

# **CarbonKernel**

**Real-time Operating System Simulator**

**Version 1.4**

**Simulated Driver  
Development Kit**

October 2001

# Table of Contents

<b>1.INTRODUCTION</b>	<b>1</b>
<b>2.SDDK CONCEPTS</b>	<b>2</b>
2.1MODULES AND STREAMS	2
2.2DEVICE MINOR NUMBER	2
2.3THE MESSAGING SYSTEM	3
2.3.1Structure of a message block header	4
2.3.2Structure of a data block header	4
2.3.3Message queue protocol	5
2.3.3.1Filter processing	5
2.3.3.2Driver processing	5
2.3.4Structure of an i/o block	6
2.4INTERFACE WITH THE APPLICATION	6
2.5INTERFACE WITH THE SIMULATION KERNEL	7
2.6STRUCTURE OF A MODULE	7
2.7MODULE IDENTIFICATION AND ATTACHMENT	7
2.7.1Description of the module information block	8
2.7.2Module attachment protocol	8
<b>3.SDDK INTERFACE</b>	<b>9</b>

# 1. Introduction

This document describes the Simulated Device Development Kit (aka SDDK) which is part of **CarbonKernel**, the real-time operating system simulator. The SDDK is an API built on top of the simulation kernel which allows developing simulated device drivers.

The main goal of a simulated device driver is to implement the simulation counter-part of a "real" driver accessing "real" hardware for the application, by providing a normalized way for sending and receiving data to/from a pseudo-device faking the real hardware during the simulation process.

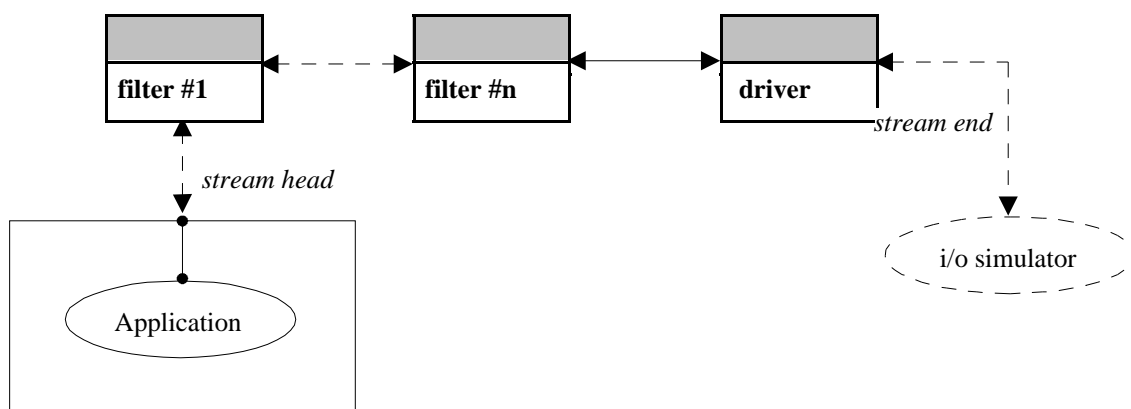
The simulated device driver is in some respects structured like a usual kernel driver. It displays canonical entry points allowing the simulation kernel to complete i/o operations through them. For the people who are familiar with UNIX(tm) kernel programming principles, the SDDK specifications are a simplified combination of legacy and STREAMS driver concepts. These characteristics make a driver rather simple to implement, whilst it can provide powerful device virtualization support to the application.

## 2. SDDK Concepts

### 2.1 Modules and streams

A *module* is the generic name for a piece of software which may act like a filter, or a device driver end-point.

An application can communicate with a simulated device through a collection of modules, linked together by a bi-directional message queue. The logical path of data created by such collection is named a *stream*. Each module may refine the messages obtained on its input queue from the precedent module in the stream and pass the resulting messages to the next module through its output queue. This process ends when the first or last module of the stream is reached, depending on the direction imposed by the current i/o operation.



The device driver is closest to the simulated device in the stream: it is known as the *stream end*. An unlimited number of filter modules may be pushed between the application and the device driver: the module closest to the application is seen as the *stream head*. The simplest stream makes a given device driver next to both the *stream head* and the *stream end*.

A direct application of this approach would allow pushing modules aimed to emit perturbations or have otherwise impact on the message flow for testing purposes.

The driver module can be coupled to any piece of code which can operate as a data source and/or sink, in order to fake the behaviour of a simulated hardware device. CarbonKernel provides a straightforward communication channel between simulation threads known as the *message port system* based on the VRTOS' internal event bus which is first candidate to relay bulks of data between a driver and the hardware simulation code. But one can use almost any other method to establish a dialog between these components, including shared data regions and so on.

Messages flowing from the *stream head* to the *stream end* are said to go *downstream*. Conversely, messages flowing from the *stream end* to the *stream head* are said to go *upstream*.

### 2.2 Device minor number

A communication path is always established between the application and a single instance of a simulated device, at least through a driver module, and eventually

across one or more filter modules. The concept of a device minor number allows a single driver to manage multiple instances of a given simulated device. For instance, an application may refer to the same keyboard driver while opening communication streams to several distinct instances of simulated keyboards.

At any time in the module's code, the routine `sddGetMinor()` can be called with the current *stream* handle to retrieve the device minor number concerned by the current i/o operation. A minor number is 0-based.

In the CarbonKernel's (inner) world, the driver's name (i.e. `mod_name`) and the device minor number passed to `ckOpenDevice()` respectively stand for the device's node directory entry and the minor number carried by its i-node in a UNIX world.

## 2.3 The messaging system

Modules pushed on a given stream communicate through message blocks, each holding a variable size data block. A linked-list of message blocks forming a single logical message is passed for input by the simulation kernel to a module, as the result of the output of the previous module in the stream. Hence, a simple or multi-parts logical message can be composed using one or several message blocks and channeled through the stream by the mean of input and output queues. The active module may then :

- push the incoming message blocks to its output queue without modification;
- compose a refined logical message by changing in whole or in part the incoming message block(s) before pushing them to the output queue;
- abort the current i/o operation, usually returning an error code to the simulation kernel.
- complete the current i/o operation before returning a success status to the simulation kernel.

When a message arrives at the *stream end*, the driver should be able to deliver the received data to any piece of code faking the behaviour of a simulated device. Sometimes the driver embodies such code. However this code can also be independent from the driver's, which should communicate with it through any available means, such as message ports.

When a read request is emitted, always starting from the *stream end*, the driver can obtain the data which should be passed to the application from any relevant source.

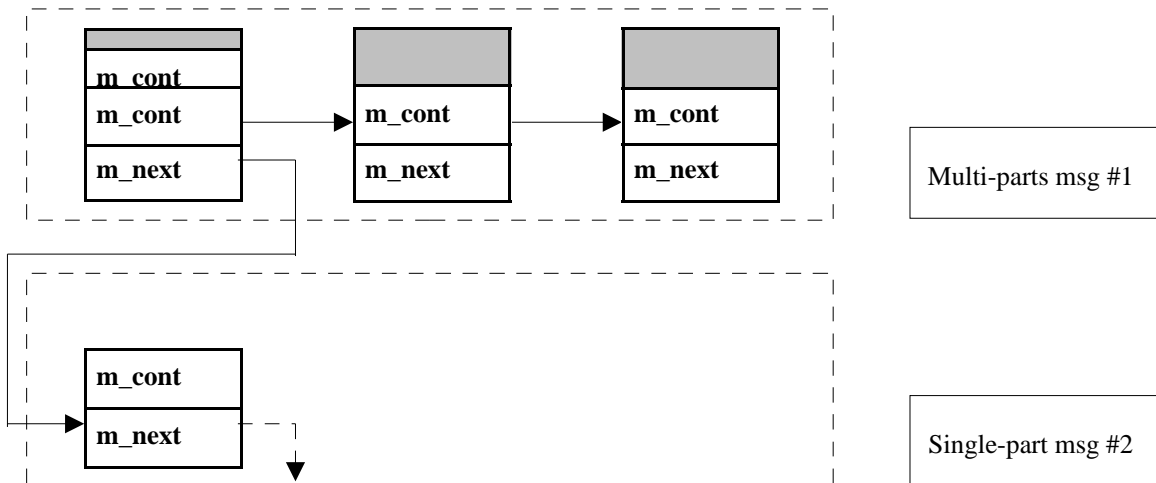
When a message arrives at the *stream head*, the contents of the output queue of the first module is simply copied into the receiving application's buffer.

When a write request is emitted, always started from the *stream head*, the message flows across the stream until the driver is reached.

### 2.3.1 Structure of a message block header

<b>m_wptr</b> ( <i>write</i> pointer inside the data )
<b>m_rptr</b> ( <i>read</i> pointer inside the data)
<b>m_data</b> (address of attached data)
<b>m_cont</b> (link to continued logical message)
<b>m_next</b> (link to next logical message)

A message block header can be traversed by two concurrent queues. First, it can be linked to other messages blocks forming a single logical message having multiple parts through the **m\_cont** member field. Second, it can be linked to the next logical message in a multi-messages queue through the **m\_next** field.

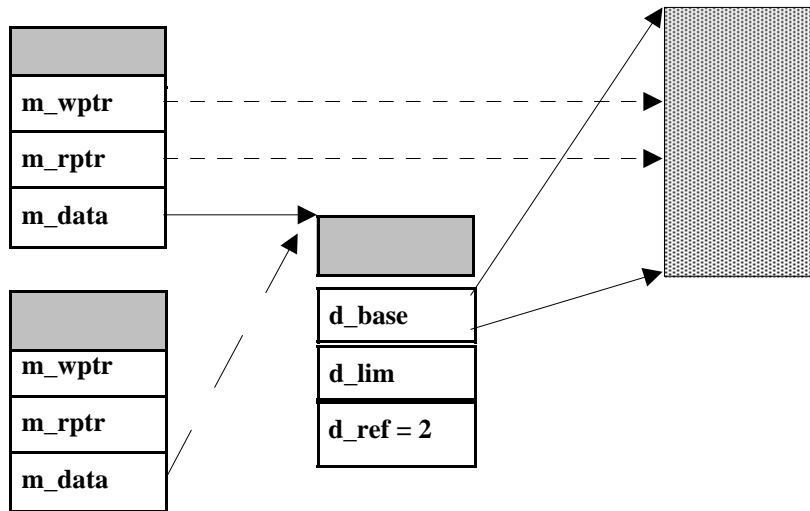


### 2.3.2 Structure of a data block header

A data block header is held by a message block header, and carries the valuable information attached to a logical message segment. A reference count is maintained by the SDDK's message manipulation routines allowing a given data block to be held by several message blocks. A data block header contains a pointer to the memory area holding the valuable information. The recycling method of such memory is also specified at the data block level.

<b>d_free</b> (pointer to the free policy descriptor )
<b>d_base</b> (pointer to the beginning of the information data)
<b>d_lim</b> (pointer after the end of the information block)
<b>d_ref</b> (reference count)

A message block contains two dedicated pointers, namely **m\_rptr** and **m\_wptr**, to designate respectively the current read and write addresses inside the region of memory laid between the **d\_base** and **d\_lim** pointer values. It should be noted that the SDDK routines and the i/o device support code from the simulation kernel both exclusively use the ranges  $[d\_base .. m\_rptr]$  and  $[d\_base .. m\_wptr]$  to compute the effective size of what has been read or written from/to the data region. The module should update these pointers consistently as needed.



Data block headers carrying no data are allowed; in such a case, the **d\_base** and **d\_lim** fields are set to NULL, and so should remain the **m\_wptr** and **m\_rptr** from the holding message block header.

### 2.3.3 Message queue protocol

The protocol used between the simulation kernel and a module to pass messages back and forth on a given stream differs whether the current module is a filter or a driver.

#### 2.3.3.1 Filter processing

**mod\_read** and **mod\_write** routines should extract the incoming logical message from the input queue, process -and eventually alter/refine its contents- then push the resulting message to its output queue using the dedicated SDDK services.

#### 2.3.3.2 Driver processing

**mod\_read** should collect the next bulk of data to pass *upstream* by any mean it has access to, including starting a simulated physical read operation, during which a piece of code faking the hardware device behaviour produces the necessary input data. The available data should be packaged in a sequence of message block headers forming a single logical message which should be pushed on the driver's output queue.

If the driver provides support for reading messages ahead from the simulated hardware, outside any outstanding read operation, it should be noted that a

special queue slot is transparently used by the SDDK routines to store those messages until they are finally claimed during a subsequent **mod\_read** execution. This slot is known as the "wait queue", and is transparently selected by the message queuing routine (i.e. `sddPutMessage()`) when it is invoked on behalf of an asynchronous context, such as an interrupt handler and so on. In order to pass the available read-ahead data *upstream* before attempting to simulate a physical read operation, the **mod\_read** routine should call the `sddGetMessage()` to extract those pending messages. It should be noted that unlike the input queue, the wait queue is a multi-messages queue, linking the heading blocks of distinct logical messages through their **m\_next** field. Each heading message block may be continued through the **m\_cont** field, thus forming a multi-parts logical message.

**mod\_write** should extract the logical message to emit from its input queue, then it may use the attached data blocks freely, including starting a simulated physical write operation, during which a piece of code faking the hardware device behaviour consumes the available output data.

It should be noted that the simulation kernel automatically discards the module's input queue contents on return of the **mod\_read** or **mod\_write** routines. Similarly, the output queue gets automatically flushed after the *stream head* or *end* is reached, depending on the direction of the undergoing i/o operation.

### 2.3.4 Structure of an i/o block

The i/o block is a data structure passed by the simulation kernel to the **mod\_read** and **mod\_write** routines, describing the undergoing i/o operation. This information block also contains the input and output queue slots the module will refer to when consuming messages from the previous module in the stream, and pushing new ones to the next module.

The pointer to the active i/o block is generally requested by SDDK services operating on messages queues or simulating a physical read/write operation.

The member fields of the i/o block should never be directly manipulated by the module's code, but rather through the documented SDDK API. Otherwise, incompatibilities with later simulation kernel or SDDK versions will possibly arise.

The i/o block contains an error field which should be updated using the `sddIoError()` macro when a failure to complete the undergoing i/o operation is encountered. Conversely, the macro `sddGetIoError()` can be used to retrieve the current error code set for a given i/o block.

## 2.4 Interface with the application

One of the significant contribution of the CarbonKernel's device i/o scheme is to allow the application to access any kind of simulated device through a small set of generic primitives defined in the CarbonKernel's CKPI interface. Moreover, the services called by those primitives are not "tainted" by the underlying simulated RTOS personality. This means that both the SDDK drivers and the primitives allowing to access their services are portable across all supported RTOS semantics.

These five primitives are :



- ▣ `ckOpenDevice()` opens a bi-directional communication stream with a designated device;
- ▣ `ckCloseDevice()` closes a previously opened communication stream;
- ▣ `ckReadDevice()` reads a stream of bytes from the device;
- ▣ `ckWriteDevice()` writes a stream of bytes to the device;
- ▣ `ckIoctlDevice()` sends an i/o control command to the device.

Information is passed back and forth between the simulation kernel and the driver by the mean of data and message blocks. These data structures allow composing messages incrementally which can then be passed through the modules that form a SDDK stream.

## 2.5 Interface with the simulation kernel

The SDDK interface provides a comprehensive set of services for implementing a module. Because the module's code is run on behalf of the calling thread's context by the simulation kernel (except the interrupt handlers), all the services from the CKPI are still available in the context of a module; this way, both APIs do not overlap functionally.

The services can be dispatched in four distinct groups :

- ▣ the binding services
- ▣ the physical i/o simulation services
- ▣ the context management services
- ▣ the message and data blocks management services

## 2.6 Structure of a module

A SDDK module is always structured the same way, whether it acts like a filter or the final driver for the simulated device. It should display a set of canonical routines used by the simulation kernel to request a well-defined set of i/o operations. These operations currently are :

- ▣ attachment procedure of the module to the running kernel (*mod\_attach, optional*)
- ▣ detachment procedure of the module from the running kernel (*mod\_detach, optional*)
- ▣ creation of a new logical path of communication identified by a stream (*mod\_open, mandatory*)
- ▣ destruction of a logical path of communication ( *mod\_close, optional* )
- ▣ data reception from the device on a given stream ( *mod\_read, optional* )
- ▣ data emission to the device on a given stream ( *mod\_write, optional* )
- ▣ module's state control and status request ( *mod\_ioctl, optional* )

Each routine should return a POSIX status code to the simulation kernel after completion, taken from the available set defined in the standard header `<errno.h>`.

## 2.7 Module identification and attachment

A SDDK module must be identified before it can be successfully attached to a running kernel. The mean of identification is a simple (static) routine embodied in the module's code which *must* be named *sdd\_info()*. This routine must return a pointer to a *sdd\_modinfo\_t* structure defined in *ck/sddk.h*, describing the module's interface to the simulation kernel.

### 2.7.1 Description of the module information block

The *sdd\_modinfo\_t* structure contains a set of pointers to functions which must be filled in with the addresses of the supplied canonical routines. When a canonical service is optional, the SDD\_NODEV value can be used to give an unspecified entry point. This structure contains four other members giving additional information on the module :

- a type field, defining whether the module is a driver or a filter;
- a name field, pointing to a null-terminated characters string, which will identify the module at the application level;
- a 32-bits version code; the *sdd\_get\_minver()* and *sdd\_get\_majver()* macros can be used to crack this value to obtain respectively the minor and major version numbers.
- an auto-push list field, which is a null-terminated array of characters string pointers identifying the modules which should be considered as prerequisites for the current module to operate. Those modules will be pushed on top of the stream head before attempting to push the current one.

### 2.7.2 Module attachment protocol

A module's object code must be statically linked to the simulator's executable image in order to be identified. During its early stages of initialization, the simulator searches its own symbol table for entries named *sdd\_info* stored in the TEXT section of the running executable image. Each found entry's address is resolved in the simulator's address space, and called in order to get back a pointer to the *sdd\_modinfo\_t* structure each identified module should export this way.

The found modules are then registered under the name given by the current value of the *mod\_name* field from their respective information block. This external name should be known by the application wanting to open a stream to this module using the *ckOpenDevice()* service. Only communication paths to *stream end* modules (i.e. drivers) can be opened by applications. Filter modules may not be directly targeted by the *ckOpenDevice()* service.

For each registered module, the *mod\_type* field is checked to see whether the module should be statically and permanently attached during the simulation, or if a dynamic protocol should be involved. In the latter case, a module whose *mod\_type* field contains the SDD\_MOD\_DYNAMIC flag will be attached when the first stream is created to it, then detached when its last stream has been closed. Otherwise, if the module attachment protocol is static, the simulation kernel invokes the attachment routine (if given) as a part of its own initialization procedure.

### **3. SDDK Interface**

## **mod\_attach**

### **SYNOPSIS**

```
#include <ck/sddk.h>

int mod_attach(sdd_devinfo_t *devinfo);
```

### **DESCRIPTION**

The module should provide for an attachment procedure to initialize its global context before accepting ulterior creation of communication paths. In the case of a static attachment protocol, this routine is called once during the system initialization. If the attachment protocol is dynamic, this routine is called each time the system is about to open the first active stream to this module.

One of the standard initializations performed by a module is allocating enough memory to store the module's private information on a per-instance basis. The `sddInitSoftState()` service from the SDDK's "software state" facility should be used for this.

### **PARAMETERS**

*devinfo* the handle of an internal information block which can be passed to the SDDK services requiring such input. This information has no direct value for the current module, thus it is given as an opaque handle.

### **RETURN VALUES**

`ENODEV` should be returned by the attachment procedure during a static attachment phase if the module denies ulterior creation of any stream. Doing so will prevent the simulation kernel to register the module as a valid target for subsequent `ckOpenDevice()` calls.

`SDD_SUCCESS` should be returned on success.

Any other standard error code found in `errno.h` can be returned.

### **CONTEXT**

When called during a static attachment phase, the running node is partially initialized, and the current context should be considered as an asynchronous one, with all the limitations which apply to, especially concerning the set of CKPI services that can be invoked.

When called during a dynamic attachment phase, the routine is run on behalf of the context of the simulation thread which issued the outstanding `ckOpenDevice()` request creating the first active stream to the module.

When called during an explicit push of the module on the given stream through and i/o control operation, the routine is run on behalf the context of the simulation thread which issued the outstanding `ckIoctlDevice()` submitting a `SDIO_PUSH` command.

### **SEE ALSO**

`mod_detach()`, `sddInitSoftState()`, `ckOpenDevice()`

## **mod\_detach**

### **SYNOPSIS**

```
#include <ck/sddk.h>

int mod_detach(sdd_devinfo_t *devinfo);
```

### **DESCRIPTION**

The module should provide for a detachment procedure to perform all the housekeeping chores it needs before being removed from the list of active modules. In the case of a static attachment protocol, this routine is currently never called. If the attachment protocol is dynamic, this routine is called each time the system closes the last active stream to this module, after the *mod\_close* has returned for the stream.

One of the standard housekeeping chores performed by a module is freeing the memory it currently holds to store its private information using the *sddFreeSoftState()* service.

### **PARAMETERS**

*devinfo* the handle of an internal information block which can be passed to the SDDK services requiring such input. This information has no direct value for the current module, thus it is given as an opaque handle.

### **RETURN VALUES**

SDD\_SUCCESS should be returned on success.

Any other standard error code found in *errno.h* can be returned.

### **CONTEXT**

When called during a dynamic detachment phase, this routine is run on behalf of the context of the simulation thread which issued the outstanding *ckCloseDevice()* request closing the last active stream to the module.

When called during an explicit pop of the module on the given stream through and i/o control operation, the routine is run on behalf of the context of the simulation thread which issued the outstanding *ckIoctlDevice()* submitting a *SDIO\_POP* command.

### **SEE ALSO**

*mod\_attach()*, *sddFreeSoftState()*, *ckCloseDevice()*

## **mod\_open**

### **SYNOPSIS**

```
#include <ck/sddk.h>

int mod_open(sdd_stream_t *stream, int mode);
```

### **DESCRIPTION**

The module should provide for an open procedure which is responsible for initializing a new communication path. If the SDDK's "software state" facility is used to manage the module's private information, `sddGetSoftState()` can be used to get a pointer to the memory block that will hold the stream's state information during its lifetime.

### **PARAMETERS**

*stream*            an opaque handle which uniquely identifies the new communication path during its lifetime.

*mode*            a mask describing the current operation. A combination of the following flags can be passed through this parameter :

- ▣ SDEV\_READ tells the module that the stream is open for reading.
- ▣ SDEV\_WRITE tells the module that the stream is open for writing.
- ▣ SDEV\_EXCL tells the module that the device instance should not be shared. It is up to the *mod\_open* routine to perform the necessary steps in identifying conflicts with existing streams and returning the appropriate error status accordingly (usually EBUSY). If the module operates on a device that cannot be shared by nature (e.g. a keyboard), it is free to enforce the exclusiveness of the access even if the SDEV\_EXCL flag is not set in the *mode* mask.

### **RETURN VALUES**

SDD\_SUCCESS should be returned on success.

Any other standard error code found in `errno.h` can be returned.

### **CONTEXT**

This routine is run on behalf of the context of the simulation thread which issued the outstanding `ckOpenDevice()` request.

### **SEE ALSO**

`mod_close()`, `ckOpenDevice()`

## **mod\_close**

### **SYNOPSIS**

```
#include <ck/sddk.h>
int mod_close(sdd_stream_t *stream);
```

### **DESCRIPTION**

The module should provide for a close procedure which is responsible for releasing any resource held by a communication path before it is destroyed.

### **PARAMETERS**

*stream*            an opaque handle which identifies the closed communication path.

### **RETURN VALUES**

SDD\_SUCCESS should be returned on success.

Any other standard error code found in errno.h can be returned.

### **CONTEXT**

This routine is run on behalf of the context of the simulation thread which issued the outstanding ckCloseDevice() request.

### **SEE ALSO**

mod\_open(), ckCloseDevice()



## mod\_read

### SYNOPSIS

```
#include <ck/sddk.h>

int mod_read(sdd_stream_t *stream, sdd_iob_t *iob);
```

### DESCRIPTION

The module should provide for a read procedure which is responsible for retrieving and sending the next data available *upstream*. Once the *stream head* is reached, the data collected on the output queue of the first module is made available to the application by the simulation kernel.

The **mod\_read** routine of a driver module usually starts a simulated physical read operation from the faked hardware device, unless some read-ahead data exists in the stream's wait queue. If the stream is bound to a message port as a result of a call to `sddBindStream()`, the straightforward manner to initiate a simulated physical read operation stands in using the `sddPhysRead()` service.

### PARAMETERS

*stream*            an opaque handle which identifies the stream to read from.

*iob*                an i/o block describing the undergoing i/o operation, which also holds the input and output queues for the active stream in the current module.

### RETURN VALUES

SDD\_SUCCESS should be returned on success.

Any other standard error code found in `errno.h` can be returned.

### CONTEXT

This routine is run on behalf of the context of the simulation thread which issued the outstanding `ckReadDevice()` request.

### SEE ALSO

`mod_write()`, `ckReadDevice()`

## **mod\_write**

### **SYNOPSIS**

```
#include <ck/sddk.h>

int mod_write(sdd_stream_t *stream, sdd_iob_t *iob);
```

### **DESCRIPTION**

The module should provide for a write procedure which is responsible for sending the submitted output data *downstream*. Once the *stream end* is reached, the output data should be made available to the piece of code faking the simulated hardware device.

The `mod_write` routine of a driver module usually starts a simulated physical write operation to the faked hardware device. If the stream is bound to a message port as a result of a call to `sddBindStream()`, the straightforward manner to initiate a simulated physical write operation stands in using the `sddPhysWrite()` service.

### **PARAMETERS**

*stream*            an opaque handle which identifies the stream to write to.

*iob*                an i/o block describing the undergoing i/o operation, which also holds the input and output queues for the active stream in the current module.

### **RETURN VALUES**

`SDD_SUCCESS` should be returned on success.

Any other standard error code found in `errno.h` can be returned.

### **CONTEXT**

This routine is run on behalf of the context of the simulation thread which issued the outstanding `ckWriteDevice()` request.

### **SEE ALSO**

`mod_read()`, `ckWriteDevice()`

## mod\_ioctl

### SYNOPSIS

```
#include <ck/sddk.h>

int mod_ioctl(sdd_stream_t *stream, int cmd, void *arg, int
*retval);
```

### DESCRIPTION

The module can provide for a control procedure which is responsible for changing the current module's or stream's state, or return some valuable information to its caller.

The **mod\_ioctl** routine is called in turn for each module present on a given stream as a result of the application layer issuing the `ckIoctlDevice()` call. Hence, the direction of the i/o control stream walk is always *downstream*.

The `SDEV_IOx()` macros from *ck/sdevice.h* can be used to encode i/o control commands. A sub-set of these commands allows an application to push/pop **SDDK** filter modules on an open stream. See the **SDDIO** documentation for more on the standard i/o control commands.

### PARAMETERS

*stream*            an opaque handle which identifies the stream to write to.

*cmd*              an integer value encoding the command word and miscellaneous information used by the simulation kernel for transferring data to or from the module.

*arg*              a generic pointer passed by the application layer as an argument to the command to apply. The length of the region of memory this pointer refers to and the type of the pointed object depend on the command word. The imposed maximum size of an argument is 255 bytes (inclusive).

*retval*           an integer value that will be returned to the application layer as a result of calling *ckIoctlDevice()*.

### RETURN VALUES

`SDD_SUCCESS` should be returned on success.

Any other standard error code found in `errno.h` can be returned. If one of the **mod\_ioctl** routines defined for the stream returns a non-zero error code, then `ckIoctlDevice()` will return -1 to the application, and `errno` will be set to that error code.

### CONTEXT

This routine is run on behalf of the context of the simulation thread which issued the outstanding `ckIoctlDevice()` request.

### SEE ALSO

`mod_read()`, `ckWriteDevice()`

## **sddBindStream** - bind a stream to a message port

### **SYNOPSIS**

```
#include <ck/sddk.h>

int sddBindStream(sdd_stream_t *stream, sdd_iohandler_t
*r_handler, sdd_iohandler_t *w_handler, ckhandle_t
*handlep);
```

### **DESCRIPTION**

This service binds an i/o stream to a message port. Using message ports is the easiest way of exchanging variable size data bulks within an application. Binding a stream to a message port implies using this CKPI's messaging service for reading data from and writing data to the code faking a hardware device's behaviour. `sddBindStream()` takes internal dispositions for the module to call `sddPhysRead()` and `sddPhysWrite()` in order to simulate physical i/o operations on a given stream. This is the reason why this service is exclusively called by drivers from their `mod_open()` routine, and never by filter modules.

In order to exchange data through a message port, a conventional port name should be agreed between the sender(s) and the receiver(s). `sddBindStream()` determines the implicit port name to which the stream can send and/or receive data to/from remote threads, using the following rule:

portName = "<mod\_name>:<minor\_number>".

For instance, if the **kbd** driver invokes `sddBindStream()` in its `mod_open()` routine, as a result of an application request to open instance #0 of the simulated keyboard device, then the implicit port name will be "kbd:0". This way, any thread sending data through the message port named "kbd:0" will provide input to the keyboard driver for the corresponding stream. Conversely, writing to this port will cause the driver to send messages concerning this device instance to any thread reading it.

If a message port with the given name pre-exists before `sddBindStream()` is called, the stream will be bound to it. Otherwise, a new message port will be built internally for the stream to be bound to.

Magnets, which are GUI front-ends connected to message ports, can be used to feed drivers with input and display their output in an interactive, user-friendly manner.

### **PARAMETERS**

*stream*            the stream handle passed by the simulation kernel to the `mod_open()` routine.

*r\_handler*        a pointer to a user routine which is expected to simulate the physical reading of data from a faked hardware device. This routine will be called in response to the invocation of the `sddPhysRead()` service by the `mod_read()` routine of the current module. This indirection allows passing the address of a routine conforming to the SDDK standards, which can be implemented in a foreign piece of code, or directly accessible from the current module, simulating the behaviour of some hardware device.

*w\_handler* a pointer to a user routine which is expected to simulate the physical writing of data to a faked hardware device. This routine will be called in response to the invocation of the `sddPhysWrite()` service by the `mod_write()` routine of the current module. This indirection allows passing the address of a routine conforming to the SDDK standards, which can be implemented in a foreign piece of code, or directly accessible from the current module, simulating the behaviour of some hardware device.

*handlep* a pointer to an opaque object's handle identifying the bi-directional message port to which the stream is bound. On error, the value of this handle is undefined.

## RETURN VALUES

`SDD_SUCCESS` is returned if the stream was successfully bound to the message port. A non-zero port handle has been written to the memory pointed to by *handlep*.

`SDD_FAILURE` is returned on error.

## CONTEXT

`sddBindStream()` can be called on behalf of the context of any canonical service routine. However, **mod\_open** is best candidate for binding a new stream to a message port.

Calling `sddBindStream()` more than once for a given stream is allowed. The previous binding will be merely replaced by the new one.

## ERRORS

Two causes of failure may arise :

- ▯ the calling module is a filter. Only drivers can invoke this service, because they are exclusively responsible for simulating physical i/o.
- ▯ there is no receiver waiting for messages on the designated port.

## SEE ALSO

`ckBindPort()`, `ckReadPort()`, `ckWritePort()`, `sddPhysRead()`, `sddPhysWrite()`

## **sddUnbindStream - unbind a stream from a message port**

### **SYNOPSIS**

```
#include <ck/sddk.h>

int sddUnbindStream(sdd_stream_t *stream, ckhandle_t
handle);
```

### **DESCRIPTION**

This service unbinds an i/o stream from a message port. This operation reverts the actions of a previous call to `sddBindStream()` for the given stream. It has no effect on the message port itself. After this service has returned, calls to `sddPhysRead()` and `sddPhysWrite()` will beget an error.

### **PARAMETERS**

*stream*            the stream handle passed by the simulation kernel to the `mod_open()` routine.

*handle*            the opaque object's handle identifying the bi-directional message port which should have been returned by a previous call to `sddBindStream()`.

### **RETURN VALUES**

`SDD_SUCCESS` is always returned.

### **CONTEXT**

`sddUnbindStream()` can be called on behalf of the context of any canonical service routine.

### **SEE ALSO**

`sddBindStream()`, `sddPhysRead()`, `sddPhysWrite()`

**sddGetMinor** - get minor device number from a stream

## **SYNOPSIS**

```
#include <ck/sddk.h>
int sddGetMinor(sdd_stream_t *stream);
```

## **DESCRIPTION**

This service returns the minor number identifying the device instance the stream is bound to. This number is a 0-based integer.

## **PARAMETERS**

*stream* the stream handle.

## **RETURN VALUES**

The minor number.

## **CONTEXT**

sddGetMinor() can be called on behalf of any context.

## **SEE ALSO**

sddOpenDevice()



## **sddInitSoftState      -      initialize the software states allocator**

### **SYNOPSIS**

```
#include <ck/sddk.h>

int sddInitSoftState(void **statesp, size_t nbytes, int
ninstances);
```

### **DESCRIPTION**

This service initializes a local allocator for the calling module which provides chunks of memory for storing any kind of private state information on a per-device instance basis.

### **PARAMETERS**

*statesp*            the address of a generic pointer into which the sddInitSoftState() service will store the new allocation root. This information should be seen as an opaque handle and remain private to the module.

*Nbytes*            the size (in bytes) of the memory chunk needed to hold a single software state. One usually provides the size of a C structure defining the private state components.

*ninstances*        the maximum number of active instances of the simulated device which can be managed by the module. The minor number ranges from 0 to *ninstances* - 1 (inclusive).

### **RETURN VALUES**

SDD\_SUCCESS is returned if the state allocator was successfully initialized for the calling module.

SDD\_FAILURE is returned on error.

### **CONTEXT**

sddInitSoftState() can be called on behalf of the context of any canonical service routine. Nevertheless, the **mod\_attach** canonical routine is first candidate for allocating this kind of resources.

### **ERRORS**

Two causes of failure may arise :

- ▣ the calling module supplied a null *nbytes* parameter.
- ▣ there is not enough available memory to complete the operation.

### **SEE ALSO**

sddFreeSoftState(), sddGetSoftState()

## **sddGetSoftState      -      retrieve a software state block**

### **SYNOPSIS**

```
#include <ck/sddk.h>

void sddGetSoftState(void *states, int minor);
```

### **DESCRIPTION**

This service returns the address of the software state block assigned to a specific instance of a simulated device managed by the module.

### **PARAMETERS**

*states*            the allocation handle returned by a previous call to      sddInitSoftState().

*minor*            the minor number of the device instance. One should use sddGetMinor() to retrieve the minor number attached to the current stream.

### **RETURN VALUES**

The address of the private state block is return on success.

A null pointer is returned on error.

### **CONTEXT**

sddGetSoftState() can be called on behalf of any context.

### **ERRORS**

The sddGetSoftState() routine returns an error if the minor number is negative, or greater or equal than the maximum number of instances declared to the sddInitSoftState() service.

### **SEE ALSO**

sddInitSoftState()

## **sddFreeSoftState      -      release all software states**

### **SYNOPSIS**

```
#include <ck/sddk.h>

void sddFreeSoftState(void *states);
```

### **DESCRIPTION**

This service frees all the resources attached to a given state allocator, such as returning the allocated memory to the simulation system. This operation is usually part of the housekeeping chores performed by a detaching module.

### **PARAMETERS**

*states*            the allocation handle returned by a previous call to      sddInitSoftState().

### **RETURN VALUES**

none.

### **CONTEXT**

sddFreeSoftState() can be called on behalf of any context. Nevertheless, the **mod\_detach** canonical routine is first candidate to release all the resources attached to a module.

### **SEE ALSO**

sddInitSoftState()

## **sddSetPrivate - set stream's private cookie**

### **SYNOPSIS**

```
#include <ck/sddk.h>

void sddSetPrivate(sdd_stream_t *stream, void *cookie);
```

### **DESCRIPTION**

This service stores a cookie defined by the module in a reserved field of the stream descriptor. This cookie can be used to hold private data on a per-stream basis, and in any case remains opaque to the simulation sytem.

### **PARAMETERS**

*stream*            the opaque handle of the stream to attach the cookie to.

*Cookie*           the new value of the stream's cookie.

### **RETURN VALUES**

none.

### **CONTEXT**

sddSetPrivate() can be called on behalf of any context.

### **SEE ALSO**

sddGetPrivate()

## **sddGetPrivate - get stream's private cookie**

### **SYNOPSIS**

```
#include <ck/sddk.h>

void *sddGetPrivate(sdd_stream_t *stream);
```

### **DESCRIPTION**

This service returns the cookie attached to a stream by a previous call to `sddSetCookie()`.

### **PARAMETERS**

*stream*            the opaque handle of the stream to retrieve the cookie from.

### **RETURN VALUES**

The current value of the stream's cookie. If the cookie was unset, NULL is returned.

### **CONTEXT**

`sddGetPrivate()` can be called on behalf of any context.

### **SEE ALSO**

`sddSetPrivate()`

## **sddPhysRead - simulate a physical read operation**

### **SYNOPSIS**

```
#include <ck/sddk.h>

int sddPhysRead(sdd_stream_t *stream, sdd_iob_t *iob);
```

### **DESCRIPTION**

This service should be called from the **mod\_read** routine of a driver which uses i/o streams bound to message ports (see `sddBindStream()`). This routine arranges for the physical read handler to be called on behalf of a safe context before returning to the caller.

The read handler will be passed the same parameters on entry than `sddPhysRead()` has received (i.e. *stream* and *iob* handles). It should perform the necessary steps to collect the next data available for input from the given instance of the simulated hardware device. The message port obtained from a previous call to `sddBindStream()` should be used to get them.

The read handler should return `SDD_SUCCESS` on successful completion, or `SDD_FAILURE` otherwise. In the later case, the appropriate error code should also be set in the i/o block structure before returning to the caller, using the `sddIoError()` macro.

### **PARAMETERS**

*stream*            the opaque handle identifying the stream to read from.

*iob*                the i/o block describing the undergoing i/o operation.

### **RETURN VALUES**

`SDD_FAILURE` is returned if the stream is not bound to any message port.

Otherwise, the status returned by the read handler is passed back to the caller.

### **ERRORS**

`ENXIO` is set in the i/o block error field if the stream is not bound to any message port.

Any error code set by the read handler in the i/o block error field can be present if the operation failed.

### **CONTEXT**

This service should be called    on behalf of    the context of the    **mod\_read** routine.

### **SEE ALSO**

`mod_read()`, `ckReadDevice()`

## **sddPhysWrite            -        simulate a physical write operation**

### **SYNOPSIS**

```
#include <ck/sddk.h>

int sddPhysWrite(sdd_stream_t *stream, sdd_iob_t *iob);
```

### **DESCRIPTION**

This service should be called from the **mod\_write** routine of a driver which uses i/o streams bound to message ports (see `sddBindStream()`). This routine arranges for the physical write handler to be called on behalf of a safe context before returning to the caller.

The write handler will be passed the same parameters on entry than `sddPhysWrite()` has received (i.e. *stream* and *iob* handles). It should perform the necessary steps to send the current output data to the given instance of the simulated hardware device. The message port obtained from a previous call to `sddBindStream()` should be used to send them.

The write handler should return `SDD_SUCCESS` on successful completion, or `SDD_FAILURE` otherwise. In the later case, the appropriate error code should also be set in the i/o block error field before returning to the caller, using the `sddIoError()` macro.

### **PARAMETERS**

*stream*            the opaque handle identifying the stream to write to.  
*iob*                the i/o block describing the undergoing i/o operation.

### **RETURN VALUES**

`SDD_FAILURE` is returned if the stream is not bound to a message port.

Otherwise, the status returned by the write handler is passed back to the caller.

### **ERRORS**

`ENXIO` is set in the i/o block error field if the stream is not bound to any message port.

Any error code set by the write handler in the i/o block error field can be present if the operation failed.

### **CONTEXT**

This service should be called    on behalf of    the context of the    **mod\_write** routine.

### **SEE ALSO**

`mod_write()`, `ckWriteDevice()`

## **sddDataAlloc - allocate a new data block header**

### **SYNOPSIS**

```
#include <ck/sddk.h>

sdd_dblk_t *sddDataAlloc(void *base, size_t len, sdd_fdata_t
*fdata);
```

### **DESCRIPTION**

This service returns a new data block header set to hold the specified memory region. The allocation/free policy of this region is defined by the **fdata** parameter.

If **fdata** equals SDD\_FDATA\_VOLATILE, the memory area ranging from **base** to **base + len - 1** is copied to a private memory block attached to the new header, unless **base** is NULL, so as to define an empty data block. This memory will be automatically returned to the system when the header is freed.

If **fdata** equals SDD\_FDATA\_STATIC, this service will assume it is safe to hold a pointer to the original memory area ranging from **base** to **base + len - 1**, with no private copy needed. No action will be performed to release this memory block when its holder is freed.

If **fdata** is a valid pointer to a *sdd\_fdata\_t* structure, this service will assume it is safe to hold a pointer to the original memory area ranging from **base** to **base + len - 1**, with no private copy needed. Moreover, it will call the free handler whose address is defined in the structure to release the original chunk of memory. The free policy information structure has the following fields :

```
struct sdd_fdata {
    void (*fdata_handler)(void *base, void *cookie);
    void *fdata_cookie;
}
```

The *fdata\_handler* field is a pointer to the handler which is passed the base address of the memory region to free, and a private cookie the user may set in the *fdata\_cookie* field.

### **PARAMETERS**

*base*            the base address of the data. A NULL value is allowed, standing for an empty data block.

*len*            the length of the data area.

*fdata*           a pointer to the allocation/free policy information.

### **RETURN VALUES**

NULL is returned if there is not enough memory to allocate the data block and/or its header.

A pointer to the new data block header is returned on success.



## **CONTEXT**

sddDataAlloc() can be called on behalf of any context.

## **SEE ALSO**

sddDataFree(), sddMsgAlloc()

## **sddDataFree - free a data block**

### **SYNOPSIS**

```
#include <ck/sddk.h>

void sddDataFree(sdd_dblk_t *dblk);
```

### **DESCRIPTION**

This service releases the memory used by the data block header and the memory region it holds, according to the free policy defined at the time the header was allocated.

### **PARAMETERS**

*dblk* a pointer to the data block header to free, as returned by the `sddDataAlloc()` service.

### **RETURN VALUES**

none.

### **CONTEXT**

`sddDataFree()` can be called on behalf of any context. This service is usually not called directly by the driver's code, but rather indirectly through the `sddMsgFree()` service. However, data blocks which have not been attached to any message should be freed this way.

### **SEE ALSO**

`sddDataAlloc()`, `sddMsgAlloc()`

## **sddMsgAlloc - allocate a new message block header**

### **SYNOPSIS**

```
#include <ck/sddk.h>

sdd_mblk_t *sddMsgAlloc(sdd_dblk_t *dblk);
```

### **DESCRIPTION**

This service returns a new message block header set to hold the specified data block. The data block header should have been returned by the `sddDataAlloc()` service.

On success, the reference count maintained for the data block is incremented.

### **PARAMETERS**

*dblk* the address of a valid data block header which will be held by the new message block.

### **RETURN VALUES**

NULL is returned if there is not enough memory to allocate the message block header.

A pointer to the new message block header is returned on success.

### **CONTEXT**

`sddMsgAlloc()` can be called on behalf of any context.

### **SEE ALSO**

`sddMsgFree()`, `sddDataAlloc()`

## **sddMsgFree     -     free a message block**

### **SYNOPSIS**

```
#include <ck/sddk.h>
void sddMsgFree(sdd_mblk_t *mblk);
```

### **DESCRIPTION**

This service releases the memory used by a message block header, and attempts to free the associated data block.

sddMsgFree() decrements the reference count of the data block header and calls sddDataFree() for it if this count reaches zero.

### **PARAMETERS**

*dblk*            a pointer to the message block header to free, as returned by the sddMsgAlloc() service.

### **RETURN VALUES**

none.

### **CONTEXT**

sddMsgFree() can be called    on behalf of    any context.

### **SEE ALSO**

sddMsgAlloc(), sddDataFree()

## **sddMsgPut      -      queue a message block**

### **SYNOPSIS**

```
#include <ck/sddk.h>

int sddMsgPut(sdd_stream_t *stream, sdd_iob_t *iob,
sdd_mblk_t *mblk);
```

### **DESCRIPTION**

This service links the specified message block to the appropriate message queue.

If the calling context is asynchronous (e.g. an interrupt handler), the message is linked to the stream's wait queue. Otherwise the message is linked to the current module's output queue.

### **PARAMETERS**

<i>stream</i>	the handle of the stream sending/receiving data.
<i>iob</i>	the information block describing the undergoing i/o operation.
<i>mblk</i>	the message block to link to a queue.

### **RETURN VALUES**

This service returns the new count of message blocks currently linked to the affected queue.

### **CONTEXT**

sddMsgPut() should be called:

- ▣ from the mod\_read() routine (or on behalf of it) to pass data *upstream*.
- ▣ from the mod\_write() routine (or on behalf of it) when acting as a filter, to pass data *downstream*.
- ▣ from other contexts, for the purpose of adjusting the wait queue's contents.

### **SEE ALSO**

sddMsgAlloc(), sddMsgGet()

## **sddMsgGet      -      extract a message block**

### **SYNOPSIS**

```
#include <ck/sddk.h>

sdd_mblk_t *sddMsgGet(sdd_stream_t *stream, sdd_iob_t *iob);
```

### **DESCRIPTION**

This service extracts the next available message block from the appropriate message queue.

If this service is called on behalf of the `mod_write()` routine, the message is obtained from the current module's input queue. Otherwise, the message is obtained from the stream's wait queue.

### **PARAMETERS**

*stream*            the handle of the stream sending/receiving data.

*iob*                the information block describing the undergoing i/o operation.

### **RETURN VALUES**

This service returns the next available message block unlinked from the appropriate queue.

### **CONTEXT**

`sddMsgGet()` should be called:

- ▣ from the `mod_read()` routine (or on behalf of it), to fetch the current read-ahead data.
- ▣ from the `mod_write()` routine (or on behalf of it) when acting as a filter, to fetch the data to be passed *downstream*.
- ▣ from the `mod_write()` routine (or on behalf of it) when acting as a driver, to fetch the data to output to the simulated device.
- ▣ from an asynchronous context, for the purpose of adjusting the wait queue's contents.

### **SEE ALSO**

`sddMsgPut()`

## **sddIoWait        -        wait for i/o completion**

### **SYNOPSIS**

```
#include <ck/sddk.h>
int sddIoWait(sdd_iob_t *iob);
```

### **DESCRIPTION**

This service blocks the calling simulation thread until the service `sddIoSignal()` is called for the same i/o information block from another thread.

This service can be used to synchronize a thread with an asynchronous event, such as an interrupt.

`sddIoWait()` is aware of the timeout value associated to the i/o operation by the application, and enforces this constraint automatically.

### **PARAMETERS**

*stream*            the handle of the stream sending/receiving data.  
*iob*                the information block describing the undergoing i/o operation.

### **RETURN VALUES**

`SDD_FAILURE` is returned upon timeout, if the event was not signaled within the allotted amount of time. The i/o block is also marked for error.

`SDD_SUCCESS` is returned on success.

### **CONTEXT**

`sddIoWait()` must be called from a synchronous context, such as the `mod_read()/mod_write()` routines. It is usually called from a driver module, simulating the interaction between synchronous and asynchronous code (such as an interrupt handler).

The same kind of synchronization can be achieved using the "condition variables" services from the CKPI. Nevertheless, this SDDK-specific service silently handles a few housekeeping chores for the driver, such as dealing with the operation timeout.

### **SEE ALSO**

`sddIoSignal()`

## **sddIoSignal      -      signal i/o completion**

### **SYNOPSIS**

```
#include <ck/sddk.h>
void sddIoSignal(sdd_iob_t *iob);
```

### **DESCRIPTION**

This service signals the simulation thread pending for i/o completion (i.e. `sddIoWait()`) on the specified information block. This thread is woken up and resumes execution according to its priority.

If more than one thread pend on the i/o block, all of them are resumed by the signal.

### **PARAMETERS**

*iob*      the information block describing the undergoing i/o operation.

### **RETURN VALUES**

none.

### **CONTEXT**

`sddIoSignal()` can be called    on behalf of    any context.

### **SEE ALSO**

`sddIoWait()`



## SDDIO - Device control commands

### SYNOPSIS

```
#include <ck/sdevice.h>
```

### DESCRIPTION

A few standard device control commands are currently supported. The corresponding command words and parameters should be passed to the `ckIoctlDevice()` by the application to obtain the expected results.

### COMMANDS

▣ `ckIoctlDevice(stream,SDIO_PUSH,mod_name,retval)`

Requests to push the filter module named **mod\_name** on top of the communication channel identified by **stream**. Modules are always pushed on the stream head side of an existing channel.

As a consequence of this call, the `mod_attach()` routine may be called for dynamic filters before `mod_open()` is invoked for the first stream.

▣ `ckIoctlDevice(stream,SDIO_POP,mod_name,retval)`

Requests to pop the filter module named **mod\_name** from the communication channel identified by **stream**.

As a consequence of this call, the `mod_detach()` routine may be called for dynamic filters after `mod_close()` is invoked for the last stream.

The operation status is returned in **retval**. Zero is returned on success. Otherwise, an error occurred:

- ▣ `ENODEV` is returned if **mod\_name** is not a known module name.
- ▣ `EBUSY` is returned when an attempt is made to push a filter module which is already active for the stream.
- ▣ `ENXIO` is returned when an attempt is made to push a driver on a stream, or to pop the driver off the stream, which are both illegal operations.
- ▣ Any other error code whether returned by the attachment or detachment routines of the target filter module.

### SEE ALSO

`ckIoctlDevice()`, `mod_attach()`, `mod_detach()`

## ***Index***

### **C**

ckCloseDevice 6, 11, 13  
ckIoctlDevice 6, 10p., 16, 37  
ckOpenDevice 3, 6, 8, 10, 12  
ckReadDevice 6, 14, 26  
ckWriteDevice 6, 15p., 27

### **M**

m 6, 21, 23  
mod\_attach 7, 10p., 37  
mod\_attach 21  
mod\_close 7, 11pp, 37  
mod\_close 11  
mod\_detach 7, 10p., 37  
mod\_detach 23  
mod\_ioctl 7, 16  
mod\_open 7, 12p., 17pp, 37  
mod\_read 5pp, 14pp, 26, 33pp  
mod\_read 5p., 14, 26  
mod\_write 7, 14p., 17, 27, 33pp  
mod\_write 6, 15

### **S**

sddBindStream 14p., 17pp, 26p.  
sddDataAlloc 28pp  
sddDataFree 29p., 32  
sddFreeSoftState 11, 21, 23  
sddGetMinor 3, 20, 22  
sddGetPrivate 24p.  
sddGetSoftState 12, 21p.  
sddInitSoftState 10, 21pp  
SDDIO 16, 37  
sddIoError 6, 26p.  
sddIoSignal 35p.  
sddIoWait 35p.  
sddMsgAlloc 29pp  
sddMsgFree 30pp  
sddMsgGet 33p.  
sddMsgPut 33p.  
sddPhysRead 14, 17pp, 26  
sddPhysWrite 15, 17pp, 27  
sddSetPrivate 24p.  
sddUnbindStream 19  
SDIO\_POP 11, 37  
SDIO\_PUSH 10, 37