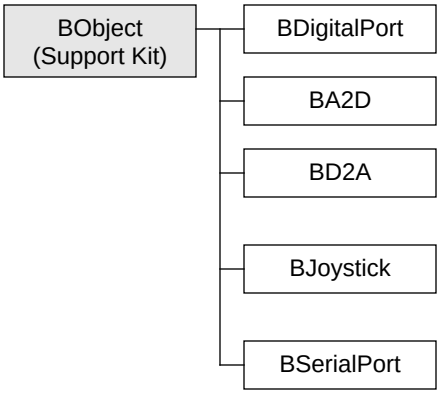
9 The Device Kit

Device Kit Inheritance Hierarchy



9 The Device Kit

The Device Kit contains software for controlling various input/output devices and for writing your own device drivers. You'll find two kinds of software documented in this chapter:

- Encapsulated interfaces to some of the ports found on the back of the BeBox. Currently, this part of the kit contains five classes—BJoystick, BSerialPort, BDigitalPort, BA2D (digital to analog), and BD2A (analog to digital). A BJoystick object represents a joystick connection to the BeBox. A BSerialPort object can represent any of the four RS–232 serial ports that are visible on the back of the machine. The other three classes represent particular functions of the GeekPort™.

  These classes are all part of the shared system library, **libbe.so**. Their header files are collected in **DeviceKit.h** and are precompiled with the header files of other kits.

- The programming interfaces and protocols for developing your own drivers for input/output devices. All drivers are dynamically loaded, add-on modules that run as extensions either of the kernel or of a specific server. Most drivers run as part of the kernel, but drivers for graphics cards extend the Application Server and printer drivers connect to the Print Server.

  The programming interfaces for device drivers are *not* included in the master **DeviceKit.h** header file or the precompiled headers; this part of the Kit doesn't belong to the Be system library. A driver links only against its host module (or perhaps statically against a private library), not against the system library.

If you're interested in the interface to a joystick or serial port, you need read only about the BJoystick or BSerialPort class. If you're interested in the GeekPort interface, there's a small section that introduces the port and its three classes; look at it before turning to the particular class that interests you. If you're interesting in writing a device driver, skip the first part of the chapter and begin with "Developing a Device Driver" on page 39.

The GeekPort and its Classes

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

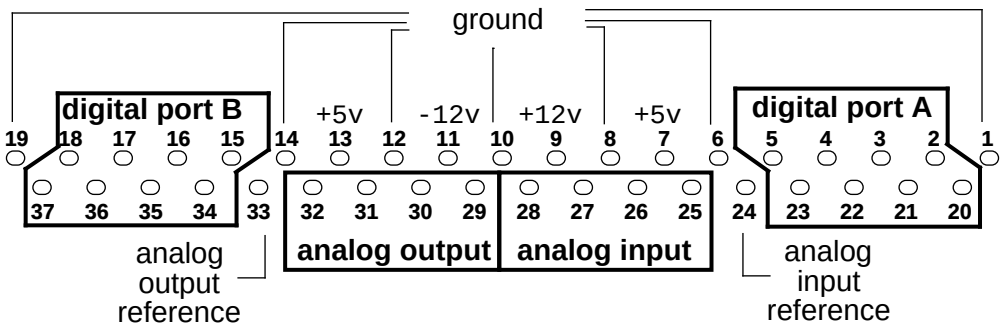
- Four 12-bit analog input channels.
- Four 8-bit analog output channels
- Two 8-bit wide digital ports (16 paths, total) that can act as inputs or outputs.

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 BA2D, BD2A, and BDigitalPort class descriptions re-visit this illustration to provide more detailed examinations of the specific GeekPort pins.

BA2D and BD2A

Derived from: public BObject

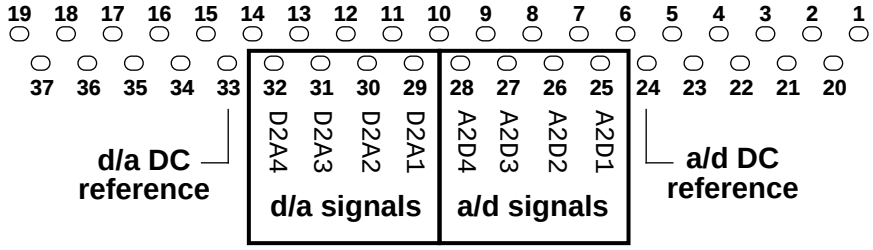
Declared in: <device/A2D.h>
<device/D2A.h>

Overview

The BA2D and BD2A classes let you talk to the GeekPort's analog-to-digital converter (ADC) and digital-to-analog converter (DAC). Before we examine the classes, let's visit the GeekPort.

The GeekPort

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.

Note: 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 hold—it 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 application—you can treat the ADC as if it were all your own—but 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."

BA2D

The BA2D class lets you create objects that can read the GeekPort's a/d channels. Each BA2D object can read a single channel at a time; if you want to read all four channels simultaneously, you have to create four separate objects.

To retrieve a value from one of the a/d channels, you create a new BA2D object, open it on the channel you want (using the labels shown above), and then (repeatedly) invoke the object's **`Read()`** function. When you're through reading, you call **`Close()`** so some other object can open the channel.

Reading is a one-shot deal: For each **`Read()`** invocation, you get a single **`ushort`** that stores the 12-bit ADC value in its least significant 12 bits. To get a series of successive values, you have to put the **`Read()`** call in a loop. Keep in mind that there's no sampling rate or other automatic time tethering. For example, if you want to read the ADC every tenth of a second, you have to impose the waiting period yourself (by snoozing between reads, for example).

The outline of a typical a/d-reading setup is shown below:

```
#include <A2D.h>

void ReadA2D1()
{
    ushort val;
    BA2D *a2d = new BA2D();

    if (a2d->Open("A2D1") <= 0)
        return;

    while ( /* whatever */ ) {

        /* Read() returns the number of bytes that were
         * read; a successful read returns the value 2.
         */
        if (a2d->Read(&val) != 2)
            break;

        /* Apply val here. */
        ...
```

```
        /* Snooze for a bit. */
        snooze(1000);
    }
    a2d->Close();
    delete a2d;
}
```

BD2A

Creating and using a BD2A object follows the same outline as shown above, but instead of reading a **ushort** value, you write a **uchar**. The **Write()** function returns 1 if successful:

```
#include <D2A.h>

void WriteD2A1()
{
    uchar val;
    BD2A *d2a = new BD2A();

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

    while ( /* whatever */ ) {
        /* Get an 8-bit value from somewhere. */
        val = ...;

        if (d2a->Write(val) != 1)
            break;

        snooze(1000);
    }
    d2a->Close();
    delete d2a;
}
```

The DAC performs a “sample and hold”: The voltage that the DAC produces on a particular channel (and to which it sets the appropriate GeekPort pin) is maintained until another **Write()** call (on the same channel) changes the setting. Furthermore, the “hold” persists across BD2A objects: Neither closing nor deleting a BD2A object affects the voltage that’s produced 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

BA2D(), BD2A()

long **BA2D(**void**)**

long **BD2A(**void**)**

Creates a new object that can open an ADC or DAC channel (respectively). The particular channel is specified in a subsequent **Open()** call. Constructing a new BA2D or BD2A object doesn't affect the state of the ADC or DAC.

~BA2D(), ~BD2A()

virtual **~BA2D(**void**)**

virtual **~BD2A(**void**)**

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

Important: 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()

long **Open(**const char **name***)**

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

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

Open() opens the named ADC (BA2D) or DAC (BD2A) channel. The channel names (as you would pass them to **Open()**)are:

BA2D Channels	BD2A Channels
"A2D1"	"D2A1"
"A2D2"	"D2A2"
"A2D3"	"D2A3"
"A2D4"	"D2A4"

See the GeekPort connector illustration, above, for the correspondences between the channel names and the GeekPort connector pins.

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 BA2D or 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 closed—even if the channel that you're attempting to open is the channel that the object already has open.

Opening an ADC or 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.

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

IsOpen() returns **TRUE** if the object holds its assigned channel open channel is successfully opened. 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()**.

Read()

BA2D:

long **Read(**ushort ******adc_12_bit***)**

BD2A:

long **Read(**uchar ******dac_8_bit***)**

BA2D's **Read()** function causes the ADC to sample and convert (within a 12-bit range) the voltage level on the GeekPort pin that corresponds to the object's ADC channel. The 12-bit unsigned value is returned by reference in the *adc_12_bit* argument.

BD2A's **Read()** returns, by reference in *dac_8_bit*, the value that was most recently written to the object's particular DAC channel. The value needn't have been written by this object—it could have been written by the channel's previous opener.

Important: The BD2A **Read()** function returns a value that's cached by the DAC driver—it 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 an ADC or DAC channel before calling **Read()**. The functions return **B_ERROR** if a channel isn't open, or if, for any other reason, the read failed. Otherwise they return the number of bytes that were read: 2 in the case of a BA2D, 1 for a BD2A object. Note that it's not an error to read the DAC before a value has been written to it.

Write()

BD2A only:

long **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()** call—possibly by a different BD2A object—changes 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).

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

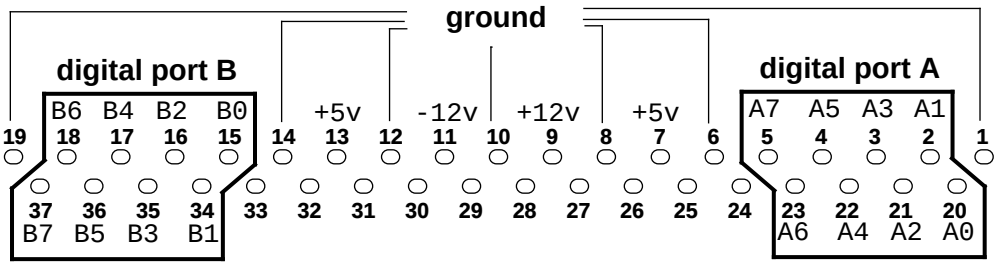
BDigitalPort

Derived from: public BObject

Declared in: <device/DigitalPort.h>

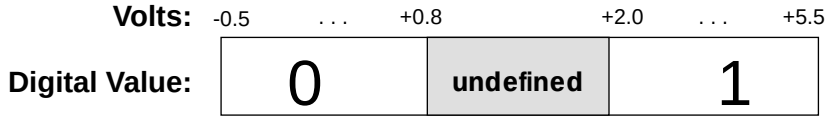
Overview

The BDigitalPort class is the programmer's interface to the GeekPort's two *digital ports*. Each digital port is an 8-bit wide device that can be set for input or output. 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 "BA2D and BD2A" on page 9 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.

BDigitalPort Objects

To access a digital port, you construct a BDigitalPort object, open it on the port you want, assign the object to work as either an input or an output, and then read or write a series of bytes from or to the object.

In the following example, we open and read from digital port A:

```
#include <DigitalPort.h>

void ReadDigitalPortA()
{
    char val;
    BDigitalPort *dPortA = new BDigitalPort();

    if (dPortA->Open("DigitalA") <= 0 ||
        dPortA->SetAsInput() != B_NO_ERROR) {
        ~dPortA;
        return;
    }

    while ( /* whatever */ ) {

        /* Read() returns the number of bytes that were
         * read; a successful read returns the value 1.
         */
        if (dPortA->Read(&val) != 1)
            break;

        /* Do something with the value. */
        ...

        /* Snooze for a bit. */
        snooze(1000);
    }
    dPortA->Close();
    delete dPortA;
}
```

As shown here, the BDigitalPort is constructed without reference to a specific port. It's not until you actually open the object (through **Open()**) that you have to identify the port that you want; identification is by name, "DigitalA" or "DigitalB". The **Read()** function returns only one value per invocation, and is untimed—if you don't provide some sort of tethering (as we do with **snooze()**, above) the read loop will spin as fast as possible.

To safeguard against an inadvertant burst of equipment-destroying output, the digital port is set to be an input when it's opened, and automatically reset to be an input when you close it.

Using Both Digital Ports at the Same Time

To access both digital ports at the same time, you have to construct two BDigitalPort objects. One of the objects can be used as an output and the other an input, both as outputs, or both as inputs.

In the following example, digital port A is used to write data to an external device, while digital port B is used for acknowledgement signalling: Before each write we set port B to 0, and after the write we wait for port B to be set to 1. We're assuming that the external device will write a 1 to port B when it's ready to receive the next 8-bits of data.

```
void WriteAndAck()
{
    char val;
    BDigitalPort *dPortA = new BDigitalPort();
    BDigitalPort *dPortB = new BDigitalPort();

    if (dPortA->Open("DigitalA") <= 0 ||
        dPortA->SetAsOutput() != B_NO_ERROR)
        goto error_tag;

    if (dPortB->Open("DigitalB") <= 0 ||
        dPortB->SetAsOutput() != B_NO_ERROR) {
        goto error_tag;

    while ( /* whatever */ ) {

        /* Clear the acknowledgement signal. */
        val = 0;
        if (dPortB->Write(&val) != 1)
            break;

        /* Reset val to the data we want to send. */
        val = ...;

        if (dPortA->Write(val) != 1)
            break;

        /* Reset digital port B to be an input. */
        if (dPortB->SetAsInput() != B_NO_ERROR)
            break;

        /* Wait for the acknowledgement. */
        while (1) {
            if (dPortB->Read(&val) != 1)
                goto error_tag;
            if (val == 1)
                break;
            snooze(1000);
        }
```

```
        /* Reset digital port B to be an output. */
        if (dPortB->SetAsOutput() != B_NO_ERROR)
            break;
    }
error_tag:
    delete dPortA;
    delete dPortB;
}
```

Notice that the acknowledgement signal only takes one bit of digital port B. This leaves seven bits that the external device can use to send additional data (triggers or gates, for example). The restriction in this scheme, given the structure shown above, is that this additional data would have to be synchronized with the acknowledgement signal.

By extension, if the data that you want to write to the external device is, at most, only seven-bits wide, then you could rewrite this example to use a single port: You would mask one of the bits as the acknowledgment carrier, and let the other seven bits carry the data, toggling the port between input and ouput as needed; the actual implementation is left as an exercise for the reader.

Overdriving an Output Pin

One of the features of the digital ports is that you can "overdrive" a pin from the outside. This means that you can set a port to be an output, and then force a voltage back onto the pin from an external device and read that voltage with the **Read()** function *without having to reset the port to be an input.* Keep in mind that there's a 1 KOhm resistor behind the pin (on the BeBox side), so your "overdrive" circuit has to be hot enough to balance the resistance.

When you overdrive an output pin, the voltage on the pin is altered for as long as the external force keeps it there. If you write an "opposing" value to an overdriven pin (through **Write()**), the written value won't pull the pin—the overdriven value will still be enforced. As soon as the overdrive voltage is removed, the pin will produce the voltage that was more recently written to it by the **Write()** function.

Constructor and Destructor

BDigitalPort()

long **BDigitalPort(**void**)**

Creates a new object that can open one of the digital ports. The particular port is specified in a subsequent **Open()** call.

~BDigitalPort

virtual **~BDigitalPort(**void**)**

Destroys the object, but not before closing the port that the object holds open (if any).

Deleting a BDigitalPort object sets the port (at the driver level) to be an input. The values at the port's pins are, at that point, undefined.

Member Functions

Open(), IsOpen(), Close()

long **Open(**const char ***name***)**

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

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

Open() opens the named digital port; the *name* argument should be either "DigitalA" or "DigitalB". See the GeekPort illustration in the "Overview" section for the correspondences between the port names and the GeekPort connector pins.

A digital port can only be held open by one BDigitalPort object at a time; you should close the port as soon as you're finished with it. Furthermore, each BDigitalPort object can only hold one port open at a time. When you invoke **Open()**, the port that the object currently has open is automatically closed—even if the port that you're attempting to open is the port that the object already has open.

When you open a digital port, the device is automatically set to be an input. If you want the port to be an output, you must follow this call with a call to **SetAsOutput()**. Just to be safe, it couldn't hurt to explicitly set the port to be an input (through **SetAsInput()**) if that's what you want.

Open() returns a positive integer if the named port is successfully opened. Otherwise, it returns **B_ERROR**.

IsOpen() returns **TRUE** if the object currently has a port open, and **FALSE** if not.

Close() does the obvious. When a digital port is closed, it's set to be an input at the driver level.

Read()

long **Read(**char ***buf***)**

Reads the data that currently lies on the digital ports pins, and returns this data as a single word in *buf*. Although you usually read a digital port that's been set to be an input, it's also possible to read an output port. In any case, the port must be open.

If the port was successfully read, the function returns 1 (the number of bytes read). Otherwise, it returns **B_ERROR**.

SetAsInput(), SetAsOutput(), IsInput(), IsOutput()

long **SetAsInput(**void**)**

long **SetAsOutput(**void**)**

bool **IsInput(**void**)**

bool **IsOutput(**void**)**

SetAsInput() and **SetAsOutput()** set the object's port to act as an input or output. They return **B_ERROR** if the object isn't open, and **B_NO_ERROR** otherwise.

IsInput() and **IsOutput()** return **TRUE** and **FALSE** much as you would expect them to.

Write()

long **Write(**char *value***)**

Sends *value* to the object's port. The port continues to produce the written data until another **Write()** call changes the setting.

The object must be open as an output for this function to succeed. Success is indicated by a return value of 1 (the number of bytes that were written). Failure returns **B_ERROR**.

BJoystick

Derived from: public BObject

Declared in: <device/Joystick.h>

Overview

A BJoystick object provides an interface to a joystick connected to the BeBox. There are two joystick ports on the back of the machine, one above the other. With the aid of a simple Y connector, each of them can support two joysticks for a total of four 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.

Data Members

double **timestamp**	The time of the most recent update, as measured in microseconds from the time the machine was last booted.
short **horizontal**	The horizontal position of the joystick at the time of the last update.
short **vertical**	The vertical position of the joystick at the time of the last update.
bool **button1**	**TRUE** if the first button was pressed at the time of the last update, and **FALSE** if not.
bool **button2**	**TRUE** if the second button was pressed at the time of the last update, and **FALSE** if not.

horizontal and **vertical** 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 linear—identical 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 BeBox hardware < which will be more fully documented in a later release >.

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

Open(), Close()

long **Open(**const char *_name_**)**

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

These functions open the *name* joystick port and close it again. There are two ports on the back panel of the BeBox, and they have names that correspond to their labels on the machine (and in *The Be 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.

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.

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

Update()

long **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.

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

BSerialPort

Derived from: public BObject

Declared in: <device/SerialPort.h>

Overview

A BSerialPort object represents an RS-232 serial port connection to the BeBox. There are four such ports on the back of the machine.

Through BSerialPort functions, you can read data received at a serial port and write data over the connection. You can also configure the connection—for 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. For example:

```
BSerialPort *connection = new BSerialPort;
if ( connection->Open("serial2") > 0 ) {
    . . .
}
```

The BSerialPort object communicates with the driver for the port it has open. The driver maintains an input buffer of 1K bytes to collect incoming data and an output buffer half that size to hold outgoing data. When the object reads and writes data, it reads from and writes to these buffers.

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, but with a timeout of 0.0 microseconds, for 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(**void**)**

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()

DataRate() *see* SetDataRate()

FlowControl() *see* SetFlowControl()

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()

long **Open(**const char ***name***)**

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

These functions open the *name* serial port and close it again. Ports are identified by names that correspond to their labels on the back panel of the BeBox:

"serial1"
"serial2"
"serial3"
"serial4"

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.)

If it's able to open the port, **Open()** returns a positive integer. If unable, it returns **B_ERROR**.

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

See also: **Read()**

ParityMode() *see* SetDataBits()

Read(), SetBlocking(), SetTimeout()

long **Read(**void **buffer*, long *maxBytes***)**

void **SetBlocking(**bool *shouldBlock***)**

void **SetTimeout(**double *timeout***)**

Read() takes incoming data from the serial port driver and places it in the data *buffer* specified. In no case will it read more than *maxBytes*—a value that should reflect the capacity of the *buffer*; it returns the actual number of bytes read. **Read()** fails if the BSerialPort object doesn't have a port open.

The number of bytes that **Read()** reads 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.

SetTimeout() sets a time limit on how long **Read()** will block while waiting for data to arrive at the 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 if the limit is greater than 0.0, regardless of the *shouldBlock* flag.)

The *timeout* is expressed in microseconds and is limited to 25,500,000.0 (25.5 seconds); it's set to the maximum value if a greater amount of time is specified. Differences less than 100,000.0 microseconds (0.1 second) are not recognized; they're rounded to the nearest tenth of a second. If the *timeout* is set to 0.0 microseconds, **Read()** (and **WaitForInput()**) will not block.

The default *shouldBlock* setting is **TRUE**, but the default *timeout* is 0.0, which prevents blocking in any case. < In future releases, the default timeout will be an infinite amount of time; it won't impose a time limit on blocking. >

Like the standard **read()** system function, **Read()** returns the number of bytes it succeeded in placing in the *buffer*, which may be 0. It returns **B_ERROR** (–1) if there's an error of any kind—for example, if the BSerialPort object doesn't have a port open. It's not considered an error if a timeout expires.

See also: **Write()**, **Open()**, **WaitForInput()**

SetBlocking() *see* Read()

SetDataBits(), SetStopBits(), SetParityMode(), DataBits(), StopBits(), ParityMode()

void **SetDataBits(**data_bits *count***)**

void **SetStopBits(**stop_bits *count***)**

void **SetParityMode(**parity_mode *mode***)**

data_bits **DataBits(**void**)**

stop_bits **StopBits(**void**)**

parity_mode **ParityMode(**void**)**

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 *count* can be:

B_DATA_BITS_7 or
B_DATA_BITS_8

The default is **B_DATA_BITS_8**.

SetStopBits() sets the number of stop bits in each unit. It can be:

B_STOP_BITS_1 or
B_STOP_BITS_2

The default is **B_STOP_BITS_1**.

SetParityMode() sets whether the serial unit contains a parity bit and, if so, the type of parity used. The mode can be:

B_EVEN_PARITY,
B_ODD_PARITY, or
B_NO_PARITY

The default is **B_NO_PARITY**.

SetDataRate(), DataRate()

void **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. Permitted values are:

B_0_BPS	**B_200_BPS**	**B_4800_BPS**
B_50_BPS	**B_300_BPS**	**B_9600_BPS**
B_75_BPS	**B_600_BPS**	**B_19200_BPS**
B_110_BPS	**B_1200_BPS**	**B_38400_BPS**
B_134_BPS	**B_1800_BPS**	**B_57600_BPS**
B_150_BPS	**B_2400_BPS**	**B_115200_BPS**

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.

SetDTR()

long **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**.

See also: **SetRTS()**

SetFlowControl(), FlowControl()

void **SetFlowControl(**ulong *mask***)**

ulong **FlowControl(**void**)**

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

B_SOFTWARE_CONTROL	Control is maintained through XON and XOFF characters inserted into the data stream.
B_HARDWARE_CONTROL	Control is maintained through the Clear to Send (CTS) and Request to Send (RTS) pins.

The *mask* passed to **SetFlowControl()** and returned by **FlowControl()** can be just one of these constants—or it can be a combination of the two, in which case the driver will use both types of flow control together. It can also be 0, in which case the driver won't use any flow control. **B_HARDWARE_CONTROL** is the default.

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()

long **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**.

See also: **SetDTR()**

SetStopBits() *see* SetDataBits()

SetTimeout() *see* Read()

StopBits() *see* SetDataBits()

WaitForInput()

long **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 ready to be read when this function is called, it immediately returns without blocking and reports how many bytes there are. If data hasn't arrived, it blocks and waits for the first bytes to be transmitted. When they're detected, it immediately reports how many have arrived.

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. A timeout of 0.0 microseconds doesn't give **WaitForInput()** enough time to block; it returns immediately.

See also: **Read()**

Write()

long **Write(**const void **data*, long *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.

Like the **write()** system function, **Write()** returns the actual number of bytes written, which will never be more than *numBytes*, and may be 0. If it fails (for example, if the BSerialPort object doesn't have a serial port open) or if it's interrupted before it can write anything, it returns **B_ERROR** (–1).

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

Constants and Defined Types

This section lists all the constants and types defined for the BJoystick, BSerialPort, BDigitalPort, BA2D, and BD2A classes—though, in fact, only the BSerialPort class relies on any defined constants or types. Everything listed here is explained more fully in the descriptions of the member functions of that class.

Constants

data_bits Constants

<device/SerialPort.h>

Enumerated constant
B_DATA_BITS_7
B_DATA_BITS_8

These constants name the possible number of data bits in a serial unit.

See also: **BSerialPort::SetDataBits()**

data_rate Constants

<device/SerialPort.h>

Enumerated constant	Enumerated constant
B_0_BPS	**B_1200_BPS**
B_50_BPS	**B_1800_BPS**
B_75_BPS	**B_2400_BPS**
B_110_BPS	**B_4800_BPS**
B_134_BPS	**B_9600_BPS**
B_150_BPS	**B_19200_BPS**
B_200_BPS	**B_38400_BPS**
B_300_BPS	**B_57600_BPS**
B_600_BPS	**B_115200_BPS**

These constants give the possible rates—in bits per second (bps)—at which data can be transmitted and received over a serial connection.

See also: **BSerialPort::SetDataRate()**

Flow Control Constants

<device/SerialPort.h>

Enumerated constant

B_SOFTWARE_CONTROL
B_HARDWARE_CONTROL

These constants form a mask that records the method(s) of flow control the serial port driver should use.

See also: **BSerialPort::SetFlowControl()**

parity_mode Constants

<device/SerialPort.h>

Enumerated constant

B_NO_PARITY
B_ODD_PARITY
B_EVEN_PARITY

These constants list the possibilities for parity when transmitting data over a serial connection.

See also: **BSerialPort::SetDataBits()**

stop_bits Constants

<device/SerialPort.h>

Enumerated constant

B_STOP_BITS_1
B_STOP_BITS_2

These constants name the possible number of stop bits in a serial unit.

See also: **BSerialPort::SetDataBits()**

Defined Types

data_bits

<device/SerialPort.h>

typedef enum { . . . } **data_bits**

This type is used to set and return the number of data bits in a serial unit.

See also: "**data_bits** Constants" above, **BSerialPort::SetDataBits()**

data_rate

<device/SerialPort.h>

typedef enum { . . . } **data_rate**

This type is used to set and return the rate at which data is sent and received through a serial connection.

See also: "**data_rate** Constants" above, **BSerialPort::SetDataRate()**

parity_mode

<device/SerialPort.h>

typedef enum { . . . } **parity_mode**

This type is used to set and return the type of parity that should be used when sending and receiving data.

See also: "**parity_mode** Constants" above, **BSerialPort::SetDataBits()**

stop_bits

<device/SerialPort.h>

typedef enum { . . . } **stop_bits**

This type is used to set and return the number of stop bits in a serial unit.

See also: "**stop_bits** Constants" above, **BSerialPort::SetDataBits()**

Developing a Device Driver

A device driver ties an input/output hardware device to the computer's operating system. To develop a driver, you have to know about both ends of that link:

- On the one hand, you need to be thoroughly familiar with the hardware device and its particular interface.
- On the other hand, you must understand the operating system and the demands it places on the driver.

Hardware specifications and manuals can provide you with the first kind of information; this documentation can help only with the second—that is, with information specific to the Be operating system. On the next page, you'll see a list of recommended documentation for the DMA controller, the PCI bus, and other hardware found inside the BeBox. This book is concerned solely with how a driver must be structured to work with Be system software.

Overview

On the BeBox, device drivers run as dynamically loaded add-on modules—as extensions of a host component of the operating system. The Application Server, the part of the operating system that's responsible for all graphics operations, is the host for graphics card drivers. The Print Server hosts drivers for printers. The kernel acts as the host for all other drivers. The kernel and the two servers load drivers on demand, and can unload them when they're no longer needed.

Because drivers are linked to their hosts and run in the host's address space, they must play by the host's rules. The kernel and servers impose three different kinds of restrictions on loadable drivers:

- A driver must be constructed so that it can respond to its host. It has to be able to inform the host of the device or devices it drives, and it has to provide functions that the host can call to operate the driver. As an add-on module, a driver lacks an independent main thread of execution (and a **main()** function). Instead, it provides the host with entry points to driver functionality and responds only to the host's instructions.
- A driver can call only those functions that it implements itself or that the host makes available to it. It cannot, for example, link against the shared system library and call any function it wants from the Kernel Kit or Storage Kit. It might statically link against a private library, but it typically links only to the host and is limited to calling functions that the host exports.

- The driver must be compiled as an add-on module and it must be installed in a directory where the host expects to find it.

Although the kernel, the Application Server, and the Print Server all impose these three kinds of restrictions on their loadable drivers, they each impose a different set of restrictions. The kernel's rules are not the Application Server's rules, and the Application Server's are not the same as the Print Server's.

To learn the rules that apply to the type of driver you intend to develop, begin with the section listed in the following chart:

To develop:	Go to:
A driver for a graphics card	"Developing a Driver for a Graphics Card" on page 77
A driver for a printer	< Wait until the next release, or contact Be developer support. The interface for printer drivers is under development and not yet documented. >
All other drivers	"Developing a Kernel-Loadable Driver" on page 41

Recommended Reading

For information on the PCI bus:

> *PCI Local Bus Specification*, revision 2.1, June 1, 1995, PCI Special Interest Group, PO Box 14070, Portland OR 97214, (800) 433-5177 or (503) 797-4207

For information on the ISA bus, ISA 8259 interrupt controller, and ISA-standard 8237 DMA controller:

> *82378 System I/O (SIO)*, August 1994, order number 290473-004, Intel Corporation, 2200 Mission College Boulevard, PO Box 58119, Santa Clara, CA 95052

For information on the SCSI common access method (CAM):

> Draft Proposed American National Standard *SCSI-2 Common Access Method Transport and SCSI Interface Module*, ASC X3T–10, revision 12, December 14, 1994, American National Standards Institute, 11 West 42nd Street, New York, NY 10036

Developing a Kernel-Loadable Driver

At the most basic level, devices (other than graphics cards) are controlled by system calls that the kernel traps and translates for the driver. Five different kinds of functions control input/output devices on the BeBox:

open()	Opens a device for reading or writing.
close()	Closes a device that was previously opened.
read()	Reads data from the device.
write()	Writes data to the device.
ioctl()	Formats, initializes, queries, and otherwise controls the device.

All these functions, except **ioctl()**, are Posix-compliant. Instead of **ioctl()**, Posix defines a set of functions like **tcsetattr()** and **tcflush()** to control data terminals. These functions are supported, but they can be treated as special cases of **ioctl()**. (Posix also defines a **fcntl()** function for file control that has the same syntax as **ioctl()**.)

When **open()**, **close()**, **read()**, **write()**, or **ioctl()** is called for a device, the kernel expects a driver to do the work that's required. Each driver must implement a set of functions that correspond directly to the five system calls. Everything the driver does to operate the device is initiated through one of these functions.

Because drivers run in the kernel's address space as extensions of the kernel, they must conform to the kernel's expectations. Separate sections discuss the three types of restrictions that the kernel imposes on drivers:

- "Entry Points" on page 42 describes how drivers must be constructed to work with the kernel. The driver must provide the kernel with entry points to its functionality and follow the kernel's instructions.
- "Exported Functions" on page 49 discusses the kinds of functions that the kernel exports for drivers. These include some from the Kernel Kit, Support Kit, and standard C library, and some that are defined especially for drivers. The driver is limited to calling functions that it itself implements or that the kernel exports.
- "Installation" on page 54 discusses how to compile a driver and install it on the BeBox. The driver must be installed where the kernel can find it.

The exported functions that the kernel defines specifically for drivers are documented following these three sections.

Entry Points

The kernel loads a device driver when it's needed—typically when someone first attempts to open the device for reading or writing. Opening a device is a prerequisite to using it.

To the **open()** function, drivers are identified by a fictitious pathname beginning with **/dev/**. For example, this code opens the parallel port driver for writing:

```
int fd = open("/dev/parallel", O_WRONLY);
```

The first thing the kernel must do is match the device name—"/dev/parallel" in this case—to a driver; it must find a driver for the device. The driver might be one that has already been loaded, or it might be one that the kernel must search for and load. Loadable drivers reside in the **/system/drivers** directory; this is where the kernel looks for drivers and where they all must be installed.

Once a driver has been located and loaded, the kernel begins communicating with it—first to get information from it and test whether it's the right driver, then to initialize it and have it open the device.

One key piece of information that the kernel needs from the driver is the names of all its devices. Another is a list of the functions it can call to exercise those devices. For each device, the driver implements a set of hook functions that open the device, control it, read data from it, write data to it, and perhaps eventually close it. These hooks correspond to the system functions discussed above.

To give the kernel initial access to this information, drivers make declarations—of functions and data structures—using names the kernel will look for. Five such names will be discussed in the following sections:

init_driver()	Initializes the driver after it's loaded.
uninit_driver()	Cleans up after the driver before it's unloaded.
devices	Declares the devices and their hook functions.
publish_device_names()	Lists the devices the driver handles.
find_device_entry()	Associates a device with its hook functions.

These are the main entry points for driver control.

Driver Initialization

Immediately after loading a driver, the kernel gives it a chance to initialize itself. If the driver implements a function called **init_driver()**, the kernel will call it before proceeding with anything else—before asking the driver to open a device. The function should expect no arguments and return either **B_ERROR** or **B_NO_ERROR**:

long **init_driver(**void**)**

A return of **B_ERROR** means that the driver can't continue; the kernel will consequently unload it. A return of **B_NO_ERROR** means that all is well; the kernel will continue by

asking the driver to open a device. The absence of an **init_driver()** function is equivalent to a return of **B_NO_ERROR**.

init_driver() might go a long way toward initializing the data structures the driver uses to do its work—for example, setting up needed semaphores. However, details specific to a particular device should be left to the hook function that opens that device.

When the kernel is about to get rid of a driver, it gives the driver a chance to undo what **init_driver()** did. If the driver implements a function called **uninit_driver()**, the kernel will call it immediately before unloading the driver. This function has the same syntax as the initialization function:

long **uninit_driver(**void**)**

This function can do nothing to prevent the driver from being unloaded. It should simply clean up after the driver—for example, delete semaphores—and return **B_NO_ERROR**.

Driver initialization and its opposite happen just once—when the driver is loaded and unloaded. In contrast, devices might be opened and closed many times while the driver continues to reside in the kernel.

Device Declarations

For the kernel to find the driver for a given device, all drivers must declare the names of the devices they control. For the kernel to be able to communicate with the driver to operate the device, every driver must declare a set of device-specific hook functions the kernel can call.

These declarations are made in a **device_entry** structure that maps the device name to the set of hook functions. This structure is declared in **device/Drivers.h** and contains the following fields:

const char ***name**	The name of the device—for example, "/dev/serial". This is the same name that's passed to **open()**. Driver code can assign any name it wants to the device, but it must begin with the "/dev/" prefix, which distinguishes devices from ordinary files.
device_open_hook **open**	The function that the kernel should call to open the device. The kernel will invoke this function to respond to **open()** system calls.
device_close_hook **close**	The function that should be called to close the device. It corresponds to the **close()** system call.
device_control_hook **control**	The function that the kernel should call to control the device, including querying the driver for information about it. The kernel will invoke this function to respond to **ioctl()** calls.

device_io_hook **read**	The function that should be called to read data from the device. The kernel will invoke this function to respond to the **read()** system call.
device_io_hook **write**	The function that should be called to write data to the device. It corresponds to the **write()** system call.

(The five functions are described in more detail under “Hook Functions” below.)

A driver declares one **device_entry** structure for each device it can drive. If it can handle more than one device, it must provide a **device_entry** structure for each one. If it permits a device to be referred to by more than one name, it must provide a structure for each name it recognizes.

There are two ways for a driver to provide the kernel with the information in a **device_entry** structure: If the list of devices is known at compile time, the driver can declare them statically. If the list might change at run time, it can return them dynamically.

Static Drivers

Most drivers are designed to handle a fixed set of known devices—perhaps a single device or perhaps many. Such drivers should declare a null-terminated array of **device_entry** structures under the global name **devices**:

device_entry **devices**[]

For example, the serial port driver might declare a **devices** array that looks like this:

```
device_entry devices[7] = {
    {"/dev/serial1", open_func, close_func, control_func,
            read_func, write_func},
    {"/dev/serial2", open_func, close_func, control_func,
            read_func, write_func},
    {"/dev/serial3", open_func, close_func, control_func,
            read_func, write_func},
    {"/dev/serial4", open_func, close_func, control_func,
            read_func, write_func},
    {"/dev/com3", open_com_func, close_func, control_com_func,
            read_func, write_func},
    {"/dev/com4", open_com_func, close_func, control_com_func,
            read_func, write_func},
    0
};
```

In this case, the driver handles the four serial ports seen on the back of the BeBox, each of which it identifies by a different name. It can also control “com3” and “com4” ports on an add-on board.

As this example illustrates, the hook functions declared in a **device_entry** structure are specific to the device. For the most part, the serial port driver above uses the same set of

functions to operate all the devices, but declares special functions for opening and controlling “com3” and “com4”.

Note also that the array is null terminated.

Dynamic Drivers

A driver can also provide **device_entry** information dynamically. Instead of a **devices** array, it implements two functions, **publish_device_names()** and **find_device_entry()**.

The first of these functions should be declared as follows:

char ****publish_device_names(**const char **deviceName***)**

If passed a proposed *deviceName* that matches the name of a device the driver handles, or if passed a **NULL** device name, this function should return a null-terminated array of all the names of all the devices that it handles. For example, the serial port driver described above would return the following array:

```
"/dev/serial1",
"/dev/serial2",
"/dev/serial3",
"/dev/serial4",
"/dev/com3",
"/dev/com4",
0
```

However, if the proposed device name doesn’t match any that the driver handles, **publish_device_names()** should return **NULL**.

While **publish_device_names()** informs the kernel of the devices that the driver handles, **find_device_entry()** returns entry information about a particular device. It has the following form:

device_entry ***find_device_entry(**const char **deviceName***)**

If passed the name of a device that the driver knows about, this function should return the **device_entry** for that name. If the *deviceName* doesn’t match one of the driver’s devices, it should return **NULL**.

The kernel first calls **publish_device_names()** during the boot sequence to find what devices the driver handles. It may call the function again to update the list when it tries to match a driver to a specific device. If a match is made, it calls **find_device_entry()** to get the list of hook functions for the device.

Hook Functions

The five hook functions that are declared in a **device_entry** structure can have any names that you want to give them, provided that they don't clash with names that the kernel exports (see "Exported Functions" on page 49). However, their syntax is strictly prescribed by the kernel (through type definitions found in **device/Drivers.h**).

The five functions have two points in common: First, they each return an error code, which should be 0 (**B_NO_ERROR**) if there is no error. The error value is passed through as the return value for the **open()**, **close()**, **read()**, **write()**, or **ioctl()** system call that caused the kernel to invoke the driver function. The driver should return error values that are compatible with ones that are expected from those functions.

Second, all five functions are passed information identifying the device. As their first argument, they receive a pointer to a **device_info** structure (also defined in **device/Drivers.h**), which contains just two fields:

device_entry ***entry**	The **device_entry** structure for the device that's being operated on. This is a copy of information that the driver declared in its **devices** array or that its **find_device_entry()** function returned.
void ***private_data**	Arbitrary data that describes the device. This data is a way for the driver to record information about the device and have it persist between function calls. Although the kernel stores this data and passes it to the driver, the driver initializes it and maintains it; the kernel doesn't query or modify it.

Thus, all of the device-specific hook functions have the same return types and initial arguments:

long **function(**device_info **info*, . . .**)**

Differences among the functions are discussed below.

Opening and Closing a Device

The function that opens a device is of type **device_open_hook** and the one that closes it is of type **device_close_hook**. They're defined as follows:

typedef long (***device_open_hook**)**(**device_info **info*, ulong *flags***)**

typedef long (***device_close_hook**)**(**device_info **info***)**

The *flags* mask that's passed to the **open()** system call is passed through to the device function. It typically will contain a flag like **O_RDONLY**, **O_RDWR**, or **O_WRONLY**.

Since the hook function that opens the device is the first one that's called, it might set up the **device_info** description of the device (to the extent that **init_driver()** hasn't already

done so). It might also use that description, or some static data, to record whether or not the device is currently open. Typically, only one process can have a device open at a time. If the hook function sees that the device is already open, it can refuse to open it again.

Whatever values these functions return will also be returned by the **open()** and **close()** system calls.

Reading and Writing Data

The driver functions that read data from and write data to a device must be of type **device_io_hook**, which is defined as follows:

```
typedef long (*device_io_hook)(device_info *info,
                               void *buffer, ulong numBytes, ulong position)
```

The function that reads data from the device should place up to *numBytes* of data into the specified *buffer* beginning at *position*. The hook that writes to the device should take *numBytes* of data from the *buffer* beginning at *position*.

Whatever values these functions return will also be returned by the **read()** and **write()** system calls.

Controlling the Device

The hook function that initializes, formats, queries, and otherwise controls a device is of type **device_control_hook**, defined as follows:

```
typedef long (*device_control_hook)(device_info *info, ulong op, void *data)
```

The second argument, *op*, is a constant that specifies the particular control operation that the function should perform. The third argument, *data*, points either to some information that the control function needs to carry out the *op* operation or to a data structure that it should fill in with information that the operation requests. The interpretation of the data pointer depends entirely on the nature of the operation and will differ from operation to operation; the *op* and *data* arguments go hand-in-hand.

For example, if the *op* code is **SET_CONFIG**, *data* might point to a structure with values that the control function should use to re-configure the device. If the operation is **GET_ENABLED_STATE**, *data* might point to an integer that the function would be expected to set to either 1 or 0. If it's **RESTART**, *data* might simply be **NULL**.

The kernel defines a number of control operations (which are explained in the next section). These are operations that the kernel might call upon any driver to perform.

If you define your own control operations (for an **ioctl()** call on your driver), you should be sure that they aren't confused with any that the kernel currently defines—or any that it will define in the future. We pledge that all system-defined control constants will have values

below **B_DEVICE_OP_CODES_END**. The constants you define should be increments above this value. For example:

```
enum {
    REPORT_STATUS = B_DEVICE_OP_CODES_END + 1,
    SET_TIMER,
    . . .
}
```

If a control function doesn't recognize the *op* code it's passed or can't perform the requested operation, it should return **B_ERROR** (–1).

Control Operations

Several control operations are defined by the kernel. The kernel can request any driver to perform these operations, even in the absence of an **ioctl()** call. A control function should respond to as many of these requests as it can. It should respond to inappropriate or unrecognized requests by returning **B_ERROR**.

The set of system-defined control operations is described below.

B_GET_SIZE and B_SET_SIZE

These control operations request the driver to get and set the memory capacity of the physical device. The capacity is measured in bytes and is recorded as a **ulong** integer. For a **B_GET_SIZE** request, the control function should write this number to the location referred to by the *data* pointer. For **B_SET_SIZE**, *data* will be the requested number of bytes (not a pointer to it).

B_SET_BLOCKING_IO and B_SET_NONBLOCKING_IO

These operations determine whether or not the driver should block when reading and writing data. **B_SET_BLOCKING_IO** requests the driver to put itself in blocking mode. Its read function should wait for data to arrive if none is readily available and its write function should wait for the device to be ready to accept data if it's not immediately free to take it. If **B_SET_NONBLOCKING_IO** is requested, the read function should return immediately if there is no data available to read and the write function should return immediately if the device isn't ready to accept written data.

For these operations, the *data* argument doesn't contain a meaningful value.

B_GET_READ_STATUS and B_GET_WRITE_STATUS

These control operations request the driver to report whether or not it's ready to read and write without blocking. The control function should respond by placing **TRUE** or **FALSE** as a **ulong** integer in the location that *data* points to.

For **B_GET_READ_STATUS**, it should respond **TRUE** if there's data waiting to be read, and **FALSE** if not. For **B_GET_WRITE_STATUS**, it should respond **TRUE** if the device is free to accept data, and **FALSE** if not.

B_GET_GEOMETRY

This *op* code requests the driver to supply information about the physical configuration of the device; it's generally appropriate only for mass storage devices. The control function should write the requested information into the **device_geometry** structure that the *data* pointer refers to. A **device_geometry** structure contains the following fields:

ulong **bytes_per_sector**	The number of bytes in each sector of storage.
ulong **sectors_per_track**	The number of sectors in each track.
ulong **cylinder_count**	The number of cylinders.
ulong **head_count**	The number of heads.
bool **removable**	Whether or not the storage medium can be removed (**TRUE** if it can be, **FALSE** if not).
bool **read_only**	Whether or not the medium can be read but not written (**TRUE** if it cannot be written, **FALSE** if it can).
bool **write_once**	Whether or not the medium can be written once, after which it becomes read-only (**TRUE** if it can be written only once, **FALSE** if it cannot be written or can be written more than once).

B_FORMAT

This operation requests the control function to format the device. The *data* argument doesn't contain any valid information.

Exported Functions

After a driver has been loaded, it runs as part of the kernel in the kernel's address space. It therefore is restricted to calling functions (a) that it implements or (b) that the kernel makes available to it. The driver links against the kernel alone; it cannot also independently link to something else, even the standard C library.

The kernel exports five kinds of functions so that they're available to a driver:

- Support Kit functions, such as **read_32_swap()** and **atomic_or()**. Like the error constants and data types that are defined in the Support Kit, these functions are available to drivers.
- Standard kernel functions, such as **area_for()** and **write_port()**, that were documented in the chapter on the Kernel Kit. Because the driver runs in the kernel's address space, it accesses these functions directly, not through the Kit. For this reason, not all of the functions are available to drivers; there are some services the kernel can provide to others, but not to itself.
- Library functions that the kernel incorporates. These are functions from the standard C library that have been adopted by the kernel and that the kernel, in turn, exports to the driver. They're a small, selected subset of library functions.
- Standard system calls, such as **read()** and **ioctl()**.
- Special functions that are implemented specifically for device drivers.

Functions from all five groups are listed in the sections below. Special driver functions are documented in detail in the section entitled "Functions for Drivers" on page 55.

Support Kit Functions

The kernel exports the following Support Kit functions:

read_16_swap()
read_32_swap()
write_16_swap()
write_32_swap()
atomic_and()
atomic_or()
atomic_add()
real_time_clock()

See the chapter on the Support Kit for descriptions of these functions.

Kernel Kit Functions

Most functions from the Kernel Kit are available to drivers. However, a few are not, sometimes because it would make no sense for a driver to call the function, and sometimes because it's difficult for the kernel to provide its very basic services to its own modules. In some cases, a special function is defined for drivers that takes the place of the missing Kit function. For example, **spawn_thread()** can't spawn a thread in the kernel. Since drivers run in the kernel, they need to use the special **spawn_kernel_thread()** instead. Similarly, **debugger()** can't be used to debug the kernel. Drivers should call **kernel_debugger()** instead.

The following Kernel Kit functions are exported for drivers:

Semaphores

create_sem()
acquire_sem()
acquire_sem_etc()
release_sem()
release_sem_etc()
get_sem_info()
get_nth_sem_info()
get_sem_count()
set_sem_owner()
delete_sem()

Threads

find_thread()
rename_thread()
set_thread_priority()
get_thread_info()
get_nth_thread_info()
suspend_thread()
resume_thread()
wait_for_thread()
exit_thread()
kill_thread()

Teams

kill_team()
get_team_info()
get_nth_team_info()

Ports

create_port()
read_port()
read_port_etc()
write_port()
write_port_etc()
get_port_info()
get_nth_port_info()
find_port()
port_count()
port_buffer_size()
port_buffer_size_etc()
set_port_owner()
delete_port()

Time

snooze()
system_time()

Other

area_for()
get_system_info()

C Library Functions

The kernel exports a small number of functions from the standard C library. They include:

Functions declared in stdlib.h

atof()
atoi()
atol()
strtod()
strtol()
strtoul()
bsearch()
qsort()
malloc()
calloc()
free()
abs()
div()
labs()
ldiv()

Functions and macros declared in ctype.h

isalnum()
isalpha()
iscntrl()
isdigit()
isxdigit()
islower()
isupper()
ispunct()
isspace()
isprint()
isgraph()
tolower()
toupper()

Functions declared in string.h

strlen()
strcat()
strncat()
strcpy()
strncpy()
strcmp()
strncmp()
strchr()
strrchr()
strspn()
strcspn()
strstr()
strpbrk()
memset()
memchr()
memcmp()
memcpy()
memmove()

Functions declared in stdio.h

sprintf()
vsprintf()

The driver accesses these functions from the kernel, not from the library.

System Calls

The kernel also exports the five system calls that control devices:

open()
close()
read()
write()
ioctl()

Kernel Functions for Drivers

The kernel defines the following functions especially for drivers. For full documentation of these functions, see “Functions for Drivers” on page 55.

Spinlocks:

acquire_spinlock()
release_spinlock()

Disabling interrupts:

disable_interrupts()
restore_interrupts()

Interrupt handling:

set_io_interrupt_handler()
disable_io_interrupt()
enable_io_interrupt()
set_isa_interrupt_handler()
disable_isa_interrupt()
enable_isa_interrupt()

Memory management:

lock_memory()
unlock_memory()
get_memory_map()
isa_address()
ram_address()

ISA DMA:

start_isa_dma()
start_scattered_isa_dma()
make_isa_dma_table()
lock_isa_dma_channel()
unlock_isa_dma_channel()

PCI:

read_pci_config()
write_pci_config()
get_nth_pci_info()

Debugging:

dprintf()
set_dprintf_enabled()
kernel_debugger()

Hardware versions:

motherboard_version()
io_card_version()

SCSI common access method:

xpt_init()
xpt_ccb_alloc()
xpt_ccb_free()
xpt_action()
xpt_bus_register()
xpt_bus_deregister()

Other

spin()
spawn_kernel_thread()

Installation

The driver must be compiled as an add-on image, which in practical terms is much the same as compiling a shared library. *The Kernel Kit* chapter explains add-on images, and the Metrowerks *CodeWarrior* manual gives compilation instructions. In summary, you'll need to specify the following options for the linker (as **LDFLAGS** in the **makefile**):

- Instruct the linker to produce an add-on image by listing the **–G** (or **–sharedlibrary**) option.
- Disable the default behavior of linking against the shared system library by including the **–nodefaults** option.
- Export the driver's entry points so that the kernel can access them. The simplest way to do this is to export everything with the **–export all** option.
- Link the driver against the kernel by specifying the **/system/kernel** file. This is the only file that the driver should be linked against.

For the kernel to be able to find the compiled driver, it must be installed in the **/system/drivers** directory. This is the only place that the kernel looks for drivers to load.

When an attempt is made to open a device, the kernel first looks for its driver among those that are already loaded. Failing that, it looks on a floppy disk (in **/fd/system/drivers**). Failing to find one there, it looks next on the boot disk (in **/boot/system/drivers**).

If the **/system/drivers** directory contains more than one driver for the same device, it's indeterminate which one will be loaded.

You can give your driver any name you wish, as long as it doesn't match the name of another file in **/system/drivers**.

Functions for Drivers

The kernel exports a number of functions for the benefit of device drivers. These are functions that drivers can call to do their work; they're not functions that are available to applications. Although implemented by the kernel, they're not part of the Kernel Kit. The device driver accesses these functions directly from the kernel, not through a library.

acquire_spinlock(), release_spinlock()

<device/KernelExport.h>

void **acquire_spinlock(**spinlock *_lock_**)**

void **release_spinlock(**spinlock *_lock_**)**

These functions acquire and release the *lock* spinlock. Spinlocks, like semaphores, are used to protect critical sections of code that must remain on the same processor for a single path of execution—for example, code that atomically accesses a hardware register or a shared data structure. A common use for spinlocks is to protect data structures that both an interrupt handler and normal driver code must access.

However, spinlocks work quite differently from semaphores. No count is kept of how many times a thread has acquired the lock, for example, so calls to **acquire_spinlock()** and **release_spinlock()** should not be nested. More importantly, **acquire_spinlock()** spins while attempting to acquire the lock; it doesn't block or release its hold on the CPU.

These functions assume that interrupts have been disabled. They should be nested within calls to **disable_interrupts()** and **restore_interrupts()** as follows:

```
spinlock lock;
cpu_status former = disable_interrupts();
acquire_spinlock(&lock);
/* critical code goes here */
release_spinlock(&lock);
restore_interrupts(former);
```

These two pairs of functions enable the thread to get into the critical code without rescheduling. Disabling interrupts ensures that the thread won't be preemptively rescheduled. Because **acquire_spinlock()** doesn't block, it provides the additional assurance that the thread won't be voluntarily rescheduled.

Executing the critical code under the protection of the spinlock guarantees that no other thread will execute the same code at the same time on another processor. Spinlocks should be held only as long as necessary and released as quickly as possible.

See also: **create_spinlock()**

create_spinlock(), delete_spinlock()

<device/KernelExport.h>

spinlock ***create_spinlock(**void**)**

void **delete_spinlock(**spinlock **lock***)**

< These functions will, when implemented and exported, produce and destroy spinlocks. Currently, they're declared but not exported. To create a spinlock at present, simply declare a **spinlock** variable and pass a pointer to it to **acquire_spinlock()**. >

See also: **acquire_spinlock()**

disable_interrupts(), restore_interrupts()

<device/KernelExport.h>

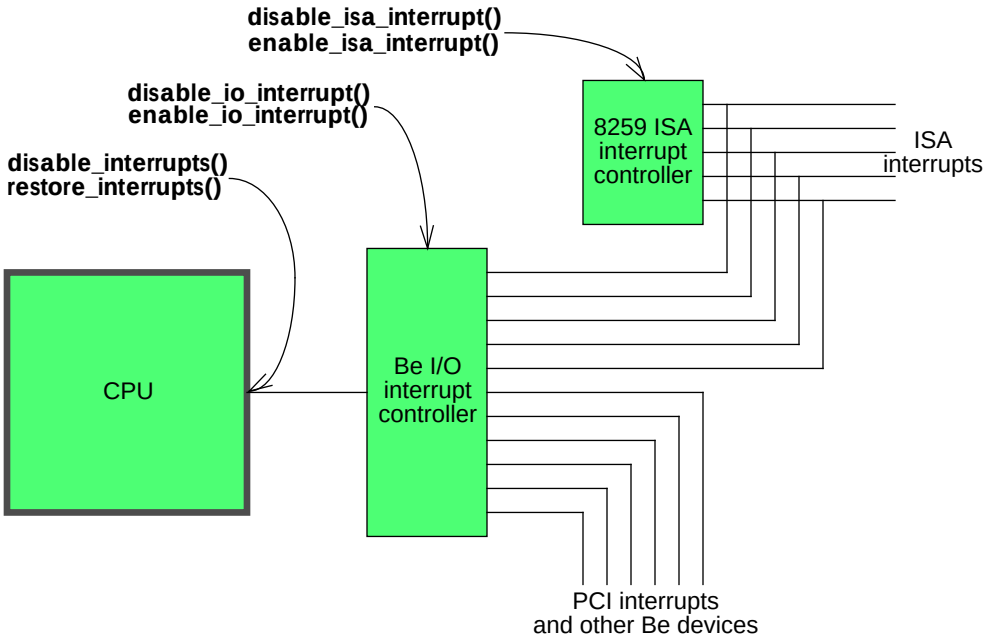
cpu_status **disable_interrupts(**void**)**

void **restore_interrupts(**cpu_status *status***)**

These functions disable interrupts at the CPU (the one the caller is currently running on) and restore them again. **disable_interrupts()** prevents the CPU from being interrupted and returns its previous status—whether or not interrupts were already disabled before the **disable_interrupts()** call. **restore_interrupts()** restores the previous *status* of the CPU, which should be the value that **disable_interrupts()** returned. Passing the status returned by **disable_interrupts()** to **restore_interrupts()** allows these functions to be paired and nested.

As diagrammed below, individual interrupts can be enabled and disabled at two other hardware locations. **disable_isa_interrupt()** and **enable_isa_interrupt()** work at the ISA

standard 8259 interrupt controller, and **disable_io_interrupt()** and **enable_io_interrupt()** act at the Be-defined I/O interrupt controller that combines ISA and PCI interrupts.



Interrupts that have been disabled by **disable_interrupts()** must be reenabled by **restore_interrupts()**.

See also: **acquire_spinlock()**, **set_io_interrupt_handler()**, **set_isa_interrupt_handler()**

disable_io_interrupt() *see* set_io_interrupt_handler()

disable_isa_interrupt() *see* set_isa_interrupt_handler()

dprintf(), set_dprintf_enabled(), kernel_debugger()

<device/KernelExport.h>

void **dprintf(**const char **format*, ...**)**

bool **set_dprintf_enabled(**bool *enabled***)**

void **kernel_debugger(**const char **string***)**

dprintf() is a debugging function that has the same syntax and behavior as standard C **printf()**, except that it writes its output to the fourth serial port (“/dev/serial4”) at a data rate of 19,200 bits per second. By default, **dprintf()** is disabled.

set_dprintf_enabled() enables **dprintf()** if the *enabled* flag is **TRUE**, and disables it if the flag is **FALSE**. It returns the previous enabled state. Calls to this function can be nested by

caching the return value of a call that disables printing and passing it to the paired call that restores the previous state.

kernel_debugger() drops the calling thread into a debugger that writes its output to the fourth serial port at 19,200 bits per second, just as **dprintf()** does. This debugger produces *string* as its first message; it's not affected by **set_dprintf_enabled()**.

kernel_debugger() is identical to the **debugger()** function documented in the Kernel Kit, except that it works in the kernel and engages a different debugger. Drivers should use it instead of **debugger()**.

See also: **debugger()** in the Kernel Kit

enable_io_interrupt() *see* set_io_interrupt_handler()

enable_isa_interrupt() *see* set_isa_interrupt_handler()

get_memory_map()

<device/KernelExport.h>

long **get_memory_map(**void **address*, ulong *numBytes*,
physical_entry **table*, long *numEntries***)**

Locates the separate pieces of physical memory that correspond to the contiguous buffer of virtual memory beginning at *address* and extending for *numBytes*. Each piece of physical memory is described by a **physical_entry** structure. It has just two fields:

void ***address**	The address of a block of physical memory.
ulong **size**	The number of bytes in the block.

This function is passed a pointer to a *table* of **physical_entry** structures. It fills in the table, stopping when the entire buffer of virtual memory has been described or when *numEntry* entries in the table have been written, whichever comes first.

If the *table* provided isn't big enough, you'll need to call **get_memory_map()** again and ask it to describe the rest of the buffer. If the table is too big, this function sets the **size** field of the entry following the last one it needed to 0. This indicates that it has finished mapping the entire *address* buffer.

Memory should be locked while it is being mapped. Before calling **get_memory_map()**, call **lock_memory()** to make sure that it all stays in place:

```
physical_entry table[count];
lock_memory(someAddress, someNumberOfBytes, FALSE);
get_memory_map(someAddress, someNumberOfBytes, table, count);
. . .
unlock_memory(someAddress, someNumberOfBytes);
```

< This function consistently returns **B_NO_ERROR**. >

See also: **lock_memory()**, **start_isa_dma()**

get_nth_pci_info()

<device/PCI.h>

long **get_nth_pci_info(**long *index*, pci_info **info***) **

This function looks up the PCI device at *index* and provides a description of it in the **pci_info** structure that *info* refers to. Indices begin at 0 and there are no gaps in the list.

The **pci_info** structure contains a number of fields that report values found in the configuration register space for the device and it also describes how the device has been mapped into the system. The following fields are common to all devices:

ushort **vendor_id**	An identifier for the manufacturer of the device.
ushort **device_id**	An identifier for the particular device of the vendor, assigned by the vendor.
uchar **bus**	The bus number.
uchar **device**	The number that identifies the location of the device on the bus.
uchar **function**	The function number in the device.
uchar **revision**	A device-specific version number, assigned by the vendor.
uchar **class_api**	The type of specific register-level programming interface for the device (the lower byte of the class code field).
uchar **class_sub**	The specific type of function the device performs (the middle byte of the class code field).
uchar **class_base**	The broadly-defined device type (the upper byte of the class code field).
uchar **line_size**	The size of the system cache line, in units of 32-bit words.
uchar **latency**	The latency timer for the PCI bus master.
uchar **header_type**	The header type.
uchar **bist**	The contents of the register for the built-in self test.
uchar **u**	A union of structures, one for each header type.

Currently, there's only one header (type 0x00), but in the future there may be others. Consequently, header-specific information is recorded in a union of structures, one for each header type. The union (named simply **u**) at present has just one member, a structure for the current header (named **h0**):

```
typedef struct {
        . . .
        union {
                struct {
                        . . .
                } h0;
        } u;
} pci_info
```

The fields of the **h0** structure are:

ulong **cardbus_cis**	The CardBus CIS pointer.
ushort **subsystem_id**	The vendor-assigned identifier for the add-in card containing the device.
ushort **subsystem_vendor_id**	The identifier for manufacturer of the add-in card that contains the device.
ulong **rom_base**	The base address for the expansion ROM, as viewed from the host processor.
ulong **rom_base_pci**	The base address for the expansion ROM, as viewed from the PCI bus. This is the address a bus master would use.
ulong **rom_size**	The amount of memory in the expansion ROM, in bytes.
ulong **base_registers**[6]	The base addresses of requested memory spaces and I/O spaces, as viewed from the host processor.
ulong **base_registers_pci**[6]	The base addresses of requested memory spaces and I/O spaces, as viewed from the PCI bus. This is the address a bus master would use.
ulong **base_register_sizes**[6]	The sizes of requested memory spaces and I/O spaces.
uchar **base_register_flags**[6]	The flags from the base-address registers.

uchar **interrupt_line**	The interrupt line. This number identifies the interrupt associated with the device. See **set_io_interrupt_handler()**.
uchar **interrupt_pin**	The interrupt pin that the device uses.
uchar **min_grant**	The minimum burst period the device needs, assuming a clock rate of 33 MHz.
uchar **max_latency**	The maximum frequency at which the device needs access to the PCI bus.

In **device/PCI.h**, you'll find a number of constants that you can use to test various fields of a **pci_info** structure. See the *PCI Local Bus Specification*, published by the PCI Special Interest Group (Portland, OR) for more information on the configuration of a PCI device.

get_nth_pci_info() returns **B_NO_ERROR** if it successfully describes a PCI device, and **B_ERROR** if it can't find the device (for example, if *index* is out-of-range).

See also: **read_pci_config()**

io_card_version() *see* motherboard_version()

isa_address()

<device/KernelExport.h>

void ***isa_address(**long *offset***)**

Returns the virtual address corresponding to the specified *offset* in the ISA I/O address space. By passing an *offset* of 0, you can find the base address that's mapped to the ISA address space.

kernel_debugger() *see* dprintf()

lock_isa_dma_channel(), unlock_isa_dma_channel()

<device/KernelExport.h>

long **lock_isa_dma_channel(**long *channel***)**

long **unlock_isa_dma_channel(**long *channel***)**

These functions reserve an ISA DMA *channel* and release a channel previously reserved. They return **B_NO_ERROR** if successful, and **B_ERROR** if not. Like semaphores, these functions work only if all participating parties adhere to the protocol.

There are 7 ISA DMA channels. In general, they're used as follows:

Channel	Use
0	Unreserved, available
1	Unreserved, available
2	Reserved for the floppy disk controller
3	Reserved for the parallel port driver
5	Reserved for IDE
6	Reserved for sound
7	Reserved for sound

Channel 4 is taken by the system; it cannot be used.

lock_memory(), unlock_memory()

<device/KernelExport.h>

long **lock_memory(**void **address*, ulong *numBytes*, bool *willChange***)**

long **unlock_memory(**void **address*, ulong *numBytes***)**

lock_memory() makes sure that all the memory beginning at the specified virtual *address* and extending for *numBytes* is resident in RAM, and locks it so that it won't be paged out until **unlock_memory()** is called. It pages in any of the memory that isn't resident at the time it's called.

The *willChange* flag should be **TRUE** if any part of the memory range will be altered while it's locked—especially if the hardware device will do anything to modify the memory, since that won't otherwise be noticed by the system and the modified pages may not be written. The *willChange* flag should be **FALSE** if the memory won't change while it's locked.

Each of these functions returns **B_NO_ERROR** if successful and **B_ERROR** if not. The main reason that **lock_memory()** would fail is that you're attempting to lock more memory than can be paged in.

make_isa_dma_table() *see* start_isa_dma()

motherboard_version(), io_card_version()

<device/KernelExport.h>

long **motherboard_version(**void**)**

long **io_card_version(**void**)**

These functions return the current versions of the motherboard and of the I/O card.

ram_address()

<device/KernelExport.h>

void ***ram_address(**void **physicalAddress***)**

Returns the address of a physical block of system memory (RAM) as viewed from the PCI bus. If passed **NULL** as the *physicalAddress*, this function returns a pointer to the first byte of RAM; otherwise it returns a pointer to the *physicalAddress*.

This information is needed by bus masters—components, such as the ethernet and some SCSI controllers, that can perform DMA reads and writes (directly read from and write to system memory without CPU intervention).

Memory must be locked when calling this function. For example:

```
physical_entry table[count];
void *where;

lock_memory(someAddress, someNumberOfBytes, FALSE);
get_memory_map(someAddress, someNumberOfBytes, table, count);
where = ram_address(table[i].address)
. . .
unlock_memory(someAddress, someNumberOfBytes);
```

See also: **get_memory_map()**, **lock_memory()**

read_pci_config(), write_pci_config()

<device/PCI.h>

long **read_pci_config(**uchar *bus*, uchar *device*, uchar *function*,
long *offset*, long *size***)**

void **write_pci_config(**uchar *bus*, uchar *device*, uchar *function*,
long *offset*, long *size*, long *value***)**

These functions read from and write to the PCI configuration register space. The *bus*, *device*, and *function* arguments can be read from the **bus**, **device**, and **function** fields of the **pci_info** structure provided by **get_nth_pci_info()**. They identify the configuration space that belongs to the device.

The *offset* is an offset to the location in the 256-byte configuration space that is to be read or written and *size* is the number of bytes to be read from that location or written to it. Permitted sizes are 1, 2, and 4 bytes. **read_pci_config()** returns the bytes that are read, **write_pci_config()** writes *size* bytes of *value* to the *offset* location.

See also: **get_nth_pci_info()**

release_spinlock() *see* acquire_spinlock()

restore_interrupts() *see* disable_interrupts()

set_dprintf_enabled() *see* dprintf()

set_io_interrupt_handler(), disable_io_interrupt(), enable_io_interrupt()

<device/KernelExport.h>

long **set_io_interrupt_handler(**long *interrupt*,
interrupt_handler *function*, void **data***)*

long **disable_io_interrupt(**long *interrupt***)**

long **enable_io_interrupt(**long *interrupt***)**

These functions manage interrupts at the Be-designed I/O interrupt controller that combines ISA and PCI interrupts. The *interrupt* can be an ISA IRQ value or the **interrupt_line** field read from the **pci_info** structure provided by **get_nth_pci_info()**.

set_io_interrupt_handler() installs the handler *function* that will be called each time the specified *interrupt* occurs. This function should have the following syntax:

bool **handler(**void **data***)*

The *data* that's passed to **set_io_interrupt_handler()** will be passed to the handler function each time it's called. It can be anything that might be of use to the handler, or **NULL**. This function should always return **TRUE**.

set_io_interrupt_handler() itself returns **B_NO_ERROR** if successful in installing the handler, and **B_ERROR** if not.

disable_io_interrupt() disables the named *interrupt*, and **enable_io_interrupt()** reenables it. Both functions return **B_ERROR** for an invalid *interrupt* number, and **B_NO_ERROR** otherwise. Neither function takes into account the disabled or enabled state of the interrupt as it might be affected by other functions, such as **disable_isa_interrupt()** or **restore_interrupts()**. An interrupt that has been disabled by **disable_io_interrupt()** must be reenabled by **enable_io_interrupt()**.

See also: **get_nth_pci_info()**, **disable_interrupts()**

set_isa_interrupt_handler(), disable_isa_interrupt(), enable_isa_interrupt()

<device/KernelExport.h>

long **set_isa_interrupt_handler(**long *interrupt*,
interrupt_handler *function*, void **data***)*

long **disable_isa_interrupt(**long *interrupt***)**

long **enable_isa_interrupt(**long *interrupt***)**

These functions manage interrupts at the 8259 ISA-compatible interrupt controller. The *interrupt* is identified by its standard IRQ value.

set_isa_interrupt_handler() installs the handler *function* for the specified *interrupt*. This function should take one argument and return a **bool**:

bool **handler(**void **data***)*

The argument is the same *data* that's passed to **set_isa_interrupt_handler()**; it can be any kind of data the *function* might need, or **NULL**. The return value indicates whether the interrupt was handled—**TRUE** if it was and **FALSE** if not. By returning **FALSE**, the handler function can indicate that the device didn't generate the interrupt. The system can then try a different handler installed for a different device at the same interrupt number.
< However, this architecture is not currently supported, so the handler function should always return **TRUE**. >

set_isa_interrupt_handler() returns **B_NO_ERROR** if it can install the handler for the interrupt, and **B_ERROR** if not.

disable_isa_interrupt() disables the specified ISA *interrupt*, and **enable_isa_interrupt()** reenables it. Both functions return **B_ERROR** if the interrupt passed is not a valid IRQ value. Neither function considers whether the interrupt might be disabled or enabled by some other function, such as **disable_io_interrupt()**. An interrupt that has been disabled by **disable_isa_interrupt()** must be reenabled by **enable_isa_interrupt()**.

ISA interrupts can also be managed at the Be-designed interrupt dispatcher that controls PCI interrupts. The Be interrupt controller is somewhat faster than the edge-sensitive ISA controller. If your device can generate a level-sensitive interrupt, it should use the counterpart **set_io_interrupt_handler()** function instead of **set_isa_interrupt_handler()**. However, if it depends on the edge-sensitive ISA interrupt controller widely found in the PC world, it needs to use these ISA functions.

See also: **set_io_interrupt_handler()**, **disable_interrupts()**

spawn_kernel_thread()

<device/KernelExport.h>

thread_id **spawn_kernel_thread(**thread_entry *func*, const char **name*,
long *priority*, void **data***)**

This function is a counterpart to **spawn_thread()** in the Kernel Kit, which is not exported for drivers. It has the same syntax as the Kernel Kit function, but is able to spawn threads in the kernel itself.

See also: **spawn_thread()** in the Kernel Kit

spin()

<device/KernelExport.h>

void **spin(**double *microseconds***)**

Executes a delay loop lasting at least the specified number of *microseconds*. It could last longer, due to rounding errors, interrupts, and context switches.

start_isa_dma(), start_scattered_isa_dma(), make_isa_dma_table()

<device/KernelExport.h>

long **start_isa_dma(**long *channel*, void **address*, long *transferCount*,
uchar *mode*, uchar *eMode***)**

long **start_scattered_isa_dma(**long *channel*, isa_dma_entry **table*,
uchar *mode*, uchar *eMode***)**

long **make_isa_dma_table(**void **address*, long *numBytes*,
ulong *numTransferBits*,
isa_dma_entry **table*, long *numEntries***)**

These functions initiate ISA DMA memory transfers for the specified *channel*. They engage the ISA 8237 DMA controller.

start_isa_dma() starts the transfer of a contiguous block of physical memory beginning at the specified *address*. It requests *transferCount* number of transfers, which cannot be greater than **B_MAX_ISA_DMA_COUNT**. Each transfer will move 8 or 16 bits of memory, depending on the *mode* and *eMode* flags. These arguments correspond to the mode and extended mode flags recognized by the DMA controller.

The physical memory *address* that's passed to **start_isa_dma()** can be obtained by calling **get_memory_map()**.

start_scattered_isa_dma() starts the transfer of a memory buffer that's physically scattered in various pieces. The separate pieces of memory are described by the *table* passed as a second argument and provided by **make_isa_dma_table()**.

make_isa_dma_table() provides a description of the separate chunks of physical memory that make up the contiguous virtual buffer that begins at *address* and extends for *numBytes*. This function anticipates a subsequent call to **start_scattered_isa_dma()**, which initiates a DMA transfer. It ensures that the information it provides is in the format expected by the 8237 DMA controller. This depends in part on how many bits will be transferred at a time. The third argument, *numTransferBits*, provides this information. It can be **B_8_BIT_TRANSFER** or **B_16_BIT_TRANSFER**.

Each chunk of physical memory is described by a **isa_dma_entry** structure, which contains the following fields (not that its arcane details matter, since you don't have to do anything with the information except pass it to **start_scattered_isa_dma()**):

ulong **address**	A physical memory address (in little endian format).
ushort **transfer_count**	The number of transfers it will take to move all the physical memory at that address, minus 1 (in little endian format). This value won't be greater than **B_MAX_ISA_DMA_COUNT**.
int **flags**.**end_of_list**:1	A flag that's set to mark the last chunk of physical memory corresponding to the virtual buffer.

make_isa_dma_table() is passed a pointer to a *table* of **isa_dma_entry** structures. It fills in the table, stopping when the entire buffer of virtual memory has been described or when *numEntry* entries in the table have been written, whichever comes first. It returns the number of bytes from the virtual *address* buffer that it was able to account for in the *table*.

start_isa_dma() and **start_scattered_isa_dma()** both return **B_NO_ERROR** if successful in initiating the transfer, and **B_ERROR** if the channel isn't free.

unlock_isa_dma_channel() *see* lock_isa_dma_channel()

unlock_memory() *see* lock_memory()

write_pci_config() *see* read_pci_config()

xpt_init(), xpt_ccb_alloc(), xpt_ccb_free(), xpt_action(), xpt_bus_register(), xpt_bus_deregister()

<device/CAM.h>

long **xpt_init (**void**)**

CCB_HEADER ***xpt_ccb_alloc(**void**)**

void **xpt_ccb_free(**void **ccb**)**

long **xpt_action(**CCB_HEADER ***ccbHeader**)**

long **xpt_bus_register(**CAM_SIM_ENTRY ***entryPoints**)**

long **xpt_bus_deregister(**long *pathID*)**

These functions conform to the SCSI common access method (CAM) specification. See the draft ANSI standard *SCSI-2 Common Access Method Transport and SCSI Interface Modules* for information.

< The current implementation doesn't support asynchronous callback functions. All CAM requests are executed synchronously in their entirety. >

Constants and Defined Types for Kernel-Loadable Drivers

This section lists the constants and types that are defined for drivers the kernel loads. Everything listed here was explained in the previous sections on "Developing a Kernel-Loadable Driver" and "Functions for Drivers".

Constants

Control Operations

<device/Drivers.h>

Enumerated constant

B_GET_SIZE
B_SET_SIZE
B_SET_NONBLOCKING_IO
B_SET_BLOCKING_IO
B_GET_READ_STATUS
B_GET_WRITE_STATUS
B_GET_GEOMETRY
B_FORMAT

B_DEVICE_OP_CODES_END = 9999

These constants name the control operations that the kernel defines. You should expect the control hook function for any driver you develop to be called with these constants as the operation code (*op*).

All system-defined control constants are guaranteed to have values less than **B_DEVICE_OP_CODES_END**. Since additional constants might be defined for future releases, any that you define should be greater than **B_DEVICE_OP_CODES_END**.

See also: "Control Operations" on page 48

ISA DMA Transfer Maximum

<device/KernelExport.h>

Defined constant	Value
B_MAX_ISA_DMA_COUNT	0x10000

This constant indicates the maximum number of transfers for a single DMA request.

See also: **start_isa_dma()** on page 66

ISA DMA Transfer Sizes

<device/KernelExport.h>

Enumerated constant

B_8_BIT_TRANSFER
B_16_BIT_TRANSFER

These constants are passed to **make_isa_dma_table()** to indicate the size of a single DMA transfer.

See also: **start_isa_dma()** on page 66

Defined Types

cpu_status

<device/KernelExport.h>

typedef ulong **cpu_status**

This defined type is returned by **disable_interrupts()** to record whether interrupts were already disabled or not. It can be passed to **restore_interrupts()** to restore the previous state.

See also: **disable_interrupts()** on page 56

device_close_hook

<device/Drivers.h>

typedef long (***device_close_hook**)**(**device_info **info***)**

The hook function that the kernel calls to close a device must conform to this type.

See also: "Opening and Closing a Device" on page 46

device_control_hook

<device/Drivers.h>

typedef long (***device_control_hook*****)(**device_info **info*, ulong *op*, void **data***)**

The hook function for controlling a device must conform to this type.

See also: "Controlling the Device" on page 47

device_entry

<device/Drivers.h>

```
typedef struct {
        const char *name;
        device_open_hook open;
        device_close_hook close;
        device_control_hook control;
        device_io_hook read;
        device_io_hook write;
} device_entry
```

This structure declares the name of a device and the hook functions that the kernel can call to operate that device. The driver must provide one **device_entry** declaration for each of its devices.

See also: "Device Declarations" on page 43

device_geometry

<device/Drivers.h>

```
typedef struct {
        ulong bytes_per_sector;
        ulong sectors_per_track;
        ulong cylinder_count;
        ulong head_count;
        bool removable;
        bool read_only;
        bool write_once;
} device_geometry
```

Drivers use this structure to report the physical configuration of a mass-storage device.

See also: "**B_GET_GEOMETRY**" on page 49

device_info

<device/Drivers.h>

```
typedef struct {
        device_entry *entry;
        void *private_data;
} device_info
```

This structure contains publicly declared and private information about a device. It's passed as the first argument to each of the device-specific hook functions.

See also: "Hook Functions" on page 46

device_io_hook

<device/Drivers.h>

typedef long (***device_io_hook**)**(**device_info **info*, void **data*, ulong *numBytes*, ulong *position***)**

The hook functions that the kernel calls to read data from or write it to a device must conform to this type.

See also: "Reading and Writing Data" on page 47

device_open_hook

<device/Drivers.h>

typedef long (***device_open_hook**)**(**device_info **info*, ulong *flags***)**

The hook function that opens a device must conform to this type.

See also: "Opening and Closing a Device" on page 46

interrupt_handler

<device/KernelExport.h>

typedef bool (***interrupt_handler**)(void **data*)

The functions that are installed to handle interrupts must conform to this type.

See also: **set_io_interrupt_handler()** on page 64, **set_isa_interrupt_handler()** on page 65

isa_dma_entry

<device/KernelExport.h>

```
typedef struct {
        ulong address;
        ushort transfer_count;
        uchar reserved;
        struct {
                int end_of_list:1;
                int reserved:7;
        } flags;
} isa_dma_entry
```

This structure is filled in by **make_isa_dma_table()** and is passed unchanged to **start_scattered_isa_dma()**.

See also: **start_isa_dma()** on page 66

pci_info

```
<device/PCI.h>

typedef struct {
        ushort vendor_id;
        ushort device_id;
        uchar bus;
        uchar device;
        uchar function;
        uchar revision;
        uchar class_api;
        uchar class_sub;
        uchar class_base;
        uchar line_size;
        uchar latency;
        uchar header_type;
        uchar bist;
        uchar reserved;
        union {
                struct {
                        ulong cardbus_cis;
                        ushort subsystem_id;
                        ushort subsystem_vendor_id;
                        ulong rom_base;
                        ulong rom_base_pci;
                        ulong rom_size;
                        ulong base_registers[6];
                        ulong base_registers_pci[6];
                        ulong base_register_sizes[6];
                        uchar base_register_flags[6];
                        uchar interrupt_line;
                        uchar interrupt_pin;
                        uchar min_grant;
                        uchar max_latency;
                } h0;
        } u;
} pci_info
```

This structure reports values from the PCI configuration register space and describes how the device has been mapped into the system.

See also: **get_nth_pci_info()** on page 59

physical_entry

```
<device/KernelExport.h>

typedef struct {
        void *address;
        ulong size;
} physical_entry
```

This structure is used to describe a chunk of physical memory corresponding to some part of a contiguous virtual buffer.

See also: **get_memory_map()** on page 58

spinlock

```
<device/KernelExport.h>

typedef vlong spinlock
```

This data type serves the **acquire_spinlock()**/**release_spinlock()** protocol.

See also: **acquire_spinlock()** on page 55

Developing a Driver for a Graphics Card

Like other drivers, drivers for graphics cards are dynamically loaded modules—but they're loaded by the Application Server, the software component that's responsible for graphics operations, not by the kernel. Because they're add-on modules, these drivers share some similarities with other drivers:

- They lack a **main()** function, but must provide entry points where the host software (the Application Server in this case) can access driver functionality.
- They're mostly limited to calling functions that they implement themselves. They don't link against the system library or the host Application Server, < but they can link against a private library that gives them the ability to make some system calls >.
- They must be compiled as add-on modules and installed in a place well-known to the host.

Because the host module is the Application Server rather than the kernel, graphics card drivers must follow protocols that the Server defines, not those that the kernel imposes on other drivers. The entry points, exported functions, and installation directory are all specific to graphics card drivers. Therefore, if you're developing a driver for a graphics card, disregard the preceding sections of this chapter dealing with kernel drivers and follow the rules outlined in this section instead.

Control of a graphics card driver resides only with the host module; there are no functions (like **open()** or **ioctl()**) through which a program can control the driver.

However, applications can get direct access to the graphics card through the BWindowScreen class in the Game Kit. Access is provided by making a clone of the graphics card driver and attaching it to the application. The original driver remains running as part of the Application Server, but its connection to the screen is suspended while the clone is active.

Entry Point

Every graphics card driver must implement a function called **control_graphics_card()**. This is the Application Server's main entry point into the driver; it's the function the

Server calls to set up the driver, query it for information, pass it configuration instructions, and generally control what it does. It has the following syntax:

long **control_graphics_card(**ulong *op*, void **data***data***)**

The first argument, *op*, names the operation the driver is requested to perform. The second argument, *data*, points either to some information that will help the driver carry out the request or to a location where it should write some information as a result of the operation. The exact type of data in either case depends on the nature of the operation. The *op* and *data* arguments are inextricably linked.

The return value is an error code. In general, the control function should return **B_NO_ERROR** if it can successfully respond to a particular *op* request, and **B_ERROR** if it cannot. It should also respond **B_ERROR** to any undefined *op* code requests it doesn't understand.

Main Control Operations

There are seventeen control operations that a driver's **control_graphics_card()** function can be requested to perform (seventeen *op* codes defined in **device/GraphicsCard.h**). Nine of these operations give the Application Server general control over the driver. The other eight concern the cloning of the driver and the direct control of the frame buffer through the Game Kit. Those operations are discussed under "Control Operations for Cloning the Driver" and "Control Operations for Manipulating the Frame Buffer" below. This sections lists and discusses the nine main control operations.

B_OPEN_GRAPHICS_CARD

This *op* code requests the driver to open and initialize the graphics card specified by the *data* argument. If the driver can open the card, it should do so and return **B_NO_ERROR**. If it can't, it should return **B_ERROR**. The *data* pointer refers to a **graphics_card_spec** structure with the following fields:

void ***screen_base**	The beginning of memory on the graphics card. The driver can locate the frame buffer somewhere in this memory, but not necessarily at the base address.
void ***io_base**	The base address for the I/O registers that control the graphics card. Registers are addressed by 16-bit offsets from this base address.
ulong **vendor_id**	The number that identifies the manufacturer of the graphics chip on the card.
ulong **device_id**	A number that identifies the particular graphics chip of that manufacturer.

If the driver can open the graphics card, it should take the opportunity to initialize any data structures it might need. However, it should wait for further instructions—particularly a

B_CONFIG_GRAPHICS_CARD request—before initializing the frame buffer or turning on the video display.

If the driver returns **B_ERROR**, indicating that it's not the driver for the specified graphics card, it will immediately get a request to close the card as a prelude to being unloaded.

B_CLOSE_GRAPHICS_CARD

This operation notifies the graphics card driver that it's about to be unloaded. The *data* argument is meaningless (it doesn't point to any valid information). The Application Server ignores the return value and unloads the driver no matter what.

B_SET_INDEXED_COLOR

This operation is used to set up the palette of colors that can be displayed when the frame buffer is 8 bits deep. It requests the driver to place a particular color at a particular position in the list of 256 colors that's kept on the card. The *data* argument points to a **indexed_color** structure with two pieces of information:

long **index**	The index of the color in the list. This value is used as the color value in the **B_COLOR_8_BIT** color space. Indices begin at 0.
rgb_color **color**	The full 32-bit color that should be associated with the index.

A driver can expect a series of **B_SET_INDEXED_COLOR** requests soon after it is opened. It might get subsequent requests when an application (through the Game Kit) modifies the color list, and when the game returns control to the Application Server.

B_GET_GRAPHICS_CARD_HOOKS

This *op* code requests **control_graphics_card()** to supply the Application Server with an array of function pointers. Each pointer is to a hook function that the Server can call to carry out a specific graphics task. A total of **B_HOOK_COUNT** (48 at present) pointers must be written, although only a quarter of that number are currently used. The full array should be written to the location the *data* argument points to, with **NULL** values inserted for undefined functions.

A later section, "Hook Functions" on page 87, describes the hook functions, the tasks they should perform, their arguments and return types, and their positions in the array.

A driver can expect a **B_GET_GRAPHICS_CARD_HOOKS** request soon after it is opened, and again any time the screen configuration changes. The hook functions can be tailored to a specific screen dimension and depth.

B_GET_GRAPHICS_CARD_INFO

This *op* code requests the **control_graphics_card()** function to supply information about the driver and the current configuration of the screen. The *data* argument points to a **graphics_card_info** structure where it should write this information. This structure contains the following fields:

short **version**	The version of the Be architecture for graphics cards that the driver was designed to work with. The current version is 2.
short **id**	An identifier for the driver, understood in relation to the version number. The Application Server doesn't check this number; it can be set to any value you desire.
void ***frame_buffer**	A pointer to the first byte of the frame buffer.
char **rgba_order**[4]	The order of color components as the bytes for those components are stored in video memory (in the frame buffer). This array should arrange the characters 'r' (red), 'g' (green), 'b' (blue), and 'a' (alpha) in the order in which those components are intermeshed for each pixel in the frame buffer; a typical order is "bgra". This field is valid only for screen depths of 32 bits per pixel.
short **flags**	A mask containing flags that describe the ability of the graphics card driver to perform particular tasks.
short **bits_per_pixel**	The depth of the screen in bits per pixel. Only 32-bit (**B_RGB_32_BIT**) and 8-bit (**B_COLOR_8_BIT**) depths are currently supported.
long **bytes_per_row**	The offset, in bytes, between two adjacent rows of pixel data in the frame buffer (the number of bytes assigned to each row).
short **width**	The width of the frame buffer in pixels (the number of pixel columns it defines).
short **height**	The height of the frame buffer in pixels (the number of pixel rows it defines).

Three constants are currently defined for the **flags** mask:

B_CRT_CONTROL	Indicates that the driver is able to control, to any extent, the position or the size of the CRT display on the monitor—that there's a provision for controlling the CRT through software, not just hardware.
B_GAMMA_CONTROL	Indicates that the driver is able to make gamma corrections that compensate for the particular characteristics of the display device.

B_FRAME_BUFFER_CONTROL	Indicates that the driver allows clients to set arbitrary dimensions for the frame buffer and to control which portion of the frame buffer (the display area) is mapped to the screen.

The driver will receive frequent **B_GET_GRAPHICS_CARD_INFO** requests. The **graphics_card_info** structure it supplies should always reflect the values currently in force.

B_GET_REFRESH_RATES

This *op* code asks **control_graphics_card()** to place the current refresh rate, as well as the maximum and minimum rates, in the **refresh_rate_info** structure referred to by the *data* pointer. This structure contains the following fields:

float **min**	The minimum refresh rate that the graphics card is capable of, given the current configuration.
float **max**	The maximum refresh rate that the graphics card is capable of, given the current configuration.
float **current**	The current refresh rate.

All values should be provided in hertz.

B_GET_SCREEN_SPACES

This *op* code requests the driver to supply a mask containing all possible configurations of the screen space—all supported combinations of pixel depth and dimensions of the pixel grid. The mask is formed from the following constants—which are defined in **interface/InterfaceDefs.h**—and is written to the location indicated by the *data* pointer:

B_8_BIT_640x480	**B_16_BIT_640x480**	**B_32_BIT_640x480**
B_8_BIT_800x600	**B_16_BIT_800x600**	**B_32_BIT_800x600**
B_8_BIT_1024x768	**B_16_BIT_1024x768**	**B_32_BIT_1024x768**
B_8_BIT_1152x900	**B_16_BIT_1152x900**	**B_32_BIT_1152x900**
B_8_BIT_1280x1024	**B_16_BIT_1280x1024**	**B_32_BIT_1280x1024**
B_8_BIT_1600x1200	**B_16_BIT_1600x1200**	**B_32_BIT_1600x1200**

For example, if the mask includes **B_8_BIT_1600x1200**, the driver can configure a frame buffer that's simultaneously 8 bits deep (the **B_COLOR_8_BIT** color space), 1,600 pixel columns wide, and 1,200 pixel rows high. The mask should include all configurations that the graphics card is capable of supporting.

< The Application Server currently doesn't permit depths of 16 bits. >

(**InterfaceDefs.h** defines one other screen space, **B_8_BIT_640x400**, but this is reserved for the default "supervga" driver provided by Be.)

B_CONFIG_GRAPHICS_CARD

This *op* code asks the control function to configure the display according to the values set in the **graphics_card_config** structure that the *data* argument points to. This structure contains the following fields:

ulong **space**	The size of the pixel grid on-screen and the depth of the frame buffer in bits per pixel. This field will be one of the constants listed above for the **B_GET_SCREEN_SPACES** control operation.
float **refresh_rate**	The refresh rate of the screen in hertz.
uchar **h_position**	The horizontal position of the CRT display on the monitor.
uchar **v_position**	The vertical position of the CRT display on the monitor.
uchar **h_size**	The horizontal size of the CRT display on the monitor.
uchar **v_size**	The vertical size of the CRT display on the monitor.

The most important configuration parameter is the **space** field. The driver should reconfigure the screen to the depth and size requested and return **B_NO_ERROR**. If it can't carry out the request, it should return **B_ERROR**.

Failure to comply with the other fields of the **graphics_card_config** structure should not result in a **B_ERROR** return value. The driver should come as close as it can to the requested refresh rate. The last four fields are appropriate only for drivers that reported that they could control the positioning of the CRT display (by setting the **B_CRT_CONTROL** flag in response to a **B_GET_GRAPHICS_CARD_INFO** request).

The values for the four CRT configuration fields range from 0 through 100, with 50 as the default. Values of less than 50 for **h_position** and **v_position** should move the display toward the left and top; those greater than 50 should move it to the right and bottom. Values of less than 50 for **h_size** and **v_size** should make the display narrower and shorter, squeezing it into a smaller area; values greater than 50 should make it wider and taller.

B_SET_SCREEN_GAMMA

This operation asks the driver to set up a table for adjusting color values to correct for the peculiarities of the display device. The *data* argument points to a **screen_gamma** structure with gamma corrections for each color component. It contains the following three fields:

uchar **red**[256]	Mappings for the red component.
uchar **green**[256]	Mappings for the green component.
uchar **blue**[256]	Mappings for the blue component.

Each field is a component-specific array. The stated color value is used as an index into the array; the value found at that index substitutes for the stated value. For example, if the value at **blue**[152] is 154, all blue component values of 152 should be replaced by 154, essentially adding to the blueness of the color as displayed.

Only drivers that indicated they could make gamma corrections (by setting the **B_GAMMA_CONTROL** flag in response to a **B_GET_GRAPHICS_CARD_INFO** request) need to respond to **B_SET_SCREEN_GAMMA** requests.

< The control function is currently not requested to perform this operation. >

Control Operations for Cloning the Driver

Normally, an application's access to the screen is mediated by the Application Server. The application can draw in windows the Server provides through BView objects with graphics environments kept by the Server. The Application Server doesn't let applications communicate directly with the graphics card driver.

To give an application direct access to the screen, as the Game Kit does, the Application Server must get out of the way and the driver must be attached directly to the application. This is accomplished, not by detaching the driver from the Server, but by making a copy of it—a clone—for the application. While the clone is active, the Server suspends its graphic operations.

Graphics card drivers must therefore be prepared to clone themselves—to respond to the four control operations described below. Two of the requests are made of a driver the Application Server has loaded, and two are made of the clone.

B_GET_INFO_FOR_CLONE

This *op* code requests **control_graphics_card()** to write information about the current state of the driver to the location referred to by the *data* pointer. This request is made of a driver loaded by the Application Server; the information it provides is passed to the clone (in a **B_SET_CLONED_GRAPHICS_CARD** request) so that the clone can duplicate the state of the driver.

The driver should package the requested information in a data structure it defines; it can be any structure you desire. The package should include all the driver's variable settings—everything from the current configuration of the screen to the location of the frame buffer

in card memory. For example, if the structure is called **info_for_clone**, driver code might look something like this:

```
case B_GET_INFO_FOR_CLONE:
    ((info_for_clone *)data)->depth = info.bits_per_pixel;
    ((info_for_clone *)data)->height = info.height;
    ((info_for_clone *)data)->width = info.width;
    ((info_for_clone *)data)->row_byte = info.bytes_per_row;
    ((info_for_clone *)data)->frame_base = info.frame_buffer;
    ((info_for_clone *)data)->io_base = spec.io_base;
    ((info_for_clone *)data)->available_mem = unused_memory;
    ((info_for_clone *)data)->refresh_rate = rate.current;
    . . .
    break;
```

Of course, information that's kept on the card itself, such as the current color map, does not have to be duplicated for the clone.

Since an attempt is made to keep the driver and its clone in the same state, you can expect numerous **B_GET_INFO_FOR_CLONE** requests while the clone is active.

B_GET_INFO_FOR_CLONE_SIZE

This operation requests the driver to inform the Application Server how many bytes of information it will provide in response to a **B_GET_INFO_FOR_CLONE** request. The control function should write the size of the data structure as a **long** integer in the location that the *data* pointer refers to. For example:

```
*((long *)data) = sizeof(info_for_clone);
```

This information enables the Application Server to allocate enough memory to hold the data it will receive.

B_SET_CLONED_GRAPHICS_CARD

This operation sets up the clone. In the *data* pointer, it passes the clone's **control_graphics_card()** function all the information that the driver provided in response to a **B_GET_INFO_FOR_CLONE** request. The clone should read the information from the *data* pointer and set all the parameters that are provided.

The clone receives a **B_SET_CLONED_GRAPHICS_CARD** request instead of a **B_OPEN_GRAPHICS_CARD** notification when it first is created and loaded by the Game Kit. It subsequently will receive the request many more times—whenever it must be synchronized with the driver loaded by the Application Server.

B_CLOSE_CLONED_GRAPHICS_CARD

This *op* code is passed to the clone's **control_graphics_card()** function to signal that the clone is about to be unloaded. The clone receives this notification instead of **B_CLOSE_GRAPHICS_CARD**. The *data* pointer should be ignored.

Control Operations for Manipulating the Frame Buffer

The BWindowScreen class of the Game Kit defines a set of four functions that give applications more or less arbitrary control over the frame buffer:

ProposeFrameBuffer()
SetFrameBuffer()
SetDisplayArea()
MoveDisplayArea()

Each of these functions translates to an identically named operation that the driver's **control_graphics_card()** function can be requested to perform. Graphics card drivers announce their ability to respond to these requests by including a constant in the **flags** field of the **graphics_card_info** structure they report in response to a **B_GET_GRAPHICS_CARD_INFO** request. The constant is **B_FRAME_BUFFER_CONTROL**.

All four of the control operations use the same structure to pass data to the driver, though they don't all make use the same set of fields within the structure. The structure is called **frame_buffer_info** and it contains the following fields:

short **bits_per_pixel**	The depth of the frame buffer; the number of bits assigned to a pixel.
short **bytes_per_row**	The number of bytes that are used to store one row of pixel data in the frame buffer.
short **width**	The width of the frame buffer in pixels (the total number of pixel columns).
short **height**	The height of the frame buffer in pixels (the total number of pixel rows.
short **display_width**	The width of the screen display in pixels (the number of pixel columns displayed on-screen).
short **display_height**	The height of the screen display in pixels (the number of pixel rows displayed on-screen).
short **display_x**	The pixel column in the frame buffer that's mapped to the leftmost column of pixels on the screen, where columns are indicated by a left-to-right index beginning with 0.
short **display_y**	The pixel row in the frame buffer that's mapped to the topmost row of pixels on the screen, where rows are indicated by a top-to-bottom index beginning with 0.

The first four fields of this structure are identical to the last four of the **graphics_card_info** structure. However, **graphics_card_info** is used only to return information to the host, whereas **frame_buffer_info** can pass requests to the driver. It's possible for those four fields to be set to arbitrary values, so the frame buffer isn't limited to the standard configurations of depth, width, and height described under "**B_GET_SCREEN_SPACES**" above. (Of course, the driver can reject proposed configurations that it can't accommodate.)

The last four fields of the **frame_buffer_info** structure distinguish between the frame buffer itself and the part of the frame buffer that's displayed on-screen—the *display area*. This distinction permits the display area to be moved and resized on a (possibly) much larger area defined by the frame buffer. For buffered drawing, the frame buffer can be partitioned into discrete sections and the display area moved from one to another. For hardware scrolling, the display area can be moved repeatedly by small increments. For simulated zooming, it's size can be incrementally reduced or expanded.

Both areas are defined by a width (the number of pixel columns the area includes) and a height (the number of pixel rows). The display area is located in the frame buffer by the index to the column (**display_x**) and row (**display_y**) of its left top pixel. See the **SetDisplayArea()** function on page 14 in *The Game Kit* chapter for an illustration

The four operations that exercise control over the frame buffer are described below.

B_PROPOSE_FRAME_BUFFER

This *op* code proposes a particular width and depth for the frame buffer to the driver. The only valid fields of the **frame_buffer_info** structure passed through the *data* pointer are **bits_per_pixel** and **width**. If the driver can configure a frame buffer with those dimensions, it should fill in the rest of frame buffer description and return **B_NO_ERROR**. In the **bytes_per_row** field, it should write the minimum number of bytes required to store each row of pixel data given the proposed depth and width. In the **height** field, it should report the maximum number of pixel rows it can provide given the other dimensions. The fields of the **frame_buffer_info** structure that describe the display area can be ignored.

The driver should not actually configure the frame buffer in response to the proposal; it should wait for a **B_SET_FRAME_BUFFER** instruction. **B_PROPOSE_FRAME_BUFFER** merely tests the driver's capabilities.

If the driver can't accommodate a frame buffer with the proposed dimensions, it should place –1 in the **bytes_per_row** and **height** fields and return **B_ERROR**.

B_SET_FRAME_BUFFER

This operation requests the driver's **control_graphics_card()** function to configure the frame buffer according to the description in the **frame_buffer_info** structure passed through the *data* pointer. All fields in the structure contain meaningful values and should be read.

The specified configuration ought to have been previously tested through a **B_PROPOSE_FRAME_BUFFER** operation, and therefore should be one the driver can accommodate. If it's not, **control_graphics_card()** should do nothing and return **B_ERROR**. If it can configure the frame buffer according to the request, it should return **B_NO_ERROR**.

B_SET_DISPLAY_AREA

This *op* code requests the control function to set the display area, as specified by the last four fields of the **frame_buffer_info** structure passed through the *data* pointer. The other fields should be ignored.

If the driver can map the display area as requested, **control_graphics_card()** should return **B_NO_ERROR**. Otherwise, it should return **B_ERROR**.

B_MOVE_DISPLAY_AREA

This *op* code requests the control function to move the display area without resizing it, as specified by the **display_x** and **display_y** fields of the **frame_buffer_info** structure that the *data* pointer refers to. The other fields of the structure should be ignored.

The driver should move the display area so that the left top pixel displayed on-screen is the one located at (**display_x**, **display_y**) in the frame buffer and return **B_NO_ERROR**. If it can't move the display area to that location, it should return **B_ERROR**.

Hook Functions

A graphics card driver can implement hook functions to manage the cursor and perform particular, well-defined drawing tasks on behalf of the Application Server. Drivers should implement as many of these functions as they can to speed on-screen graphics performance.

The driver informs the Application Server about these functions soon after it's loaded when its **control_graphics_card()** function receives a **B_GET_GRAPHICS_CARD_HOOKS** request (see page 79 above). In response to this request, the driver needs to place an array of **B_HOOK_COUNT** (48) function pointers at the location the *data* argument points to. The request is repeated whenever the configuration of the frame buffer (its dimensions and depth) changes. The driver can provide hook functions specific to a particular configuration.

Currently, only the first 12 slots in the array are defined. These functions fall into four groups:

- Indices 0–2: The first three functions define and manage the cursor. Drivers must implement all three of these functions, or none of them. The Application Server defers to driver-defined cursors because of the significant performance improvements they offer.

- Indices 3–9: The next seven hook functions take on specific drawing tasks, such as stroking a minimum-width line or filling a rectangle. You can choose which of these functions to implement.
- Index 10: The function at this index is used to synchronize the Application Server with the driver. Drivers should implement it only if the Server might sometimes need to wait for the driver to finish the drawing undertaken by any of the other hook functions.
- Index 11: The final function inverts the colors in a rectangle.

Each undefined slot in the array of hook functions should be filled with a **NULL** pointer. Similarly, the driver should place a **NULL** value in any defined slot if it can't usefully implement the function.

Although all pointers in the array are declared to be of type **graphics_card_hook**,

typedef void (***graphics_card_hook**)**(**void**)**

each function has its own set of arguments and returns a meaningful error value, declared as a **long**. The functions should be implemented to return **B_NO_ERROR** if all goes well and they're successful in performing the task at hand, and **B_ERROR** if unsuccessful. It's better by far to place a **NULL** pointer in the array than to define a function that always returns **B_ERROR**.

The coordinate system that the Application Server assumes for all hook functions equates one coordinate unit to one screen pixel. The origin is at the pixel in the left top corner of the screen. In other words, an *x* coordinate value is a left-to-right index to a pixel column and a *y* coordinate value is a top-to-bottom index to a pixel row.

The following sections discuss each of the hook functions in turn.

Index 0: Defining the Cursor

The function at index 0 is called to set the cursor image. It has the following syntax:

long **define_cursor(**uchar **xorMask*, uchar **andMask*, long *width*, long *height*,
long *hotX*, long *hotY***)**

The first two arguments, *xorMask* and *andMask*, together define the shape of the cursor. Each mask has a depth of 1 bit per pixel, yielding a total of four possible values for each cursor pixel. They should be interpreted as follows:

xorMask	andMask	meaning
0	0	Transparency; let the color of the screen pixel under the cursor pixel show through.
1	0	Inversion; invert the color of the screen pixel.

0	1	White; replace the screen pixel with a white cursor pixel.
1	1	Black; replace the screen pixel with a black cursor pixel.

Inversion in its simplest form is accomplished by taking the complement of the color index or of each color component. For example:

```
color = 255 - color;
```

< However, the results of inversion may not be very pleasing given the current color map. Therefore, none of the Be-defined cursors will use inversion until a future release. It would be better for your drivers to avoid it as well. The color map will be corrected in a future release. >

The second two arguments, *width* and *height*, determine the size of the cursor image in pixels. Currently, the Application Server supports only one cursor size; they must be 16 pixels wide and 16 pixels high.

The (*hotX*, *hotY*) arguments define the hot pixel in the image—the pixel that's used to report the location of the cursor. They assume a coordinate system where the pixel at the left top corner of the image is (0, 0) and the one at the right bottom corner is (15, 15).

This function should change the cursor image on-screen, if the cursor is currently displayed on-screen. But if the cursor is hidden, it should not show it. Wait for explicit calls to the next two functions to move the cursor or change its on-screen status.

Index 1: Moving the Cursor

The function at index 1 changes the location of the cursor image. It should expect two arguments:

long **move_cursor(**long *screenX*, long *screenY***)**

In response, this function should move the cursor so that its hot pixel corresponds to (*screenX*, *screenY*).

Index 2: Showing and Hiding the Cursor

The function at index 2 shows and hides the cursor:

long **show_cursor(**bool *flag***)**

If the *flag* argument is **TRUE**, this function should show the cursor image on-screen; if it's **FALSE**, it should remove the cursor from the screen.

< If this function is asked to show the cursor before the function at index 1 is called, it should show it at (0, 0). >

Index 3: Drawing a Line with an 8-Bit Color

The function at index 3 draws a straight line in the **B_COLOR_8_BIT** color space. It takes 10 arguments:

long **draw_line_with_8_bit_depth(**long *startX*, long *startY*, long *endX*, long *endY*,
uchar *colorIndex*, bool *clipToRect*, short *clipLeft*,
short *clipTop*, short *clipRight*, short *clipBottom***)**

The first four arguments define the starting and ending points of the line; it begins at (*startX*, *startY*) and ends at (*endX*, *endY*). Both points are included within the line. The fifth argument, *colorIndex*, is the color of the line; it's an index into the map of 256 colors.

< In the current release, the second and third arguments are inverted; the first four arguments are ordered: *startX*, *endX*, *startY*, *endY*. >

If the sixth argument, *clipToRect*, is **TRUE**, the function should draw only the portion of the line that lies within the clipping rectangle defined by the last four arguments. The sides of the rectangle are included within the drawing area—they're inside the visible region; everything outside the rectangle is clipped.

If *clipToRect* is **FALSE**, the final four arguments should be ignored.

This function should draw a line of minimal thickness, which means a line no thicker than one pixel at any given point. If the line is more vertical than horizontal, only one pixel per row between the start and end points should be colored; if it's more horizontal than vertical, only one pixel per column should be colored.

Index 4: Drawing a Line with a 32-Bit Color

The function at index 4 is like the one at index 3, except that it draws a line in the **B_RGB_32_BIT** color space:

long **draw_line_with_32_bit_depth(**long *startX*, long *startY*, long *endX*, long *endY*,
ulong *color*, bool *clipToRect*, short *clipLeft*,
short *clipTop*, short *clipRight*, short *clipBottom***)**

The only difference between this and the previous function is the *color* argument. Here the color is specified as a full 32-bit quantity with 8-bit red, green, blue, and alpha components. The *color* argument arranges the components in the order that the driver asked for them (in the **rgba_order** field of the **graphics_card_info** structure that it provided in response to a **B_GET_GRAPHICS_CARD_INFO** request).

Otherwise, this function should work just like the one at index 3. < And like the function at index 3, the second and third arguments are inverted in the current release; the first four arguments are ordered: *startX*, *endX*, *startY*, *endY*. >

Index 5: Drawing a Rectangle with an 8-Bit Color

The function at index 5 should be implemented to fill a rectangle with a color specified by its index:

long **draw_rect_with_8_bit_depth(**long *left*, long *top*, long *right*, long *bottom*,
uchar *colorIndex***)**

The *left*, *top*, *right*, and *bottom* sides of the rectangle should be included in the area being filled.

Index 6: Drawing a Rectangle with a 32-Bit Color

The function at index 6, like the one at index 5, fills a rectangle:

long **draw_rect_with_32_bit_depth(**long *left*, long *top*, long *right*, long *bottom*,
ulong *color***)**

The *color* value contains the four color components—red, green, blue, and alpha—arranged in the natural order for the device (the same order that the driver recorded in the **rgba_order** field of the **graphics_card_info** structure it provided to the Application Server).

The sides of the rectangle should be included in the area being filled.

Index 7: Copying Pixel Data

The function at index 7 should copy pixel values from a source rectangle on-screen to a destination rectangle:

long **blit(**long *sourceX*, long *sourceY*, long *destinationX*, long *destinationY*,
long *width*, long *height***)**

The left top corner of the source rectangle is the pixel at (*sourceX*, *sourceY*). The left top pixel of the destination rectangle is at (*destinationX*, *destinationY*). Both rectangles are *width* pixels wide and *height* pixels high, and both are guaranteed to always lie entirely on-screen. The *width* and *height* arguments will always contain positive values.

Index 8: Drawing a Line Array with an 8-Bit Color

The function at index 8 should draw an array of lines in the **B_COLOR_8_BIT** color space. It takes the following set of arguments:

long **draw_array_with_8_bit_depth(**indexed_color_line **array*, long *numItems*,
bool *clipToRect*, short *clipLeft*, short *clipTop*,
short *clipRight*, short *clipBottom***)**

The line *array* holds a total of *numItems*. Each item is specified as an **indexed_color_line** structure, which contains the following fields:

short **x1**	The *x* coordinate of one end of the line.
short **y1**	The *y* coordinate of one end of the line.
short **x2**	The *x* coordinate of the other end of the line.
short **y2**	The *y* coordinate of the other end of the line.
uchar **color**	The color of the line, expressed as an index into the color map.

The function should draw each line from (**x1**, **y1**) to (**x2**, **y2**) using the **color** specified for that line.

If the *clipToRect* flag is **TRUE**, nothing should be drawn that falls outside the clipping rectangle defined by the final four arguments. The sides of the rectangle are included in the visible region. If *clipToRect* is **FALSE**, the final four arguments should be ignored.

Each line in the array should be drawn with the minimal possible thickness, as described under “Index 3: Drawing a Line with an 8-Bit Color” on page 90 above.

Index 9: Drawing a Line Array with a 32-Bit Color

The function at index 9 has the same syntax as the one at index 8, except for the first argument:

long **draw_array_with_32_bit_depth(**rgb_color_line **array*, long *numItems*,
bool *clipToRect*, short *clipLeft*, short *clipTop*,
short *clipRight*, short *clipBottom***)**

Here, each line in the array is specified as an **rgb_color_line** structure, rather than as an **indexed_color_line**. The two structures differ only in how the color is specified:

short **x1**	The *x* coordinate of one end of the line.
short **y1**	The *y* coordinate of one end of the line.
short **x2**	The *x* coordinate of the other end of the line.
short **y2**	The *y* coordinate of the other end of the line.
rgb_color **color**	The color of the line, expressed as a full 32-bit value.

In all other respects, this function should work like the one at index 8.

Index 10: Synchronizing Drawing Operations

The Application Server calls the function at index 10 to synchronize its activities with the driver. It takes no arguments:

long **sync(**void**)**

This function simply returns when the driver is finished modifying the frame buffer—when it's finished touching video RAM.

If any of the other hook functions works asynchronously—if it returns before the drawing it's asked to do is complete—the synchronizing function should wait until all drawing operations have been completed before it returns. The return value is not important; the Application Server ignores it.

However, if all the other hook functions are synchronous—if they don't return until the drawing is finished—this function would simply return; it would be empty. It's better not to implement such a function. It's preferable to put a **NULL** pointer at index 10 and save the Application Server a function call.

Therefore, your driver should implement this function only if at least one of the other hook functions draws asynchronously.

Index 11: Inverting Colors

The function at index 11 should invert the colors in a rectangle. It has the following syntax:

long **invert_rect(**long *left*, long *top*, long *right*, long *bottom***)**

Inversion is typically defined as taking the complement of each color component. For example:

```
color.red = 255 - color.red;
color.green = 255 - color.green;
color.blue = 255 - color.blue;
```

The inversion rectangle includes the pixel columns and rows that four arguments designate.

Exported Functions

Graphics card drivers are not linked against the Application Server, so the Server cannot export functions to them. They are also not linked against any library. Consequently, they're generally limited to calling functions that they implement themselves.

However, in the current release, a graphics card driver can be statically linked against **scalls.o**, located with the libraries in **/develop/libraries**. < A future release will replace this file with a private library. >

Linking against **scalls.o** makes it possible for the driver to call any system function. The future library won't be as liberal, however, so system calls should be limited to the following functions:

Functions defined in the Kernel Kit

create_sem()
acquire_sem()
release_sem()
delete_sem()

system_time()
snooze()

spawn_thread()
resume_thread()

Functions defined in the Support Kit

atomic_add()

Other functions

dprintf() < accessed through the name **_kdprintf_()** >
set_dprintf_enabled() < accessed through the name **_kset_dprintf_enabled_()** >

Functions from the first two groups are documented in the respective chapters on the Kernel Kit and the Support Kit. Functions in the last group are documented in the section on "Functions for Drivers" in this chapter on page 55.

Installation

The graphics card driver should be compiled as an "add-on image," as described in *The Kernel Kit* chapter. This means following the directions for compiling a shared library presented in the Metrowerks *CodeWarrior* manual. In summary, you'll need to specify the following options for the linker (as **LDFLAGS** in the **makefile**):

- Tell the linker to produce an add-on image by including the **–G** (or **–sharedlibrary**) option.
- Turn off the default behavior—which is to link against the Be system library, **libbe.so**—by specifying the **–nodefaults** flag.
- Export the driver's entry point, **control_graphics_card()**, so that the Application Server can call it. The most direct way to do this is to surround its definition with explicit directives that turn exporting on and off,

```
#pragma export on
long control_graphics_card(ulong op, void *data)
{
    . . .
}
#pragma export off
```

  and inform the linker with an **–export pragma** flag.
- Specify the **scalls.o** file if you want your driver to call any of the system functions listed under "Exported Functions" on page 94 above. Don't link the driver against any other file.

After the driver has been compiled, it should be installed in:

/system/add-ons/app_server

This is the only place where the Application Server will look for graphics card drivers to load.

The Server first looks for a driver for the graphics card in an **app_server** directory on a floppy disk (**/fd/system/add-ons/app_server**). Failing to find one, it looks next on the boot disk (**/boot/system/add-ons/app_server**).

When it searches each disk for a driver, the Application Server begins by looking for one developed specifically for the installed graphics card. If there are more than one, it's indeterminate which one it will choose to load. If there aren't any, the Server looks for the generic driver called **supervga**. This driver should be able to do a minimal job of putting a display on-screen, but probably won't be able to exploit the full potential of the graphics card.

You can give your driver any name you wish. However, the name "supervga" is reserved for the generic driver provided by Be.

Constants and Defined Types for Graphics Card Drivers

This section lists the various constants and types that are defined for graphics card drivers. Explanations for all of them can be found in the preceding section, “Developing a Driver for a Graphics Card”.

Constants

Control Operations

<device/GraphicsCard.h>

Enumerated constant	Enumerated constant
B_OPEN_GRAPHICS_CARD	**B_GET_INFO_FOR_CLONE**
B_CLOSE_GRAPHICS_CARD	**B_GET_INFO_FOR_CLONE_SIZE**
B_GET_GRAPHICS_CARD_INFO	**B_SET_CLONED_GRAPHICS_CARD**
B_GET_GRAPHICS_CARD_HOOKS	**B_CLOSE_CLONED_GRAPHICS_CARD**
B_SET_INDEXED_COLOR	
B_GET_SCREEN_SPACES	**B_PROPOSE_FRAME_BUFFER**
B_CONFIG_GRAPHICS_CARD	**B_SET_FRAME_BUFFER**
B_GET_REFRESH_RATES	**B_SET_DISPLAY_AREA**
B_SET_SCREEN_GAMMA	**B_MOVE_DISPLAY_AREA**

These constants define the various control operations that the Application Server or the Game Kit can request a driver to perform.

See also: “Main Control Operations” on page 78

Hook Count

<device/GraphicsCard.h>

Defined constant	Value
B_HOOK_COUNT	48

This constant is the number of hook function pointers that a driver must provide. Most will be **NULL** pointers.

See also: “Hook Functions” on page 87

Info Flags

<device/GraphicsCard.h>

Defined constant

B_CRT_CONTROL
B_GAMMA_CONTROL
B_FRAME_BUFFER_CONTROL

These flags report the driver's ability to control the CRT display, make gamma corrections, and permit nonstandard configurations of the frame buffer.

See also: "**B_GET_GRAPHICS_CARD_INFO**" on page 80

Defined Types

frame_buffer_info

<device/GraphicsCard.h>

```
typedef struct {
        short bits_per_pixel;
        short bytes_per_row;
        short width;
        short height;
        short display_width;
        short display_height;
        short display_x;
        short display_y;
} frame_buffer_info
```

This structure is used to pass information to the driver on how the frame buffer should be configured.

See also: the BWindowScreen class in the Game Kit, "Control Operations for Manipulating the Frame Buffer" on page 85 above

graphics_card_config

<device/GraphicsCard.h>

```
typedef struct {
        ulong space;
        float refresh_rate;
        uchar h_position;
        uchar v_position;
        uchar h_size;
        uchar v_size;
} graphics_card_config
```

This structure is used to pass the driver a set of parameters describing how the graphics card should be configured.

See also: "**B_CONFIG_GRAPHICS_CARD**" on page 82

graphics_card_hook

<device/GraphicsCard.h>

```
typedef void (*graphics_card_hook)(void)
```

This is the general type declaration for a hook function. Specific hook functions will in fact declare various sets of arguments and all return a **long** error code rather than **void**.

See also: "Hook Functions" on page 87

graphics_card_info

<device/GraphicsCard.h>

```
typedef struct {
        short version;
        short id;
        void *frame_buffer;
        char rgba_order[4];
        short flags;
        short bits_per_pixel;
        short bytes_per_row;
        short width;
        short height;
} graphics_card_info
```

Drivers use this structure to supply information about themselves and the current configuration of the frame buffer to the Application Server and to the BWindowScreen class in the Game Kit.

See also: "**B_GET_GRAPHICS_CARD_INFO**" on page 80

graphics_card_spec

```
<device/GraphicsCard.h>

typedef struct {
        void *screen_base;
        void *io_base;
        ulong vendor_id;
        ulong device_id;
        ulong _reserved1_;
        ulong _reserved2_;
} graphics_card_spec
```

This structure informs the driver about the graphics card and how it's mapped into the system.

See also: "**B_OPEN_GRAPHICS_CARD**" on page 78

indexed_color

```
<device/GraphicsCard.h>

typedef struct {
        long index;
        rgb_color color;
} indexed_color
```

This structure is used to set up the list of 256 colors in the **B_COLOR_8_BIT** color space. It locates a particular color at a particular index in the list.

See also: "**B_SET_INDEXED_COLOR**" on page 79

indexed_color_line

```
<device/GraphicsCard.h>

typedef struct {
        short x1;
        short y1;
        short x2;
        short y2;
        uchar color;
} indexed_color_line
```

This structure defines a colored line in the **B_COLOR_8_BIT** color space.

See also: "Index 8: Drawing a Line Array with an 8-Bit Color" on page 92

refresh_rate_info

<device/GraphicsCard.h>

```
typedef struct {
        float min;
        float max;
        float current;
} refresh_rate_info
```

Drivers use this structure to report the current refresh rate, and the maximum and minimum possible rates.

See also: "**B_GET_REFRESH_RATES**" on page 81

rgb_color_line

<device/GraphicsCard.h>

```
typedef struct {
        short x1;
        short y1;
        short x2;
        short y2;
        rgb_color color;
} rgb_color_line
```

This structure defines a colored line in the **B_RGB_32_BIT** color space.

See also: "Index 9: Drawing a Line Array with a 32-Bit Color" on page 92

screen_gamma

<device/GraphicsCard.h>

```
typedef struct {
        uchar red[256];
        uchar green[256];
        uchar blue[256];
} screen_gamma
```

This structure defines the table used to make gamma corrections for the screen display.

See also: "**B_SET_SCREEN_GAMMA**" on page 82