# The Device Kit

# The Device Kit − Table of Contents

# The Device Kit

The Device Kit contains software for controlling hardware that's connected to the computer.

- A BSerialPort object represents a RS232 serial port.
- A BJoystick object represents a joystick connection.

# The GeekPort and its Classes

This documentation applies to the BeBox hardware only.

The GeekPort is a piece of hardware on the back of the BeBox that communicates with external devices. Depending on how you use the GeekPort's ports, you can get up to 24 independent data paths:

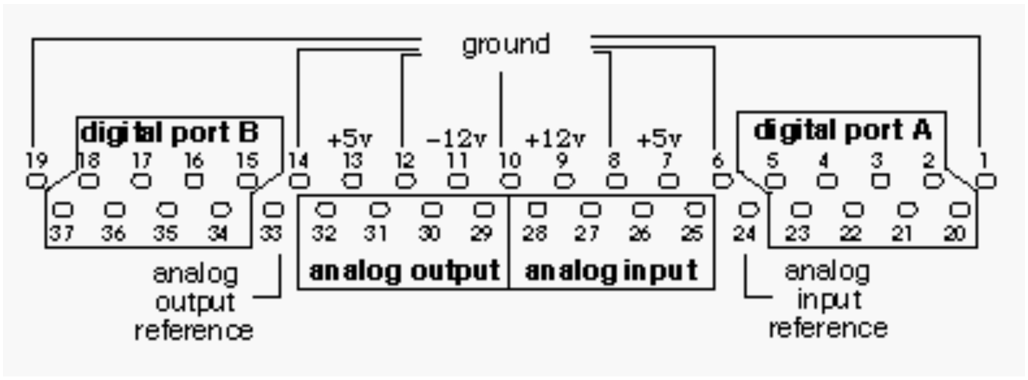
- Four 12−bit analog input channels.
- Four 8−bit analog output channels
- Two 8−bit wide digital ports (16 paths, total) that can act as inputs or outputs.

To provide high−level access to these data paths, the Device Kit defines three classes:

- The BA2D class ("analog to digital") lets you get at the analog input channels.
- The BD2A class ("digital to analog") does the same for the analog output channels.
- The BDigitalPort class lets you configure, read, and write the digital ports.

The signals and data that these classes read and write appear at the GeekPort connector, a 37−pin female connector that you'll find at the back of every BeBox. In addition to the pins that correspond to the analog and data paths, the GeekPort provides power and ground pins. Everything you need to feed your external gizmo is right there.

The GeekPort connector's pins are assigned thus:



## The ADC and DAC

The GeekPort provides four channels of simultaneous analog−to−digital (a/d) and four channels of simultaneous digital−to−analog (d/a) conversion. The signals that feed the ADC arrive on pins 25−28 of the GeekPort connector; the signals that are produced by the DAC depart through pins 29−32 (as depicted below). Pins 24 and 33 are DC reference levels (ground) for the a/d and d/a signals, respectively (*don't* use pins 24 or 33 as power grounds):

In the illustration, the a/d and d/a pins are labelled ("A2D1", "A2D2", etc.) as they are known to the BA2D and BD2A classes.

If you've read the GeekPort hardware specification, you'll have discovered that the ADC can be placed in a few different modes (the DAC is less flexible). The BA2D and BD2A classes (more accurately, the ADC and DAC drivers) refine the GeekPort specification, as described in the following sections.

Keep in mind that the a/d and d/a converters that the GeekPort uses are *not* part of the Crystal codec that's used by the audio software (and brought into your application through the Media Kit). The two sets of converters are completely separate and can be used independently and simultaneously. If you're doing on−board high−fidelity sound processing (or generation) in real time, you should stick with the Crystal convertors.

### The ADC

The ADC accepts signals in the range [0, +4.096] Volts, performs a linear conversion, and spits out unsigned 12−bit data. The 4.096V to 12−bit conversion produces a convenient one−digital−step per milliVolt of input.

A/d conversion is performed on−demand: When you read a value from the ADC, the voltage that lies on the specified pin is immediately sampled (this is the "Single Shot" mode described in the GeekPort hardware specification). In other words, the ADC doesn't perform a sample and holdit doesn't constantly and regularly measure the voltages at its inputs. Nonetheless, you *can't* retrieve samples at an arbitrarily high frequency simply by reading in a tight loop. This is because of the "sampling latency": When you ask for a sample, it takes the driver about ten milliseconds to process the request, not counting the (slight) overhead imposed by the C++ call (from your BA2D object). Therefore, the fastest rate at which you can get samples from the ADC is a bit less than 100 kHz.

Furthermore, the ADC driver "fakes" the four channels of a/d conversion. In reality, there's only one ADC data path; the driver multiplexes the path to create four independent signals. This means that the optimum 100 kHz sampling frequency is divided by the number of channels that you want to read. If all four channels are being read at the same time, you'll find that successive samples on a *particular* channel arrive slightly less often than once every 40 milliseconds (a rate of < 25 kHz).

Finally, the ADC hardware is shared by the GeekPort and the two joysticks. This cooperative use shouldn't affect your applicationyou can treat the ADC as if it were all your ownbut this increases the multiplexing. In general, joysticks shouldn't need to sample very often, so while the theoretical "worst hit" on the ADC is a sample every 60 milliseconds, the reality should be much better. If we can assume that a joystick−reading application isn't oversampling, then the BA2D "sampling latency" should stay near the 10 milliseconds per channel measurement.

### The DAC

The DAC accepts 8−bit unsigned data and converts it, in 16 mV steps, to an analog signal in the range [0, +4.080] Volts. Again, the quantization is linear. The DAC output isn't filtered; if you need to smooth the stair−step output, you have to build a filter into the gizmo that you're connecting to the GeekPort.

Each of the d/a pins is protected by an in−series 4.7 kOhm resistor; however, pin 33, the d/a DC reference (ground) pin, is not similarly impeded. If you want to attach an op−amp circuit to the DAC output, you should hang a 4.7 kOhm resistor on the ground pin that you're using.

When you write a digital sample to the DAC, the specified pin is immediately set to the converted voltage. The pin continues to produce that voltage until you write another sample.

Unlike the ADC, the DAC is truly a four−channel device, so there's no multiplexing imposition to slow things down. Furthermore, writing to the DAC is naturally faster than writing to the ADC. You should be able to write to the DAC as frequently as you want, without worrying about a hardware−imposed "sampling latency."

## The Digital Ports

The following illustration shows the disposition of the GeekPort connector pins as they are assigned to the digital ports:

Each pin in a digital port transmits the value of a single bit; the pins are labelled by bit position. Thus, A0 is the least significant bit of digital port A, and A7 is its most significant bit. You can use any of the seven ground pins (1, 6, 8, 10, 12, 14, and 19) in your digital port circuit. The unmarked pins (24−33) are the analog ports; see the BA2D and BD2A classes for more information on these ports.

Devices that you connect to the digital ports should send and (expect to) receive voltages that are below 0.8 Volts or above 2.0 Volts. These thresholds correspond, respectively, to the greatest value for digital 0 and the least for digital 1 (as depicted below). The correspondence to bit value for voltages between these limits is undefined.

Although there's no lower voltage limit for digital 0, nor upper limit for digital 1, the BeBox outputs voltages that are no less than 0 Volts, nor no more than +5 Volts. Your input device can exceed this range without damaging the BeBox circuitry: Excessive input emf is clipped to fall within [−0.5V, +5.5V].

Be aware that behind each digital port pin lies a 1 kOhm resistor.

# BD2A

Derived from: none

Declared in: be/device/D2A.h

Library: libdevice.so

***Summary***

The BD2A class lets you write to (and, less importantly, read from) the GeekPort's digital−to−analog converter (DAC). Each BD2A object can write a single channel at a time; if you want to read all four channels simultaneously, you have to create four separate objects.

To write a value to one of the d/a channels, you create a new BD2A object, open it on the channel you want, and then (repeatedly) invoke the object's **`Write()`** function. When you're through reading, you call **`Close()`** so some other object can open the channel. The **`Write()`** function writes a single byte at each call and returns 1 if it was successful:

```
#include <D2A.h>

void WriteD2A1()
{
   uint8 val;
   BD2A d2a;

   if (d2a.Open("D2A1") <= 0)
      return;

   while ( /* whatever */ ) {
      /* Get a byte from somewhere. */
      val = ...;

      if (d2a.Write(val) != 1)
         break;
      snooze(1000);
   }
   d2a.Close();
}
```

The DAC performs a "sample and hold": The voltage that the DAC produces is maintained until another **`Write()`** call (on that channel) changes the setting. Furthermore, the "hold" persists across BD2A objects: Neither closing nor deleting a BD2A object affects the voltage that's held by the corresponding GeekPort pin.

The BD2A class also implements a **`Read()`** function. This function returns the value that was most recently written to the particular DAC channel.

## Constructor and Destructor

### BD2A()

**`BD2A(`**void**`)`**

Creates a new object that can open a DAC channel. The particular channel is specified in a subsequent **`Open()`** call. Constructing a new BD2A object doesn't affect the state of the DAC.

### ~BD2A()

virtual **`~BD2A()`**

Closes the channel that the object holds open (if any) and then destroys the object.

Deleting a BD2A object *doesn't* affect the DAC channel's output voltage. If you want the voltage cleared (for example), you have to set it to 0 explicitly before deleting (or otherwise closing) the BD2A object.

## Member Functions

### Open() , IsOpen() , Close()

status_t **`Open(`**const char **name***`)`**

bool **`IsOpen(`**void**`)`**

Open() opens the named DAC (BD2A) channel. The channel names are:

| |
|---|
| "D2A1" |
| "D2A2" |
| "D2A3" |
| "D2A4" |

Each channel can only be held open by one object at a time; you should close the channel as soon as you're finished with it. Furthermore, each BD2A object can only hold one channel open at a time. When you invoke **`Open()`**, the channel that the object currently has open is automatically closedeven if the channel that you're attempting to open is the channel that the object already has open.

Opening a DAC channel doesn't affect the data in the channel itself. In particular, when you open a DAC channel, the channel's output voltage isn't changed.

**`IsOpen()`** returns **`true`** if the object holds a DAC channel open. Otherwise, it returns **`false`**.

**`Close()`** does the obvious without affecting the state of the ADC or DAC channel. If you want to set a DAC channel's output voltage to 0 (for example), you must explicitly write the value before invoking **`Close()`**.

**RETURN CODES**

**`Open()`** returns a positive integer if the channel is successfully opened; otherwise, it returns **`B_ERROR`**.

## Read()

ssize_t **`Read(`**uint8 **dac_8_bit*`)`**

Returns, by reference in *dac_8_bit*, the value that was most recently written to the object's DAC channel. The value needn't have been written by this objectit could have been written by the channel's previous opener.

❗ The BD2A **`Read()`** function returns a value that's cached by the DAC driverit doesn't actually tap the GeekPort pin to see what value it's currently carrying. This should only matter to the clever few who will attempt (unsuccessfully) to use the d/a pins as input paths.

The object must open a DAC channel before calling **`Read()`**.

**RETURN CODES**

**`Read()`** return **`B_ERROR`** if a channel isn't open, or if, for any other reason, the read failed. Otherwise it returns 1 (the number of bytes that were read).

## Write()

ssize_t **`Write(`**uchar *dac_8_bit***`)`**

Sends the *dac_8_bit* value to the object's DAC channel. The DAC converts the value to an analog voltage in the range [0, +4.080] Volts and sets the corresponding GeekPort pin. The pin continues to produce the voltage until another **`Write()`** callpossibly by a different BD2A objectchanges the setting.

The DAC's conversion is linear: Each digital step corresponds to 16 mV at the output. The analog voltage midpoint, +2.040V, can be approximated by a digital input of 0x7F (which produces +2.032V) or 0x80 (+2.048V).

**RETURN CODES**

If the object isn't open, this function returns **`B_ERROR`**, otherwise it returns 1 (the number of bytes that were written).

# BJoystick

Derived from: none

Declared in: be/device/Joystick.h

Library: libdevice.so

Allocation: Constructor only

***Summary***

A BJoystick object provides an interface to a joystick (or other game controller) connected to the computer. The BeOS supports joysticks on the BeBox and Intel platforms.

The BeBox supports up to four analog joysticks, each of which can have up to two axes and two buttons; digital joysticks aren't supported by the built−in game ports. You can install a card that provides additional game ports (such as the Sound Blaster AWE64), and use digital joysticks on those ports.

BeOS for Intel systems supports joysticks through game ports on cards, as well as some built−in game ports.

Unlike the event and message−driven interface to the mouse and keyboard, the interface to a joystick is strictly demand−driven. An application must repeatedly poll the state of the joystick by calling the BJoystick object's **`Update()`** function. **`Update()`** queries the port and updates the object's data members to reflect the current state of the joystick.

There are two modes available. Standard mode supports only two axes per joystick, and two buttons per joystick. This mode has been available since the early days of the BeOS. You read the joystick in standard mode by looking at the BJoystick member variables **`vertical`** and **`horizontal`** to determine the joystick's axis values, and **`button1`** and **`button2`** to determine the state of the buttons.

Enhanced mode supports up to 32 buttons per joystick, and an indefinite number of axes and hats (thumb controls, usually located at the top of a stick). It also supports multiple joysticks chained to a single game port. Instead of reading variables to determine the state of the joystick, there are several functions provided to let you do this.

In addition, enhanced mode provides a mechanism for determining what joysticks are available and what types of (and how many) controls are available on the joysticks. There's also a preference application (cleverly named "Joysticks") that lets the user select and configure joysticks connected to their computer.

## Data Members

| | |
|---|---|
| bigtime_t **`timestamp`** | The time of the most recent update, as measured in microseconds from the beginning of 1970. |
| int16 **`horizontal`** | The horizontal position of the joystick at the time of the last update. Values increase as the joystick moves from left to right. |
| int16 **`vertical`** | The vertical position of the joystick at the time of the last update. Values decrease as the joystick moves forward and increase as it's pulled back. |
| bool **`button1`** | **`false`** if the first button was pressed at the time of the last update, and **`true`** if not. |
| bool **`button2`** | **`false`** if the second button was pressed at the time of the last update, and **`true`** if not. |

The **`horizontal`** and **`vertical`** data members record values read directly from the ports, values that simply digitize the analog output of the joysticks. This class makes no effort to translate the values to a standard scale or range. Values can range from 0 through 4,095, but joysticks typically don't use the full range and some don't register all values within the range that is used. The scale is not linearidentical increments in different parts of the range can reflect differing amounts of horizontal and vertical movement. The exact variance from linearity and the extent of the usable range are partly characteristics of the individual joystick and partly functions of the computer's hardware.

Typically you won't use these data members if you're using enhanced mode; they're provided primarily for backward compatibility.

## Constructor and Destructor

### BJoystick()

**`BJoystick(`**void**`)`**

Initializes the BJoystick object so that all values are set to 0. Before using the object, you must call **`Open()`** to open a particular joystick port. For the object to register any meaningful values, you must call **`Update()`** to query the open port.

**See also:** **`Open()`**, **`Update()`**

### ~BJoystick()

virtual **~BJoystick(**void**)**

Closes the port, if it was not closed already.

## Member Functions

### ButtonValues() see CountButtons()

### Close() see Open()

### CountAxes() , GetAxisValues()

int32 **CountAxes(**void**)**

status_t **GetAxisValues(**int16 **outValues*, int32 *forStick* = 0**)**

**CountAxes()** returns the number of axes on the opened game port.

**GetAxisValues()** fills the array pointed to by *outValues* with the values of all the axes connected to the specified joystick. The *forStick* parameter lets you choose which joystick on the game port to read the values from (the default is to read from the first stick on the port).

The returned values range from −32,768 to 32,767.

The array *outValues* must be large enough to contain the values for all the axes. You can ensure that this is the case by using the following code:

```
int16 *axes;

axes = (int16 *) malloc(sizeof(int16) * stick->CountAxes());
stick->GetAxisValues(axes);
```

These functions can only be used in enhanced mode.

**RETURN CODES**

**B_OK.** The name was returned successfully.

- **B_BAD_VALUE**. The joystick specified by *forStick* doesn't exist.

### CountButtons() , GetButtonValues()

int32 **CountButtons(**void**)**

uint32 **ButtonValues(**void**)**

**CountButtons()** returns the number of buttons on the opened game port.

**ButtonValues()** returns a 32−bit number in which each bit represents the state of one button. The *forStick* parameter lets you choose which joystick on the game port to read the values from (the default is to read from the first stick on the port). You can deterimine if a particular button is down by using the following code:

```
uint32 buttonValues = stick->ButtonValues();

if (buttonValues & (1 << whichButton)) {
   /* button number whichButton is pressed */
}
```

These functions can only be used in enhanced mode.

## CountDevices() , GetDeviceName()

int32 **`CountDevices(`**void**`)`**

status_t **`GetDeviceName(`**int32 *index*, char **outName*, size_t *bufSize* = B_OS_NAME_LENGTH**`)`**

**`CountDevices()`** returns the number of game ports on the computer.

**`GetDeviceName()`** returns the name of the device specified by the given *index*. The buffer pointed to by *outName* is filled with the device name; *bufSize* indicates the size of the buffer.

The names returned by **`GetDeviceName()`** can be passed into the **`Open()`** function to open a device.

The BJoystick doesn't need to have an open device before you use these functions; in fact, your application will typically use these to provide user interface allowing the user to choose the joystick device they'd like to use.

**RETURN CODES**

**`B_OK`**. The name was returned successfully.

- B_BAD_INDEX. The specified *index* is invalid.
- **`B_NAME_TOO_LONG`**. The device name is too long for the buffer.

## CountHats() , GetHatValues()

int32 **`CountHats(`**void**`)`**

status_t **`GetHatValues(`**int8 **outHats*, int32 *forStick* = 0**`)`**

**`CountHats()`** returns the number of hats on the opened game port.

**`GetHatValues()`** fills the array pointed to by *outHats* with the values of all the hats connected to the specified joystick. The *forStick* parameter lets you choose which joystick on the game port to read the values from (the default is to read from the first stick on the port).

The return value means the following:

- 0: Centered
- 1: Up
- 2: Up and right
- 3: Right
- 4: Down and right
- 5: Down
- 6: Down and left
- 7: Left
- 8: Up and left

The array *outHats* must be large enough to contain the values for all the hats. You can ensure that this is the case by using the following code:

```
int8 *hats;
hats = (int8 *) malloc(sizeof(int8) * stick->CountAxes());
```

```
stick->GetHatValues(hats);
```

These functions can only be used in enhanced mode.

**RETURN CODES**

**B_OK.** The name was returned successfully.

- **B_BAD_VALUE**. The joystick specified by *forStick* doesn't exist.

## CountSticks()

int32 **CountSticks(**void**)**

Returns the number of joysticks connected to the opened game port.

This function can only be used in enhanced mode.

## GetAxisNameAt() , GetHatNameAt() , GetButtonNameAt()

status_t **GetAxisNameAt(**int32 *index*, BString **outName***)**

status_t **GetHatNameAt(**int32 *index*, BString **outName***)**

status_t **GetButtonNameAt(**int32 *index*, BString **outName***)**

Returns the name of the control specified by the given *index*. The BString object pointed to by *outName* is set to the control's name.

**GetAxisNameAt()** returns the specified axis' name, **GetHatNameAt()** returns the specified hat's name, and **GetButtonNameAt()** returns the specified button's name.

These functions can only be used in enhanced mode.

**RETURN CODES**

**B_OK.** The name was returned successfully.

- **B_BAD_INDEX**. The specified *index* is invalid.

## GetAxisValues() see CountAxes()

## GetButtonNameAt() see GetAxisNameAt()

## GetControllerModule() , GetControllerName()

status_t **GetControllerModule(**BString **outName***)** status_t **`GetControllerName(`**BString **outName* **`)`**

**`GetControllerModule()`** returns the name of the joystick module that represents the opened joystick device. If the device isn't in enhanced mode, this always returns "Legacy".

**`GetControllerName()`** returns the name of the joystick that's been configured for the opened device. This is the same string that appears in the Joysticks preference application. The returned string is always "2−axis" if the device isn't in enhanced mode.

These functions can only be used in enhanced mode.

**RETURN CODES**

**`B_OK`**. The name was returned successfully.

- **`B_BAD_INDEX`**. The specified *index* is invalid.

## GetDeviceName() see CountDevices()

## GetHatNameAt() see GetAxisNameAt()

## GetHatValues() see CountHats()

## EnableCalibration() see IsCalibrationEnabled()

## EnterEnhancedMode()

bool **`EnterEnhancedMode(`**const entry_ref **ref* = NULL**`)`**

Switches the device into enhanced mode. If *ref* isn't **`NULL`**, it's treated as a reference to a joystick description file (such as those in /boot/beos/etc/joysticks). If *ref* is **`NULL`**, the currently−configured joystick settings (as per the Joysticks preference application) are used.

If enhanced mode is entered successfully (or the device is already in enhanced mode), **`true`** is returned. Otherwise, **`EnterEnhancedMode()`** returns **`false`**.

## IsCalibrationEnabled() , EnableCalibration()

bool **`IsCalibrationEnabled(`**void**`)`**

status_t **`EnableCalibration(`**bool *calibrates* = true**`)`**

**`IsCalibrationEnabled()`** returns **`true`** if axis values returned by the joystick will be calibrated automatically into the range −32,768 to 32,767, or **`false`** if the raw values will be returned.

**`EnableCalibration()`** enables or disables automatic calibration for the joystick's axes. Specify a value of **`true`** for *calibrates* to enable calibration; otherwise, specify **`false`**.

The names returned by **`GetDeviceName()`** can be passed into the **`Open()`** function to open a device.

The Joysticks preference application lets the user calibrate the joystick. Calibration is enabled by default. These functions may only be used in enhanced mode.

**RETURN CODES**

**`B_OK`**. The name was returned successfully.

- B_NO_INIT. The device isn't in enhanced mode.

## Open() , Close()

status_t **`Open(`**const char **devName*`)`

status_t **`Open(`**const char **devName*, bool *enterEnhanced*`)`

void **`Close(`**void`)`

These functions open the joystick port specified by *devName* and close it again.

On the BeBox, there are two ports on the back of the computer, and they have names that correspond to their labels on the machine (and in **The BeOS User's Guide** diagram):

"joystick1" (on the top)
"joystick2" (on the bottom)

By attaching a Y cable to a machine port, you can make it support two joysticks. Cables, therefore, add two additional ports:

"joystick3" (on the top)
"joystick4" (on the bottom)

The cable maps the bottom row of pins on a machine port to the top row on a cable port. Therefore, the first two names listed above correspond to the top row of pins on a machine port; the last two names correspond to the bottom row of pins.

On other systems, the *devName* should be the name of a game port device. The easiest way to determine valid device names is by using the **`CountDevices()`** and **`GetDeviceName()`** functions.

If it's able to open the port, **`Open()`** returns a positive integer. If unable or if the *name* isn't valid, it returns **`B_ERROR`**. If the *name* port is already open, **`Open()`** tries to close it first, then open it again.

By default, **`Open()`** opens the device in enhanced mode. If you want to use the old, unenhanced mode, you can use the second form of **`Open()`** to specify whether or not you want to use enhanced mode; set *enterEnhanced* to **`true`** to use enhanced mode; specify **`false`** if you don't want enhanced mode.

Even in enhanced mode, the classic BJoystick data members are valid (for compatibility with older applications). However, they only give you access to the first two axes and buttons of the joystick.

To be able to obtain joystick data, a BJoystick object must have a port open.

## SetMaxLatency()

status_t **`SetMaxLatency(`**bigtime_t *maxLatency*`)`

Specifies the maximum latency to allow when accessing the joystick. Returns **`B_OK`** if the change is applied successfully, otherwise returns an error code.

## Update()

status_t **`Update(`**void**`)`**

Updates the data members of the object so that they reflect the current state of the joystick. An application would typically call **`Update()`** periodically to poll the condition of the device, then read the values of the data members, orif in enhanced modecall the various functions for obtaining joystick state information.

This function returns **`B_ERROR`** if the BJoystick object doesn't have a port open, and **`B_OK`** if it does.

# BSerialPort

Derived from: none

Declared in: be/device/SerialPort.h

Library: libdevice.so

***Summary***

A BSerialPort object represents an RS−232 serial connection to the computer. Through BSerialPort functions, you can read data received at a serial ports and write data over the connection. You can also configure the connectionfor example, set the number of data and stop bits, determine the rate at which data is sent and received, and select the type of flow control (hardware or software) that should be used.

To read and write data, a BSerialPort object must first open one of the serial ports by name. To find the names of all the serial ports on the computer, use the **`CountDevices()`** and **`GetDeviceName()`** functions:

```
BSerialPort serial;
char devName[B_OS_NAME_LENGTH];
int32 n = 0;

for (int32 n = serial.CountDevices()−1; n >= 0; n−−) {
   serial.GetDeviceName(n, devName);

   if ( serial.Open(devName) > 0 )
       ....
}
```

The BSerialPort object communicates with the driver for the port it has open. The driver maintains an input buffer to collect incoming data and a smaller output buffer to hold outgoing data. When the object reads and writes data, it reads from and writes to these buffers.

The serial port drivers, and therefore BSerialPort objects, send and receive data asynchronously only.

## Constructor and Destructor

### BSerialPort()

**`BSerialPort(`**void**`)`**

Initializes the BSerialPort object to the following default values:

- Hardware flow control (see **`SetFlowControl()`**)
- A data rate of 19,200 bits per second (see **`SetDataRate()`**)
- A serial unit with 8 bits of data, 1 stop bit, and no parity (see **`SetDataBits()`**)
- Blocking with no time limitan infinite timeoutfor reading data (see **`Read()`**)

The new object doesn't represent any particular serial port. After construction, it's necessary to open one of the ports by name.

The type of flow control must be decided before a port is opened. But the other default settings listed above can be changed before or after opening a port.

**See also:** **`Open()`**

### ~BSerialPort()

virtual **`~BSerialPort()`**

Makes sure the port is closed before the object is destroyed.

## Member Functions

### ClearInput() , ClearOutput()

void **`ClearInput(`**void**`)`**

void **ClearOutput(**void**)**

These functions empty the serial port driver's input and output buffers, so that the contents of the input buffer won't be read (by the **`Read()`** function) and the contents of the output buffer (after having been written by **`Write()`**) won't be transmitted over the connection.

The buffers are cleared automatically when a port is opened.

**See also:** **`Read()`**, **`Write()`**, **`Open()`**

---

### Close() see Open()

---

### DataBits() see SetDataBits()

---

### IsCTS()

bool **IsCTS(**void**)**

Returns **`true`** if the Clear to Send (CTS) pin is asserted, and **`false`** if not.

---

### IsDCD()

bool **IsDCD(**void**)**

Returns **`true`** if the Data Carrier Detect (DCD) pin is asserted, and **`false`** if not.

---

### IsDSR()

bool **IsDSR(**void**)**

Returns **`true`** if the Data Set Ready (DSR) pin is asserted, and **`false`** if not.

---

### IsRI()

bool **IsRI(**void**)**

Returns **`true`** if the Ring Indicator (RI) pin is asserted, and **`false`** if not.

---

### Open() , Close()

status_t **Open(**const char **name***)**

void **Close(**void**)**

These functions open the *name* serial port and close it again. To get a serial port name, use the GetDeviceName() function; an example is shown in the introduction.

To be able to read and write data, the BSerialPort object must have a port open. It can open first one port and then another, but it can have no more than one open at a time. If it already has a port open when **`Open()`** is called, that port is closed before an attempt is made to open the *name* port. (Thus, both **`Open()`** and **`Close()`** close the currently open port.)

**`Open()`** can't open the *name* port if some other entity already has it open. (If the BSerialPort itself has *name* open, **`Open()`** first closes it, then opens it again.)

When a serial port is opened, its input and output buffers are emptied and the Data Terminal Ready (DTR) pin is asserted.

**RETURN CODES**

*positive integers* (not 0). Success.

- `B_PERMISSION_DENIED. The port is already open.`

## ParityMode() see SetDataBits()

## Read() , SetBlocking() , SetTimeout()

```
ssize_t Read(void *buffer, size_t maxBytes)

void SetBlocking(bool shouldBlock)

status_t SetTimeout(bigtime_t timeout)
```

`Read()` takes incoming data from the serial port driver and places it in the data *buffer* provided. In no case will it read more than *maxBytes*a value that should reflect the capacity of the *buffer*. The input buffer of the driver, from which `Read()` takes the data, holds a maximum of 2,024 bytes (2048 on Mac hardware). This function fails if the BSerialPort object doesn't have a port open.

The number of bytes that `Read()` will read before returning depends not only on *maxBytes*, but also on the *shouldBlock* flag and the *timeout* set by the other two functions.

- `SetBlocking()` determines whether `Read()` should block and wait for *maxBytes* of data to arrive at the serial port if that number isn't already available to be read. If the *shouldBlock* flag is `true`, `Read()` will block. However, if *shouldBlock* is `false`, `Read()` will take however many bytes are waiting to be read, up to the maximum asked for, then return immediately. If no data is waiting at the serial port, it returns without reading anything.

  The default *shouldBlock* setting is `true`.

- `SetTimeout()` sets a time limit on how long `Read()` will block while waiting for data to arrive at the port's input buffer. The *timeout* is relevant to `Read()` only if the *shouldBlock* flag is `true`. However, the time limit also applies to the `WaitForInput()` function, which always blocks, regardless of the *shouldBlock* setting.

  There is no time limit if the *timeout* is set to `B_INFINITE_TIMEOUTRead()` (and `WaitForInput()`) will block forever. Otherwise, the *timeout* is expressed in microseconds and can range from a minimum of 100,000 (0.1 second) through a maximum of 25,500,000 (25.5 seconds); differences less than 100,000 microseconds are not recognized; they're rounded to the nearest tenth of a second.

**RETURN CODES**

`Read()` returns...

- *non−negative integer*. Success; the value is the number of bytes that were read.

**RETURN CODES**

`SetTimeout()` returns...

- `B_OK`. Success.

## SetBlocking() see Read()

## SetDataBits() , SetStopBits() , SetParityMode() , DataBits() , StopBits() , ParityMode() , data_bits , stop_bits , parity_mode

```
void SetDataBits(data_bits count)

void SetStopBits(stop_bits count)

data_bits DataBits(void)

stop_bits StopBits(void)

typedef enum { B_DATA_BITS_7, B_DATA_BITS_8 } data_bits

typedef enum { B_STOP_BITS_1, B_STOP_BITS_2 } stop_bits
```

These functions set and return characteristics of the serial unit used to send and receive data.

- `SetDataBits()` sets the number of bits of data in each unit; the default is **`B_DATA_BITS_8`**.
- `SetStopBits()` sets the number of stop bits in each unit; the default is **`B_STOP_BITS_2`**.
- `SetParityMode()` sets whether the serial unit contains a parity bit and, if so, the type of parity used; the default is **`B_NO_PARITY`**.

---

## SetDataRate() , DataRate() , data_rate

status_t **`SetDataRate(`**data_rate *bitsPerSecond***`)`**

data_rate **`DataRate(`**void**`)`**

These functions set and return the rate (in bits per second) at which data is both transmitted and received.

The default data rate is **`B_19200_BPS`**. If the rate is set to 0 (**`B_0_BPS`**), data will be sent and received at an indeterminate number of bits per second.

**RETURN CODES**

**`SetDataRate() returns...`**

- **`B_OK. The rate was successfully set.`**

---

## SetDTR()

status_t **`SetDTR(`**bool *pinAsserted***`)`**

Asserts the Data Terminal Ready (DTR) pin if the *pinAsserted* flag is **`true`**, and de−asserts it if the flag is **`false`**. The function should always returns **`B_OK`**.

---

## SetFlowControl() , FlowControl()

void **`SetFlowControl(`**uint32 *mask***`)`**

uint32 **`FlowControl(`**void**`)`**

These functions set and return the type of flow control the driver should use. There are four possibilities:

| | |
|---|---|
| **`B_SOFTWARE_CONT ROL`** | Control is maintained through XON and XOFF characters inserted into the data stream. |
| **`B_HARDWARE_CONT ROL`** | Control is maintained through the Clear to Send (CTS) and Request to Send (RTS) pins. |
| **`B_SOFTWARE_CONT ROL`** + **`B_HARDWARE_CONT ROL`** | Both of the above. |
| 0 (zero) | No control. |

`SetFlowControl()` should be called before a specific serial port is opened. You can't change the type of flow control the driver uses in midstream.

---

## SetParityMode() see SetDataBits()

---

## SetRTS()

status_t **`SetRTS(`**bool *pinAsserted***`)`**

Asserts the Request to Send (RTS) pin if the *pinAsserted* flag is **`true`**, and de−asserts it if the flag is **`false`**. The function always returns **`B_OK`**.

## SetStopBits() see SetDataBits()

## SetTimeout() see Read()

## WaitForInput()

```
ssize_t WaitForInput(void)
```

Waits for input data to arrive at the serial port and returns the number of bytes available to be read. If data is already waiting, the function returns immediately.

This function doesn't respect the flag set by **`SetBlocking()`**; it blocks even if blocking is turned off for the **`Read()`** function. However, it does respect the timeout set by **`SetTimeout()`**. If the timeout expires before input data arrives at the serial port, it returns 0.

## Write()

```
ssize_t Write(const void *data, size_t numBytes)
```

Writes up to *numBytes* of *data* to the serial port's output buffer. This function will be successful in writing the data only if the BSerialPort object has a port open. The output buffer holds a maximum of 512 bytes (1024 on Mac hardware).

**RETURN CODES**

*non−negative integer*. Success; the value is the number of bytes that were written.

- **`B_INTERRUPTED`**. The operation was interrupted by a signal.

# The Device Kit: Master Index

C

| | |
|---|---|
| ClearOutput() | BSerialPort |
| Close() | BJoystick |
| Close() | BSerialPort |
| Constructor and Destructor | BJoystick |
| Constructor and Destructor | BSerialPort |
| CountAxes() | BJoystick |
| CountButtons() | BJoystick |
| CountDevices() | BJoystick |
| CountHats() | BJoystick |
| CountSticks() | BJoystick |

D

| | |
|---|---|
| Data Members | BJoystick |
| DataRate() | BSerialPort |
| data_bits | BSerialPort |
| data_rate | BSerialPort |
| The Device Kit | The Device Kit |
| The Device Kit | The Device Kit |

E

| | |
|---|---|
| EnterEnhancedMode() | BJoystick |

F

G

| | |
|---|---|
| GetAxisValues() | BJoystick |
| GetButtonNameAt() | BJoystick |
| GetButtonValues() | BJoystick |
| GetControllerModule() | BJoystick |