The Input Server

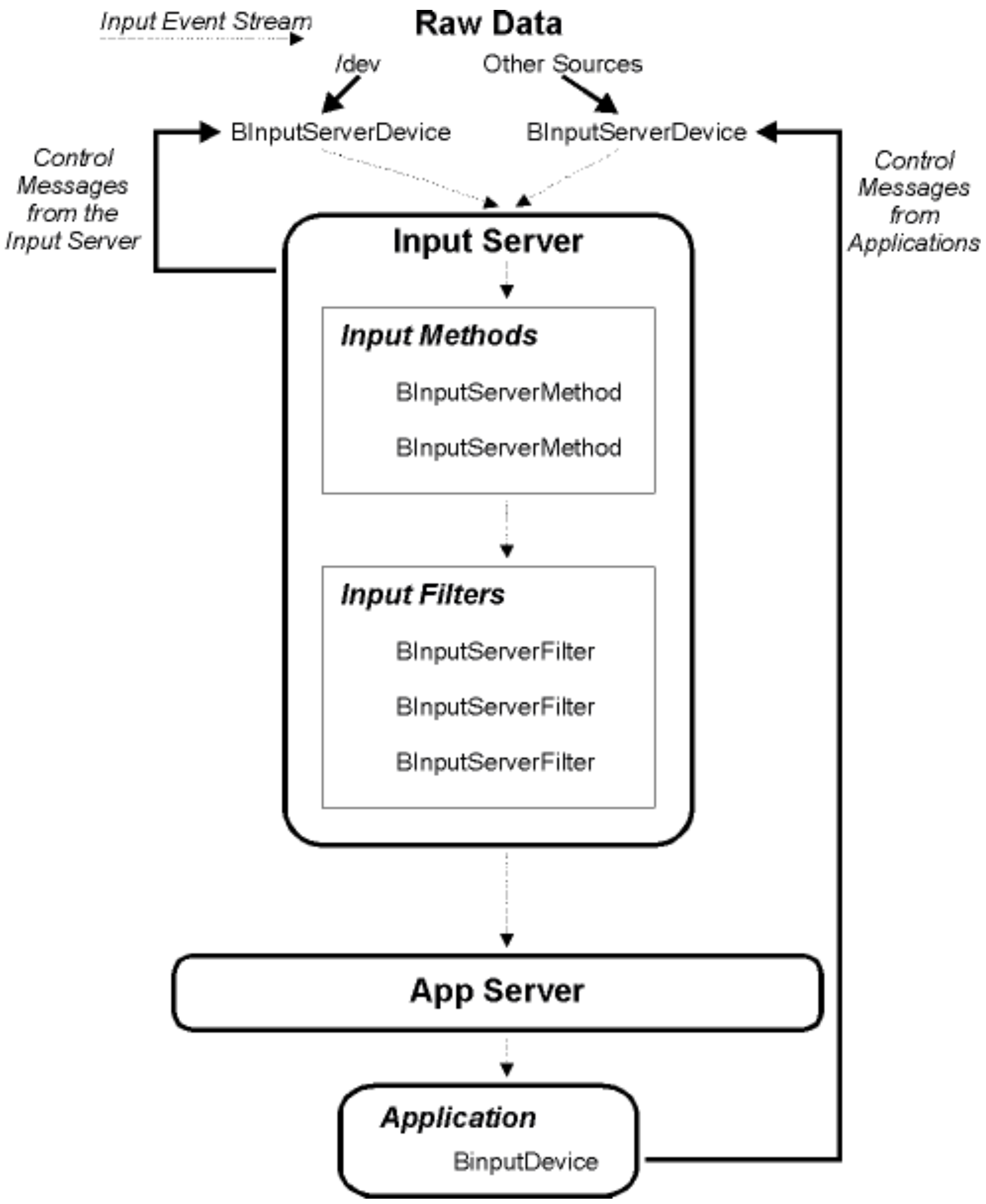
The Device Kit − Table of Contents

The Input Server

The Input Server is a system service that accepts user events (on the mouse and keyboard, typically) at one end, and dispatches them to the App Server at the other. In between, the Input Server runs the events through a series of filters that can inspect and modify them. The generation and filtering of events is performed by the add−ons that the Input Server loads; the Server itself just provides the plumbing. Event−generating add−ons (called *input devices*) typically correspond to one or more device drivers, although this isn't a requirement. An event−filtering add−on (*input filter*) processes the events that are fed to it; input filters aren't intended to correspond to hardware. A third type of Input Server add−onan *input method*is used to implement input method mechanisms, which convert keyboard events into character sets that can't be easily represented on a standard keyboard, such as Kanji.

Each of these add−on types (input devices, filters, and methods) is represented by a C++ class: BInputServerDevice, BInputServerFilter, and BInputServerMethod. For each add−on you want to create, you subclass the appropriate class and override its hook functions. An additional classBInputDevicelets a "normal" application send messages back through the Input Server to the input devices; a BInputDevice object can be useful if you're creating a preference app for a custom Input Server add−on, for example.

A map of the Input Server world looks like this:



Note that the Input Server and its add−ons (and BInputDevice) all live in user space, so, in theory, there's nothing that a "normal" application can do that an Input Server add−on can't do. However, Input Server add−ons are loaded early in the boot process, before some system services (such as the Media and Network servers) have started. Attempting to use services from these servers before they've started is a good way to wedge the system.

The BeOS provides a few Input Server add−ons: It installs input devices that handle a variety of mice and keyboard drivers, and an input filter that the Screen Saver engine uses to detect user activity (on the mouse and keyboard). BeOS's only built−in input method is installed when you choose the Japanese language option during the installation process.

Currently, events that are generated by the BeOS joystick drivers do not go through the Input Server.

Drivers and Input Devices

As mentioned above, most input devices (i.e. input−generating add−ons) correspond to one or more device drivers. For example, the BeOS **mouse** input device manages all the mouse drivers that the OS provides.

It's important to keep in mind that an input device is not the same as the device driver(s) it managesthey're separate pieces of code that execute in separate address spaces: the drivers run in the kernel, the add−ons run in the Input Server. An input device can **`open()`** a driver, but it must not explicitly load the driver. In other words, the add−on shouldn't re−invent or subvert the kernel's driver−loading mechanism.

Similar to drivers, Input Server add−ons must be scrupulous about managing their memory and threads:

- Memory that an add−on allocates must be freed when the add−on is unloaded, otherwise the add−on will leak.
- The hook functions that are invoked on your add−on are executed in threads that must stay as "live" as possible. If your add−on does a lot of processing that can be performed asynchronouslyfor example, if it's an input device that's "watching" a piece of hardwarethe add−on should spawn a thread.

Building

Like all add−ons, Input Server add−ons are compiled as shared libraries. The add−ons must link against **input_server**, renamed (as a symbolic link) to **_APP_**. In other words, you set up a symbolic link like this:

```
$ cd <yourProjectDirectory>
$ ln −s /boot/beos/system/servers/input_server _APP_
```

And then link against **_APP_**.

Installing

The input server looks for add−ons in the "**input_server**" directory within **`B_BEOS_ADDONS_DIRECTORY`**, **`B_COMMON_ADDONS_DIRECTORY,`** and **`B_USER_ADDONS_DIRECTORY`**. Where you install your add−ons depends on what type of add−on it is:

- **input_server/devices** is for input devices
- **input_server/filters** is for input filters
- **input_server/methods** is for input methods
- You can install your input devices in the latter two directoriesi.e. those under **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.
- The **`B_BEOS_ADDONS_DIRECTORY`** is reserved for add−ons that are supplied by the BeOS.

Loading

The Input Server automatically loads (or attempts to load) all add−ons at boot time.

Currently, the Input Server doesn't dynamically load add−ons. This is a particular annoyance if you're developing and testing an add−on. To work around this lack, move your add−on into the appropriate directory, and then quit and restart the Input Server from a Terminal:

```
/system/servers/input_server −q
```

This will gracefully shutdown the Input Server and then re−launch it. The first thing the Server does when it comes back up is re−load the add−ons from its add−on directories.

Your mouse and keyboard (and other input devices) will go dead for a moment while this is happening. This is normal.

Input Server and You

The Input Server gives applications a chance to take advantage of useful features present in input devices more interesting than your typical 101−key keyboard and 3−button mouse.

Mice and Tablets

The Input Server extends the plain **`B_MOUSE_MOVED`** message (which triggers a BView>s **`MouseMoved()`** function) beyond its ordinary existence to let things like tablets pass along extra information about a user>s actions. For example, drawing tablets can track the user>s movement with greater precision than a mouse, and can include drawing pressure and tilt information. Some also include an "eraser."

If an application can do something useful with this information (and let's face it; drawing applications that respond to pressure and tilt on a drawing pad are useful as well as being cool), it>ll be present in the **B_MOUSE_MOVED** message:

```
void MyView::MouseMoved(BPoint *where, uint32 transit, BMessage *drag_msg)
{
   BMessage *moved_msg = Window()->CurrentMessage();
   ...
}
```

The extra information that a "mouse" input device could add to the **B_MOUSE_MOVED** messages includes:

- more precise position information
- drawing pressure
- pen tilt
- "eraser" mode

Precision Position Information

Tablets store the absolute position of the pointer with as much precision as they can in the *be:tablet_x* and *be:tablet_y* fields:

```
float x, y;
x = moved_msg-FindFloat( "be:tablet_x" );
y = moved_msg->FindFloat( "be:tablet_y" );
```

These entries will be scaled to the range [0.0 to 1.0].

Pressure

Tablet pressure is stored as a float in the range [0.0 to 1.0] (minimum to maximum), present in the *be:tablet_pressure* field:

```
float pressure;
pressure = moved_msg->FindFloat( "be:tablet_pressure" );
```

Tilt

Pen tilt is expressed as a pair of floats in the range [0.0 to 1.0], where (−1.0, −1.0) tilts to the left−top, (1.0, 1.0) tilts to the right−bottom, and (0.0, 0.0) is no tilt. These floats are found in the *be:tablet_tilt_x* and *be:tablet_tilt_y* fields:

```
float tilt_x, tilt_y;
tilt_x = moved_msg->FindFloat( "be:tablet_tilt_x" );
tilt_y = moved_msg->FindFloat( "be:tablet_tilt_y" );
```

Eraser Mode

The pen>s eraser mode is expressed as an int32 in the *be:tablet_eraser* field:

```
int32 erase_mode;
erase_mode = moved_msg->FindInt32( "be:tablet_eraser" );
```

A value of 1 means the pen is reversed (i.e. the eraser is on) and 0 means the pen is behaving normally. Other eraser modes may be defined in the future.

Supporting Input Methods in Views

When the user is entering text using an input method, such as the Japanese language input method that became an installation option in R4, there are two ways that applications can handle their input:

- *in−line*: the text entry interface object lets them enter text directly
- *bottom−line*: the input method itself pops up a window to accept the user>s input, and then passes **B_KEY_DOWN** messages simulating the characters to the application; the app doesn't have to do anything to support bottom−line input

If your application>s text−entry needs are met by the Interface Kit>s BTextControl and BTextView objects, it>ll automatically use the in−line mode, which gives the user a much better experience. If you>re writing your own text widget, you>ll have to do a little work to let the user input text directly.

Doing this is a very good idea; making your application behave well when dealing with foreign (to you) languages will improve your application>s acceptance around the world.

Messages from Input Methods

When interacting with an input method, your view>s **MessageReceived()** function will receive **B_INPUT_METHOD_EVENT** messages; inside is a *be:opcode* field (an int32 value) indicating the kind of event:

- **B_INPUT_METHOD_STARTED**
- **B_INPUT_METHOD_STOPPED**
- **B_INPUT_METHOD_CHANGED**
- **B_INPUT_METHOD_LOCATION_REQUEST**

B_INPUT_METHOD_STARTED tells your view that a new input transaction has begun. Inside the message is a BMessenger named *be:reply_to*; you should store this because it>s your only way of talking to the input method while the transaction is going on.

B_INPUT_METHOD_STOPPED lets you know the transaction is over; you should discard the BMessenger at this point because it>s gone stale.

In between the **B_INPUT_METHOD_STARTED** and **B_INPUT_METHOD_STOPPED** messages, you>ll receive various **B_INPUT_METHOD_CHANGED** and **B_INPUT_METHOD_LOCATION_REQUEST** messages as the transaction proceeds.

B_INPUT_METHOD_CHANGED does most of the work in an input transaction; its message contains the following important fields:

be:string	**B_STRING_TYPE**	The text the user is currently entering; display it at the insertion point. BTextView also highlights the text in blue to show that it>s part of a transitory transaction.
be:selection	**B_INT32_TYPE**	A pair of **B_INT32_TYPE** offsets into the *be:string* if any of the text *be:string* is currently selected by the user. BTextView highlights this selection in red instead of drawing it in blue.
be:clause_start	**B_INT32_TYPE**	Zero or more offsets into the *be:string* for handling languages (such as Japanese) that separate a sentence or phrase into numerous clauses. An equal number of *be:clause_start* and *be:clause_end* pairs delimit these clauses; BTextView separates the blue/red highlighting wherever there is a clause boundary.
be:clause_end	**B_INT32_TYPE**	Zero or more offsets into *be:string*; there will be as many *be:clause_end* entries as there are *be:clause_start.*
be:confirmed	**B_BOOL_TYPE**	True when the user has entered and "confirmed" the current string and wishes to end the transaction. BTextView unhighlights the blue/red text and waits for a **B_INPUT_METHOD_STOPPED** (to close the transaction) or another **B_INPUT_METHOD_CHANGED** (to start a new transaction immediately).

B_INPUT_METHOD_LOCATION_REQUEST is the input method>s way of asking you for the on−screen location of each character in your representation of the *be:string.* This information can be used by the input method to pop up additional windows giving the user an opportunity to select characters from a list or anything else that makes sense. When you get a **B_INPUT_METHOD_LOCATION_REQUEST**, reply to the *be:reply_to* messenger (that you saved from the **B_INPUT_METHOD_STARTED** message) with a **B_INPUT_METHOD_EVENT** message, filling in the following fields:

be:opcode	**B_INT32_TYPE**	Use **B_INPUT_METHOD_LOCATION_REQUEST** for the opcode.
be:location_reply	**B_POINT_TYPE**	The co−ordinates of each character (there should be one *be:location_reply* for every character in *be:string*) in screen co−ordinates not view or window co−ordinates).
be:height_reply	**B_FLOAT_TYPE**	The height of each character in *be:string*.

App and Input Events

If you're writing an application and want to record or react to input events without writing an Input Server add−on (which, of course, requires an Input Server restart), you can:

1. Create a window off−screen, at a co−ordinate like (−10.0, −10.0).

2. Add a view to the window at (0.0, 0.0).

3. **Show()** and then **Hide()** the window; this is necessary or the App Server won't send you any messages.

4. Move the hidden window to (0.0, 0.0).

5. Implement the window's **DispatchMessage()** function to handle **B_KEY_DOWN**, **B_MOUSE_UP**, or whatever other input events you're interested in observing.

Modifying these messages won't affect any other applications in the system; by the time they reach your application, they've already passed through the Input Server.

You can see this trick in action in Doug Fulton's masterful Whistle application (found at **ftp://ftp.be.com/pub/samples/midi_kit/Whistle.zip**).

BInputDevice

Derived from: none

Declared in: be/interface/InputDevice.h

Library: libbe.so

Allocation: By the system only. See

Summary

A BInputDevice object is a "downstream" representation of an Input Server device, such as a mouse or a keyboard, within a "regular" application. The BInputDevice can `Start()` and `Stop()` the device it represents, and can send it *input device control messages* through its `Control()` function.

You never create BInputDevice objects yourself; instead, you ask the system to return one or more instances to you through the `find_input_device()` or `get_input_devices()` functions. Alternatively, you can work without an object by invoking the static versions of `Start()`, `Stop()`, and `Control()`. Note, however, that the static functions control *all* devices of a given type, whereas a BInputDevice instance can talk to a specific device.

BInputDevice objects don't live in the Input Serverthey're used in "normal" applications as a means to control an Input Server device add−on.

The BInputDevice object is provided, primarily, to let an application talk to a custom input device.

You never subclass BInputDevice.

Constructor and Destructor

BInputDevice()

`The constructor is private. Use` `find_input_device()` or `get_input_devices()` to retrieve a BInputDevice instance.

~BInputDevice()

```
~BInputDevice()
```

Deletes the BInputDevice object. Deleting this object doesn't affect the device that it represents.

Member Functions

Control()

```
status_t Control(uint32 code, BMessage *message)

static status_t Control(input_device_type type,
     uint32 code,
          BMessage *message)
```

Sends an input device control message to the object's input device or, in the static version, to all devices of the given type, where, *type* can be `B_POINTING_DEVICE`, `B_KEYBOARD_DEVICE`, or `B_UNDEFINED_DEVICE`. Input devices receive these messages in their `Control()` function.

The control message is described by the *code* value; it can be supplemented or refined by *message*. For example, *code* can indicate that a certain parameter should be set, and *message* can supply the requested value.

In general, you only use this function to send custom messages to a (custom) device. You never use it to send messages to a Be−defined input device since the messages that these devices respond to are covered by Be−defined functions. See "Input Device Control Messages" for a list of the messages that the Be−defined devices respond to, the functions that cover them, and the German women who love them.

Start()

Name() , Type()

```
const char *Name(void) const

input_device_type Type(void) const
```

Name()Start() returns a pointer to the input device's name. The name, which is set when the device is registered, is meant to be human−readable and appropriate for use as the label of a UI element (such as a menu field). Device names are not unique.

Type() returns the input device's type, one of **B_POINTING_DEVICE**, **B_KEYBOARD_DEVICE**, and **B_UNDEFINED_DEVICE**.

Start() , Stop() , IsRunning()

```
status_t Start(void)

status_t Start(void)

bool IsRunning(void) const

static status_t Start(input_device_type type)

static status_t Stop(input_device_type type)
```

Start() tells the object's input device to start generating events; **Stop()** tells it to stop generating events. **IsRunning()** returns true if the device is currently generating events (i.e. if it has been started and hasn't been stopped).

The static versions of **Start()** and **Stop()** start and stop all devices of the given *type*.

RETURN CODES

B_OK. The device is now started (**Start()**) or stopped (**Stop()**)even if the device was already started or stopped.

- **B_ERROR**. The device couldn't be found.

The Input Server tells a device to start and stop without asking the device if the operation was successful. This means, for example, that **Start()** can return **B_OK** (and **IsRunning()** can return **true**) even if the device isn't really running. For the Be−provided devices this isn't an issuestarting and stopping always succeeds (as long as the device exists).

Start()

Type() see Name()

C Functions

find_input_device() , get_input_devices()

```
BInputDevice* find_input_device(const char *name)

status_t get_input_devices(BList *list)
```

These functions get BInputDevice objects for you.

find_input_device() creates and hands you a BInputDevice object that represents the Input Server device registered as *name*. If *name* is invalid, the function returns **NULL**. The caller is responsible for deleting the object. Note that **find_input_device()** returns a new BInputDevice object for each (valid) call, even if you ask for the same device more than once.

get_input_devices() creates a new BInputDevice object for each registered device, and puts the objects in your *list* argument. *list* must already be allocated, and is automatically emptied by the function (even if the function fails). If the function succeeds, the caller owns the contents, and needs to delete the items in the list:

```
#include <interface/Input.h>
#include <support/List.h>

...

static bool del_InputDevice( void *ptr )
{
   if( ptr ) {
      BInputDevice *dev = (BInputDevice *)ptr;
      delete dev;

      return false;
   }

   return true;
}

...

void SomeFunc( void )
{
   // Get a list of all input devices.
   BList list_o_devices;

   status_t retval = get_input_devices( &list_o_devices );
   if( retval != B_OK ) return;

   // Do something with the input devices.
   ...

   // Dispose of the device list.
   list_o_devices.DoForEach( del_InputDevice );
   list_o_devices.MakeEmpty();
}
```

RETURN CODES

get_input_devices() returns:

- **B_OK**. Success.
- **B_ERROR**. General failure.
- **B_BAD_PORT_ID**. Couldn't talk to the Input Server.
- **B_TIMED_OUT**. The Input Server is no longer on speaking terms with your application.
- **B_WOULD_BLOCK**. More trouble communicating with the Input Server.

watch_input_devices()

status_t **watch_input_devices(**BMessenger *target*, bool *start***)**

Tells the Input Server to start or stop watching (as *start* is **true** or **false**) for changes to the set of registered devices. Change notifications are sent to *target*. The set of messages that the Server may send are listed in **Input Server Messages**.

watch_input_devices() is not currently implemented.

BInputServerDevice

Derived from: none

Declared in: be/add−ons/input_server/InputServerDevice.h

Library: The Input Server

Allocation: By the Input Server only

Summary

BInputServerDevice is a base class for *input devices*; these are instances of BInputServerDevice subclasses that generate *input events*. In most cases, an input device corresponds to a device driver that handles a specific brand or model of hardware (mouse, keyboard, tablet, etc.), but it doesn't have to: an input device can get events from the Net, or generate them algorithmically, for example. Also, a single BInputServerDevice can handle more than one device driver.

BInputServerDevice objects are created and deleted by the Input Server onlyyou never create or delete these objects themselves.

Starting and Sending Messages

For each device that your object registers, it gets a **`Start()`** function call. This is the Input Server's way of telling your object that it can begin generating input events (for the designated device). So far, all of thisfrom the add−on load to the **`Start()`** callhappens within a single Input Server thread (for all input devices). When your **`Start()`** function is called, you should spawn a thread so your object can generate events without blocking the Server. Events are generated and sent through the **`EnqueueMessage()`** function.

Device Types and Control Messages

The Input Server knows about two types of devices: keyboards, and pointing devices (mice, tablets, etc). When you register your object's devices (through **`RegisterDevices()`**) you have to indicate the *device type*. The Input Server uses the device type to predicate the *input device control messages* it sends to the devices. These messages, delivered in **`Control()`** calls, tell a device that there's been a change downstream that applies specifically to that type of device. For example, when the user changes the mouse speed, each pointing device receives a **`B_MOUSE_SPEED_CHANGED`** notification.

The Be−defined control messages are predicated on device type only.

If your BInputServerDevice object manages a device other than a pointer or a keyboard, you tell the Input Server that the device is undefined. In this case, the Input Server won't send your device any device−specific messages; to send your device a message you (or an application that knows about your device) have to use a BInputDevice object.

Pointing Devices

Pointing devices such as mice, trackballs, drawing tablets, etc. generate **`B_MOUSE_MOVED`** messages (which trigger a BView>s **`MouseMoved()`** function) featuring a *where* field representing the cursor>s location in view co−ordinates. Unfortunately, your BInputServerDevice doesn>t know anything about views; that>s the App Server>s job. You'll still need to add this information to the **`B_MOUSE_MOVED`** messages generated by your BInputServerDevice, and the App Server will adjust it to view co−ordinates for you.

When generating a **`B_MOUSE_MOVED`** message, you add *x* and *y* fields in one of two ways:

- an offset relative to the cursor>s previous position (**`B_INT32_TYPE`** values)
- an absolute position expressed in the range 0.0 to 1.0 (**`B_FLOAT_TYPE`** values)

Mice always use relative locations; tablets can use either (though they usually provide absolute values).

Relative Locations

All mice (and some drawing tablets) express the pointer location relative to its previous position. If your pointing device is operating in relative co−ordinate mode, you add *x* and *y* entries as **`B_INT32_TYPE`** values in device−defined units. The App Server interprets these units as pixels, so you may need to scale your output:

```
int32 xVal, yVal;
...
event−>AddInt32( "x", xVal );
event−>AddInt32( "y", yVal );
```

Absolute Locations

Drawing tablets or other pointing devices that provide absolute locations add the *x* and *y* entries as **`B_FLOAT_TYPE`**s:

```
float xVal, yVal;
...
event−>AddFloat( "x", xVal );
event−>AddFloat( "y", yVal );
```

These values must be in the range [0.0 to 1.0]. The app_server scales them to the screen>s co−ordinate system so (0.0, 0.0) is the left−top, and (1.0, 1.0) is the right−bottom of the screen. This lets the pointing device work with any screen resolution, automatically.

Now where?

When the Application Server receives one of these **`B_MOUSE_MOVED`** messages, it converts the *x* and *y* values into absolute values in the target view>s co−ordinate system, and then throws away the *x* and *y* entries in the message. Because of this, and the fact that some applications might want more accurate positional information from tablets, fill in the *be:tablet_x* and *be:tablet_y* fields as well:

```
float xVal, yVal;
...
event->AddFloat( "x", xVal );
event->AddFloat( "y", yVal );
event->AddFloat( "be:tablet_x", xVal );
event->AddFloat( "be:tablet_y", yVal );
```

Other Useful Information

Pressure information is stored in the *be:tablet_pressure* field, as a float in the range [0.0 to 1.0] (minimum pressure to maximum pressure):

```
float pressure;
...
event->AddFloat( "be:tablet_pressure", pressure );
```

If the tablet supports tilt information, store it in *be:tablet_tilt_x* and *be:tablet_tilt_y*, scaling the information to the range [0.0 to 1.0]. A tilt of (−1.0, −1.0) tilts to the left−top, (1.0, 1.0) tilts to the right−bottom, and (0.0, 0.0) is no tilt.

```
float tilt_x, tilt_y;
...
event->AddFloat( "be:tablet_tilt_x", tilt_x );
event->AddFloat( "be:tablet_tilt_y", tilt_y );
```

Tablets with pens that support an eraser store the eraser>s state in the *be:tablet_eraser* field. A value of 1 means the pen is reversed (i.e. the eraser is on), and 0 means it should behave normally.

```
int32 erase_mode;
...
event->AddInt32( "be:tablet_eraser", erase_mode );
```

Device State

The **`Control()`** protocol is designed to accommodate queries (in addition to commands). Currently, however, the Input Server maintains the keyboard and pointing device state and answers these queries itself; it doesn't forward any of the Be−defined query messages. For example, when an application asks for the current mouse speed setting (through **`get_mouse_speed()`**), the query gets no further than the Input Server itselfit doesn't get passed as a control message to a pointing device.

If you're designing a BInputServerDevice that manages a keyboard or pointing device, you must keep in mind that your device is not responsible for its "Be−defined" state. The elements of the statemouse speed, key map, etc.correspond to the control messages listed in "Input Device Control Messages".

Dynamic Devices

As hardware devices are attached and detached from the computer, you can add and remove items from your BInputServerDevice's list of registered devices (by calling **`RegisterDevice()`**/**`UnregisterDevice()`**). But your object has to first *notice* that a physical device has been added or removed. It does this by placing a node monitor on the device directory (**/dev**). As a convenienceand to help conserve resourcesthe BInputServerDevice class provides the **`Start`**/**`StopMonitoringDevices()`** functions which install and remove node monitors for you.

Creating and Registering

To create a new input device, you must:

- create a subclass of BInputServerDevice
- implement the **`instantiate_input_device()`** C function to create an instance of your BInputServerDevice subclass
- compile the class and the function as an add−on
- install the add−on in one of the *input device directories*

At boot time, the Input Server loads the add−ons it finds in the input device directories. For each add−on it loads, the Server invokes **`instantiate_input_device()`** to get a pointer to the add−on's BInputServerDevice object. After constructing the object, the Server calls **`InitCheck()`** to give the add−on a chance to bail out if the constructor failed. If the add−on wants to continue, it calls **`RegisterDevices()`** (from within **`InitCheck()`**) to tell the Server which physical or virtual devices it handles.

Installing an Input Device

The input server looks for input devices in the "**input_server/devices**" subdirectories of **`B_BEOS_ADDONS_DIRECTORY`**, **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.

- You can install your input devices in the latter two directoriesi.e. those under **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.
- The **`B_BEOS_ADDONS_DIRECTORY`** is reserved for add−ons that are supplied by the BeOS.

Hook Functions

- **`Control()`**

- `InitCheck()`
- `Start()`
- `Stop()`
- `SystemShuttingDown()`

Constructor and Destructor

BInputServerDevice()

```
BInputServerDevice(void)
```

Creates a new BInputServerDevice object. You can initialize your objectset initial values, spawn (but not necessarily resume; do that in `Start()`) threads, open drivers, etc.either here or in the `InitCheck()` function, which is called immediately after the constructor.

~BInputServerDevice()

```
~BInputServerDevice()
```

Deletes the BInputServerDevice object. The destructor is invoked by the Input Server onlyyou never delete a BInputServerDevice object from your own code. When the destructor is called, the object's devices will already be unregistered and `Stop()` will already have been called. If this object spawned its own threads or allocated memory on the heap, it must clean up after itself here.

Member Functions

Control()

```
virtual status_t Control(const char *name,
     void *cookie,
     uint32 command,
          BMessage *message)
```

The `Control()` hook function is invoked by the Input Server to send an *input device control message* or a Node Monitor message to this object. *name* and *cookie* are the readable name and pointer−to−whatever−you−want that you used when registering the device (with the `RegisterDevices()` function).

The function's return value is ignored.

Input Device Control Messages

An input device control message is sent when a downstream change needs to be propagated to an input device. For example, when the user resets the mouse speed (through the **Mouse** preference), a `B_MOUSE_SPEED_CHANGED` control message is sent to all objects that have registered a `B_POINTING_DEVICE` device (see `RegisterDevices()`). *name* and *cookie* identify the device that this message applies to. The control message itself is represented by the *command* constant, optionally supplemented by *message*.

See "Input Device Control Messages" for a list of the control messages that the BeOS defines, and instructions for how to respond to them. An application can send a custom control message through a BInputDevice object; see `BInputDevice::Control()` for details.

Node Monitor Messages

A Node Monitor message is sent if an entry is added to or removed from one of the device directories that the object is monitoring, as set through `StartMonitoringDevice()`. In this case, *name* and *cookie* are `NULL`, command is `B_NODE_MONITOR`, and *message* describes the file that was added or deleted. The message's `opcode` field will be `B_ENTRY_CREATED` or `B_ENTRY_REMOVED` (or, potentially but nonsensically, `B_ENTRY_MOVED`). For instructions on how to read these messages, see "The Node Monitor" in the Storage Kit (or click on the opcode constants).

EnqueueMessage()

status_t **EnqueueMessage(**BMessage **message***)**

Sends an event message to the Input Server, which passes it through the input methods and input filters before sending it to the App Server. The message you create should be appropriate for the action you're trying to depict. For example, if the user presses a key, you should create and send a **B_KEY_DOWN** message. A list of the system−defined event messages that an input device is expected to create and send is given in "Input Device Event Messages".

RETURN CODES

B_OK. The message was sent.

- *Anything else*. The connection to the App Server has been brokenthis isn't good, and you may want to check the **is_computer_on_fire()** function found in the Kernel Kit.

InitCheck()

virtual status_t **InitCheck(**void**)**

Invoked by the Input Server immediately after the object is constructed to test the validity of the initialization. If the object is properly initialized (i.e. all required resources are located or allocated), this function should return **B_OK**. **Start()** will be invoked soon if you need to do any extra initialization. If the object returns non−**B_OK**, the object is deleted and the add−on is unloaded.

The default implementation returns **B_OK**.

RegisterDevices() , UnregisterDevices()

status_t **RegisterDevices(**input_device_ref ***devices***)**

status_t **UnregisterDevices(**input_device_ref ***devices***)**

RegisterDevices() tells the Input Server that this object is responsible for the listed *devices*. This means that when a control message is sent back upstream, the messagewhich is tagged as being relevant for a specific device, or type of devicewill be forwarded (through the **Control()** hook) to the responsible BInputServerDevice object(s). Typically, you initially register your devices as part of the constructor or **InitCheck()**. Registration is cumulativeeach **RegisterDevices()** call adds to the object's current list of devices.

UnregisterDevices() tells the Input Server that this object is no longer responsible for the listed devices. The devices are automatically unregistered when your object is deleted.

RegisterDevices() invokes **Start()** for each device in the *devices* list; **UnregisterDevices()** invokes **Stop()**.

For both functions, the *devices* list must be **NULL**−terminated, and the caller retains ownership of the list and its contents.

Note that the BeOS currently only targets the device types when sending a **Control()** message. For example, let's say you've registered two pointing devices and a keyboard:

```
status_t MyISDevice::InitCheck()
{
...
   input_device_ref **devices =
      (input_device_ref **)malloc(sizeof(*input_device_ref * 4));
   input_device_ref mouse1 = {"Mouse 1", B_POINTING_DEVICE,
                              (void *)this)};
   input_device_ref mouse2 = {"Mouse 2", B_POINTING_DEVICE,
                              (void *)this)};
   input_device_ref keyboard = {"Keyboard", B_KEYBOARD_DEVICE,
                                (void *)this)};
   devices[0] = &mouse1;
   devices[1] = &mouse2;
   devices[2] = &keyboard;
   devices[3] = NULL;
   RegisterDevices(devices);
   ...
}
```

When the user fiddles with the **Mouse** preference (more specifically, if an application calls **set_mouse_speed()** et. al.), this object will receive two **Control()** messages: one targets "Mouse 1", and the other targets "Mouse 2". That's because the mouse and keyboard functions (as defined by the BeOS and as used by the system preferences) know which type of device to control, but they don't provide a means for more granular identification. If you need a UI that identifies specific devices, you have to create the UI yourself, and use a BInputDevice object to tune the control messages that are sent back upstream.

RETURN CODES

B_OK. At least one of the devices was registered.

- **B_ERROR**. None of the devices were registered.

> The functions don't let you un/register the same device definition twice, and **RegisterDevices()** won't register a device that doesn't have a name (although the name can be **""**). However, the functions don't complain about violations of these conditions as long as at least one definition is properly formed.

Start() , Stop()

virtual status_t **Start(**const char **name*, void **cookie***)**

virtual status_t **Stop(**const char **name*, void **cookie***)**

Start() is invoked by the Input Server to tell the object that it can begin sending events for the registered device identified by the arguments. The values of the arguments are taken from the **input_device_ref** structure you used to register the device (see **RegisterDevices()**). If your object needs to resume a thread (spawned in the constructor, in **InitCheck()**, or here), this is the place to do it.

Stop() is invoked to tell the object to stop sending events for the registered device. The device is not unregisteredyou can still receive **Control()** messages for the device while it's stopped. You should pause or kill any threads associated with the device (that were spawned by this object) from here.

> The return value (for both of these functions) is ignored.

StartMonitoringDevice() , StopMonitoringDevice()

status_t **StartMonitoringDevice(**const char **deviceDir***)**

status_t **StopMonitoringDevice(**const char **deviceDir***)**

These are convenient covers for the Node Monitor's **watch_node()** and **stop_watching()** functions. You use them to watch for physical devices that are attached and detached, as indicated by changes to subdirectories of the system device directory (**/dev**).

deviceDir is the name of the device subdirectory that you want to watch. The "**/dev/**" root is automatically prepended; for example, if you want to watch for new ps2 mice, you would pass "**input/mouse/ps2**" as the *deviceDir* name. The Node Monitor is told to look for changes to the directory (**B_WATCH_DIRECTORY** opcode). When an entry is added or removed, this object receives a **B_NODE_MONITOR** message delivered to its **Control()** function.

RETURN CODES

B_OK. Success.

- **B_ERROR**. Unspecified failure.
- **B_NOT_A_DIRECTORY**. You're trying to monitor a node that isn't a directory.
- **B_BAD_VALUE**. *deviceDir* not found.

SystemShuttingDown()

virtual status_t **SystemShuttingDown(**void**)** const

Tells the object that the Input Server is in the process of shutting down. Unless something interrupts the shutdown, this notification will be followed by a **Stop()** and **delete**, thus you don't have to do much from this function (other than note that the end is near).

The return value is ignored.

UnregisterDevices see RegisterDevices()

BInputServerFilter

Derived from: none

Declared in: be/add−ons/input_server/InputServerFilter.h

Library: libbe.so

Allocation: By the Input Server only

Summary

BInputServerFilter is a base class for *input filters*; these are instances of BInputServerFilter that modify, generate, or eat *input events*. An input filter add−on is privy to all the events that pass through the Input Server>s event stream. A filter is similar to the Interface Kit>s BMessageFilter, but at a much lower level. The BInputServerFilter also sees *all* events, while a BMessageFilter only sees the events targeted at its BLooper. BInputServerFilters can also generate additional events in place of, or in addition to, the original input event.

BInputServerFilter objects are created and deleted by the Input Server onlyyou never create or delete these objects in your code.

Creating

To create a new input filter, you must:

- create a subclass of BInputServerFilter
- implement the **`instantiate_input_filter()`** C function to create an instance of your BInputServerFilter subclass
- compile the class and function as an add−on
- install the add−on in one of the input filter directories

At boot time (or whenever the Input Server is restarted; see "Loading" in **The Input Server**), the Input Server loads the add−ons it finds in the input filter directories. For each add−on it finds, the Server invokes **`instantiate_input_filter()`** to get a pointer to the add−ons>s BInputServerFilter object. After constructing the object, the Server calls **`InitCheck()`** to give the add−on a chance to bail out if the constructor failed.

Installing an Input Filter

The input server looks for input filters in the "**input_server/filters**" subdirectories of **`B_BEOS_ADDONS_DIRECTORY`**, **`B_COMMON_ADDONS_DIRECTORY,`** and **`B_USER_ADDONS_DIRECTORY`**.

- You can install your input devices in the latter two directoriesi.e. those under **`B_COMMON_ADDONS_DIRECTORY`**, and **`B_USER_ADDONS_DIRECTORY`**.
- The **`B_BEOS_ADDONS_DIRECTORY`** is reserved for add−ons that are supplied with BeOS.

Hook Functions

- **`Filter()`**
- **`InitCheck()`**

Constructor and Destructor

BInputServerFilter()

```
BInputServerFilter(void)
```

Creates a new BInputServerFilter object. You can initialize the objectset initial values, spawn threads, etc.either here or in the **`InitCheck()`** function, which is called immediately after the constructor.

~BInputServerFilter()

```
~BInputServerFilter()
```

Deletes the BInputServerFilter object. The destructor is invoked by the Input Server onlyyou never delete a BInputServerFilter object from your own code. If this object has spawned its own threads or allocated memory on the heap, it must clean up after itself here.

Member Functions

Filter()

virtual filter_result Filter(BMessage **message*, BList **outList*)

The **Filter()** hook function is invoked by the Input Server to filter the in−coming *message*. You can add fields to message, remove them, or otherwise modify the object. When you're finished with the message, your implementation of **Filter()** should return **B_SKIP_MESSAGE** if this input event message should be dropped (or replaced by *outList*), or **B_DISPATCH_MESSAGE** if it should continue through the Input Server towards its destination.

To insert new messages into the event stream, fill the empty *outList* with pointers to new BMessage objects filled in with the proper input event fields, and return **B_SKIP_MESSAGE**. The Input Server will ignore *message* if you use *outList*, so you'll need to make a copy if you want it to continue downstream:

```
BMessage *new_message = new BMessage( message );
outList->AddItem( new_message );
```

The Input Server owns all of the messages you pass back in *outList*, so don't delete them yourself. Events added via *outList* are processed in the order they appear in the list and are inserted into the event stream in place of *message*. If **Filter()** returns **B_DISPATCH_MESSAGE**, messages in *outList* are ignored.

The default implementation returns **B_DISPATCH_MESSAGE** without modifying the *message*.

GetScreenRegion()

status_t GetScreenRegion(BRegion **region*) const

The **GetScreenRegion()** function returns the screen's region in *region*. This is the most efficient way for an input filter to get the screen's region. The system screen saver's input filter uses this for its "sleep now"/"sleep never" corners.

GetScreenRegion() returns **B_OK**.

InputCheck()

virtual status_t InitCheck(void)

Invoked by the Input Server immediately after the object is constructed to test the validity of the initialization. If the object is properly initialized (i.e. all required resources are located or allocated), this function should return **B_OK**. If the object returns non−**B_OK**, the object is deleted and the add−on is unloaded.

The default implementation returns **B_OK**.

BInputServerMethod

Derived from: BInputServerFilter

Declared in: be/add−ons/input_server/InputServerMethod.h

Library: libbe.so

Allocation: By the Input Server only

Summary

BInputServerMethod is a base class for *input methods*; these are instances of BInputServerMethod that act as an interface between the user and languages using character sets that can>t be easily represented on standard keyboards, such as the Japanese input method that comes with BeOS.

Input methods generally handle **B_KEY_DOWN** messages in their **Filter()** function (see BInputServerFilter), keeping some sort of state around to translate these standard keyboard messages into new **B_KEY_DOWN** messages representing another character set. An input method can handle any input event, they>re not limited to keyboard events.

Writing an input method is an involved process, even though the BInputMethod protocol is relatively simple. If you>re working on an input method, please feel free to contact Be Developer Technical Support (devsupport@be.com) for additional information.

Input Method Events

Input methods insert **B_INPUT_METHOD_EVENT** messages (using their **EnqueueMessage()** function) into the Input Server>s event stream. These messages let BView subclasses work together with your input method to create a seamless experience for the user.

Each **B_INPUT_METHOD_EVENT** message contains a *be:opcode* field (an int32 value) indicating the kind of event:

- **B_INPUT_METHOD_STARTED**
- **B_INPUT_METHOD_STOPPED**
- **B_INPUT_METHOD_CHANGED**
- **B_INPUT_METHOD_LOCATION_REQUEST**

B_INPUT_METHOD_STARTED indicates that a new input transaction has begun. Add a BMessenger in the *be:reply_to* field; the receiver of the message will use this messenger to communicate with you during the transaction.

B_INPUT_METHOD_STOPPED indicates that the transaction is over.

In between the **B_INPUT_METHOD_STARTED** and **B_INPUT_METHOD_STOPPED** messages, you>ll send **B_INPUT_METHOD_CHANGED** and **B_INPUT_METHOD_LOCATION_REQUEST** messages as the transaction proceeds.

B_INPUT_METHOD_CHANGED does most of the work in an input transaction; add the following important fields:

be:string	**B_STRING_TYPE**	The text the user is currently entering; the receiver will display it at the current insertion point. BTextView also highlights the text in blue to show that it>s part of a transitory transaction.
be:selection	**B_INT32_TYPE**	A pair of **B_INT32_TYPE** offsets into the *be:string* if part of *be:string* is current selected. BTextView highlights this selection in red instead of drawing it in blue.
be:clause_start	**B_INT32_TYPE**	Zero or more offsets into the *be:string* for handling languages (such as Japanese) that separate a sentence or phrase into numerous clauses. An equal number of *be:clause_start* and *be:clause_end* pairs delimit these clauses; BTextView separates the blue/red highlighting wherever there is a clause boundary.
be:clause_end	**B_INT32_TYPE**	Zero or more offsets into *be:string*; there must be as many *be:clause_end* entries as there are *be:clause_start*.
be:confirmed	**B_BOOL_TYPE**	True when the user has entered and "confirmed" the current string and wishes to end the transaction. BTextView unhighlights the blue/red text and waits for a **B_INPUT_METHOD_STOPPED** (to close the transaction) or another **B_INPUT_METHOD_CHANGED** (to start a new transaction immediately).

B_INPUT_METHOD_LOCATION_REQUEST is the input method>s way of asking for the on−screen location of each character in *be:string*. This information can be used by the input method to pop up additional windows giving the user an opportunity to select characters from a list or anything else that makes sense. When you send a **B_INPUT_METHOD_LOCATION_REQUEST**, the receiver will reply to the *be:reply_to* messenger (that you sent in your **B_INPUT_METHOD_STARTED** message) with a **B_INPUT_METHOD_EVENT** message, filling in the following fields:

be:opcode	**B_INT32_TYPE**	Set to **B_INPUT_METHOD_LOCATION_REQUEST**.

be:location_reply	**B_POINT_TYPE**	The co−ordinates of each character (there should be one *be:location_reply* for every character in *be:string*) relative to the display (not your view or your window).
be:height_reply	**B_FLOAT_TYPE**	The height of each character in *be:string*.

Creating

To create a new input method, you must:

- create a subclass of BInputServerMethod
- implement the **instantiate_input_method()** C function to create an instance of your BInputServerMethod subclass
- compile the class and function as an add−on
- install the add−on in one of the input method directories

At boot time (or whenever the Input Server is restarted; see "Dynamic Loading"), the Input Server loads the add−ons it finds in the input method directories. For each add−on it finds, the Server invokes **instantiate_input_method()** to get a pointer to the add−on>s BInputServerMethod object. After constructing the object, the Server calls **InitCheck()** to give the add−on a chance to bail out if the constructor failed.

Installing an Input Method

The input server looks for input methods in the "**input_server/methods**" subdirectories of **B_BEOS_ADDONS_DIRECTORY**, **B_COMMON_ADDONS_DIRECTORY,** and **B_USER_ADDONS_DIRECTORY**.

- You can install your input devices in the latter two directoriesi.e. those under **B_COMMON_ADDONS_DIRECTORY**, and **B_USER_ADDONS_DIRECTORY**.
- The **B_BEOS_ADDONS_DIRECTORY** is reserved for add−ons that are supplied by the BeOS.

Hook Functions

- **MethodActivated()**

Constructor and Destructor

BInputServerMethod()

BInputServerMethod(const char **name*, const uchar **icon***)**

Creates a new BInputServerMethod object. You can initialize the objectset initial values, spawn threads, etc.either here or in the **InitCheck()** function, which is called immediately after the constructor.

name is a textual name describing the input method, and *icon* is the raw data for a 16x16 8−bit icon built from the standard BeOS palette. This *name* and *icon* will be displayed in the input method menu (the little keyboard icon in the Deskbar). When the user selects your input method from the menu, your **MethodActivated()** function is called.

~BInputServerMethod()

~BInputServerMethod()

Deletes the BInputServerMethod object. The destructor is invoked by the Input Server onlyyou never delete a BInputServerMethod object from your own code. If this object has spawned its own threads or allocated memory on the heap, it must clean up after itself here.

Member Functions

EnqueueMessage()

status_t EnqueueMessage(BMessage **message***)**

Inserts the specified *message* into the Input Server>s event queue; the message continues down−stream from this point, passing through additional active input methods and input filters on its way to the App Server.

MethodActivated()

virtual status_t MethodActivated(bool *active*)

The MethodActivated() hook function is invoked by the Input Server when the user activates your input method (*active* is true) or deactivates it (*active* is false). This is your chance to activate any helper threads or loopers to the fact that you>ll be handling input events soon.

Return **B_OK** if that>s OK, or something else if it>s not. The default implementation returns **B_OK**.

SetName()

status_t SetName(const char **name*)

Changes your input method>s name, as found in the input method menu of the Deskbar.

The Input Server makes a copy of *name*.

SetIcon()

status_t SetIcon(const uchar **icon*)

Changes your input method>s icon, as found in the input method menu of the Deskbar. icon should be raw data for a 16x16 8−bit icon built from the standard BeOS palette.

The Input Server makes a copy of *icon*.

SetMenu()

status_t SetMenu(const BMenu **menu*, const BMessenger *target*)

Lets you assign a *menu* to your input method>s entry in the input method menu of the Deskbar. Messages generated by the menu are sent to *target*.

Passing a **NULL** menu disables your input method>s menu.

You retain ownership of *menu*; be careful not to delete it while the Input Server is still using it.

Input Server Messages

This section describes the messages that your Input Server objects are expected to create and send, and that they're expected to respond to.

Input Device Event Messages

Declared in: be/app/AppDefs.h

This section lists the event messages that a BInputServerDevice is expected to create and send through its **`EnqueueMessage()`** function. The primary documentation for these messages is in the System Messages appendix (click on an item in the lists below to be taken to a specific definition).

Pointing Device Event Messages

`B_MOUSE_DOWN`
`B_MOUSE_UP`
`B_MOUSE_MOVED`

Note that a pointing device isn't expected to send the **`B_MOUSE_ENTER_EXIT`** message.

Keyboard Device Event Messages

`B_KEY_DOWN`
`B_UNMAPPED_KEY_DOWN`
`B_KEY_UP`
`B_UMAPPED_KEY_UP`
`B_MODIFIERS_CHANGED`

Input Device Control Messages

Declared in: be/add−ons/input_server/InputServerDevice.h

This section lists the control messages that are defined by the BeOS for pointing and keyboard devices. These are messages that appear in the **`BInputServerDevice::Control()`** function. Each control message is identified by the value that appears as the *command* argument in the **`Control()`** function. None of the Be−defined control messages use the additional BMessage argument.

Control messages are used to notify input devices of downstream requests. For example, when the user changes the mouse speed, a **`B_MOUSE_SPEED_CHANGED`** command is sent back upstream. It's expected that an input device that receives this message will tune subsequent event messages that it generates to match the requested mouse speed.

The messages listed below are defined by the BeOS; you can send custom control messages back upstream through the **`BInput::Control()`** function. Of course, this is only effective if you install a custom input device that can handle the messages.

Note that the Be−defined control messages ask a device to set parameters (such as mouse speed), but they never ask a device for the value of a parameter. For example, a pointing device is never asked what the mouse speed is. This is because the Input Server maintains the state of the keyboard and pointing device environments and can answer these requests itself.

Furthermore, the Be−defined control messages don't contain the value of the parameter that's being set. For example, the **`B_MOUSE_SPEED_CHANGED`** message doesn't contain the requested mouse speed. The input device must ask the Input Server for the new value through a global function (**`get_mouse_speed()`**, in this case). The functions that correspond to the messages are listed in the descriptions below.

Pointing Device Control Messages

B_CLICK_SPEED_CHANGED

Requests that the receiver change the mouse double−click speed to the value retrieved through **`get_click_speed()`**.

B_MOUSE_MAP_CHANGED

Requests that the receiver change the mouse map (the correspondence between physical mouse buttons and the **`B_PRIMARY_MOUSE_BUTTON`**, et. al., constants) to the map retrieved through **`get_mouse_map()`**.

B_MOUSE_SPEED_CHANGED

Requests that the receiver change the mouse speed to the value retrieved through **`get_mouse_speed()`**.

B_MOUSE_TYPE_CHANGED

Requests that the receiver change the mouse type (the number of buttons) to the type retrieved through **`get_mouse_type()`**.

Keyboard Device Control Messages

B_KEY_LOCKS_CHANGED

Requests that the receiver change the state of the locked keys (caps lock, num lock, etc.). To get the desired state of the locking keys, read the states out of the key map returned by `get_key_map()`.

B_KEY_MAP_CHANGED

Requests that the receiver change the keyboard's key mapthe mapping between physical keys and the character codes they generate. The new key map is returned by `get_key_map()`.

B_KEY_REPEAT_DELAY_CHANGED

Requests that the receiver change the delay before a held key starts generating repeated characters to the value retrieved through `get_key_repeat_delay()`.

B_KEY_REPEAT_RATE_CHANGED

Requests that the receiver change the speed at which a held key generates repeated characters to the value retrieved through `get_key_repeat_rate()`.

Device Monitoring

The `watch_input_devices()` function lets you ask the Input Server to send you a message when a device starts or stops, or when the set of registered devices changes. These "device monitoring" notifications are sent to the target specified in the function. The command constant is always `B_INPUT_DEVICES_CHANGED`. The *be:opcode* field will be one of:

B_INPUT_DEVICE_ADDED

An input device has been added to the system.

B_INPUT_DEVICE_REMOVED

An input device has been removed from the system.

B_INPUT_DEVICE_STARTED

An input device has been started.

B_INPUT_DEVICE_STOPPED

An input device has been stopped.

Input Method Events

Active input methods send input method events (`B_INPUT_METHOD_EVENT` messages) downstream to application views to help integrate the method>s work with the view>s display. Inside each `B_INPUT_METHOD_EVENT` message is a *be:opcode* field indicating the type of input method event:

B_INPUT_METHOD_CHANGED

Sent whenever the user changes the text during an input transaction.

B_INPUT_METHOD_LOCATION_REQUEST

Sent whenever the input method needs to know the on−screen locations of characters in the input transaction.

B_INPUT_METHOD_STARTED

Sent when a new input transaction is beginning.

B_INPUT_METHOD_STOPPED

Sent when an input transaction is completed.

Input Functions

Declared in: be/interface/InterfaceDefs.h

Library: libbe.so

This section describes the global mouse and keyboard functions.

Mouse Functions

get_click_speed() see set_click_speed()

get_mouse_map() see set_mouse_map()

get_mouse_speed() see set_mouse_map()

get_mouse_acceleration() see set_mouse_acceleration()

get_mouse_type() see set_mouse_map()

set_click_speed() , get_click_speed()

Declared in: be/interface/InterfaceDefs.h

```
status_t set_click_speed(bigtime_t interval)
status_t get_click_speed(bigtime_t *interval)
```

These functions set and report the timing for multiple−clicks. For successive mouse−down events to count as a multiple−click, they must occur within the *interval* set by **`set_click_speed()`** and provided by **`get_click_speed()`**. The interval is measured in microseconds; it's usually set by the user in the Mouse preferences application. The smallest possible interval is 100,000 microseconds (0.1 second).

If successful, these functions return **`B_OK`**; if unsuccessful, they return an error code, which may be just **`B_ERROR`**.

set_mouse_map() , get_mouse_map() , set_mouse_type() , get_mouse_type() , set_mouse_speed() , get_mouse_speed()

Declared in: be/interface/InterfaceDefs.h

```
status_t set_mouse_map(mouse_map *map)
status_t get_mouse_map(mouse_map *map)
status_t set_mouse_type(int32 numButtons)
status_t get_mouse_type(int32 *numButtons)
status_t set_mouse_speed(int32 speed)
status_t get_mouse_speed(int32 *speed)
status_t set_mouse_acceleration(int32 acceleration)
status_t get_mouse_acceleration(int32 *acceleration)
```

These functions configure the mouse and supply information about the current configuration. The configuration should usually be left to the user and the Mouse preferences application.

`set_mouse_map()` maps the buttons of the mouse to their roles in the user interface, and **`get_mouse_map()`** writes the current map into the variable referred to by *map*. The **`mouse_map`** structure has a field for each button on a three−button mouse:

uint32 **`left`**

The button on the left of the mouse

uint32 **`right`**
The button on the right of the mouse

uint32 **`middle`**
The button in the middle, between the other two buttons

Each field is set to one of the following constants:

`B_SECONDARY_MOUSE_BUTTON`
`B_TERTIARY_MOUSE_BUTTON`

The same role can be assigned to more than one physical button. If all three buttons are set to **`B_PRIMARY_MOUSE_BUTTON`**, they all function as the primary button; if two of them are set to **`B_SECONDARY_MOUSE_BUTTON`**, they both function as the secondary button; and so on.

`set_mouse_type()` informs the system of how many buttons the mouse actually has. If it has two buttons, only the **`left`** and **`right`** fields of the **`mouse_map`** are operative. If it has just one button, only the **`left`** field is operative. **`set_mouse_type()`** writes the current number of buttons into the variable referred to by *numButtons*.

`set_mouse_speed()` sets the speed of the mousethe rate at which the cursor image moves on−screen relative to the actual speed at which the user moves the mouse on its pad. A *speed* value of 0 is the slowest movement rate. The maximum rate is 20, though even 10 is too fast for most users. **`get_mouse_speed()`** writes the current speed into the variable referred to by *speed*.

`set_mouse_acceleration()` sets the mouse's accelerationthe rate at which the cursor image gains and loses speed as the user begins and ceases moving the mouse. An *acceleration* value of 0 is the slowest movement rate. The maximum rate is 20, though even 10 is too fast for most users. **`get_mouse_acceleration()`** writes the current acceleration into the variable referred to by *acceleration*.

All six functions return **`B_OK`** if successful, and an error code, typically **`B_ERROR`**, if not.

Keyboard Functions

get_key_info()

Declared in: be/interface/InterfaceDefs.h

status_t **`get_key_info(`**key_info **keyInfo***`)`**

Writes information about the state of the keyboard into the **`key_info`** structure referred to by *keyInfo*. This function lets you get information about the keyboard in the absence of **`B_KEY_DOWN`** messages. The **`key_info`** structure has just two fields:

uint32 **`modifiers`**
A mask indicating which modifier keys are down and which keyboard locks are on.

uint8 **`key_states`** [16]
A bit array that records the state of all the keys on the keyboard, and all the keyboard locks. This array works identically to the "states" array passed in a key−down message. See "Key States" in the **Keyboard Information** appendix for information on how to read information from the array.

`get_key_info()` returns **`B_OK`** if it was able to get the requested information, and **`B_ERROR`** if the return results are unreliable.

See also: `BView::KeyDown()`, the **Keyboard Information** appendix, **`modifiers()`**

get_key_map()

Declared in: be/interface/InterfaceDefs.h

void **`get_key_map(`**key_map ***keys*, char ***chars***`)`**

Provides a pointer to a copy of the system key mapthe structure that describes the role of each key on the keyboard. The pointers returned by the function are yours; you must **`free()`** them when you're finished with them.

In versions of the BeOS before Release 4, the pointers used to belong to the operating system. Now they're yours to do with as you please. Please update your applications as necessary to avoid leaking memory.

Through the Keymap preferences application, users can configure the keyboard to their liking. The user's preferences are stored in a file (**Key_map** within the `B_USER_SETTINGS_DIRECTORY`, returned by the `find_directory()` function). When the machine reboots, the key map is read from this file. If the file doesn't exist, the original map encoded in the Application Server is used.

The `key_map` structure contains a large number of fields, but it can be broken down into these six parts:

- A version number.
- A series of fields that determine which keys will function as modifier keyssuch as Shift, Control, or Num Lock.
- A field that sets the initial state of the keyboard locks in the default key map.
- A series of ordered tables that assign character values to keys. Except for a handful of modifier keys, all keys are mapped to characters, though they may not be mapped for all modifier combinations.
- A series of tables that locate the dead keys for diacritical marks and determine how a combination of a dead key plus another key is mapped to a particular character.
- A set of masks that determine which modifier keys are required for a key to be considered dead.

The following sections describe the parts of the `key_map` structure.

Version

The first field of the key map is a version number:

uint32 **`version`**
An internal identifier for the key map.

The version number doesn't change when the user configures the keyboard, and shouldn't be changed programmatically either. You can ignore it.

Modifiers

Modifier keys set states that affect other user actions on the keyboard and mouse. Eight modifier states are definedShift, Control, Option, Command, Menu, Caps Lock, Num Lock, and Scroll Lock. These states are discussed under "Modifier Keys" in the **Keyboard Information** appendix. They fairly closely match the key caps found on a Macintosh keyboard, but only partially match those on a standard PC keyboardwhich generally has a set of Alt(ernate) keys, rarely Option keys, and only sometimes Command and Menu keys. Because of these differences, the mapping of keys to modifiers is the area of the key map most open to the user's personal judgement and taste, and consequently to changes in the default configuration.

Since two keys, one on the left and one on the right, can be mapped to the Shift, Control, Option, and Command modifiers, the keyboard can have as many as twelve modifier keys. The `key_map` structure has one field for each key:

uint32 **`caps_key`**
The key that functions as the Caps Lock key; by default, this is the key labeled "Caps Lock," key 0x3b.

uint32 **`scroll_key`**
The key that functions as the Scroll Lock key; by default, this is the key labeled "Scroll Lock," key 0x0f.

uint32 **`num_key`**
The key that functions as the Num Lock key; by default, this is the key labeled "Num Lock," key 0x22.

uint32 **`left_shift_key`**
A key that functions as a Shift key; by default, this is the key on the left labeled "Shift," key 0x4b.

uint32 **`right_shift_key`**
Another key that functions as a Shift key; by default, this is the key on the right labeled "Shift," key 0x56.

uint32 **`left_command_key`**
A key that functions as a Command key; by default, this is key 0x5d, sometimes labeled "Alt."

uint32 **`right_command_key`**
Another key that functions as a Command key; by default, this is key 0x5f, sometimes labeled "Alt."

uint32 **`left_control_key`**
A key that functions as a Control key; by default, this is the key labeled "Control" on the left, key 0x5c.

uint32 **`right_control_key`**
Another key that functions as a Control key; by default on keyboards that have Option keys, this key is the key labeled "Control" on the right, key 0x60. For keyboards that don't have Option keys, this field is unmapped (its value is 0); key 0x60 is used as an Option key.

uint32 **`left_option_key`**
A key that functions as an Option key; by default, this is key 0x66, which has different labels on different keyboards"Option," "Command," or a Windows symbol. This key doesn't exist on, and therefore isn't mapped for, a standard 101−key keyboard.

uint32 **`right_option_key`**
A key that functions as an Option key; by default, this is key 0x67, which has different labels on different keyboards"Option," "Command," or a Windows symbol. For keyboards without this key, the field is mapped to the key labeled "Control" on the right, key 0x60.

uint32 **`menu_key`**
A key that initiates keyboard navigation of the menu hierarchy; by default, this is the key labeled with a menu symbol, key 0x68. This key doesn't exist on, and therefore isn't mapped for, a standard 101−key keyboard.

Each field names the key that functions as that modifier. For example, when the user holds down the key whose code is set in the **`right_option_key`** field, the `B_OPTION_KEY` and `B_RIGHT_OPTION_KEY` bits are turned on in the modifiers mask that the `modifiers()` function returns. When the user then strikes a character key, the `B_OPTION_KEY` state influences the character that's generated.

If a modifier field is set to a value that doesn't correspond to an actual key on the keyboard (including 0), that field is not mapped. No key fills that particular modifier role.

Keyboard locks

One field of the key map sets initial modifier states:

uint32 **lock_settings**
A mask that determines which keyboard locks are turned on when the machine reboots or when the default key map is restored.

The mask can be 0 or may contain any combination of these three constants:

B_SCROLL_LOCK
B_NUM_LOCK

It's 0 by default; there are no initial locks.

Altering the **lock_settings** field has no effect unless the altered key map is made the default.

Character maps

The principal job of the key map is to assign character values to keys. This is done in a series of nine tables:

int32 **control_map** [128]
The characters that are produced when a Control key is down but both Command keys are up.

int32 **option_caps_shift_map** [128]
The characters that are produced when Caps Lock is on and both a Shift key and an Option key are down.

int32 **option_caps_map** [128]
The characters that are produced when Caps Lock is on and an Option key is down.

int32 **option_shift_map** [128]
The characters that are produced when both a Shift key and an Option key are down.

int32 **option_map** [128]
The characters that are produced when an Option key is down.

int32 **caps_shift_map** [128]
The characters that are produced when Caps Lock is on and a Shift key is down.

int32 **caps_map** [128]
The characters that are produced when Caps Lock is on.

int32 **shift_map** [128]
The characters that are produced when a Shift key is down.

int32 **normal_map** [128]
The characters that are produced when none of the other tables apply.

Each of these tables is an array of 128 offsets into another array, the *chars* array of Unicode UTF−8 character encodings. **get_key_map()** provides a pointer to the *chars* array as its second argument.

Key codes are used as indices into the character tables. The offset stored at any particular index maps a character to that key. For example, the code assigned to the *M* key is 0x52; at index 0x52 in the **option_caps_map** is an offset; at that offset in the *chars* array, you'll find the character that's mapped to the *M* key when an Option key is held down and Caps Lock is on.

This indirectionan index to an offset to a characteris required because characters are encoded as Unicode UTF−8 strings. Character values of 127 or less (7−bit ASCII) are just a single byte, but UTF−8 takes two, three, or (rarely) four bytes to encode values over 127.

The *chars* array represents each character as a Pascal stringthe first byte in the string tells how many other bytes the string contains. For example, the string for the trademark symbol ((TM)) looks like this:

```
x03xE2x84xA2
```

The first byte (x03) indicates that Unicode UTF−8 takes 3 bytes to represent the trademark symbol, and those bytes follow (xE2x84xA2). Pascal strings are not null−terminated.

Here's an example showing you how to decode the character tables. This sample prints out a simple chart of the **normal_map**, **shift_map**, **option_map**, and **option_shift_map** characters:

```
#include <interface/InterfaceDefs.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void print_key( char *chars, int32 offset )
{
   int size = chars[offset++];

   switch( size ) {
   case 0:
      // Not mapped
      printf( "N/A" );
```

```
         break;

      case 1:
         // 1−byte UTF−8/ASCII character
         printf( "%c", chars[offset] );
         break;

      default:
         // 2−, 3−, or 4−byte UTF−8 character
         {
            char *str = new char[size + 1];
            strncpy( str, &(chars[offset]), size );
            printf( "%s", str );
            delete [] str;
         }
         break;
      }

      printf( "t" );
   }

   int main( void )
   {
      // Get the current key map.
      key_map *keys;
      char *chars;
      get_key_map( &keys, &chars );

      // Print a chart of the normal, shift, option, and option+shift
      // keys.
      printf( "Key #tNormaltShifttOptiontOption+Shiftn" );
      for( int idx = 0; idx < 128; idx++ ) {
         printf( " %3dt", idx );
         print_key( chars, keys−>normal_map[idx] );
         print_key( chars, keys−>shift_map[idx] );
         print_key( chars, keys−>option_map[idx] );
         print_key( chars, keys−>option_shift_map[idx] );
         printf( "n" );
      }

      // Free our copy of the key map.
      free( chars );
      free( keys );

      return EXIT_SUCCESS;
   }
```

The character map tables are ordered. Values from the first applicable table are used, even if another table might also seem to apply. For example, if Caps Lock is on and a Control key is down (and both Command keys are up), the **control_map** array is used, not **caps_map**. If a Shift key is down and Caps Lock is on, the **caps_shift_map** is used, not **shift_map** or **caps_map**.

Notice that the last eight tables (all except **control_map**) are paired, with a table that names the Shift key (...**_shift_map**) preceding an equivalent table without Shift:

- **option_caps_shift_map** is paired with **option_caps_map**,
- **option_shift_map** with **option_map**,
- **caps_shift_map** with **caps_map**, and
- **shift_map** with **normal_map**.

These pairings are important for a special rule that applies to keys on the numerical keypad when Num Lock is on:

- If the Shift key is down, the non−Shift table is used.
- However, if the Shift key is *not* down, the Shift table is used.

In other words, Num Lock inverts the Shift and non−Shift tables for keys on the numerical keypad.

Not every key needs to be mapped to a character. If the *chars* array has a 0−length string for a key, the key is not mapped to a character (given the particular modifier states the table represents). Generally, modifier keys are not mapped to characters, but all other keys are, at least for some tables. Key−down events are not generated for unmapped keys.

Dead keys

Next are the tables that map combinations of keys to single characters. The first key in the combination is "dead"it doesn't produce a key−down event until the user strikes another character key. When the user hits the second key, one of two things will happen: If the second key is one that can be used in combination with the dead key, a single key−down event reports the combination character. If the second key doesn't combine with the dead key, two key−down events occur, one reporting the dead−key character and one reporting the second character.

There are five dead−key tables:

int32 **acute_dead_key** [32]
The table for combining an acute accent (xab) with other characters.

int32 **grave_dead_key** [32]
The table for combining a grave accent (Q) with other characters.

int32 **circumflex_dead_key** [32]
The table for combining a circumflex (xf6) with other characters.

int32 **dieresis_dead_key** [32]
The table for combining a dieresis (xac) with other characters.

int32 **`tilde_dead_key`** [32]
The table for combining a tilde (xf7) with other characters

The tables are named after diacritical marks that can be placed on more than one character. However, the name is just a mnemonic; it means nothing. The contents of the table determine what the dead key is and how it combines with other characters. It would be possible, for example, to remap the **`tilde_dead_key`** table so that it had nothing to do with a tilde.

Each table consists of a series of up to 16 offset pairswhere, as in the case of the character maps, each offset picks a character from the *chars* character array. The first character in the pair is the one that must be typed immediately after the dead key. The second character is the resulting character, the character that's produced by the combination of the dead key plus the first character in the pair. For example, if the first character is 'o', the second might be 'ô'meaning that the combination of a dead key plus the character 'o' produces a circumflexed 'ô'.

The character pairs for the default **`grave_dead_key`** array look something like this:

```
> >, >'>,
>A>, >À>,
>E>, >È>,
>I>, >Ì>,
>O>, >Ò>,
>U>, >Ù>,
>a>, >à>,
>e>, >è>,
>i>, >ì>,
>o>, >ò>,
>u>, >ù>,
. . .
```

By convention, the first offset in each array is to the **`B_SPACE`** character and the second is to the dead−key character itself. This pair does double duty: It states that the dead key plus a space yields the dead−key character, and it also names the dead key. The system understands what the dead key is from the second offset in the array.

Character tables for dead keys

As mentioned above, for a key to be dead, it must be mapped to the character picked by the second offset in a dead−key array. However, it's not typical for every key that's mapped to the character to be dead. Usually, there's a requirement that the user must hold down certain modifier keys (often the Option key). In other words, a key is dead only if selected character−map tables map it to the requisite character.

Five additional fields of the **`key_map`** structure specify what those character−map tables arewhich modifiers are required for each of the dead keys:

uint32 **`acute_tables`**
The character tables that cause a key to be dead when they map it to the second character in the **`acute_dead_key`** array.

uint32 **`grave_tables`**
The character tables that cause a key to be dead when they map it to the second character in the **`grave_dead_key`** array.

uint32 **`circumflex_tables`**
The character tables that cause a key to be dead when they map it to the second character in the **`circumflex_dead_key`** array.

uint32 **`dieresis_tables`**
The character tables that cause a key to be dead when they map it to the second character in the **`dieresis_dead_key`** array.

uint32 **`tilde_tables`**
The character tables that cause a key to be dead when they map it to the second character in the **`tilde_dead_key`** array.

Each of these fields contains a mask formed from the following constants:

B_OPTION_CAPS_SHIFT_TABLE	**B_CAPS_TABLE**
B_OPTION_CAPS_TABLE	**B_SHIFT_TABLE**
B_OPTION_SHIFT_TABLE	**B_NORMAL_TABLE**
B_OPTION_TABLE	

The mask designates the character−map tables that permit a key to be dead. For example, if the mask for the **`grave_tables`** field is,

```
B_OPTION_TABLE | B_OPTION_CAPS_SHIFT_TABLE
```

a key would be dead whenever either of those tables mapped the key to the character of the second offset in the **`grave_dead_key`** array ('Q' in the example above). A key mapped to the same character by another table would not be dead.

See also: **`get_key_info()`**, **`modifiers()`**, the **Keyboard Information** appendix, **`set_modifier_key()`**

get_key_repeat_delay() see set_key_repeat_rate()

get_key_repeat_rate() see set_key_repeat_rate()

get_keyboard_id()

Declared in: be/interface/InterfaceDefs.h

```
status_t get_keyboard_id(uint16 *id)
```

Obtains the keyboard identifier from the Application Server and device driver and writes it into the variable referred to by *id*. This number reveals what kind of keyboard is currently attached to the computer.

The identifier for the standard 101−key PC keyboardand for keyboards with a similar set of keysis 0x83ab.

If unsuccessful for any reason, `get_keyboard_id()` returns `B_ERROR`. If successful, it returns `B_OK`.

modifiers()

Declared in: be/interface/InterfaceDefs.h

```
uint32 modifiers(void)
```

Returns a mask that has a bit set for each modifier key the user is holding down and for each keyboard lock that's set. The mask can be tested against these constants:

B_CONTROL_KEY	**B_MENU_KEY**	**B_SCROLL_LOCK**
B_OPTION_KEY		**B_NUM_LOCK**

No bits are set (the mask is 0) if no locks are on and none of the modifiers keys are down.

If it's important to know which physical key the user is holding down, the one on the right or the one on the left, the mask can be further tested against these constants:

B_LEFT_CONTROL_KEY	**B_RIGHT_CONTROL_KEY**
B_LEFT_OPTION_KEY	**B_RIGHT_OPTION_KEY**
B_LEFT_COMMAND_KEY	**B_RIGHT_COMMAND_KEY**

By default, the keys closest to the space bar function as Command keys, no matter what their labels on particular keyboards. If a keyboard doesn't have Option keys (for example, a standard 101−key keyboard), the key on the right labeled "Control" functions as the right Option key, and only the left "Control" key is available to function as a Control modifier. However, users can change this configuration with the **/bin/keymap** application.

set_key_repeat_rate() , get_key_repeat_rate() , set_key_repeat_delay() , get_key_repeat_delay()

Declared in: be/interface/InterfaceDefs.h

```
status_t set_key_repeat_rate(int32 rate)

status_t get_key_repeat_rate(int32 *rate)

status_t set_key_repeat_delay(bigtime_t delay)

status_t get_key_repeat_delay(bigtime_t *delay)
```

These functions set and report the timing of repeating keys. When the user presses a character key on the keyboard, it produces an immediate `B_KEY_DOWN` message. If the user continues to hold the key down, it will, after an initial delay, continue to produce messages at regularly spaced intervalsuntil the user releases the key or presses another key. The delay and the spacing between messages are both preferences the user can set with the Keyboard application.

`set_key_repeat_rate()` sets the number of messages repeating keys produce per second. For a standard PC keyboard, the *rate* can be as low as 2 and as high as 30; `get_key_repeat_rate()` writes the current setting into the integer that *rate* refers to.

`set_key_repeat_delay()` sets the length of the initial delay before the key begins repeating. Acceptable values are 250,000, 500,000, 750,000 and 1,000,000 microseconds (.25, .5, .75, and 1.0 second); `get_key_repeat_delay()` writes the current setting into the variable that *delay* points to.

All four functions return **`B_OK`** if they successfully communicate with the Application Server, and **`B_ERROR`** if not. It's possible for the **`set`**...**`()`** functions to communicate with the server but not succeed in setting the *rate* or *delay* (for example, if the *delay* isn't one of the listed four values).

set_keyboard_locks()

Declared in: be/interface/InterfaceDefs.h

void **`set_keyboard_locks(`**uint32 *modifiers***`)`**

Turns the keyboard locksCaps Lock, Num Lock, and Scroll Lockon and off. The keyboard locks that are listed in the *modifiers* mask passed as an argument are turned on; those not listed are turned off. The mask can be 0 (to turn off all locks) or it can contain any combination of the following constants:

`B_NUM_LOCK`
`B_SCROLL_LOCK`

See also: **`get_key_map()`**, **`modifiers()`**

set_modifier_key()

Declared in: be/interface/InterfaceDefs.h

void **`set_modifier_key(`**uint32 *modifier*, uint32 *key***`)`**

Maps a *modifier* role to a particular *key* on the keyboard, where *key* is a key identifier and *modifier* is one of the these constants:

`B_NUM_LOCK`	**`B_LEFT_CONTROL_KEY`**	**`B_RIGHT_CONTROL_KEY`**
`B_SCROLL_LOCK`	**`B_LEFT_OPTION_KEY`**	**`B_RIGHT_OPTION_KEY`**
`B_MENU_KEY`	**`B_LEFT_COMMAND_KEY`**	**`B_RIGHT_COMMAND_KEY`**

The *key* in question serves as the named modifier key, unmapping any key that previously played that role. The change remains in effect until the default key map is restored. In general, the user's preferences for modifier keysexpressed in the Keymap applicationshould be respected.

Modifier keys can also be mapped by calling **`get_key_map()`** and altering the **`key_map`** structure directly. This function is merely a convenient alternative for accomplishing the same thing. (It's currently not possible to alter the key map; **`get_key_map()`** looks at a copy.)

Input Server Structures and Constants

Structures

input_device_ref

Declared in: be/add−ons/input_server/InputServerDevice.h

```
struct input_device_ref {
    char *name;
    input_device_type type;
    void *cookie;
    }
```

The **input_device_ref** structure is used to identify specific devices that a BInputServerDevice manages. To tell the Input Server about the devices your object manages, you create an array of these structures and pass them in a **BInputServerDevice::RegisterDevices()** call.

The fields are:

const char ***name**	An arbitrarybut preferably human−readablename that you invent. Although the name is used as an identifier (in **Control()** calls), you should choose a name that's "UI−suitable" (e.g. "ADB Mouse", or "USB Keyboard").
input_device_type **type**	This is either **B_POINTING_DEVICE** (i.e. mice), **B_KEYBOARD_DEVICE**, or **B_UNDEFINED_DEVICE**, where the last of these is a catchall for anything that isn't a mouse or a keyboard.
void ***cookie**	This is an arbitrary piece of data that you can associate with the device. Use it to conveniently store some cogent piece of data (such as ***this**), or to more specifically identify the device (such as the pathname of the driver it corresponds to), and so on.

Constants

Device Types

Declared in: be/interface/Input.h

```
B_POINTING_DEVICE

B_KEYBOARD_DEVICE

B_UNDEFINED_DEVICE
```

These constants represent the different types of input devices; they're used when defining an **input_device_ref** structure, and when sending control messages through a BInputDevice object.

- **B_POINTING_DEVICE** signifies devices, such as mice and tablets, that are used to control the cursor.
- **B_KEYBOARD_DEVICE** signifies keyboards.
- **B_UNDEFINED_DEVICE** is anything that isn't a pointer or keyboard.

Note that there is no "joystick device"the Input Server doesn't currently handle joysticks.

Input , Method , Operations

Declared in: be/interface/Input.h

```
enum input_method_op {
      B_INPUT_METHOD_STARTED,
      B_INPUT_METHOD_STOPPED,
      B_INPUT_METHOD_CHANGED,
      B_INPUT_METHOD_LOCATION_REQUEST
        }
```

Input , Device , Notifications

Declared in: be/interface/Input.h

```
enum input_device_notification {
      B_INPUT_DEVICE_ADDED,
      B_INPUT_DEVICE_STARTED,
      B_INPUT_DEVICE_STOPPED,
      B_INPUT_DEVICE_REMOVED
        }
```

Input , Device , Control , Messages

Declared in: be/add−ons/input_server/InputServerDevice.h

```
B_KEY_MAP_CHANGED

B_KEY_LOCKS_CHANGED

B_KEY_REPEAT_DELAY_CHANGED

B_KEY_REPEAT_RATE_CHANGED

B_MOUSE_TYPE_CHANGED

B_MOUSE_MAP_CHANGED

B_MOUSE_SPEED_CHANGED

B_CLICK_SPEED_CHANGED
```

These constants are used in the **`BInputServerDevice::Control()`** function to tell a BInputServerDevice that a change to a device parameter has been requested.

The Input Server: Master Index

A

acute_dead_key	Input Functions
acute_tables	Input Functions
App and Input Events	The Input Server

B

C

caps_key	Input Functions
B_CAPS_LOCK	Input Functions
B_CAPS_LOCK	Input Functions
caps_map	Input Functions
caps_shift_map	Input Functions
B_CAPS_SHIFT_TABLE	Input Functions
B_CAPS_TABLE	Input Functions
Character maps	Input Functions
Character tables for dead keys	Input Functions
circumflex_dead_key	Input Functions
circumflex_tables	Input Functions
B_CLICK_SPEED_CHANGED	Input Server Messages
B_COMMAND_KEY	Input Functions
Constants	Input Server Structures and Constants
Constructor and Destructor	BInputDevice
Constructor and Destructor	BInputServerDevice
Constructor and Destructor	BInputServerFilter
Constructor and Destructor	BInputServerMethod
Control()	BInputDevice
Control()	BInputServerDevice
Control	Input Server Structures and Constants