap initialization code so it is executed only once.

Synopsis

```
#ifndef KD_NO_STATIC_DATA
typedef struct KDThreadOnce {
    void *impl;
} KDThreadOnce;
#define KD_THREAD_ONCE_INIT { 0 }

KDint kdThreadOnce(KDThreadOnce *once_control, void (*init_routine)(void));

#endif /* ndef KD_NO_STATIC_DATA */
```

Description

This function ensures that an application-supplied function is run only once in the process. The first time `kdThreadOnce` is called with a given *`once_control`*, the function pointed to by *`init_routine`* is called. On subsequent calls, the function is not called. For a particular *`once_control`* pointer, no thread returns from `kdThreadOnce` until the first thread to reach it has finished running the *`init_routine`*.

One thread being in an *`init_routine`* via a call to `kdThreadOnce` does not block another thread calling `kdThreadOnce` with a different *`once_control`*. This even applies to the same thread; one *`init_routine`* can call `kdThreadOnce` with a different *`once_control`*.

In an OpenKODE Core implementation that does not support writable static data (`KD_NO_STATIC_DATA` is defined), this function is not present at all.

In an OpenKODE Core implementation that does support writable static data but does not support threading, this function works as normal.

If *`once_control`* is not a pointer to a global/static readable and writable KDThreadOnce location which was initialized to `KD_THREAD_ONCE_INIT` (all zero words) before the first call to `kdThreadOnce` using it, or if *`init_routine`* is not a pointer to a function taking void and returning void, then undefined behavior results.

Return value

On success, the function returns 0. There is no defined error case.

Rationale

`kdThreadOnce` is based on [POSIX] `pthread_once`.

Note that `kdThreadOnce` cannot fail through lack of memory or other resources. This implies that it is implemented using underlying synchronization primitives (such as a mutex and a condition variable) which are created at application startup time, and it does not need any memory- or resource-using creation action when `kdThreadOnce` is called.

The requirement that one thread being in an *`init_routine`* via a call to `kdThreadOnce` does not block another thread calling `kdThreadOnce` with a different *`once_control`* means that `kdThreadOnce` cannot be implemented with a single mutex covering all *`once_control`*s.

7.3.11. kdThreadMutexCreate

Create a mutex.

Synopsis

```
typedef struct KDThreadMutex KDThreadMutex;

KDThreadMutex *kdThreadMutexCreate(const void *mutexattr);
```

Description

This function creates a mutex, returning a valid handle to it. The new mutex is initially unlocked.

`mutexattr` must be `KD_NULL`.

Mutexes work on an OpenKODE Core implementation that does not support threading.

If `mutexattr` is not `KD_NULL`, then undefined behavior results.

Return value

On success, the function returns the mutex handle. 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_EAGAIN`	Not enough resources (other than memory).
`KD_ENOMEM`	Out of memory.

Rationale

The mutex created by `kdThreadMutexCreate` is based on [POSIX] mutexes. POSIX has the mutex represented by a structure in user data which is initialized with `pthread_mutex_init`, or with a static initializer. OpenKODE Core has no support for mutex attributes, although `mutexattr` has been included to allow a future extension to add mutex attributes compatibly.

7.3.12. kdThreadMutexFree

Free a mutex.

Synopsis

```
KDint kdThreadMutexFree(KDThreadMutex *mutex);
```

Description

This function frees the `mutex`. The mutex must be unlocked when destroyed. Once the function has been entered, `mutex` is no longer a valid mutex handle.

If `mutex` is not a valid mutex handle, or the mutex is currently locked by some thread, then undefined behavior results.

Return value

On success, the function returns 0. There is no defined failure case.

Rationale

[POSIX] has a mutex represented by a structure in user data which is deinitialized with `pthread_mutex_destroy`.

7.3.13. kdThreadMutexLock

Lock a mutex.

Synopsis

```
KDint kdThreadMutexLock(KDThreadMutex *mutex);
```

Description

`kdThreadMutexLock` locks the specified *`mutex`* for the calling thread. Only one thread can lock the mutex at a time; if it is currently locked by another thread, the function blocks until the mutex is unlocked and it can acquire the lock for the calling thread.

If any other thread is blocked in `kdThreadMutexLock` waiting for the same mutex, it is undefined which thread acquires the mutex first.

Mutexes work on an OpenKODE Core implementation that does not support threading.

If the mutex is already locked by the calling thread, then undefined behavior results.

If *`mutex`* is not a valid mutex handle, then undefined behavior results.

Return value

On success, the function returns 0. The function has no defined failure case.

Rationale

`kdThreadMutexLock` is based on [POSIX] `pthread_mutex_lock`. The POSIX function returns any error code, rather than returning -1 and setting the error indicator. OpenKODE Core has no equivalent of POSIX's `PTHREAD_MUTEX_INITIALIZER`.

The undefined behavior on an attempt to lock a mutex that the thread already has locked is intended to allow easy and efficient implementation on OSes with differing mutex behaviors, such as recursive, non-recursive but allowed to re-lock, non-recursive and gives an error, and non-recursive and blocks. This presents a portability problem where an application could accidentally be written to assume a particular platform's behavior. Therefore it is recommended that any "debug" OpenKODE Core implementation should diagnose an attempt to re-lock an already locked mutex in a way that stops the application working, such as terminating it.

A draft of this specification contained `kdThreadMutexTryLock`. It was removed with the justification that some popular platforms do not support it directly, and it is more difficult for the OpenKODE implementation to implement it generally, with the interaction between mutexes and condition variables, than it is for an application that requires them but not in conjunction with condition variables to implement the functionality itself.

7.3.14. kdThreadMutexUnlock

Unlock a mutex.

Synopsis

```
KDint kdThreadMutexUnlock(KDThreadMutex *mutex);
```

Description

This function unlocks the specified *`mutex`* for the calling thread. If any other thread is blocked in `kdThreadMutexLock` on this mutex, then exactly one of those threads acquires the mutex and successfully returns from `kdThreadMutexLock`. If multiple threads were waiting, it is undefined which thread acquires the mutex lock.

Mutexes work on an OpenKODE Core implementation that does not support threading.

If the mutex was locked but not by the calling thread, or was already unlocked, then undefined behavior results.

If `mutex` is not a valid mutex handle, then undefined behavior results.

Return value

On success, the function returns 0. There is no defined failure case.

Rationale

`kdThreadMutexUnlock` is based on [POSIX] `pthread_mutex_unlock`.

A debugging OpenKODE implementation may want to diagnose the cases of unlocking an already-unlocked mutex and unlocking a mutex locked by another thread, perhaps causing the application to terminate with an error message.

7.3.15. kdThreadCondCreate

Create a condition variable.

Synopsis

```
typedef struct KDThreadCond KDThreadCond;
```

```
KDThreadCond *kdThreadCondCreate(const void *attr);
```

Description

This function creates a condition variable, returning a valid handle to it.

If *`attr`* is not `KD_NULL`, then undefined behavior results.

Return value

On success, the function returns the handle to the new condition variable. 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_EAGAIN`	Not enough resources (other than memory).
`KD_ENOMEM`	Out of memory.
`KD_ENOSYS`	Threading not supported.

Rationale

The condition variable created by `kdThreadCondCreate` is based on [POSIX] condition variables. POSIX has the condition variable represented by a structure in user data which is initialized with `pthread_cond_init`, or with a static initializer. OpenKODE Core has no support for condition variable attributes, although *`attr`* has been included to allow a future extension to add such attributes.

7.3.16. kdThreadCondFree

Free a condition variable.

Synopsis

```
KDint kdThreadCondFree(KDThreadCond *cond);
```

Description

This function frees the condition variable `cond`. Once the function has been entered, `cond` is no longer a valid condition variable handle.

If `cond` is not a valid condition variable handle, or the condition variable has one or more threads waiting on it, then undefined behavior results.

Return value

On success, the function returns 0. There is no defined failure case.

Rationale

[POSIX] has a condition variable represented by a structure in user data which is deinitialized with `pthread_cond_destroy`.

7.3.17. kdThreadCondSignal, kdThreadCondBroadcast

Signal a condition variable.

Synopsis

```
KDint kdThreadCondSignal(KDThreadCond *cond);

KDint kdThreadCondBroadcast(KDThreadCond *cond);
```

Description

These functions are used to unblock threads blocked on the condition variable `cond`. For `kdThreadCondSignal`, if threads exist which are being blocked by the condition variable, at least one of them is unblocked. If more than one thread is blocked on a condition variable, it is undefined which one(s) is/are unblocked. For `kdThreadCondBroadcast`, all threads waiting on the condition variable are unblocked.

Once a thread is unblocked as a consequence of some thread calling `kdThreadCondSignal`, it returns from its call to `kdThreadCondWait`, with the associated mutex (as specified to `kdThreadCondWait`) locked. If multiple threads are unblocked, they each try and lock the associated mutex before returning from `kdThreadCondWait`, and it is undefined which thread acquires the mutex first.

The functions have no effect and succeed if there is no thread waiting on the condition variable.

If `cond` is not a valid condition variable handle, then undefined behavior results.

Return value

On success, the function returns 0. There is no defined failure case.

Rationale

`kdThreadCondSignal` is based on [POSIX] `pthread_cond_signal`. `kdThreadCondBroadcast` is based on [POSIX] `pthread_cond_broadcast`.

Allowing more than one thread to be unblocked by `kdThreadCondSignal` is necessary to facilitate implementation on a multi-processor system. This means that, where a condition variable is used to signal to one of a set of waiting threads that some resource is now available for use by one of them, a waiting thread may experience a spurious wake-up. Thus, the code calling `kdThreadCondWait` must check that the resource really is available to it, and if not loop back and wait again.

OpenKODE Core has no equivalent of POSIX's `PTHREAD_COND_INITIALIZER`.

7.3.18. kdThreadCondWait

Wait for a condition variable to be signalled.

Synopsis

```
KDint kdThreadCondWait(KDThreadCond *cond, KDThreadMutex *mutex);
```

Description

This function blocks on the condition variable *`cond`*. A *`mutex`* must be associated to the condition variable, and must be locked when passed to `kdThreadCondWait`. This mutex becomes bound to the condition variable until successful return.

The function releases the mutex and causes the calling thread to block on the condition variable as a single atomic operation with respect to access by another thread to the mutex and then the condition variable. That is, if another thread is able to lock the mutex after the about-to-block thread has released it, then a subsequent call to `kdThreadCondSignal` or `kdThreadCondBroadcast` in that other thread behaves as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex has been locked and is owned by the calling thread.

If different mutexes are used for concurrent `kdThreadCondWait` operations on the same condition variable, then undefined behavior results.

If *`cond`* is not a valid condition variable handle, or *`mutex`* is not a valid mutex handle, or the mutex is not locked by the calling thread on entry, then undefined behavior results.

Return value

On success, the function returns 0. There is no defined error case.

Rationale

`kdThreadCondWait` is based on [POSIX] `pthread_cond_wait`.

OpenKODE Core has no equivalent of POSIX's `PTHREAD_COND_INITIALIZER`.

7.3.19. kdThreadSemCreate

Create a semaphore.

Synopsis

```
typedef struct KDThreadSem KDThreadSem;
```

```
KDThreadSem *kdThreadSemCreate(KDuint value);
```

Description

This function creates a semaphore, and returns a valid handle to it.

A semaphore has a non-negative integer *value*. This function uses the `value` parameter as the initial value of the semaphore.

Semaphores work on an OpenKODE Core implementation that does not support threading.

Return value

On success, the function returns the handle to the new semaphore. 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_EINVAL`	`value` is larger than the implementation-defined semaphore value limit.
`KD_ENOSPC`	Not enough resources to initialize the semaphore.

Rationale

The semaphore created by `kdThreadSemCreate` is based on [POSIX] semaphores. POSIX has the semaphore represented by a structure in user data which is initialized with `sem_init`, or with a static initializer. OpenKODE Core has no support for inter-process semaphores.

7.3.20. kdThreadSemFree

Free a semaphore.

Synopsis

```
KDint kdThreadSemFree(KDThreadSem *sem);
```

Description

This function frees the semaphore `sem`. Once the function has been entered, `sem` is no longer a valid semaphore handle.

If `sem` is not a valid semaphore handle, or the semaphore has one or more threads blocked on it, then undefined behavior results.

Return value

On success, the function returns 0. There is no defined failure case.

Rationale

[POSIX] has a semaphore represented by a structure in user data which is deinitialized with `sem_destroy`.

7.3.21. kdThreadSemWait

Lock a semaphore.

Synopsis

```
KDint kdThreadSemWait(KDThreadSem *sem);
```

Description

This function performs a lock operation on the semaphore specified by *`sem`*: If the semaphore's value is 0, then the function causes the calling thread to be blocked and added to the set of threads waiting on the semaphore, otherwise, the value is decremented and the function returns immediately.

Semaphores work on an OpenKODE Core implementation that does not support threading.

If *`sem`* is not a valid semaphore handle, then undefined behavior results.

Return value

On success, the function returns 0. There is no defined failure case.

Rationale

`kdThreadSemWait` is based on [POSIX] `sem_wait`.

7.3.22. kdThreadSemPost

Unlock a semaphore.

Synopsis

```
KDint kdThreadSemPost(KDThreadSem *sem);
```

Description

This function performs an unlock operation on the semaphore specified by *`sem`*: If the set of threads waiting on the semaphore is empty, then the semaphore's value is incremented, otherwise, one thread is removed from the set and unblocked, allowing its `kdThreadSemWait` call to return successfully. If there is more than one thread in the waiting set, then it is not specified which one is removed and unblocked.

Semaphores work on an OpenKODE Core implementation that does not support threading.

If *`sem`* is not a valid semaphore handle, then undefined behavior results.

Return value

On success, the function returns 0. There is no defined failure case.

Rationale

`kdThreadSemPost` is based on [POSIX] `sem_post`.

8. Events

8.1. Introduction

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

8.1.1. 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 (or any thread receiving events) can have the following code:

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

Here, the thread loops processing events, using callbacks that have been registered by that thread. 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. Each thread has its own event queue. 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 thread calling one of several functions that do this. Thus the callback executes in that thread's context, as a callback from the function.
- The event can be returned by the application calling `kdWaitEvent`.

An event enabled for delivery by callback (in that thread) 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 same thread's 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` or `kdPostThreadEvent`.

8.2. Yielding

A thread that is not the main thread and has destroyed any windows, sockets and timers that it has ever created (or has never created any) does not receive events.

Any other thread (including the main thread) does receive events, and it must ensure it yields frequently enough. It must yield by calling one of `kdWaitEvent`, `kdPumpEvents`, or a thread synchronization function where the call actually blocks, no more than one second since the last yield (or the time the thread started if no yield has yet been performed).

8.2.1. Rationale

Some platforms have a watchdog timer, often of a few seconds, that kills an application that does not yield frequently enough.

A 'testing' OpenKODE implementation might like to impose its own one second deadline in order to ensure that applications do not exceed the limit.

8.3. Types

8.3.1. KDEvent

Struct type containing an event.

Synopsis

```
typedef struct KDEvent KDEvent;
#define KD_EVENT_USER 0x40000000

struct KDEvent {
    KDust timestamp;
    KDint32 type;
    void *userptr;
    union KDEventData {
        KDEventState state;
        KDEventInput input;
        KDEventInputJog inputjog;
        KDEventInputPointer inputpointer;
        KDEventInputStick inputstick;
        KDEventSocketReadable socketreadable;
        KDEventSocketWritable socketwritable;
        KDEventSocketConnect socketconnect;
        KDEventSocketIncoming socketincoming;
        KDEventNameLookup namelookup;
#ifdef KD_WINDOW_SUPPORTED
        KDEventWindowProperty windowproperty;
        KDEventWindowFocus windowfocus;
#endif /* KD_WINDOW_SUPPORTED */
        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.

8.4. Functions

8.4.1. kdWaitEvent

Get next event from thread’s event queue.

Synopsis

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

Description

This function is used in the thread’s event loop to get the next event in the thread’s event queue whose event type and user pointer combination is not covered by an installed callback for this thread (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 in this thread, 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 in this thread. 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.

It is allowed to call `kdWaitEvent` when in an event callback called back from `kdWaitEvent` or `kdPumpEvents`. Such a recursive call functions as normal in its processing of events in the queue.

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` (in the same thread) with it before calling `kdWaitEvent` (in the same thread) 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.

Order of event delivery

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`.

Callback recursion

See the notes on callback recursion in `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.

8.4.2. 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.

A global event is always sent to the main thread (the one in which `kdMain` was called).

8.4.3. 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`.

If `kdDefaultEvent` is called in a different thread from that in which the event was delivered (by being returned by `kdWaitEvent` or passed into a callback), then undefined behavior results.

8.4.4. kdPumpEvents

Pump the thread's event queue, performing callbacks.

Synopsis

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

Description

This function performs an *event pump*. Each event in order in the thread's event queue whose type and userptr combination is callback enabled in this thread is delivered by the applicable installed callback. 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.

It is allowed to call `kdPumpEvents` when in an event callback called back from `kdWaitEvent` or `kdPumpEvents`. Such a recursive call functions as normal in its processing of events in the queue.

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.

Order of event delivery

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.

Callback recursion

Calling `kdWaitEvent` or `kdPumpEvents` from within a callback can be useful to allow two (or more) tiers of event responsiveness in a single-threaded application. For example, an application may take tens or hundreds of milliseconds to process a timer event (its render loop), but need a much quicker response to an event signalling that an audio buffer needs refilling. (Note however that OpenKODE 1.0 does not include any audio API.)

It is up to the application to deal with the effects of this recursion, if it chooses to use it:

- A callback may be re-entered when already in use (if another event of the type it handles has been queued).
- Recursion into `kdWaitEvent` or `kdPumpEvents` is related to the number of events arriving, so the application needs to control it to avoid running out of stack. (A sensible approach might be for the application to allow only two levels.)
- If the application receives an event from `kdWaitEvent`, and, while processing that event, it calls `kdPumpEvents`, and a callback called by it uses `kdWaitEvent`, then the second use of `kdWaitEvent` invalidates the event returned by the first use earlier than the programmer might expect.

8.4.5. 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, including event type 0. Setting *`eventuserptr`* to `KD_NULL` matches any user pointer, including `KD_NULL`.

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 installed callback state is thread-local; the callback installed by this function is only ever used for events delivered in the calling thread.

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 typedef 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.

Implementation notes

The callback state of all event types and user pointers (for a particular thread) can be imagined as a grid of points, where the horizontal position gives the event type and the vertical position gives the user pointer, and each point in the grid has a callback function (`KD_NULL` if none).

Then, a call to `kdInstallCallback` changes the state of a single point if the function fully specifies both event type and user pointer, or a vertical line of points if it uses `KD_NULL` for the user pointer, or a horizontal line of points if it uses `0` for the event type, or all points in the grid if it does both.

The most obvious implementation of this specification of how OpenKODE Core chooses which callback to call if any (other than storing each grid point separately, which is impractical) is that the implementation keeps a record of all `kdInstallCallback` calls and, to determine how to dispatch an event, finds the most recent record with matching event type and user pointer.

However, this is not efficient in the common use case that the application installs a callback for a particular event type and user pointer combination, then later removes the callback for the same event type and user pointer combination by calling `kdInstallCallback` with the same parameters but with a `KD_NULL` function pointer.

To avoid old callback installation records being retained unnecessarily, `kdInstallCallback` could check the records starting with the most recent, and if a record at least has an overlap with the present call's event type and user pointer, then:

- if the event type and user pointer specification of the found record is a subset of (or exactly the same as) that of the present `kdInstallCallback`, then the found record can be removed and searching continues;
- otherwise, the found record is not removed and searching stops;
- if the search reaches the beginning of the records of earlier `kdInstallCallback` calls, and the present call has a `KD_NULL` function pointer, then no record needs to be added for the present call.

8.4.6. 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 `kdPostThreadEvent` to post the event, or calls `kdFreeEvent` to abandon the newly constructed event. The call to post or free the event must be in the same thread as the call to create the event.

Return value

On success, the function returns the pointer to a new `KDEvent`, which remains valid until it is used in a `kdPostEvent` or `kdPostThreadEvent` 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`, `kdPostThreadEvent` 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 outside any OpenKODE thread.

8.4.7. kdPostEvent, kdPostThreadEvent

Post an event into a queue.

Synopsis

```
KDint kdPostEvent(KDEvent *event);

KDint kdPostThreadEvent(KDEvent *event, KDThread *thread);
```

Description

These functions post the event pointed to by the *`event`* parameter, which is one returned by `kdCreateEvent` (in the same thread), although the fields in the KDEvent structure can have been altered to any values. If the *`timestamp`* field is 0, `kdPostEvent` and `kdPostThreadEvent` store the current time at some point during this `kdPostEvent` or `kdPostThreadEvent` call (as returned by `kdGetTimeUST`) into that field. The event is otherwise unaltered by `kdPostEvent` or `kdPostThreadEvent`.

`kdPostEvent` posts the event to the queue associated with the calling thread. `kdPostThreadEvent` posts the event to the thread whose thread ID is *`thread`* (as returned by `kdThreadCreate` or `kdThreadSelf`).

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

If the event is being posted to the calling thread, then it is available immediately.

If *`event`* is not an event struct pointer returned by an earlier `kdCreateEvent` in the same thread, or it has already been passed to `kdPostEvent`, `kdPostThreadEvent` or `kdFreeEvent`, or *`thread`* is not a valid thread ID, 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`, `kdPostThreadEvent` 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 outside any OpenKODE thread.

8.4.8. 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` in the same thread.

If *`event`* is not an event struct pointer returned by an earlier `kdCreateEvent` in the same thread, or it has already been passed to `kdPostEvent`, `kdPostThreadEvent` or 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`, `kdPostThreadEvent` 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 outside any OpenKODE thread.

9. System events

9.1. 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.

9.2. Events

9.2.1. KD_EVENT_QUIT

Event to request to quit application.

Synopsis

```
#define KD_EVENT_QUIT 43
```

Description

This event type is generated by OpenKODE Core (typically as the result of a request from the underlying OS) and sent to the main thread's event queue 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).

9.2.2. KD_EVENT_PAUSE

Application pause event.

Synopsis

```
#define KD_EVENT_PAUSE 45
```

Description

This event type, which has no associated data, is generated by OpenKODE (typically as the result of a request from the underlying OS) and sent to the main thread's event queue to signal that the application should pause.

On some implementations, a pause event may also count as a power management event for EGL purposes. In this case, EGL context(s) cannot be recreated until a `KD_EVENT_RESUME` event has been received.

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, and it is delivered to the main thread (the one in which `kdMain` was called).

Rationale

The expectation is that the application will make reasonable effort to reduce resource usage, especially CPU usage, within a reasonable time. If the application does not do this, the underlying OS may decide to kill the application. The defintion here of “reasonable” and whether the OS does enforce this policy are platform dependent. The application is not required to reduce its memory usage. The application is still expected to respond to `KD_EVENT_WINDOW_REDRAW` events, possibly by redrawing the same bitmap as last time.

The distinction between this pause event and an event signalling loss of focus is that, for an application which continues processing even when it does not have focus (for example a music player), the pause event instructs it to pause.

9.2.3. KD_EVENT_RESUME

Application resume event.

Synopsis

```
#define KD_EVENT_RESUME 46
```

Description

This event type, which has no associated data, is generated by OpenKODE (typically as the result of a request from the underlying OS) and sent to the main thread’s event queue 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, and it is delivered to the main thread (the one in which `kdMain` was called).

9.2.4. KD_EVENT_ORIENTATION

Orientation change event.

Synopsis

```
#define KD_EVENT_ORIENTATION 48
```

Description

This event type, which has no associated data, is generated by OpenKODE (typically as the result of a request from the underlying OS) and sent to the main thread’s event queue to signal that some aspect of orientation has changed, that is one or more of the state values in the `KD_IOGROUP_ORIENTATION` I/O group, or the `KD_STATE_PHONEKEYPAD_ORIENTATION` state value, has changed. An orientation change may cause more than one of those state values to change; this `KD_EVENT_ORIENTATION` event indicates that all of those state values have now changed to their new state.

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, and it is delivered to the main thread (the one in which `kdMain` was called).

Rationale

Because an orientation event (typically the user rotating the handset or rotating some part of the handset relative to some other part) can generate a change to more than one of the orientation-related state values, the recommended strategy for an application to deal with an orientation change is to ignore the events from the orientation-related state value changes, and then, when it receives this `KD_EVENT_ORIENTATION` event, check the state values then.

9.3. I/O groups and items

9.3.1. KD_IOGROUP_EVENT

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

Synopsis

```
#define KD_IOGROUP_EVENT 0x100
#define KD_STATE_EVENT_USING_BATTERY       (KD_IOGROUP_EVENT + 0)
#define KD_STATE_EVENT_LOW_BATTERY         (KD_IOGROUP_EVENT + 1)
```

Description

This I/O group defines state values which implement several OpenKODE Core system events, allowing the application to get the state using `kdStateGet*` functions, as well as receiving events.

All state values 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_STATE_EVENT_USING_BATTERY`	mandatory KDint32 state	0..1	1 if using battery, 0 if using mains power. Where an implementation is not able to give this information, the value is always 0.
`KD_STATE_EVENT_LOW_BATTERY`	mandatory KDint32 state	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, the value is always 0.

9.3.2. KD_IOGROUP_ORIENTATION

I/O group for OpenKODE Core orientation state.

Synopsis

```
#define KD_IOGROUP_ORIENTATION 0x200
#define KD_STATE_ORIENTATION_OVERALL       (KD_IOGROUP_ORIENTATION + 0)
#define KD_STATE_ORIENTATION_SCREEN        (KD_IOGROUP_ORIENTATION + 1)
#define KD_STATE_ORIENTATION_RENDERING     (KD_IOGROUP_ORIENTATION + 2)
#define KD_STATE_ORIENTATION_LOCKSURFACE   (KD_IOGROUP_ORIENTATION + 3)
```

Description

This I/O group defines state values which indicate several aspects of the orientation of the platform and parts of it.

All state values are mandatory.

I/O items

index	type	range	usage
`KD_STATE_ORIENTATION_OVERALL`	mandatory KDint32 state	0..3	The number of right angles the platform is rotated counterclockwise from its “normal” orientation. Where an implementation is not able to give this information, the state is -1. If the state is -1, it never changes.
`KD_STATE_ORIENTATION_SCREEN`	mandatory KDint32 state	0..3	The number of right angles the screen is rotated counterclockwise from its “normal” orientation. Where an implementation is not able to give this information, the state is always -1. If the state is -1, it never changes. An implementation does change the state value when it changes `KD_STATE_ORIENTATION_RENDERING` or `KD_STATE_ORIENTATION_LOCKSURFACE`.
`KD_STATE_ORIENTATION_RENDERING`	mandatory KDint32 state	0..3	The number of right angles anything rendered via OpenGL ES or OpenVG is rotated from the user’s point of view. For example, with a value of 1, an arrow rendered to point to the right is seen by the user pointing up.
`KD_STATE_ORIENTATION_LOCKSURFACE`	mandatory KDint32 state	0..3	The number of right angles anything rendered via EGL’s lock surface extension is rotated from the user’s point of view. For example, with a value of 1, an arrow rendered to point to the right is seen by the user pointing up.

Orientation rationale

The OpenKODE Core state values and event for orientation allow for many different platform configurations, where either or both of the screen and keypad may be rotated separately from the platform, or where the platform may detect the rotation of the platform as a whole.

Where the OpenGL ES, OpenVG and EGL drivers or hardware automatically compensate for rotation of the screen, the `KD_STATE_ORIENTATION_RENDERING` and `KD_STATE_ORIENTATION_LOCKSURFACE` state values are always 0, but, if a rotation of the screen causes a resize of any window belonging to the application (for example when it is a full-screen window), then a `KD_EVENT_WINDOWPROPERTY_CHANGE` event for the `KD_WINDOWPROPERTY_SIZE` property is generated.

Where the OpenGL ES, OpenVG and EGL drivers or hardware do not automatically compensate for rotation of the screen, the `KD_STATE_ORIENTATION_RENDERING` and `KD_STATE_ORIENTATION_LOCKSURFACE` change as appropriate. This is in addition to any `KD_EVENT_WINDOWPROPERTY_CHANGE` event(s) for the `KD_WINDOWPROPERTY_SIZE` property.

In either case, the platform may be capable of changing `KD_STATE_ORIENTATION_SCREEN` and/or `KD_STATE_ORIENTATION_OVERALL` for the benefit of an application that wants to change its behavior according to screen or overall orientation. This is likely to be fairly non-portable behavior, but the information is presented by OpenKODE Core in this portable way.

The `KD_IOGROUP_PHONEKEYPAD` I/O group has a state value which specifies whether the phone keypad has been rotated. The softkeys in that I/O group, and other I/O groups (such as the game keys) are defined to be remapped on a rotation (where it is detectable by the handset), so the direction keys work as expected in the new orientation.

10. Application startup and exit.

10.1. 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.

10.2. Functions

10.2.1. kdMain

The application-defined main function.

Synopsis

```
KDint kdMain(KDint argc, const KDchar *const *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.

The initial thread in which OpenKODE calls `kdMain` is the *main thread*.

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.

10.2.2. kdExit

Exit the application.

Synopsis

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

Description

This function causes the application to exit 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.

If `kdExit` is called when other threads in the application are still running, it kills those other threads before the application exits. Each of those other threads is killed at any time between immediately and the next time that the thread:

- enters, returns from a callback to or is blocked in `kdWaitEvent` or `kdPumpEvents`;
- enters or is blocked in `kdThreadJoin`, `kdThreadMutexLock`, `kdThreadCondWait` or `kdThreadSemWait`.

If multiple threads call `kdExit` simultaneously, it is undefined which one's exit status is the exit status of the process.

Rationale

`kdExit` is based on [C89] `exit`.

In [POSIX], `exit` causes an immediate exit, killing other threads. It is expected that this will be the case in OpenKODE when implemented on an OS that makes this possible, such as Symbian, Windows and Linux. On some other OSes, such semantics would have to be constructed by the OpenKODE implementation killing the other threads, which is often not possible in a clean and reliable way. This is why the specification allows for other threads not being killed until they reach defined cancel points.

Thus a portable application should not have even a non-event-receiving thread spending a long time in a loop with no calls to the functions in the above list, since it may find that the thread cannot be killed by a `kdExit` in a different thread. A non-event-receiving thread should periodically call `kdPumpEvents` to check whether the thread needs to be killed.

When using C++ with OpenKODE Core, it is likely that the implementation will not destroy automatic variables when exiting with `kdExit`, the same as `exit`.

11. Utility library functions

11.1. 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.

11.2. Functions

11.2.1. 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`.

11.2.2. 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
- 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.

An *infinity* is `"INF"`, case insensitive.

A *NaN* is `"NAN"`, case insensitive. It is not defined which exact NaN representation results.

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. It is undefined whether a number that could be represented as a denormal in the return type causes underflow or not. In the overflow and underflow cases, 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.

Error codes

`KD_ERANGE`	Result overflowed or underflowed.

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.

The specification for the input required for an infinite or NaN result is tighter than the C standards; they allow `infinity` instead of `inf`, and an arbitrary sequence of characters in parentheses after `nan`. The OpenKODE Core specification aims to allow for interchange of text files containing floating point values between different OpenKODE Core implementations.

OpenKODE Core 1.0 specified that `kdStrtof` accepts C99-style hexadecimal floating point numbers. This was an accidentally-included misfeature and has been removed from OpenKODE Core 1.0.1. In fact the specification here demands that an implementation does not accept a hexadecimal floating point number, for example treating `0x123p45` as the single-character number `0`.

11.2.3. 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 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 a minus sign was present in the position indicated above, then the result of the conversion is negated. This is the case even for `kdStrtoul`, in which case the negated value is cast to the unsigned return type.

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 (after negating if a minus sign was present) 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` (even if there was a minus sign) 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.

Error codes

`KD_ERANGE`	Result overflowed or underflowed.

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.

11.2.4. 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+2)/3+1)

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

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

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 base 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.

11.2.5. 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 is `nan`.
- If *`number`* is plus infinity, the string representation is `inf`. If *`number`* is minus infinity, the string representation is `-inf`.
- 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 [1e8,1e9), and then obtaining an integer by rounding to nearest, with tiebreak case rounded to even. 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.

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`.

The specification for the string result of an infinite or NaN input is tighter than the C standards; they allow `infinity` instead of `inf`, and an arbitrary sequence of characters in parentheses after `nan`. The OpenKODE Core specification aims to allow for interchange of text files containing floating point values between different OpenKODE Core implementations.

11.2.6. 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.

12. Locale specific functions

12.1. 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.

12.2. Functions

12.2.1. 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. The locale is intended to reflect the user's preference, and not necessarily the location in which the handset was purchased or where it is being used.

Return value

On success, the function returns a pointer to a static string with one of these forms:

- empty string if no information is available;
- the ISO 639-1 language code if no information is available for the location;
- the ISO 639-1 language code followed by an underscore if the location is explicitly "international";
- the ISO 639-1 language code, then an underscore, then an ISO 3166-1 alpha-2 location code.

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.

13. Memory allocation

13.1. Introduction

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

13.2. Functions

13.2.1. 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.

13.2.2. 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`.

13.2.3. 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.

14. Thread-local storage.

14.1. 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.

These OpenKODE Core functions allow only a single per-thread pointer. Khronos defines an extension KHR_thread_storage which provides a more flexible mechanism suitable for when independently authored libraries and application all want to use thread-local storage.

14.2. Functions

14.2.1. 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.

Return value

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

14.2.2. 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.

15. Mathematical functions

15.1. 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.

15.2. Constants and macros

`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`	Log base 2 of e.
`KD_LOG10E_F`	Log base 10 of e.
`KD_LN2_F`	Natural log of 2.
`KD_LN10_F`	Natural log of 10.
`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_FLT_EPSILON`	Difference between 1 and the smallest float greater than 1, that is, 2 to the power of -23.
`KD_FLT_MAX`	The largest possible finite float, that is, 2 to the power of 128 minus 2 to the power of 104.
`KD_FLT_MIN`	The smallest possible positive normalized float, that is, two to the power of -126.

`KD_INFINITY`	Positive infinity, with the bit pattern 0x7f800000.
`kdIsNan(x)`	1 if *x* is not-a-number, 0 otherwise Because `kdIsNan` is specified as a macro rather than a function, it is undefined how many times the argument is evaluated.
`KD_HUGE_VALF`	Equivalent to `KD_INFINITY`. Some OpenKODE functions are based on C standard functions that, in the C standard, return `HUGE_VALF` for an overflow. OpenKODE does not use `KD_HUGE_VALF` in its definition, but provides it anyway for familiarity as it is equivalent.
`KD_DEG_TO_RAD_F`	`#define KD_DEG_TO_RAD_F 0.0174532924F` Pi divided by 180, so multiply by this number to convert degrees to radians.
`KD_RAD_TO_DEG_F`	`#define KD_RAD_TO_DEG_F 57.2957802F` 180 divided by pi, so multiply by this number to convert radians to degrees.

15.2.1. 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`. The macro `kdIsNan` has a [C99] equivalent `isnan`.

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.)

15.3. Functions

15.3.1. 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,PI].

If *x* is +1, the function returns +0.

If *x* is a NAN, or a value outside the range [-1,1], the function returns a NAN value.

Rationale

`kdAcosf` is based on the [C99]/[POSIX] function `acosf`. [C89] has `acos`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.2. 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 principal value of the arc sine of *x*, in the range [-PI/2,PI/2].

If *x* is plus or minus 0, the function returns *x*.

If *x* is a NAN, or a value outside the range [-1,1], the function returns a NAN value.

Rationale

`kdAsinf` is based on the [C99]/[POSIX] function `asinf`. [C89] has `asin`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.3. 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,PI/2].

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

If *x* is plus or minus zero, the function returns *x*.

If *x* is plus or minus infinity, the function returns plus or minus PI/2 respectively.

Rationale

`kdAtanf` is based on the [C99]/[POSIX] function `atanf`. [C89] has `atan`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.4. kdAtan2f

Arc tangent function.

Synopsis

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

Description

Note that the parameters as named in this description are in the order *y*, *x*.

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,PI].

Other special values of *y* and *x* give a result as follows:

y	x	result
NAN	any	NAN
any	NAN	NAN
±0	-0	±PI
±0	+0	±0
±0	> 0	±0
< 0	±0	-PI/2
> 0	±0	+PI/2
finite > 0	-INF	+PI
finite < 0	-INF	-PI
finite > 0	+INF	+0
finite < 0	+INF	-0
±INF	finite	±PI/2
±INF	-INF	±3PI/4
±INF	+INF	±PI/4

Where the plus-or-minus sign ± appears for one of the inputs and the result, it means that the result must have the same sign as that input.

Rationale

`kdAtan2f` is based on the [C99]/[POSIX] function `atan2f`. [C89] has `atan2`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.5. 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,+1].

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

If *x* is infinite, the return value is a NAN.

Rationale

`kdCosf` is based on the [C99]/[POSIX] function `cosf`. [C89] has `cos`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.6. 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,+1].

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

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

Rationale

`kdSinf` is based on the [C99]/[POSIX] function `sinf`. [C89] has `sin`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.7. 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*.

Special values of *x* give a result as follows:

x	result
NAN	NAN
±0	±0
±INF	NAN

Where the plus-or-minus sign ± appears for one of the inputs and the result, it means that the result must have the same sign as that input.

If the result would overflow, the function returns `KD_INFINITY` of the same sign as *x*.

Rationale

`kdTanf` is based on the [C99]/[POSIX] function `tanf`. [C89] has `tan`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.8. 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 -INF, the function returns +0. If `x` is +INF, the function returns +INF.

If `x` is finite and the correct value would cause overflow, the function returns `KD_INFINITY`.

Rationale

`kdExpf` is based on the [C99]/[POSIX] function `expf`. [C89] has `exp`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.9. 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_INFINITY`.

If `x` is less than 0 (including the case of -INF), the function returns a NAN value.

Rationale

`kdLogf` is based on the [C99]/[POSIX] function `logf`. [C89] has `log`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.10. 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.

Rationale

`kdFabsf` is based on the [C99]/[POSIX] function `fabsf`. [C89] has `fabs`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors.

15.3.11. 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 the result would cause overflow, the function returns `KD_INFINITY` of the correct sign.

Other special values of `y` and `x` give a result as follows:

x	**y**	**result**
NAN	any except ±0	NAN
any except +1	NAN	NAN
finite < 0	finite non-integer	NAN
+1	any (including NAN)	+1
any (including NAN)	±0	+1
±0	positive odd integer	±0
±0	positive, not odd integer	+0
-1	±INF	+1
absolute value < 1	-INF	+INF
absolute value > 1	-INF	+0
absolute value < 1	+INF	+0
absolute value > 1	+INF	+INF
-INF	negative odd integer	-0

x	**y**	**result**
-INF	negative, not odd integer	+0
-INF	positive odd integer	-INF
-INF	positive, not odd integer	+INF
+INF	< 0	+0
+INF	> 0	+INF
±0	negative odd integer	±`KD_INFINITY`
±0	negative, not odd integer	`KD_INFINITY`

Where the plus-or-minus sign ± appears for one of the inputs and the result, it means that the result must have the same sign as that input.

Rationale

`kdPowf` is based on the [C99]/[POSIX] function `powf`. [C89] has `pow`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.12. 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`*.

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

If *`x`* is ±0 or +INF, the function returns *`x`*.

If *`x`* is negative (including -inf), the function returns a NAN value.

Rationale

`kdSqrtf` is based on the [C99]/[POSIX] function `sqrtf`. [C89] has `sqrt`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

15.3.13. 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 or ±0, the function returns *x*.

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.

15.3.14. 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 or ±0, the function returns *x*.

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.

15.3.15. 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 ±0 or ±inf, the function returns `x`.

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.

15.3.16. 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.

If the result would cause overflow, the function returns +INF.

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 graphical applications.

15.3.17. 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.

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

If *x* is ±0 and *y* is not zero, the function returns *x*.

If *x* is finite and y is ±INF, then the function returns *x*.

Rationale

`kdFmodf` is based on the [C99]/[POSIX] function `fmodf`. [C89] has `fmod`. [POSIX] and/or [POSIX]'s MX extension specify the NAN, infinity, out-of-domain and out-of-range behaviors, but the OpenKODE Core function does not follow C and/or POSIX in setting the error indicator.

16. String and memory functions

16.1. 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 [TR24731] functions for added safety.

16.2. Functions

16.2.1. 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`.

16.2.2. 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`.

16.2.3. 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`.

16.2.4. 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`.

16.2.5. 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`.

16.2.6. 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`* cast to a KDchar. The null termination byte is included in this scan, and thus matches if *`ch`* (cast to KDchar) 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`.

16.2.7. 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.

16.2.8. 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.

16.2.9. 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.

16.2.10. kdStrncat_s

Concatenate two strings.

Synopsis

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

Description

This function appends the first *`srcmaxlen`* 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 *`srcmaxlen`* bytes, or the buffers overlap, then undefined behavior results.

Return value

On success, the function returns 0. Any remaining space in the buffer after the resulting null-terminated string is set to undefined values.

If *`buflen`* is 0, the function does not write to memory, and returns a non-zero value.

If the resulting string, including the terminating null character, would not fit in *`buflen`* bytes, then `buf[0]` is set to 0 and the function fails, returning a non-zero value.

Rationale

`kdStrncat_s` is based on the [TR24731] function `strncat_s`. That function has additional null pointer checks.

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

16.2.11. 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`.

16.2.12. kdStrcpy_s

Copy a string with an overrun check.

Synopsis

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

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. Any remaining space in the buffer after the resulting null-terminated string is set to undefined values.

If `buflen` is 0, the function does not write to memory, and returns a non-zero value.

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 a non-zero value.

Rationale

`kdStrcpy_s` is based on the [TR24731] function `strcpy_s`. That function has additional null pointer checks.

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 force the programmer to specify a limit for the buffer length.

16.2.13. 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`. In either case the resulting string is null terminated.

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. Any remaining space in the buffer after the resulting null-terminated string is set to undefined values.

If `buflen` is 0, the function does not write to memory, and returns a non-zero value.

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 a non-zero value.

Rationale

`kdStrncpy_s` is based on the [TR24731] function `strncpy_s`. That function has additional null pointer checks.

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 force the programmer to specify a limit for the buffer length.

Note that `kdStrncpy_s` does not fill the remainder of the buffer after the null termination with further 0 bytes, unlike the C function `strncpy`.

17. Time functions

17.1. 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.

There is no OpenKODE Core function to sleep for a length of time equivalent to the [POSIX] `sleep` function. Such a function would not fit easily into OpenKODE Core's event-driven architecture. A timed delay can be achieved one of two ways:

- For a thread that does not receive any events (not the main thread), a simple call to `kdWaitEvent` with a timeout can be used.
- More generally, a thread can set up a timer using `kdSetTimer`, which will deliver its event into the thread's event queue after the specified delay.

17.2. Functions

17.2.1. 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.

17.2.2. 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.

17.2.3. 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;
} 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	second, in the range [0,60]

`tm_min`	minute, in the range [0,59]
`tm_hour`	hour, in the range [0,23]
`tm_mday`	day of the month, in the range [1,31]
`tm_mon`	month of the year, in the range [0,11]
`tm_year`	years since 1900
`tm_wday`	day of the week, in the range [0,6] where 0 is Sunday
`tm_yday`	day in the year, in the range [0,365]

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.

17.2.4. 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.

18. Timer functions

18.1. Introduction

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

18.2. Functions

18.2.1. kdSetTimer

Set timer.

Synopsis

```
#define KD_TIMER_ONESHOT 61
#define KD_TIMER_PERIODIC_AVERAGE 62
#define KD_TIMER_PERIODIC_MINIMUM 63

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. The interval between two fires is the difference between the timestamps of their events.

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. The interval between two fires is the difference between the timestamps of their events.

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. The event is sent to the queue for the thread that created the timer by calling kdSetTimer.

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.

A timer handle becomes invalid as soon as the thread that created the timer exits; using such an invalidated handle in an OpenKODE Core function results in undefined behavior. However it is undefined whether resources associated with the timer are actually freed then or some later point up to and including application exit.

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.

18.2.2. kdCancelTimer

Cancel and free a timer.

Synopsis

```
KDint 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.

If this function call overlaps with a function call in any other thread using the same *`timer`*, undefined behavior results.

The function must be called in the same thread that created the timer with `kdSetTimer`, otherwise the function fails with an error.

`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.

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_EINVAL`	Called from thread other than the one that created the timer.

18.3. Events

18.3.1. KD_EVENT_TIMER

Timer fire event.

Synopsis

```
#define KD_EVENT_TIMER 42
```

Description

When a timer (as configured by `kdSetTimer`) fires, it generates a `KD_EVENT_TIMER` event.

`KD_EVENT_TIMER` events are merged as follows: 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.

19. File system

19.1. 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.

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].

The pathname `/` cannot be used with `kdOpenDir`, `kdStat`, `kdAccess` or `kdGetFree`.

19.1.1. Not thread safe

For any particular file handle or directory handle, using the same handle in OpenKODE function calls in multiple threads at the same time results in undefined behavior.

19.2. 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, and using `kdAccess` on a file or directory in `/res` states that it is not writable.

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.

`/data` is writable, that is the application can create and/or modify directories and files here.

`/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.

`/tmp` is writable, that is the application can create and/or modify directories and files here.

`/removable` The location of any removable media devices on the device. This directory contains one or more subdirectories, each corresponding to a particular removable media that is currently present. Each is named after the slot in an implementation-defined way. The subdirectory for a slot is not present if there is no media in the slot. If the implementation supports no removable media slots, then `/removable` itself is absent.

> `/removable` itself may be successfully used with `kdOpenDir` (to scan it to see what subdirectories it contains), but cannot be used with `kdStat`, `kdAccess` or `kdGetFree`.

> 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.

`/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 and the functions that take them do not apply to `/native` or anything inside it. It is allowed for `/native` to be absent.

> 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, a third might not have `/native` at all, and a fourth might map some subset of the native file system.

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

An attempt to write to (create a file or directory in) `/` or `/removable`, or to access either of those with `kdStat` or `kdAccess` or to access `/` with `kdOpenDir`, results in a `KD_EACCES` error. It is allowed to use `/removable` in `kdOpenDir`.

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. It is undefined whether a filename with a trailing period refers to the same file as the same filename without the trailing period.

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 absolute, it starts with one of the top-level directories listed above, or is relative to `/res` (so prepending `/res/` gives the equivalent absolute file path.

If any component of a file path is one period "`.`" or two periods "`..`" then it is undefined what the file path refers to.

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). 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.

19.2.1. 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 canonical case, in order to ensure it works on both case-sensitive and case-insensitive file systems.

19.2.2. Rationale

OpenKODE Core 1.0 Provisional contained the notion of a current directory, together with functions to set and get it. It was removed with the justification that current directory is mostly useful only when the application is started from a user interface that has a notion of the user's current location which is passed to the application as its current directory. This is outside the scope of OpenKODE Core.

19.3. Constants

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

19.4. Functions

19.4.1. 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, at the end of the file. The position of the file, and thus the result of a tell, is undefined.
"`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 writing at the end of the file and for reading anywhere. The initial position of the file, and thus the position of a read or result of a tell performed before the first write or seek, is undefined.

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. This error is given on an attempt to create a file in / or /removable or open a file for writing in the subdirectory tree rooted at /res. There may be other implementation-dependent circumstances where this error is given.
KD_EINVAL	The specified mode is invalid. It is undefined whether an invalid mode gives this error.
KD_EIO	I/O error.
KD_EISDIR	The specified file path is a directory.
KD_EMFILE	Too many open files. The circumstances leading to this error are implementation dependent.
KD_ENAMETOOLONG	Path name is longer than the implementation-defined limit. It is undefined whether using a too-long name results in this error.
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.

Append mode

[POSIX] does not properly define the interaction between the append modes (`"a"` and `"a+"`) and the file position indicator. OpenKODE Core leaves this explicitly undefined. In particular, in `"a"` mode, a write always writes to the end of the file but the state of the file position indicator is completely undefined, whereas in `"a+"` mode, a write moves the file position to the end of the file, but the state of the file position indicator on file open is undefined.

The usual ways of using append mode files will not encounter these areas of undefinedness. However, a “testing” OpenKODE Core implementation might like to detect and abort on any attempt to rely on undefined behavior, using a read or tell before the first seek or write on an `"a+"`, or any use of tell on an `"a"` file.

19.4.2. 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 this function call overlaps with a function call in any other thread using the same *`file`*, then undefined behavior results.

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. The circumstances leading to this error are implementation defined.
`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.

19.4.3. 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 this function call overlaps with a function call in any other thread using the same `file`, then undefined behavior results.

If `file` 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. The circumstances leading to this error are implementation defined.
`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`, but without an equivalent of `fflush(NULL)` flushing all open file handles. [POSIX] adds the setting of `errno` on error.

What buffering is used, and how much `kdFclose` and `kdFflush` cause buffers to be flushed, is implementation dependent and cannot be specified by OpenKODE Core. However the intention of `kdFflush` is to flush the same amount as `kdFclose` does, so `kdFflush` is the same (as regards buffering) as closing and reopening the file.

`kdFflush` is intended to be analogous to the [C89] function `fflush`. A typical address-space-separated multi-process OS such as Unix or Windows implements these C89 file functions with buffers in the user process, and `fflush` flushes a file's buffer to the OS. The OS may then have its own buffering before the data is sent to the physical device, but `fflush` is not concerned about those OS buffers.

For such a multi-process OS, a crash of just the application process would cause the loss of data not yet flushed by `fflush` or `fclose`, but no loss of data that has been flushed. A crash of the whole OS could additionally cause the loss of data that has been flushed.

19.4.4. 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 the file is open only for writing, then it is undefined whether this read function succeeds, fails with an error, or appears to succeed.

If this function call overlaps with a function call in any other thread using the same *file*, then undefined behavior results.

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.

19.4.5. 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 the file is open in `"a"` (append) mode, then the write starts at the current end of file. The file's position indicator is left in an undefined state.

If the file is open in `"a+"` (append) mode, then the write starts at the current end of file, by setting the file's position indicator to the end of the file before the write, and (if there is no error) the write advances the position indicator by the number of bytes actually written in the normal way.

If the file is open only for reading, then it is undefined whether this write function succeeds, fails with an error, or appears to succeed.

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.

19.4.6. 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 this function call overlaps with a function call in any other thread using the same *`file`*, then undefined behavior results.

If the file is open only for writing, then it is undefined whether this read function succeeds, fails with an error, or appears to succeed.

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.

19.4.7. kdPutc

Write a byte to an open file.

Synopsis

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

Description

This function writes the byte whose value is given by the bottom 8 bits of *`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 this function call overlaps with a function call in any other thread using the same *`file`*, then undefined behavior results.

If the file is open in `"a"` (append) mode, then the write starts at the current end of file. The file's position indicator is left in an undefined state.

If the file is open in `"a+"` (append) mode, then the write starts at the current end of file, by setting the file's position indicator to the end of the file before the write, and (if there is no error) the write advances the position indicator by one in the normal way.

If the file is open only for reading, then it is undefined whether this write function succeeds, fails with an error, or appears to succeed.

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. The circumstances leading to this error are implementation defined.
`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.

19.4.8. 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 this function call overlaps with a function call in any other thread using the same *`file`*, then undefined behavior results.

If the file is open only for writing, then it is undefined whether this read function succeeds, fails with an error, or appears to succeed.

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 read succeeds in reading some data but reaches end-of-file before reaching the length limit or a newline character, the function returns *`buffer`* containing the data read (null terminated), and sets the file’s end-of-file indicator (as returned by `kdFEOF`). If the read reaches end-of-file before reading any characters, 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.

19.4.9. 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 this function call overlaps with a function call in any other thread using the same *`file`*, then undefined behavior results.

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.

19.4.10. 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 this function call overlaps with a function call in any other thread using the same *`file`*, then undefined behavior results.

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 error indicator is set.

19.4.11. 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 this function call overlaps with a function call in any other thread using the same `file`, then undefined behavior results.

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

19.4.12. 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. 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 this function call overlaps with a function call in any other thread using the same `file`, then undefined behavior results.

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. The circumstances leading to this error are implementation defined.
`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`.

19.4.13. 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 this function call overlaps with a function call in any other thread using the same *`file`*, then undefined behavior results.

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`.

19.4.14. 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 other than `/` or `/removable` 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. This error is given on an attempt to create a directory in `/` or `/removable` or in the subdirectory tree rooted at `/res`. There may be other implementation-dependent circumstances where this error is given.
`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. It is undefined whether using a too-long name results in this error.
`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.

19.4.15. 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. This error is given on an attempt to remove a directory in `/` or `/removable` or in the subdirectory tree rooted at `/res`. There may be other implementation-dependent circumstances where this error is given.
`KD_EBUSY`	*`pathname`* is in use in some undefined way which makes the operation impossible.
`KD_EEXIST`	Directory is not empty. It is undefined whether this circumstance causes this error or `KD_EBUSY`.
`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. It is undefined whether using a too-long name results in this error.
`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.

19.4.16. 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.

It is undefined whether the function succeeds or fails with an error when either filename is in `/native`. Otherwise, if the two filenames are in different OpenKODE top-level directories, `/removable/`*xxx* for some *xxx*, and the other is not in that directory, then the function fails with an error. If the two filenames are in the same OpenKODE top-level directory other than `/removable`, or both in `/removable/`*xxx* for the same *xxx*, then the function succeeds.

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. This error is given on an attempt to move a file from or to `/` or `/removable` or in the subdirectory tree rooted at `/res`. There may be other implementation-dependent circumstances where this error is given.
`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. It is undefined whether using a too-long name results in this error.
`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.

19.4.17. 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. This error is given on an attempt to remove a file from `/` or `/removable` or the subdirectory tree rooted at `/res`. There may be other implementation-dependent circumstances where this error is given.
`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. It is undefined whether using a too-long name results in this error.
`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.

19.4.18. 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.

It is undefined whether attempting to truncate an open file succeeds or fails with an error. If it does succeed, then the open file’s file position indicator remains unchanged. If that position is now beyond the end of the file, and data is subsequently written at that position, the intervening empty space is filled with 0 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. This error is given on an attempt to truncate a file in the subdirectory tree rooted at `/res`. There may be other implementation-dependent circumstances where this error is given.
`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. It is undefined whether using a too-long name results in this error.
`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.

19.4.19. 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(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. It is a bitmap, with the following bits:

- The `0x8000` bit is set if the entity is a regular file. This can be tested with the `KD_ISREG` macro, taking the *st_mode* value as its argument, returning non-zero if the file system entity is a regular file.
- The `0x4000` bit is set if the entity is a directory. This can be tested with the `KD_ISDIR` macro, taking the *st_mode* value as its argument, returning non-zero if the file system entity is a directory.

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.

Other bits in the field have undefined values.

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. This error is given on an attempt to stat `/` or `/removable`. There may be other implementation-dependent circumstances where this error is given.
`KD_EIO`	I/O error.
`KD_ENAMETOOLONG`	Path name is longer than the implementation-defined limit. It is undefined whether using a too-long name results in this error.
`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.

19.4.20. kdAccess

Determine whether the application can access a file or directory.

Synopsis

```
KDint kdAccess(const KDchar *pathname, KDint amode);
```

Description

This function checks whether the directory or file with name *`pathname`* is accessible in the way(s) defined by the bit(s) set in *`amode`*.

`amode` is a logical "or" of one or more of the following values:

`KD_R_OK` (4)	Check whether the file or directory is readable. A directory must be readable in order to successfully open it with `kdOpenDir`.
`KD_W_OK` (2)	Check whether the file or directory is writable. A directory must be writable in order to create a new file or directory in it.
`KD_X_OK` (1)	Check whether it is possible to access files or directories within the named directory. The semantics of this type of accessibility test applied to a file are undefined.

It is not guaranteed that the application will actually be able to access the named file or directory in the way(s) that this function says it can, firstly because the file or directory may be changed by another application just after the `kdAccess` call, and secondly because some implementations are not able to glean information to the detail specified here from the underlying operating system.

Results are undefined if *`pathname`* is an object in the file system other than a file or directory, or if *`amode`* is zero or has bits set other than those specified.

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

Return value

If *`pathname`* exists and is accessible in the way(s) specified in *`amode`*, then the function returns 0. Otherwise, the function returns -1 and stores one of the error codes listed below into the error indicator returned by `kdGetError`.

Error codes

`KD_EACCES`	The file or directory exists but is not accessible in the way(s) specified in *`amode`*, or one of the path elements other than the last in *`pathname`* is a directory in which the application cannot access files and directories.
`KD_EIO`	I/O error.
`KD_ENAMETOOLONG`	Path name is longer than the implementation-defined limit. It is undefined whether using a too-long name results in this error.
`KD_ENOENT`	File or directory not found.
`KD_ENOMEM`	Out of memory or other resource.

Rationale

`kdAccess` is based on the [POSIX] function `access`.

[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.

19.4.21. 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_EACCES` if the specified directory is `/` .

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. This error is given on an attempt to open `/`. There may be other implementation-dependent circumstances where this error is given.
`KD_EIO`	I/O error.
`KD_ENAMETOOLONG`	Path name is longer than the implementation-defined limit. It is undefined whether using a too-long name results in this error.
`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.

19.4.22. kdReadDir

Return the next file in a directory.

Synopsis

```
typedef struct KDDirent {
    const KDchar *d_name;
} 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, itself containing a field *d_name* pointing to the null-terminated name of the directory entity, relative to the directory being listed (thus no path separator characters in the name). The KDDirent structure and the name pointed to by *d_name* remain valid only until the next call to `kdReadDir` or `kdCloseDir` with the same *dir* parameter (even if that call is in a different thread).

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 points to the null-terminated name of the directory entity, relative to the directory being listed (thus no path separator characters in the name).

If this function call overlaps with a function call in any other thread using the same *dir*, then undefined behavior results.

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`.

19.4.23. kdCloseDir

Close a directory.

Synopsis

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

Description

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

If this function call overlaps with a function call in any other thread using the same *`dir`*, then undefined behavior results.

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.

19.4.24. 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, and whether this information is available at all, are 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. This error is given if the path is `/` or `/removable`. There may be other implementation-dependent circumstances where this error is given.
`KD_EIO`	I/O error.
`KD_ENAMETOOLONG`	Path name is longer than the implementation-defined limit. It is undefined whether using a too-long name results in this error.
`KD_ENOENT`	File or directory not found.
`KD_ENOMEM`	Out of memory or other resource.
`KD_ENOSYS`	Information not available for this part of the file system.

`KD_EOVERFLOW`	The free space size cannot be represented by a KDoff.

20. Network sockets

20.1. Introduction

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

- OpenKODE Core is event based, and exposes only non-blocking sockets that 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.
- Rather than a sockaddr struct type with a specialized struct type for each address family (sockaddr_in for IPv4), OpenKODE Core uses a single struct type `KDSockaddr`, containing a union with a struct for each supported address family (currently only the one for IPv4).

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`.

20.1.1. 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` in the thread that created the socket (if not already in such a call). The application cannot tell the difference between this and the event being generated immediately.

20.1.2. Not thread safe

For any particular socket handle, using the same handle in OpenKODE function calls in multiple threads at the same time results in undefined behavior, except that `kdSocketAccept` is allowed to be used at the same time as any socket function using the same handle.

20.2. Network connection

If a name is looked up or a socket is created and used in such a way that the platform needs to establish a connection to a network that it is not permanently connected to (for example over the air), it is undefined at what point the connection is established.

20.2.1. Network connection implementation notes

It is recommended that the platform should not establish a connection that spends the user’s money any earlier than a call to `kdNameLookup` or a call to `kdSocketConnect`.

Where the underlying OS demands connection before the socket is created, the implementation could allow creation of the OpenKODE Core socket handle without creating an OS socket, and then asynchronously establish the network connection when the application calls `kdSocketConnect`. The implementation does not then create the OS socket until the network connection has been established, and the OpenKODE Core `KD_EVENT_SOCKET_CONNECT_COMPLETE` event is not generated until the network connection is established, the socket has been created, and the socket has been connected.

20.3. Types

20.3.1. KDSockaddr

Struct type for socket address.

Synopsis

```
typedef struct KDSockaddr {
    KDuint16 family;
    union {
#define KD_AF_INET 70
        struct {
            KDuint16 port;
            KDuint32 address;
        } sin;
    } data;
} KDSockaddr;
```

Description

This struct contains a socket address. The two top-level members are *`family`*, which specifies the address family, and *`data`*, which is a union where the union member in use depends on the address family.

OpenKODE Core supports only the address family `KD_AF_INET`, for use with TCP and UDP over IPv4. When *`family`* is `KD_AF_INET`, the *`sin`* union member is used, with the following fields:

`port`	Port number in network byte order
`address`	IP address in network byte order

Any special signficance attached to certain ranges of IP addresses is implementation specific. In particular, there is no guarantee that there is a loopback interface accessed with the IP address 127.0.0.1.

20.4. Functions

20.4.1. kdNameLookup

Look up a hostname.

Synopsis

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

Description

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

`af` is the address family in which to look for the name. It is one of the following values:

`KD_AF_INET`	Search for the name in the IPv4 address family. 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, each with a single address; the `userptr` value of each event is as supplied to this function in the `eventuserptr` parameter, and each event is sent to the queue for the thread that initiated the operation by calling `kdNameLookup`.

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 `af` is not an address family that supports name lookup, then the function fails with an error as below.

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`	Address family 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.

20.4.2. kdNameLookupCancel

Selectively cancels ongoing kdNameLookup operations.

Synopsis

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

Description

This function cancels any outstanding lookup operations initiated by calls in this same thread 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 initiated by calls in this same thread 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.

20.4.3. 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` (64)	TCP over IPv4. The socket is connection-based.
`KD_SOCK_UDP` (65)	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 give the error `KD_EAGAIN`.

`eventuserptr` is the value that will be used for the *`userptr`* field in any event associated with the socket. Any event associated with the socket is sent to the queue for the thread that created the socket.

Any socket left open at application termination is automatically closed.

A socket handle becomes invalid as soon as the thread that created the socket exits; using such an invalidated handle in an OpenKODE Core function results in undefined behavior. However it is undefined whether resources associated with the socket, including any connection within or outside the platform, are actually freed then or some later point up to and including application exit.

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.

`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`).

20.4.4. 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 this function call overlaps with a function call in any other thread other than `kdSocketAccept` using the same *`socket`*, then undefined behavior results.

Otherwise, the function must be called in the same thread that created the socket with `kdSocketCreate` or `kdSocketAccept`, otherwise the function fails with an error.

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, except where noted.

Error codes

`KD_EINVAL`	Called from thread other than the one that created the socket. The socket is still open and usable.
`KD_EIO`	I/O error.
`KD_ENOMEM`	Out of memory or other resource.

20.4.5. kdSocketBind

Bind a socket.

Synopsis

KDint **kdSocketBind**(KDSocket **socket*, const struct KDSockaddr **addr*, KDboolean *reuse*);

Description

This function binds *`socket`* to the local address specified in the location pointed to by *`addr`* (of type `KDSockaddr`).

If *`addr`*`->family` is `KD_AF_INET`, then *`addr`* specifies an IPv4 address, and *`addr`*`->data.sin.address` and *`addr`*`->data.sin.port` specify the local IP address and port number to bind to. If *`addr`*`->data.sin.address` 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 a receive operation would not give the error `KD_EAGAIN` (including the case that it would give some other error).

If this function call overlaps with a function call in any other thread other than `kdSocketAccept` using the same *`socket`*, then undefined behavior results.

If *`socket`* is not a socket, or has already been closed, or *`addr`* is not a readable location of type `KDSockaddr`, 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`	*`addr`*`->family` is not `KD_AF_INET`

`KD_EINVAL`	Socket is already bound to an address.
`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.

20.4.6. kdSocketGetName

Get the local address of a socket.

Synopsis

```
KDint kdSocketGetName(KDSocket *socket, struct KDSockaddr *addr);
```

Description

This function stores the local address that *`socket`* is bound to into the location pointed to by *`addr`* (of type `KDSockaddr`).

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 an `KD_AF_INET` family address with the local IP address and port that the socket is bound to.

If this function call overlaps with a function call in any other thread other than `kdSocketAccept` using the same *`socket`*, then undefined behavior results.

If *`socket`* is not a socket, or has already been closed, or *`addr`* does not point to a writable location of type `KDSockaddr`, 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. This error cannot occur for TCP and UDP sockets, and is specified as a placeholder for possible future socket types.

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.

20.4.7. kdSocketConnect

Connects a socket.

Synopsis

```
KDint kdSocketConnect(KDSocket *socket, const KDSockaddr *addr);
```

Description

This function initiates an operation to connect *`socket`* to the remote address specified in the location pointed to by *`addr`* (which is of type `KDSockaddr`).

If *`addr`*`->family` is `KD_AF_INET`, then *`addr`* specifies an IPv4 address, the only address family supported by OpenKODE Core, giving 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 this function call overlaps with a function call in any other thread other than `kdSocketAccept` using the same *`socket`*, then undefined behavior results.

If *`socket`* is not a socket, or has already been closed, or *`addr`* does not point to a readable location of type `KDSockaddr`, 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. It is undefined whether this condition results in this `KD_ECONNREFUSED` error or the catch-all `KD_EIO`.
`KD_ECONNRESET`	The remote host reset the connection. It is undefined whether this condition results in this `KD_ECONNRESET` error or the catch-all `KD_EIO`.

`KD_EHOSTUNREACH`	Remote host cannot be reached. It is undefined whether this condition results in this `KD_EHOSTUNREACH` error or the catch-all `KD_EIO`.
`KD_EINVAL`	The socket is listening.
`KD_EIO`	General I/O or network error.
`KD_EISCONN`	Socket is connection-based and already connected.
`KD_ENOMEM`	Out of memory or other resource.
`KD_ETIMEDOUT`	Connection attempt timed out. It is undefined whether this condition results in this `KD_ETIMEDOUT` error or the catch-all `KD_EIO`.

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.

This specification does not mandate the separate detection of the errors `KD_ECONNREFUSED`, `KD_ECONNRESET`, `KD_EHOSTUNREACH` or `KD_ETIMEDOUT`, since not all OSes are able to detect these conditions separately.

[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`.

20.4.8. 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 this function call overlaps with a function call in any other thread other than `kdSocketAccept` using the same *`socket`*, then undefined behavior results.

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`.

20.4.9. kdSocketAccept

Accept an incoming connection.

Synopsis

```
KDSocket *kdSocketAccept(KDSocket *socket, KDSockaddr *addr, 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` and `KD_EVENT_SOCKET_WRITABLE` events are generated as soon as the socket would not give the error `KD_EAGAIN` on a receive or send operation 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`* (of type `KDSockaddr`).

`eventuserptr` is the value to use for the *`userptr`* field of any event generated by the new, connected, socket. Any such event is sent to the queue for the thread that created the socket by calling `kdSocketAccept`.

A socket handle becomes invalid as soon as the thread that created the socket exits, therefore using the handle in an OpenKODE Core function results in undefined behavior. However it is undefined whether resources associated with the socket, including any connection within or outside the platform, are actually freed then or some later point up to and including application exit.

If *`socket`* is not a socket, or has already been closed, or *`addr`* is not `KD_NULL` and not a pointer to a readable and writable KDsockaddr location, 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).

Unlike other functions that take a socket handle, it is allowed to use `kdSocketAccept` at the same time as another thread is calling an OpenKODE Core socket function (including another `kdSocketAccept`) with the same handle. This exception is designed to allow for the case where a server is implemented by responding to a `KD_EVENT_SOCKET_INCOMING` by starting a new thread to accept the connection and handle the resulting new socket. An implementation must ensure that `kdSocketAccept`'s thread safety is maintained even if the application is attempting to do something that causes an error return, such as accepting on a socket that is not listening, or reading/writing on a socket that is listening. It should be possible for an implementation to implement `kdSocketAccept` thread safety without needing to protect all socket functions, since `kdSocketAccept` can only legally be used on a listening socket, and the function can test the listening state of the socket by reading a simple flag without a lock.

20.4.10. 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);
```

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`.

In `kdSocketSendTo`, if *`addr`* is not `KD_NULL`, it points to a location (of type `KDSockaddr`) which specifies the remote address to send to.

Since OpenKODE Core supports only IPv4, this address is of that family, specifying the remote IP address and port.

If `kdSocketSendTo` is used on a connection-based socket with *`addr`* set to a value other than `KD_NULL`, 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.

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.

The OpenKODE Core implementation is likely to have a limit to how much data can be sent at once. For TCP, the limit is unspecified; since TCP is stream-based, the application can use more than one `kdSocketSend` call to send data if necessary. For UDP, the limit is defined to be at least 1472 bytes, in that it is possible to send a 1472 byte datagram at certain times (perhaps when no other sent data is queued). A `KD_EVENT_SOCKET_WRITABLE` event does not necessarily mean that 1472 bytes are available now, so a send of a large datagram may still fail. Since UDP is unreliable, from the application's point of view this is no different to the send succeeding but the packet being lost in transit.

If this function call overlaps with a function call in any other thread other than `kdSocketAccept` using the same *`socket`*, then undefined behavior results.

If `socket` is not a socket, or has already been closed, or `addr` is not `KD_NULL` and is not a readable location of type `KDSockaddr`, 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.

For TCP, the number of bytes sent may be less than the number requested. For UDP, either all bytes are sent or there is an error.

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_EAFNOSUPPORT`	*addr*`->family` is not `KD_AF_INET`
`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.

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:

- `EISCONN` 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.

20.4.11. 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);
```

Description

These functions receive data from a socket. `kdSocketRecv` is equivalent to `kdSocketRecvFrom` with *`addr`* 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`* (of type `KDSockaddr`).

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 this function call overlaps with a function call in any other thread other than `kdSocketAccept` using the same *`socket`*, then undefined behavior results.

If *`socket`* is not a socket, or has already been closed, or *`addr`* is not `KD_NULL` and does not point to a writable location of type `KDSockaddr`, then undefined behavior results.

Return value

The functions return the number of bytes received on success. For a connection-based socket, the functions may return 0, which indicates that the remote end has closed the connection in an orderly way and no more queued data remains to read. 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. It is undefined whether this condition results in this `KD_ETIMEDOUT` error or the catch-all `KD_EIO`.

Rationale

`kdSocketRecv` and `kdSocketRecvFrom` are based on the BSD and [POSIX] functions `recv` and `recvfrom`.

This specification does not mandate the separate detection of the error `KD_ETIMEDOUT`, since not all OSes are able to detect these conditions separately.

[POSIX] defines additional error codes, some of which are not applicable to the subset of socket functionality defined by OpenKODE Core, but also including:

- `EISCONN` 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.

20.4.12. 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`.

20.4.13. 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`.

20.4.14. 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`.

20.4.15. 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`.

20.4.16. 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. The resulting numbers are then combined into a 32-bit integer, starting with the first number in the highest order bits, and with each number except the last taking 8 bits and the last number taking the remaining lowest order bits. The 32-bit integer is then stored in network order (i.e. as if passed through `kdHtonl`) into the location pointed to by *`inp`*.

If *`inp`* is not a pointer to a writable KDuint32 location, then undefined behavior results.

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.

20.4.17. kdInetNtop

Convert a network address to textual form.

Synopsis

```
#define KD_INET_ADDRSTRLEN 16
typedef struct KDInAddr {
    KDuint32 s_addr;
} KDInAddr;

const KDchar *kdInetNtop(KDuint af, const void *src, KDchar *dst, KDsize cnt);
```

Description

This function converts a network address to its textual representation.

af is the address family of the address pointed to by *src*. OpenKODE Core supports only `KD_AF_INET`. *dst* points to a buffer to store the null-terminated string result into. *cnt* is the length in bytes of the buffer.

For address family `KD_AF_INET`, the result is the address in “dotted quad” notation. *cnt* must be at least `KD_INET_ADDRSTRLEN`. The function takes the IP address from the *s_addr* field of the KDInAddr struct pointed to by *src*, converts it from network order as if by passing it through `kdNtohl`, and splits it into four 8-bit components, with bits 31..24 (as seen after the `kdNtohl` conversion) giving the first component and bits 7..0 giving the fourth component. The components are then converted to decimal numbers (with no leading zeros but at least one digit) and separated with dots.

If *af* is not a supported address family, or *cnt* is not big enough for the specified address family, then an error is returned. If *src* is not a readable struct of the appropriate type for the address family, or *dst* is not a writable buffer of at least *cnt* bytes, then undefined behavior results.

Return value

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

Error codes

`KD_EAFNOSUPPORT`	Address family not supported.
`KD_ENOSPC`	*cnt* not big enough.

Rationale

`kdInetNtop` is based on the [POSIX] function `inet_ntop`.

20.5. Events

20.5.1. KD_EVENT_SOCKET_READABLE

Event to indicate that a socket is readable.

Synopsis

```
#define KD_EVENT_SOCKET_READABLE 49
```

Description

For a connected socket or a bound connectionless socket, this event is generated whenever `kdWaitEvent` or `kdPumpEvents` is called or an event callback returns or the former is blocked, and one of the following holds:

- The socket has changed from a non-readable state (a connection-based socket was not connected, or a connectionless socket was unbound, or a call to `kdSocketRecv` or `kdSocketRecvFrom` (as applicable) would give the error `KD_EAGAIN`) to a readable state (a call to the function would succeed or give an error other than `KD_EAGAIN`).
- Since the last time `kdWaitEvent` or `kdPumpEvents` was called or an event callback returned, a call to `kdSocketRecv` or `kdSocketRecvFrom` (as applicable) has been made which successfully read a non-zero number of bytes, yet the socket remains in a readable state (as above).

In addition, an event *may* be generated at any time for a socket that is connectionless and bound, or connection-based and connected. It is implementation dependent whether such "spurious" events are generated; an OpenKODE Core application must be able to cope with receiving the event when the socket is not readable (so `kdSocketRecv` or `kdSocketRecvFrom` would give an error of `KD_EAGAIN`).

This last provision is present for two reasons:

- OpenKODE Core 1.0 mandated that an event would be generated if any successful `kdSocketRecv` or `kdSocketRecvFrom` does not drain all available data from the socket, even if all data has been drained by event pump time. Thus an OpenKODE Core 1.0 implementation does not need to be modified in this respect to be OpenKODE Core 1.0.1 compliant.
- Certain OSes may generate their equivalents of these spurious events, so an OpenKODE Core implementation on such an OS may find it difficult to filter out the spurious ones.

Being readable includes the case where `kdSocketRecv` returns 0 due to the other end of the connection being closed in an orderly way. The definition above means that this event is generated when a socket first enters this state, but not after any subsequent read which returns 0.

`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 is sent to the queue for the thread that created the socket.

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.

20.5.2. KD_EVENT_SOCKET_WRITABLE

Event to indicate that a socket is writable.

Synopsis

```
#define KD_EVENT_SOCKET_WRITABLE 50
```

Description

For a socket, this event is generated whenever `kdWaitEvent` or `kdPumpEvents` is called or an event callback returns or the former is blocked, and one of the following holds:

- The socket has changed from a non-writable state (a connection-based socket was not connected, or a call to `kdSocketSend` or `kdSocketSendTo` (as applicable) would give the error `KD_EAGAIN`) to a writable state (a call to the function would succeed or give an error other than `KD_EAGAIN`).
- Since the last time `kdWaitEvent` or `kdPumpEvents` was called or an event callback returned, a call to `kdSocketSend` or `kdSocketSendTo` (as applicable) has been made which successfully wrote a non-zero number of bytes, yet the socket remains in a writable state (as above).

In addition, an event *may* be generated at any time for a socket that is connectionless and bound, or connection-based and connected. It is implementation dependent whether such "spurious" events are generated; an OpenKODE Core application must be able to cope with receiving the event when the socket is not writable (so `kdSocketSend` or `kdSocketSendTo` would give an error of `KD_EAGAIN`).

This last provision is present for two reasons:

- OpenKODE Core 1.0 mandated that an event would be generated if any successful `kdSocketSend` or `kdSocketSendTo` does not fill the socket's buffer, even if the buffer has been filled by event pump time. Thus an OpenKODE Core 1.0 implementation does not need to be modified in this respect to be OpenKODE Core 1.0.1 compliant.
- Certain OSes may generate their equivalents of these spurious events, so an OpenKODE Core implementation on such an OS may find it difficult to filter out the spurious ones.

`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 is sent to the queue for the thread that created the socket.

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.

20.5.3. KD_EVENT_SOCKET_CONNECT_COMPLETE

Event generated when a socket connect is complete

Synopsis

```
#define KD_EVENT_SOCKET_CONNECT_COMPLETE 51
```

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 is sent to the queue for the thread that created the socket.

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.

20.5.4. KD_EVENT_SOCKET_INCOMING

Event generated when a listening socket detects an incoming connection or an error.

Synopsis

```
#define KD_EVENT_SOCKET_INCOMING 52
```

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 is sent to the queue for the thread that created the socket.

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.

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.

20.5.5. KD_EVENT_NAME_LOOKUP_COMPLETE

`kdNameLookup` complete event.

Synopsis

```
#define KD_EVENT_NAME_LOOKUP_COMPLETE 53
```

Description

This event is generated when a lookup initiated by a call to `kdNameLookup` is complete, either successfully or with an error. A name lookup generates one or more events, with a single address or an error indication in each.

The event's *userptr* field is set to the value supplied in the *eventuserptr* parameter to `kdNameLookup`. The event is sent to the queue for the thread that initiated the lookup by calling that function.

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 KDSockaddr location pointed to by *result*. The length of the returned KDSockaddr is *resultlen*. For an IP address, the *data.sin.address* field gives the IP address, but the *data.sin.port* field has an undefined value.

If the lookup generated an error, the error code is in the *error*, and the event is the last one to be generated by the name lookup (thus the *more* field is 0).

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 event struct itself becomes invalid (when the callback returns if the event is being handled by a callback, or when the same thread next calls `kdWaitEvent` if this event was returned by a call to `kdWaitEvent`).

Error codes

`KD_EHOST_NOT_FOUND`	The specified name is not known.

`KD_EIO`	General I/O or network error.
`KD_ENOMEM`	Out of memory or buffers.
`KD_ENO_DATA`	The specified name is valid but does not have an address. An implementation that cannot distinguish this condition from `KD_EHOST_NOT_FOUND` may use that error instead.
`KD_ENO_RECOVERY`	A non-recoverable error has occurred on the name server. An implementation that cannot distinguish this condition from `KD_EHOST_NOT_FOUND` may use that error instead.
`KD_ETRY_AGAIN`	A temporary error has occurred on an authoritative name server, and the lookup may succeed if retried later. An implementation that cannot distinguish this condition from `KD_EHOST_NOT_FOUND` may use that error instead.

21. Input/output

21.1. 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 input/output API also includes a mechanism for reading global state values (such as battery status), and receiving an event when such a state changes.

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.

State values, 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.

A state value can be retrieved using one of the `kdStateGet*` functions. In addition, any change in a state value is notified by an event, `KD_EVENT_STATE`.

An input value cannot be directly retrieved, but any change to an input is notified by an event sent to at least any of the application's windows that have input focus. If the same input has multiple indexes in different I/O groups, then an event is generated for each such index 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.

21.1.1. 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.

21.1.2. Game controller

OpenKODE Core has a concept of a *game controller*, which the platform may have multiple instances of, and this is implemented by I/O groups that could be applicable to a game controller potentially appearing multiple times in the I/O index space. For instance, `KD_IOGROUP_BUTTONS` represents the buttons on the first game controller, `KD_IOGROUP_BUTTONS + KD_IO_CONTROLLER_STRIDE` represents the buttons on the second game controller, and so on.

In a handheld platform such as a handset, the handset is considered the first (and often the only) controller.

21.2. Events

21.2.1. KD_EVENT_STATE

State changed event.

Synopsis

```
#define KD_EVENT_STATE 55

typedef struct KDEventState {
    KDint32 index;
    union {
        KDint32 i;
        KDint64 l;
        KDfloat32 f;
    } value;
} KDEventState;
```

Description

A state generates this event whenever its value changes.

The event data is in the *`state`* of the event data union, of type KDEventState. Within this struct, *`index`* is the index of the state whose change caused the event, and one of *`value.i`*, *`value.l`* or *`value.f`* is the new value of the state, for state type KDint32, KDint64 or KDfloat32 respectively.

The 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, and it is delivered to the main thread (the one in which `kdMain` was called).

21.2.2. KD_EVENT_INPUT

Input changed event.

Synopsis

```
#define KD_EVENT_INPUT 56

typedef struct KDEventInput {
    KDint32 index;
    union {
        KDint32 i;
        KDint64 l;
        KDfloat32 f;
    } value;
} KDEventInput;
```

Description

Unless otherwise specified, an input 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 KDint32, KDint64 or KDfloat32 respectively.

When any of application's windows has input focus, the event is sent to that window, thus the *`eventuserptr`* parameter supplied when the window was created is used as the value of the *`userptr`* field of the input event, and the event is sent to the queue for the thread that created the window. When none of application's realized windows has input focus, it is unspecified whether one of them receives the event. When the application has no realized windows, it is unspecified whether no event is sent at all, or an event is sent to the main thread with a *`userptr`* field of `KD_NULL`.

No attempt is made to make events give a consistent state for a particular window. For example, a window with input focus may receive a key down event and then lose focus, so it never sees the key up event (which gets sent to the new input focus window). Thus an application must be aware of focus loss/gain events and modify its input event handling accordingly.

Input focus

An application is likely to maintain its own state for inputs it is interested in. If it maintains a per-window state, then a sensible approach for buttons and keys is to assume that they are all up when the window gains focus and go up when the window loses focus. This avoids the problem where the code maintaining the window's state sees a key down event but misses the subsequent key up event because the window has lost focus in the meantime.

An application that is in a mode that requires user interaction (for example a game in gameplay mode) needs to allow for possibly not receiving any more input events when the window loses focus, for example by pausing itself.

21.2.3. KD_EVENT_INPUT_JOG

Jogdial jog event.

Synopsis

```
#define KD_EVENT_INPUT_JOG 71
typedef struct KDEventInputJog {
    KDint32 index;
    KDint32 count;
} KDEventInputJog;
```

Description

When a jog dial is jogged in a particular direction, this event is generated.

The data is in the *`inputjog`* element of the event data union, with type KDEventInputJog, with the following fields:

- *`index`* is the index number of the input that actually changed;
- *`count`* contains the count of how far the jogdial has been jogged in that direction;

`KD_EVENT_INPUT_JOG` 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 for the same jogdial I/O group was for the same I/O index (i.e. jogging in the same direction), then the old event is removed from the queue as the new event is added to the end of the queue with the count summed.

When any of application's windows has input focus, the event is sent to that window, thus the *`eventuserptr`* parameter supplied when the window was created is used as the value of the *`userptr`* field of the input event, and the event is sent to the queue for the thread that created the window. When none of application’s realized windows has input focus, It is unspecified whether one of them receives the event. When the application has no realized windows, it is unspecified whether no event is sent at all, or an event is sent to the main thread with a *userptr* field of KD_NULL.

No attempt is made to make events give a consistent state for a particular window. For example, a window with input focus may receive a select button down event and then lose focus, so it never sees the button up event (which gets sent to the new input focus window). Thus an application must be aware of focus loss/gain events and modify its input event handling accordingly.

Input focus

See the note above on events and input focus.

21.2.4. KD_EVENT_INPUT_POINTER

Pointer input changed event.

Synopsis

```
#define KD_EVENT_INPUT_POINTER 57
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 if it was the button, or the index number of the X axis input if only one or both of the axes 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 or both coordinates.

When any of application’s windows has input focus, the event is sent to that window, thus the *eventuserptr* parameter supplied when the window was created is used as the value of the *userptr* field of the input event, and the event is sent to the queue for the thread that created the window. When none of application’s realized windows has input focus, It is unspecified whether one of them receives the event. When the application has no realized windows, it is unspecified whether no event is sent at all, or an event is sent to the main thread with a *userptr* field of KD_NULL.

No attempt is made to make events give a consistent state for a particular window. For example, a window with input focus may receive a select button down event and then lose focus, so it never sees the button up event (which gets sent to the new input focus window). Thus an application must be aware of focus loss/gain events and modify its input event handling accordingly.

Input focus

See the note above on events and input focus.

21.2.5. KD_EVENT_INPUT_STICK

Joystick stick changed event.

Synopsis

```
#define KD_EVENT_INPUT_STICK 58
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 X axis input of the stick that 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 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 or both coordinates.

When any of application's windows has input focus, the event is sent to that window, thus the *eventuserptr* parameter supplied when the window was created is used as the value of the *userptr* field of the input event, and the event is sent to the queue for the thread that created the window. When none of application's realized windows has input focus, It is unspecified whether one of them receives the event. When the application has no realized windows, it is unspecified whether no event is sent at all, or an event is sent to the main thread with a *userptr* field of KD_NULL.

Input focus

See the note above on events and input focus.

21.3. Functions

21.3.1. kdStateGeti, kdStateGetl, kdStateGetf

get state value(s)

Synopsis

KDint **kdStateGeti**(KDint *startidx*, KDuint *numidxs*, KDint32 **buffer*);

KDint **kdStateGetl**(KDint *startidx*, KDuint *numidxs*, KDint64 **buffer*);

KDint **kdStateGetf**(KDint *startidx*, KDuint *numidxs*, KDfloat32 **buffer*);

Description

This function retrieves *numidxs* (zero or more) state values in a contiguous index range starting at *startidx*, all of the same type.

kdStateGeti gets KDint32 state values, kdStateGetl gets KDint64 state values, and kdStateGetf gets KDfloat32 state values.

If not all of the indexes in the range are states of the applicable type or are not all in the same I/O group, then the range is cut short so all indexes are states of the applicable type and are in the same I/O group. This includes the case of cutting the range to 0 size when the specified index itself is not a state of the applicable type or is not a valid index in an I/O group present in the implementation.

A state, as reflected by a value returned by one of these functions, can change at any time. It is not limited to changing when the main thread calls kdWaitEvent or kdPumpEvents.

If *buffer* does not point to a writable 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.

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.

21.3.2. 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.

`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. This includes the case of cutting the range to 0 size when the specified index itself is not an input of the applicable type or is not a valid index in an I/O group present in the implementation.

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.

Where an application attempts to use both this API and another API outside of OpenKODE Core to control the same physical output, the effects on the output are undefined.

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.

21.4. I/O groups and items

An *I/O item* is a state, an input or an output. Each I/O item is referenced, in getting and setting functions and in events, using an index. Indexes are in the range 0..`KDINT_MAX`.

Each possible index has one of the types in the following list:

- empty;
- KDint32 state;
- KDint64 state;
- KDfloat32 state;
- 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. An I/O group specified below may or may not be available. If not available, all the indexes in it have empty type, which means that attempting to retrieve a state value 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).

Each I/O group (of index `idx`) has as its first item (also of index `idx`) a KDint32 or KDint64 state value, giving the *availability bitmap*. This is an integer where each item in the group has a bit, starting at bit 0 for the lowest-indexed item, with the bit set to 1 if the item is available. For this purpose, the "lowest-indexed item" is usually at index `idx`+1, so the availability bitmap itself is not included. Some I/O groups may have further state value items also not included in the availability bitmap.

An I/O group may be permanently unavailable, meaning that its I/O indexes are always empty. In this case, attempting to get a state or set an output in the group always returns 0 to indicate that no state values or outputs of the requested type could be found at the specified index.

An I/O group may be temporarily unavailable, meaning that it is possible for it to be available, but the platform is currently configured such that the inputs/outputs being exposed cannot be used. For example, the phone keypad I/O group might become temporarily unavailable when the keypad is pushed into the body of the handset. In this case, the items in the group are all available, but the availability bitmap has a value of 0 to indicate that no items are available. They may become available again (for example if the keypad is pulled out), and in that case the availability bitmap changes value (and thus generates an event) to indicate this.

When an I/O group is available, some of its items may be unavailable. Any such item is marked as optional in the description of its I/O group. An unavailable item may be only temporarily unavailable; when it changes between available and unavailable, the availability bitmap changes value and thus generates an event.

If an I/O group is not permanently unavailable but a particular item within it is unavailable (either by the I/O group being temporarily unavailable, or by the item it being optional and in fact not available), then the item has the same type as it would do if it were available, but:

- if it is a state, it never generates an event, and it has an undefined value, as seen when retrieving it;
- if it is an input, it never generates an event, and it has an undefined value, as seen when including its value in another input's event;
- if it is an output, setting it has no effect.

It is recommended (but not mandated) that an implementation should map inputs into multiple I/O groups where that makes sense for compatibility reasons. For example, a handset with a direction control which is analog would map that as a stick, but it is also recommended to map it as a D-pad, and as the direction keys in the game keys I/O group, such that an application which uses the game keys or a D-pad but does not understand a joystick will still work.

Certain I/O groups are defined to be part of a *game controller*, and, if the platform supports multiple game controllers, appear multiple times in the I/O index space at a stride of `KD_IO_CONTROLLER_STRIDE` (64).

21.4.1. KD_IOGROUP_GAMEKEYS

I/O group for game keys.

Synopsis

```
#define KD_IOGROUP_GAMEKEYS 0x1000
#define KD_STATE_GAMEKEYS_AVAILABILITY    (KD_IOGROUP_GAMEKEYS + 0)
#define KD_INPUT_GAMEKEYS_UP              (KD_IOGROUP_GAMEKEYS + 1)
#define KD_INPUT_GAMEKEYS_LEFT            (KD_IOGROUP_GAMEKEYS + 2)
#define KD_INPUT_GAMEKEYS_RIGHT           (KD_IOGROUP_GAMEKEYS + 3)
#define KD_INPUT_GAMEKEYS_DOWN            (KD_IOGROUP_GAMEKEYS + 4)
#define KD_INPUT_GAMEKEYS_FIRE            (KD_IOGROUP_GAMEKEYS + 5)
#define KD_INPUT_GAMEKEYS_A               (KD_IOGROUP_GAMEKEYS + 6)
#define KD_INPUT_GAMEKEYS_B               (KD_IOGROUP_GAMEKEYS + 7)
#define KD_INPUT_GAMEKEYS_C               (KD_IOGROUP_GAMEKEYS + 8)
#define KD_INPUT_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. 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 or in a D-pad. Where the platform is able to detect reorientation, the game keys, in particular the direction keys, are automatically remapped.

If this I/O group is available, then `KD_IOGROUP_GAMEKEYSNC` is also available, and the two groups represent the same keys.

I/O items

index	**type**	**range**	**usage**
`KD_STATE_GAMEKEYS_AVAILABILITY`	mandatory KDint32 state		availability bitmap
`KD_INPUT_GAMEKEYS_UP`	mandatory KDint32 input	0..1	up button
`KD_INPUT_GAMEKEYS_LEFT`	mandatory KDint32 input	0..1	left button
`KD_INPUT_GAMEKEYS_RIGHT`	mandatory KDint32 input	0..1	right button
`KD_INPUT_GAMEKEYS_DOWN`	mandatory KDint32 input	0..1	down button
`KD_INPUT_GAMEKEYS_FIRE`	mandatory KDint32 input	0..1	fire button
`KD_INPUT_GAMEKEYS_A`	optional KDint32 input	0..1	game_a button
`KD_INPUT_GAMEKEYS_B`	optional KDint32 input	0..1	game_b button
`KD_INPUT_GAMEKEYS_C`	optional KDint32 input	0..1	game_c button
`KD_INPUT_GAMEKEYS_D`	optional KDint32 input	0..1	game_d button

`KD_STATE_GAMEKEYS_AVAILABILITY` is a state whose value indicates using a bitmap which inputs are available. Bit n represents input `KD_INPUT_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 0 if the group is temporarily unavailable, 31 if the minimum set (direction keys plus fire) is available, 511 if all keys are available, and other values if some of the game A, B, C or D keys are unavailable.

The value of this state may change if the user takes some action which causes reconfiguration, for example reorienting the handset. Such a change causes an event.

It is recommended that a handset with some other means of controlling direction, such as an analog control or a D-pad, should also map it to the direction keys in the game keys I/O group, if there is no other obvious mapping for the game keys direction keys. This allows an application which does not understand a joystick or D-pad but does use the game keys to work on the handset.

The `KD_IOGROUP_GAMEKEYS` group only appears once in the I/O index space, even if the platform supports multiple game controllers. This is because `KD_IOGROUP_GAMEKEYS` is considered a stopgap measure to allow an application to work unchanged on a variety of handsets with or without specific gaming form factor. An application that wants to take advantage of a proper D-pad or joystick should use the appropriate I/O groups.

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 accurately reflect the keys pressed. Similarly for any two keys in such a three key combination.

21.4.2. KD_IOGROUP_GAMEKEYSNC

I/O group for game keys, no chording.

Synopsis

```
#define KD_IOGROUP_GAMEKEYSNC 0x1100
#define KD_STATE_GAMEKEYSNC_AVAILABILITY  (KD_IOGROUP_GAMEKEYSNC + 0)
#define KD_INPUT_GAMEKEYSNC_UP            (KD_IOGROUP_GAMEKEYSNC + 1)
#define KD_INPUT_GAMEKEYSNC_LEFT          (KD_IOGROUP_GAMEKEYSNC + 2)
#define KD_INPUT_GAMEKEYSNC_RIGHT         (KD_IOGROUP_GAMEKEYSNC + 3)
#define KD_INPUT_GAMEKEYSNC_DOWN          (KD_IOGROUP_GAMEKEYSNC + 4)
#define KD_INPUT_GAMEKEYSNC_FIRE          (KD_IOGROUP_GAMEKEYSNC + 5)
#define KD_INPUT_GAMEKEYSNC_A             (KD_IOGROUP_GAMEKEYSNC + 6)
#define KD_INPUT_GAMEKEYSNC_B             (KD_IOGROUP_GAMEKEYSNC + 7)
#define KD_INPUT_GAMEKEYSNC_C             (KD_IOGROUP_GAMEKEYSNC + 8)
#define KD_INPUT_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 available, both represent the same group of keys.

Rationale

The chording requirements of `KD_IOGROUP_GAMEKEYS` are intended to meet the playability requirements of many highly interactive applications. 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.

An application which requires the chording should use `KD_IOGROUP_GAMEKEYS`, which stops it being used on handsets that do not match the chording requirements.

An 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 usable.

21.4.3. KD_IOGROUP_PHONEKEYPAD

I/O group for phone keypad.

Synopsis

```
#define KD_IOGROUP_PHONEKEYPAD 0x2000
#define KD_STATE_PHONEKEYPAD_AVAILABILITY  (KD_IOGROUP_PHONEKEYPAD + 0)
#define KD_INPUT_PHONEKEYPAD_0             (KD_IOGROUP_PHONEKEYPAD + 1)
#define KD_INPUT_PHONEKEYPAD_1             (KD_IOGROUP_PHONEKEYPAD + 2)
#define KD_INPUT_PHONEKEYPAD_2             (KD_IOGROUP_PHONEKEYPAD + 3)
#define KD_INPUT_PHONEKEYPAD_3             (KD_IOGROUP_PHONEKEYPAD + 4)
#define KD_INPUT_PHONEKEYPAD_4             (KD_IOGROUP_PHONEKEYPAD + 5)
#define KD_INPUT_PHONEKEYPAD_5             (KD_IOGROUP_PHONEKEYPAD + 6)
#define KD_INPUT_PHONEKEYPAD_6             (KD_IOGROUP_PHONEKEYPAD + 7)
#define KD_INPUT_PHONEKEYPAD_7             (KD_IOGROUP_PHONEKEYPAD + 8)
#define KD_INPUT_PHONEKEYPAD_8             (KD_IOGROUP_PHONEKEYPAD + 9)
#define KD_INPUT_PHONEKEYPAD_9             (KD_IOGROUP_PHONEKEYPAD + 10)
#define KD_INPUT_PHONEKEYPAD_STAR          (KD_IOGROUP_PHONEKEYPAD + 11)
#define KD_INPUT_PHONEKEYPAD_HASH          (KD_IOGROUP_PHONEKEYPAD + 12)
#define KD_INPUT_PHONEKEYPAD_LEFTSOFT      (KD_IOGROUP_PHONEKEYPAD + 13)
#define KD_INPUT_PHONEKEYPAD_RIGHTSOFT     (KD_IOGROUP_PHONEKEYPAD + 14)
#define KD_STATE_PHONEKEYPAD_ORIENTATION   (KD_IOGROUP_PHONEKEYPAD + 15)
```

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. 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.

I/O items

index	type	range	usage
`KD_STATE_PHONEKEYPAD_AVAILABILITY`	mandatory KDint32 state		availability bitmap
`KD_INPUT_PHONEKEYPAD_0`	optional KDint32 input	0..1	0 key
`KD_INPUT_PHONEKEYPAD_1`	optional KDint32 input	0..1	1 key
`KD_INPUT_PHONEKEYPAD_2`	optional KDint32 input	0..1	2 key
`KD_INPUT_PHONEKEYPAD_3`	optional KDint32 input	0..1	3 key
`KD_INPUT_PHONEKEYPAD_4`	optional KDint32 input	0..1	4 key
`KD_INPUT_PHONEKEYPAD_5`	optional KDint32 input	0..1	5 key
`KD_INPUT_PHONEKEYPAD_6`	optional KDint32 input	0..1	6 key
`KD_INPUT_PHONEKEYPAD_7`	optional KDint32 input	0..1	7 key
`KD_INPUT_PHONEKEYPAD_8`	optional KDint32 input	0..1	8 key

index	type	range	usage
KD_INPUT_PHONEKEYPAD_9	optional KDint32 input	0..1	9 key
KD_INPUT_PHONEKEYPAD_STAR	optional KDint32 input	0..1	* key
KD_INPUT_PHONEKEYPAD_HASH	optional KDint32 input	0..1	# key
KD_INPUT_PHONEKEYPAD_LEFTSOFT	optional KDint32 input	0..1	left soft key
KD_INPUT_PHONEKEYPAD_RIGHTSOFT	optional KDint32 input	0..1	right soft key
KD_STATE_PHONEKEYPAD_ORIENTATION	mandatory KDint32 state	0..3	orientation

If the group is not permanently unavailable, the availability bitmap state value is present. The orientation state value is present if any of the inputs is present. The actual phone keypad keys (0-9,*,#) are either all absent or all present. The soft keys are optional. Thus it is possible for one or both of the soft keys to be present at a time when the 0-9,*,# keys are absent, for example when they have been slid away on a slider handset.

`KD_STATE_PHONEKEYPAD_AVAILABILITY` is a state whose value indicates using a bitmap which inputs are available. Bit n represents input `KD_INPUT_PHONEKEYPAD_0 + n`, set to 1 if the input is available and 0 if not, with unused bits set to 0. The bitmap does not have an entry for `KD_STATE_PHONEKEYPAD_ORIENTATION`.

Thus the value of the input is 0 if the group is temporarily unavailable, or 0x4fff if 0-9, *, # and orientation are available, 0x7fff if the two soft keys are additionally available, or 0x5fff or 0x6fff if only the left or right (respectively) softkey is additionally available, or 0x5000, 0x6000 or 0x7000 if the left, right or both softkeys (respectively) are available without the 0-9,*,# keys.

The availability bitmap state value 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, or sliding the keypad keys away. Such a change causes an event to be generated.

`KD_STATE_PHONEKEYPAD_ORIENTATION` is a state whose value indicates how many right angles counterclockwise the phone keypad is rotated from normal. The state is mandatory; an implementation that is not able to give this information always has a value of 0.

The keys in this group may also be mapped in the `KD_IOGROUP_GAMEKEYS` and `KD_IOGROUP_GAMEKEYSNC` groups, but are never mapped in any `KD_IOGROUP_DPAD` or `KD_IOGROUP_BUTTONS` group.

21.4.4. KD_IOGROUP_VIBRATE

I/O group for vibrate.

Synopsis

```
#define KD_IOGROUP_VIBRATE 0x3000
#define KD_STATE_VIBRATE_AVAILABILITY  (KD_IOGROUP_VIBRATE + 0)
#define KD_STATE_VIBRATE_MINFREQUENCY  (KD_IOGROUP_VIBRATE + 1)
#define KD_STATE_VIBRATE_MAXFREQUENCY  (KD_IOGROUP_VIBRATE + 2)
#define KD_OUTPUT_VIBRATE_VOLUME        (KD_IOGROUP_VIBRATE + 3)
#define KD_OUTPUT_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_STATE_VIBRATE_AVAILABILITY`	mandatory KDint32 state		availability bitmap
`KD_STATE_VIBRATE_MINFREQUENCY`	optional KDint32 state		frequency minimum in millihertz (constant)
`KD_STATE_VIBRATE_MAXFREQUENCY`	optional KDint32 state		frequency maximum in millihertz (constant)
`KD_OUTPUT_VIBRATE_VOLUME`	mandatory KDint32 output	0..1000	volume in permilles
`KD_OUTPUT_VIBRATE_FREQUENCY`	optional KDint32 output	see below	frequency in millihertz

Output `KD_OUTPUT_VIBRATE_VOLUME` sets the volume level in permilles (i.e. thousandths), and is mandatory. The initial value 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_OUTPUT_VIBRATE_FREQUENCY` sets the frequency in millihertz (e.g. 25000 represents 25Hz), and is optional. The range is determined by the constant state values `KD_STATE_VIBRATE_MINFREQUENCY` and `KD_STATE_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.

A change in one of the two above outputs is effective immediately (subject to the way the platform virtualizes the outputs in the presence of concurrent applications).

States `KD_STATE_VIBRATE_MINFREQUENCY` and `KD_STATE_VIBRATE_MAXFREQUENCY` have constant values which indicate the minimum and maximum (respectively) frequencies that the handset's vibrate implements. They are available if and only if output `KD_OUTPUT_VIBRATE_FREQUENCY` is available.

`KD_STATE_VIBRATE_AVAILABILITY` is a state whose constant value is a bitmap that indicates which I/O items are available, such that bit n is 1 if and only if I/O item index `KD_IOGROUP_VIBRATE_MINFREQUENCY + 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 0 if the group is temporarily unavailable, or 4 if it is not possible to set the frequency or 15 if it is possible.

21.4.5. KD_IOGROUP_POINTER

I/O group for pointer.

Synopsis

```
#define KD_IOGROUP_POINTER 0x4000
#define KD_STATE_POINTER_AVAILABILITY  (KD_IOGROUP_POINTER + 0)
#define KD_INPUT_POINTER_X             (KD_IOGROUP_POINTER + 1)
#define KD_INPUT_POINTER_Y             (KD_IOGROUP_POINTER + 2)
#define KD_INPUT_POINTER_SELECT        (KD_IOGROUP_POINTER + 3)
```

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 select input. 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 with multiple buttons and other mouse-specific features.

Where the platform is able to detect reorientation, the pointer input is automatically remapped.

Inputs and outputs

index	type	range	usage
`KD_STATE_POINTER_AVAILABILITY`	mandatory KDint32 state	7	availability bitmap
`KD_INPUT_POINTER_X`	mandatory KDint32 input		X coordinate
`KD_INPUT_POINTER_Y`	mandatory KDint32 input		Y coordinate
`KD_INPUT_POINTER_SELECT`	mandatory KDint32 input	0..1	select input

The X and Y coordinates use the top left of the window to which the event is being directed as the origin, and are mandatory inputs. The select input is also mandatory.

The select input has a value of 1 when selected (for example the touchscreen is being pressed) and 0 when released. Separate events are generated for press and release at the appropriate times, even if there is no call to `kdWaitEvent` or `kdPumpEvents` in between the press and release.

`KD_STATE_POINTER_AVAILABILITY` is a state whose constant value is a bitmap that indicates which I/O items are available, such that bit n is 1 if and only if I/O item index `KD_IOGROUP_POINTER_X + n` is available. Thus it has the value 0 if the group is temporarily unavailable, or 7 if it is available.

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`.

21.4.6. KD_IOGROUP_BACKLIGHT

I/O group for backlight.

Synopsis

```
#define KD_IOGROUP_BACKLIGHT 0x5000
#define KD_STATE_BACKLIGHT_AVAILABILITY (KD_IOGROUP_BACKLIGHT + 0)
#define KD_OUTPUT_BACKLIGHT_FORCE (KD_IOGROUP_BACKLIGHT + 1)
```

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_STATE_BACKLIGHT_AVAILABILITY`	mandatory KDint32 state	1	availability bitmap

index	type	range	usage
`KD_OUTPUT_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_OUTPUT_BACKLIGHT_FORCE` is 0.

`KD_STATE_BACKLIGHT_AVAILABILITY` is a state whose constant value is a bitmap that indicates which I/O items are available, such that bit n is 1 if and only if I/O item index `KD_IOGROUP_BACKLIGHT_FORCE + n` is available. It has the value 0 if the group is temporarily unavailable, or 1 when it is available.

21.4.7. KD_IOGROUP_JOGDIAL

I/O group for a jog dial.

Synopsis

```
#define KD_IOGROUP_JOGDIAL 0x6000
#define KD_STATE_JOGDIAL_AVAILABILITY  (KD_IOGROUP_JOGDIAL + 0)
#define KD_INPUT_JOGDIAL_UP            (KD_IOGROUP_JOGDIAL + 1)
#define KD_INPUT_JOGDIAL_LEFT          (KD_IOGROUP_JOGDIAL + 2)
#define KD_INPUT_JOGDIAL_RIGHT         (KD_IOGROUP_JOGDIAL + 3)
#define KD_INPUT_JOGDIAL_DOWN          (KD_IOGROUP_JOGDIAL + 4)
#define KD_INPUT_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.

For the up, down and (when present) left and right inputs, movement in a particular direction generates a `KD_EVENT_INPUT_JOG` event, which specifies how far in that direction the jogdial was moved. The select input generates a `KD_EVENT_INPUT` event.

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.

This I/O group can be a per-game-controller group: further groups with the same semantics may appear at `KD_IOGROUP_JOGDIAL + n * KD_IO_CONTROLLER_STRIDE` (where n is 1 for the second controller, up to 63 for the 64th controller, and `KD_IO_CONTROLLER_STRIDE` is 64).

I/O items

index	type	range	usage
`KD_STATE_JOGDIAL_AVAILABILITY`	mandatory KDint32 state		availability bitmap
`KD_INPUT_JOGDIAL_UP`	mandatory input	not supplied in event	event generated when dial clicked up
`KD_INPUT_JOGDIAL_LEFT`	optional input	not supplied in event	event generated when dial clicked left
`KD_INPUT_JOGDIAL_RIGHT`	optional input	not supplied in event	event generated when clicked right

index	type	range	usage
`KD_INPUT_JOGDIAL_DOWN`	mandatory input	not supplied in event	event generated when clicked down
`KD_INPUT_JOGDIAL_SELECT`	mandatory KDint32 input	0..1	whether dial is being pushed in (or selected in some other way

`KD_STATE_JOGDIAL_AVAILABILITY` is a state with constant value that indicates, using a bitmap, which inputs are available. Bit n represents input `KD_INPUT_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 state is 0 if the group is temporarily unavailable, 25 for a three-way jog dial, or 31 for a five-way jog dial.

21.4.8. KD_IOGROUP_STICK

I/O group for joystick.

Synopsis

```
#define KD_IOGROUP_STICK 0x7000
#define KD_STATE_STICK_AVAILABILITY    (KD_IOGROUP_STICK + 0)
#define KD_INPUT_STICK_X               (KD_IOGROUP_STICK + 1)
#define KD_INPUT_STICK_Y               (KD_IOGROUP_STICK + 2)
#define KD_INPUT_STICK_Z               (KD_IOGROUP_STICK + 3)
#define KD_INPUT_STICK_BUTTON          (KD_IOGROUP_STICK + 4)
#define KD_IO_STICK_STRIDE 8
```

Description

This I/O group defines a joystick, consisting of 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. In addition, there may be a button actually on the stick.

A particular game controller (where a handset is considered a single game controller) may have up to eight sticks, appearing at indexes `KD_IOGROUP_STICK +` *`m`* `* KD_IO_STICK_STRIDE` where *m*=0 for the first stick up to *m*=7 for the 8th stick.

This I/O group can be a per-game-controller [160] group: further groups with the same semantics may appear at `KD_IOGROUP_STICK +` *`n`* `* KD_IO_CONTROLLER_STRIDE` (where n is 1 for the second controller, up to 63 for the 64th controller). Each controller may have up to eight sticks, like the first controller.

Where the platform is able to detect reorientation, the stick input is automatically remapped as appropriate.

Inputs and outputs

index	type	range	usage
`KD_STATE_STICK_AVAILABILITY`	mandatory KDint32 state		availability bitmap
`KD_INPUT_STICK_X`	KDint32 input	-32768..+32767	X axis
`KD_INPUT_STICK_Y`	KDint32 input	-32768..+32767	Y axis
`KD_INPUT_STICK_Z`	KDint32 input	-32768..+32767	Z axis
`KD_INPUT_STICK_BUTTON`	KDint32 input	0..1	whether button is being pressed

If any stick axis input changes, the `KD_EVENT_INPUT_STICK` event is generated, rather than the normal `KD_EVENT_INPUT`.

`KD_STATE_STICK_AVAILABILITY` is a state with constant value that indicates, using a bitmap, which inputs are available. Bit *n* represents input `KD_INPUT_STICK_X + 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 0 if the group is temporarily unavailable, 3 for a two-axis stick with no button, 7 for a three-axis stick with no button, 11 for a two-axis stick with a button, or 15 for a three-axis stick with a button.

The inputs of such an analog stick may also be mapped as a digital D-pad.

21.4.9. KD_IOGROUP_DPAD

I/O group for D-pad.

Synopsis

```
#define KD_IOGROUP_DPAD 0x8000
#define KD_STATE_DPAD_AVAILABILITY     (KD_IOGROUP_DPAD + 0)
#define KD_STATE_DPAD_COPY             (KD_IOGROUP_DPAD + 1)
#define KD_INPUT_DPAD_UP               (KD_IOGROUP_DPAD + 2)
#define KD_INPUT_DPAD_LEFT             (KD_IOGROUP_DPAD + 3)
#define KD_INPUT_DPAD_RIGHT            (KD_IOGROUP_DPAD + 4)
#define KD_INPUT_DPAD_DOWN             (KD_IOGROUP_DPAD + 5)
#define KD_INPUT_DPAD_SELECT           (KD_IOGROUP_DPAD + 6)
#define KD_IO_DPAD_STRIDE 8
```

Description

This I/O group defines a D-pad, consisting of four direction buttons arranged in a diamond pattern, optionally with a “select” button in the middle.

A particular game controller (where a handset is considered a single game controller) may have up to eight D-pads, appearing at indexes `KD_IOGROUP_DPAD + m* KD_IO_DPAD_STRIDE` where *m*=0 for the first D-pad up to *m*=7 for the 8th D-pad.

This I/O group can be a per-game-controller [160] group: further groups with the same semantics may appear at `KD_IOGROUP_DPAD + n * KD_IO_CONTROLLER_STRIDE` (where n is 1 for the second controller, up to 63 for the 64th controller). Each controller may have up to eight D-pads, like the first controller.

Where the platform is able to detect reorientation, the D-pad input is automatically remapped as appropriate.

Inputs and outputs

index	type	range	usage
`KD_STATE_DPAD_AVAILABILITY`	mandatory KDint32 state		availability bitmap
`KD_STATE_DPAD_COPY`	mandatory KDint32 state		see below
`KD_INPUT_DPAD_UP`	KDint32 input	0 or 1	up
`KD_INPUT_DPAD_LEFT`	KDint32 input	0 or 1	left
`KD_INPUT_DPAD_RIGHT`	KDint32 input	0 or 1	right
`KD_INPUT_DPAD_DOWN`	KDint32 input	0 or 1	down
`KD_INPUT_DPAD_SELECT`	KDint32 input	0 or 1	select button

KD_STATE_DPAD_AVAILABILITY is a state with constant value that indicates, using a bitmap, which inputs are available. Bit *n* represents input KD_INPUT_DPAD_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 0 if the group is temporarily unavailable, 15 for a D-pad with no select button, or 31 for a D-pad with a select button.

KD_STATE_DPAD_COPY is a state with a constant value that indicates whether this D-pad is also mapped as an analog stick input. The value is -1 if it is not so mapped, or the I/O group index number of the stick if it is.

The inputs of a D-pad on the first controller may also be mapped within the game keys. However no input in a D-pad is also a phone keypad key (KD_IOGROUP_PHONEKEYPAD) or a button (KD_IOGROUP_BUTTONS). No part of a D-pad maps to a D-pad input on more than one controller.

Rationale

The KD_STATE_DPAD_COPY state allows an application to tell whether a controller has (for example) a separate stick and D-pad, or whether the D-pad is just a remapping of the stick.

21.4.10. KD_IOGROUP_BUTTONS

I/O group for buttons associated with joystick or D-pad.

Synopsis

```
#define KD_IOGROUP_BUTTONS 0x9000
#define KD_STATE_BUTTONS_AVAILABILITY  (KD_IOGROUP_BUTTONS + 0)
#define KD_INPUT_BUTTONS_0             (KD_IOGROUP_BUTTONS + 1)
```

Description

This I/O group defines buttons that are typically associated with a joystick and/or D-pad (other than a D-pad's select button).

This I/O group can be a per-game-controller [160] group: further groups with the same semantics may appear at KD_IOGROUP_BUTTONS + *n* * KD_IO_CONTROLLER_STRIDE (where n is 1 for the second controller, up to 63 for the 64th controller). Each controller may have up to 63 buttons.

Inputs and outputs

index	type	range	usage
KD_STATE_BUTTONS_AVAILABILITY	mandatory KDint64 state		availability bitmap
KD_INPUT_BUTTONS_0 + *m*	KDint32 input	0 or 1	button *m*

KD_STATE_BUTTONS_AVAILABILITY is a **64-bit** (unlike other groups' availability bitmaps) state with constant value that indicates, using a bitmap, which inputs are available. Bit *n* represents input KD_INPUT_BUTTONS_0 + *n*, set to 1 if the input is available and 0 if not, with unused bits set to 0. The value is 0 if the group is temporarily unavailable.

Button 0 is conventionally the fire button. No other semantics are defined.

The buttons on the first controller may also be mapped within the game keys. However no button input here is also a phone keypad key (KD_IOGROUP_PHONEKEYPAD) or a D-pad input (KD_IOGROUP_DPAD). No button maps to a button input on more than one controller.

21.4.11. 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.

22. Windowing

22.1. Introduction

The windowing API provides an abstraction for managing windows and performing simple operations on them.

The representation of a window on the display is platform dependent. An implementation may support a single window that spans the entire area of the display, it may support a single window that covers the display only partially or it may support multiple windows that may or not may not be visible on the display at the same time.

The OpenKODE windowing API allows the application to set window properties to request a window to be displayed in a particular fashion. The implementation may or may not support setting any of the window properties and it may change any of the supported window properties at any time. The application is notified about changes in supported window properties via the OpenKODE Core event system. Additionally, setting a window property may be supported either only before realizing a window as a visible entity on the display or at any time.

22.1.1. Not thread safe

For any particular window handle, using the same handle in OpenKODE function calls in multiple threads at the same time results in undefined behavior.

22.2. Presence of windowing API

An OpenKODE implementation that does not allow creation in EGL of a window surface does not include OpenKODE Core's windowing API, and none of the rest of this Windowing section applies.

An implementation that does include OpenKODE Core's windowing API #defines `KD_WINDOW_SUPPORTED`. Such an implementation is required to support at least one window per application.

22.3. Types

`KDWindow`	An opaque struct used to represent a window. A pointer to this type is used as a handle to a window.

22.4. Functions

22.4.1. kdCreateWindow

Create a window.

Synopsis

```
KDWindow *kdCreateWindow(EGLDisplay display, EGLConfig config, void
*eventuserptr);
```

Description

This function creates a window object.

The window object does not represent a visible display entity until it has been realized by a call to `kdRealizeWindow`.

Supported window properties are initially at their default values, as stated by the entry in the window properties table. The application can request changes to the initial representation of the window by setting the values of the appropriate window properties before realizing the window.

If the application has already created the maximum number of windows supported by the implementation, 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`. The *`config`* parameter is the EGL config which the application intends to use with the new window.

`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 handle, and any EGL resources associated with it, become invalid as soon as the thread that created the window exits; using such an invalidated handle (or any associated EGL handle) in an OpenKODE Core function results in undefined behavior. However it is undefined whether resources associated with the window, including any on-screen display, are actually freed then or some later point up to and including application exit.

If *`display`* is not an EGL display handle, or *`config`* is not a valid EGL config, then undefined behavior results.

Return value

On success, the function returns the KDWindow * pointer for the newly created window object, which is initially in an unrealized state. The window supports at least the *`config`* supplied to it. 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_EPERM`	Attempt to create a window when the application already has a the maximum number of supported windows.

22.4.2. kdDestroyWindow

Destroy a window.

Synopsis

```
KDint kdDestroyWindow(KDWindow *window);
```

Description

This function unrealizes and destroys *`window`* and all OpenKODE Core resources related to it. EGL resources associated with the window must be freed before calling this function.

All pending events for the window (including input events when the window has input focus) are removed from the queue.

If this function call overlaps with a function call in any other thread using the same *`window`*, then undefined behavior results.

Otherwise, the function must be called in the same thread that created the window with `kdCreateWindow`, otherwise the function fails with an error.

If `window` is not a window, or has already been destroyed, or has EGL resources associated with it, 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_EINVAL`	Called from thread other than the one that created the window.

22.4.3. kdSetWindowPropertybv, kdSetWindowPropertyiv, kdSetWindowPropertycv

Set a window property to request a change in the on-screen representation of the window.

Synopsis

```
KDint kdSetWindowPropertybv(KDWindow *window, KDint pname, const KDboolean
*param);

KDint kdSetWindowPropertyiv(KDWindow *window, KDint pname, const KDint32 *param);

KDint kdSetWindowPropertycv(KDWindow *window, KDint pname, const KDchar *param);
```

Description

For the specified `window`, these functions attempt to change the value of the window property `pname` to the new value given in `param`. Each window property is specified to have a value that is an array of type KDboolean, KDint or KDchar, and is set using `kdSetWindowPropertybv`, `kdSetWindowPropertyiv` or `kdSetWindowPropertycv` respectively. Refer to the window properties section for the storage types of OpenKODE Core window properties.

The functions `kdSetWindowPropertybv` and `kdSetWindowPropertyiv` read as many elements from the `param` array as defined in the specification of the corresponding property. In the case of `kdSetWindowPropertycv` where the property is defined to be a null-terminated string, elements are read up to the null termination character.

Setting a window property of a window in an unrealized state affects the initial representation of that window at the time the window becomes realized. Setting a window property of a window in a realized state changes the representation of the window immediately.

An OpenKODE Core implementation may support setting any number of the window properties listed in this specification, but the specification does not mandate support for setting any of the window properties. Additionally, an implementation may only support setting a given window property when the window is in an unrealized state. A portable application should not rely on the ability to set window properties; the window property mechanism allows the application to instruct the implementation about how its windows should be represented but the application should be written to function even when setting none of the requested properties can be supported by the system.

When setting a property of a realized window, a `KD_EVENT_WINDOWPROPERTY_CHANGE` event (or other event if applicable to the property in question) is generated with the specified property identifier if the property value was changed.

In certain implementation-defined cases, setting one property may cause other properties to change as well. For a realized window, these other property changes also cause an event each. For an unrealized window, the dependency of one property on another is not implemented until `kdRealizeWindow` is called.

If this function call overlaps with a function call in any other thread using the same *`window`*, then undefined behavior results.

If *`window`* is not a window, or has been destroyed, then undefined behavior results. If *`param`* is not a readable array of KDboolean or KDint of the length specified for the property (respectively for `kdSetWindowPropertybv` or `kdSetWindowPropertyiv`), or a readable array of KDchar forming a null-terminated UTF-8 string (for `kdSetWindowPropertycv`), 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.
`KD_EINVAL`	Incompatible storage type or unsupported value for property *`pname`*.
`KD_EOPNOTSUPP`	The implementation does not support setting the window property *`pname`*.
`KD_EPERM`	The implementation does not support setting the window property *`pname`* for an already realized window.

22.4.4. kdGetWindowPropertybv, kdGetWindowPropertyiv, kdGetWindowPropertycv

Get the current value of a window property.

Synopsis

```
KDint kdGetWindowPropertybv(KDWindow *window, KDint pname, KDboolean *param);

KDint kdGetWindowPropertyiv(KDWindow *window, KDint pname, KDint32 *param);

KDint kdGetWindowPropertycv(KDWindow *window, KDint pname, KDchar *param, KDsize
*size);
```

Description

For the specified *`window`*, these functions get the current value of a window property. The value is written into the location pointed to by *`param`*. Each window property is specified to have a value that is an array of type KDboolean, KDint or KDchar, and is read using `kdGetWindowPropertybv`, `kdGetWindowPropertyiv` or `kdGetWindowPropertycv` respectively. Refer to the window properties section for the storage types of OpenKODE Core window properties.

It is only possible to get a window property for a realized window.

`kdGetWindowPropertycv` gets a property value that is a null-terminated UTF-8 string. *`size`* points to a KDsize location used as both an input and an output to specify the size. On input, `*`*`size`* is the size of the buffer pointed to by *`param`*; the function writes as many whole UTF-8 characters followed by a null termination as will fit in that size, and stores the number of bytes written into the location pointed to by *`size`*. On entry, if `*`*`size`* is zero, then no data is copied, and the number of bytes (including null termination) that would be written if the buffer were big enough is stored into the location pointed to by *`size`*.

An implementation is required to support getting the value of all specified window properties. Where the implementation attaches no meaning to a particular property, it returns the default value.

If this function call overlaps with a function call in any other thread using the same *`window`*, then undefined behavior results.

If *`window`* is not a window, or has been destroyed, then undefined behavior results. If *`param`* is not a writable array of KDboolean or KDint of the length specified for the property (respectively for `kdGetWindowPropertybv` or `kdGetWindowPropertyiv`), or a writable array of at least *`size`* KDchar elements (for `kdSetWindowPropertycv`), 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.
`KD_EINVAL`	Incompatible storage type for property *`pname`*.
`KD_EPERM`	Window not realized.
`KD_EOPNOTSUPP`	The implementation does not support the window property *`pname`*.

22.4.5. kdRealizeWindow

Realize the window as a displayable entity and get the native window handle for passing to EGL.

Synopsis

```
KDint kdRealizeWindow(KDWindow *window, EGLNativeWindowType *nativewindow);
```

Description

For the specified *`window`*, this function realizes the window and stores the native window handle (of type EGLNativeWindowType, as required by the EGL function `eglCreateWindowSurface`) into the location pointed to by *`nativewindow`*.

The implementation is expected to realize the window according to the current values of window properties, as set by the application, or using the default values if not set by the application. The implementation may, however, change the values of window properties in the process of realizing a window. This includes the case where one property has been set, and this causes another property to change in an implementation-defined way. When any such changes happen, a `KD_EVENT_WINDOWPROPERTY_CHANGE` event (or other event as applicable to the property) is generated with the property identifier of the changed property.

The net result is that an application can assume that any property has the value it set it to before the `kdRealizeWindow`, or failing that the default, unless it receives an event to indicate otherwise.

If this function call overlaps with a function call in any other thread using the same *`window`*, then undefined behavior results.

If *`window`* is not a window, or it has been destroyed, or *`nativewindow`* does not point to a writable location of type EGLNativeWindowType, 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_ENOMEM`	Out of memory or other resource.
`KD_EPERM`	*`window`* is already realized.

22.5. Window properties

The way that an OpenKODE application communicates with the platform windowing system about how windows are represented is the window property mechanism. OpenKODE Core specifies a set of core window properties specifically targeting a platform supporting only one visible full desktop client area window, but also meaningful on platforms where an application can own multiple windows displayed at the same time.

A property may change, yielding a `KD_EVENT_WINDOWPROPERTY_CHANGE` event, by the application calling the appropriate `kdSetWindowProperty` function, or by the application calling `kdSetWindowProperty` on some other property, or by from some external influence, such as the user giving a different application or window the focus.

Properties relating to the representation of multiple windows are intentionally not exposed by OpenKODE Core, since different window managers handle such properties in a variety of ways. It is expected that windowing system specific properties will be exposed in OpenKODE Core extensions.

Each window property has an associated storage type that dictates the `kdSetWindowProperty` and `kdGetWindowProperty` variant to be used with the property, and a default value.

Unless otherwise specified, a property generates a `KD_EVENT_WINDOWPROPERTY_CHANGE` event when the property value changes. Where a property generates a different event, this is specified with the property.

22.5.1. KD_WINDOWPROPERTY_SIZE

Window client area width and height.

Synopsis

```
#define KD_WINDOWPROPERTY_SIZE 66
```

Description

This window property describes the 2D dimensions of the window client area. The property value is an array of two KDint32s, and is thus set with `kdSetWindowPropertyiv` and read with `kdGetWindowPropertyiv`. The first value in the array is the width of the window and the second value is the height.

The following are implementation dependent:

- the default value;
- whether it is possible to set the property;

- if it is possible to set the property, whether and how the implementation modifies a value that is not supported to one that is.

22.5.2. KD_WINDOWPROPERTY_VISIBILITY

Window visibility status.

Synopsis

```
#define KD_WINDOWPROPERTY_VISIBILITY 67
```

Description

This window property indicates whether the window is visible. It is an array of a single KDboolean, and is thus set with `kdSetWindowPropertybv` and read with `kdGetWindowPropertybv`. The boolean has the value `KD_TRUE` if the window is visible, or `KD_FALSE` if is not, and the default is `KD_TRUE`.

It is implementation defined whether it is possible to set this property. If it is possible, attempting to set a value other than `KD_TRUE` or `KD_FALSE` causes `kdSetWindowPropertybv` to fail with a `KD_EINVAL` error.

Link with focus property

Whether setting one property also causes another to change is implementation defined. However it is likely that, on many implementations, setting visibility to 0 also causes focus to be set to 0.

22.5.3. KD_WINDOWPROPERTY_FOCUS

Window input focus status.

Synopsis

```
#define KD_WINDOWPROPERTY_FOCUS 68
```

Description

This window property indicates whether the window has input focus. It is an array of a single KDboolean, and is thus set with `kdSetWindowPropertybv` and read with `kdGetWindowPropertybv`. The boolean has the value `KD_TRUE` if the window has focus, or `KD_FALSE` if does not, and the default is `KD_TRUE`.

It is implementation defined whether it is possible to set this property. If it is possible, attempting to set a value other than `KD_TRUE` or `KD_FALSE` causes `kdSetWindowPropertybv` to fail with a `KD_EINVAL` error.

A change of this property generates a `KD_EVENT_WINDOW_FOCUS` event.

22.5.4. KD_WINDOWPROPERTY_CAPTION

Window caption.

Synopsis

```
#define KD_WINDOWPROPERTY_CAPTION 69
```

Description

This window property indicates the caption displayed in association with the window. It is a null-terminated UTF-8 string, and is thus set with `kdSetWindowPropertycv` and read with `kdGetWindowPropertycv`. The default value is the empty string.

It is implementation defined whether it is possible to set this property.

22.6. Events

22.6.1. KD_EVENT_WINDOW_CLOSE

Event to request to close window.

Synopsis

```
#define KD_EVENT_WINDOW_CLOSE 44
```

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).

22.6.2. KD_EVENT_WINDOWPROPERTY_CHANGE

Notification about realized window property change.

Synopsis

```
#define KD_EVENT_WINDOWPROPERTY_CHANGE 47
```

Description

This event type signals that a property of a realized window being used by the application has changed. The change might have been triggered by the user setting the value of a property, the window being realized or by an external change in the window environment.

Some window properties have their own event type; such a property does not generate this event.

The event's *`userptr`* field is set to the *`eventuserptr`* value for the window of which a property has changed. This value was supplied by the application when the window was created with `kdCreateWindow`.

The event data is in `event->data.windowproperty` element of the event's data union, which has the following type:

```
typedef struct KDEventWindowProperty {
    KDint32 pname;
} KDEventWindowProperty;
```

Upon receiving this event, the application can query the new value of the window property by with the appropriate`kdGetWindowProperty` function.

`KD_EVENT_WINDOWPROPERTY_CHANGE` events are merged: if an event is generated by OpenKODE Core when another one is already in the queue that was generated by OpenKODE Core for the same window with the same property identifier, then the earlier one is removed.

22.6.3. KD_EVENT_WINDOW_FOCUS

Event for change of window's focus state.

Synopsis

```
#define KD_EVENT_WINDOW_FOCUS 60
```

Description

This event type signals that the `KD_WINDOWPROPERTY_FOCUS` property of a realized window being used by the application has changed. The change might have been triggered by the user setting the value of a property, the window being realized or by an external change in the window environment.

The focus property has its own event type because the application may need to know the sequence of focus changes relative to input events, therefore the events are not merged, and the event gives the new focus state at the time the event was posted.

The event's *`userptr`* field is set to the *`eventuserptr`* value for the window of which a property has changed. This value was supplied by the application when the window was created with `kdCreateWindow`.

The event data is in `event->data.windowfocus` element of the event's data union, which has the following type:

```
typedef struct KDEventWindowFocus {
    KDint32 focusstate;
} KDEventWindowFocus;
```

The *`focusstate`* field is 0 if the window lost input focus, or 1 if it gained focus.

22.6.4. KD_EVENT_WINDOW_REDRAW

Event to notify need to redraw the window.

Synopsis

```
#define KD_EVENT_WINDOW_REDRAW 59
```

Description

This event type is generated by OpenKODE Core (typically as the result of a request from the underlying OS) to signal that the some or all of the window is invalid due to being uncovered and therefore should be redrawn to restore consistency of display appearance.

Events of this type are merged: when posting such an event, any earlier event of this type for the same window is removed from the queue.

Rationale

There are several cases of how an application might need to respond to this event:

- When the application is rendering to the back buffer (as mandated by both OpenGL ES and EGL's lock surface extension, and as usually done with OpenVG), and the surface has a `EGL_SWAP_BEHAVIOR` value of `EGL_BUFFER_PRESERVED`, then a call to `eglSwapBuffers` restores the visible window.
- When the application renders the whole surface frequently (for example at least several times a second), no action is required since the window will get updated anyway at the next render cycle.
- Otherwise, the application is not rendering the whole surface frequently and is rendering to the front buffer, or to the back buffer of a surface where `EGL_SWAP_BEHAVIOR` has a value of `EGL_BUFFER_DESTROYED`, so it needs to act on a redraw event by rerendering the whole frame.

No information is provided about which part(s) of the window surface need redrawing, so the application must assume all of it. This may be addressed in a future version of OpenKODE Core, although it is only of use in the OpenVG rendering to front buffer case.

23. Assertions and logging

23.1. 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.

23.2. Functions

23.2.1. 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.

23.2.2. 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.

23.2.3. 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 1.0.2

OpenKODE 1.0.2 was released on November 6th 2008. It is a minor update to OpenKODE 1.0.1; all changes are clarifications to the specification such that an application using specified OpenKODE 1.0.1 features continues to work.

A.1.1. Revisions

Revision 1, 2009-02-23

No substantive changes; this revision was released simply to add an anchor for the Khronos *Implementers Guidelines* to link to.

Revision 0, 2008-11-06

Changes since OpenKODE 1.0.1 revision 0:

- Clarified that /data and /tmp file areas are writable.
- kdStrtof: Clarified that it is undefined whether a number that could be represented as a denormal in the return type counts as an underflow or not.
- Added descriptive note about clashes between kd.h and platform header files, such as s_addr on Windows.
- Clarified that the D-pad and button groups cannot overlap, and the button and phone keypad groups cannot overlap, and for each of D-pad and buttons no input can be mapped to more than one controller.
- A file opened in `"a"` mode has a completely undefined file position (but a write always happens at the end of the file). A file opened in `"a+"` mode has its file position moved to the end by a write, but its initial position on file open is undefined.
- kdFread, kdGetc, kdFgets: Clarified that if the file is open only for writing, then it is undefined whether the read succeeds, fails with an error, or appears to succeed.
- kdFwrite, kdPutc: Clarified that if the file is open only for reading, then it is undefined whether the write succeeds, fails with an error, or appears to succeed.
- kdGetFree: Clarified that KD_EACCES is given for / or /removable, and KD_ENOSYS is given if the information is not available elsewhere.
- kdFtostr: Fixed the explanation of how to calculate the “correct” result: the range to scale to before rounding to integer is [1e8,1e9), not [1e9,1e10).
- kdStrchr: Corrected to state that the function casts ch, the character to match, to a KDchar before matching. The conformance tests have always assumed this behavior.

A.2. OpenKODE 1.0.1

OpenKODE 1.0.1 was released on June 16th 2008. It is a minor update to OpenKODE 1.0; most changes are clarifications to the specification such that an application using specified OpenKODE 1.0 features continues to work.

A.2.1. Revisions

Revision 0, 2008-06-16

Changes since OpenKODE 1.0 revision 1:

- KD_EVENT_NAME_LOOKUP_COMPLETE: An implementation that cannot distinguish the error conditions for KD_ENO_DATA, KD_ENO_RECOVERY, KD_ETRY_AGAIN and KD_EHOST_NOT_FOUND is allowed to use KD_EHOST_NOT_FOUND for all such errors.
- kdSocketSend/kdSocketSendTo: The error KD_EPIPE has been removed, since the circumstance it covered could only occur if shutdown is called on the socket, and there is no OpenKODE Core equivalent of shutdown.
- kdSocketGetName: Noted that the error KD_EOPNOTSUPP does not occur for TCP and UDP sockets, and is a placeholder for future socket types.
- kdThreadJoin and kdThreadDetach: The error KD_ENOSYS (threading not supported) has been removed; it was unnecessary since attempting to join or detach the main thread was already stated to cause undefined behavior.
- kdRmdir: Stated that it is undefined whether the directory not being empty causes KD_EEXIST or KD_EBUSY.
- kdFopen: Clarified that it is undefined whether an invalid mode string gives KD_EINVAL.
- kdSocketSendTo: Noted that there may be an implementation-defined limit on how much data can be sent at once. For UDP, this is specified to be at least 1472.
- kdFflush: A descriptive note has been added regarding buffering and flushing.
- kdPutc now takes the byte to write as a KDint rather than a KDchar, to match putc in C and POSIX.
- KD_IOGROUP_PHONEKEYPAD now allows the soft keys to be present without the 0-9,*,# keys, to allow for a slider handset where the keypad can be slid away.
- kdSocketConnect no longer mandates the separate detection of the errors `KD_ECONNREFUSED`, `KD_ECONNRESET`, `KD_EHOSTUNREACH` or `KD_ETIMEDOUT`.
- kdSocketRecv and kdSocketRecvFrom no longer mandate the separate detection of the error `KD_ETIMEDOUT`.
- kdThreadDetach: Behavior is now undefined when attempting to detach the main thread.
- It is now defined that “spurious” KD_EVENT_SOCKET_READABLE and KD_EVENT_SOCKET_WRITABLE events may be generated for a bound connectionless socket or a connected connection-based socket; it is implementation defined whether such spurious events are generated.
- kdStrtof: Removed support for hexadecimal floating point. Tightened input requirements for infinity and NaN. kdFtostr: Tightened specification of results for infinity and NaN.
- Clarified that, since kdIsNan is specified as a macro, it is undefined how many times its argument is evaluated.
- Stated that it is undefined whether a filename with a trailing period refers to the same file as the same filename without the trailing period.
- Clarified that it is undefined at what point a network connection is established, but recommended that it should be no earlier than name lookup or socket connect if establishing the connection spends the user’s money.

- KD_EVENT_WINDOW_REDRAW: Clarified that the application should redraw the window to restore consistency of display appearance, rather than for any other reason (e.g. to avoid undefined behavior). Clarified that redraw events should still be honored even when paused.
- kdStrtoul: Clarified that a minus sign negates, but does not negate the KDUINT_MAX out-of-range indicator.
- Clarified that if two or more error conditions occur together, it is undefined which one's error code is set.
- KD_EVENT_TIMER: Wording of event merging description changed slightly to use the word "merge".
- kdNameLookup: Clarified that an unknown address family gives the error KD_EINVAL. Fixed some parameter naming inconsistency in the description.
- Clarified that all signed integer types are two's complement. Added new constants KDSSIZE_MIN KDSSIZE_MAX KDSIZE_MAX KDUINTPTR_MAX.
- kdLogf: Clarified that an input of -0 gives a result of -KD_INFINITY, the same as an input of +0.
- kdAtan2f: The special cases table now lists the cases when both inputs are 0, with results to match POSIX's MX extension.

A.3. OpenKODE 1.0

OpenKODE 1.0 was released on February 11th, 2008.

A.3.1. Acknowledgements

OpenKODE 1.0 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); Fredrik Kekäläinen (Acrodea); Tim Renouf (Tao/Antix); Keh-Li Sheng (Aplix); Ed Plowman (ARM); Roger Nixon (Broadcom); Scott Krig (Broadcom); Paul Novak (Ericsson); Brian Murray (Freescale); Teemu Keski-Kuha (Futuremark); Timo Suoranta (Futuremark); Petri Talala (Futuremark); Sami Niemela (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); Jani Vaisanen (Nokia); Kari Pulli (Nokia); Neil Trevett (NVIDIA); Petri Kero (Hybrid/NVIDIA); Ville Miettinen (NVIDIA); Antti Hätäla (NVIDIA); Thant Tessman (NVIDIA); Lars Bishop (NVIDIA); Aviad Lahav (Samsung); Remi Arnaud (Sony); Gabriele Svelto (STMicroelectronics); Jerry Evans (Sun); Bill Pinnell (Symbian); Robert Palmer (Symbian); Phil Huxley (Tao); Leo Estevez (TI); Marion Lineberry (TI); Tom Olsen (TI); Jon Leech.

A.3.2. Revisions

Revision 1, 2008-02-11

- Clarified that all four EGLImage EGL extensions for OpenGL ES are required if both OpenGL ES and OpenVG are present.

Revision 0, 2008-02-11

Initial revision.

A.4. OpenKODE 1.0 Provisional

OpenKODE 1.0 Provisional was approved by the Khronos Board of Promoters on February 8th, 2007. Revisions were released on March 30th 2007 and December 20th 2007.

The specification received many changes in its provisional period, and these changes are listed in a separate document.

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”)* .

[TR24731] *ISO/IEC TR 24731: Extensions to the C Library Part I: Bounds-checking interface* .

Index