CSR BlueCore 01 USB Kernel Device Driver Interface

Overview

This document documents the USB Device Driver csrbc01.sys interface. This is a kernel device driver called csrbc01.sys. An application can access a BlueCore 01 device on a USB device, by opening the device driver, sending commands, receiving events, and controlling ACL and SCO streams.

Basic Interface

Opening the Device

The device can be opened using normal file system function calls by opening the device "\\.\CSR0". This can be done with a CreateFile WIN32 system call. The result is just a file handle. If the device does not exist, because you haven't plugged it into the USB bus, then the call will fail. If more than one CSR BC01 device is on the USB bus, then they will be called CSR0, CSR1, CSR2, etc…

Example:

```
hci_handle = CreateFile
(
    "\\\\.\\CSR0",
    GENERIC_READ | GENERIC_WRITE,
    0,
    0,
    OPEN_EXISTING,
    0,
    0
);

if (hci_handle == INVALID_HANDLE_VALUE)
{
    printf ("Cannot open %s\n", filename);
}
```

Sending an HCI Command

To send HCI Command to the CSR BC01 USB device, you first need to open the device, then issue a DeviceIoControl IOCTL_CSRBC01_SEND_HCI_COMMAND. This takes a buffer and the length of data to send to the device. This call will return with the actual number of bytes written.

Example:

```
status = DeviceIoControl
(
    hci_handle,
    IOCTL_CSRBC01_SEND_HCI_COMMAND,
    buffer,
    length,
    0,
    0,
    &written,
    0
);
```

Reading an HCI Event

To read HCI Event, you again call DeviceIoControl. This takes a buffer, and a length, although the call will fail if the length is not 16. When the driver has received an event, the buffer is filled with the values of the event, and the number of valid bytes is required. This call is non-blocking, so if there is no event waiting to be sent up to the user application, then it will return 0 in the written parameter.

Example:

```
status = DeviceIoControl
(
    hci_handle,
    IOCTL_CSRBC01_GET_HCI_EVENT,
    0,
    0,
    buffer,
    length,
    &written,
    0
);
```

Closing the Device

When you have finished using the device, you will have to close the file handle that you originally opened. You can achieve this by calling the system call CloseHandle.

Example:

```
CloseHandle (hci_handle);
```

Writing ACL Data

Eventually you will need to send ACL Data. You can do this by calling WriteFile on the handle obtained from a CreateFile. The written parameter will return the number of bytes actually sent.

Example:

```
WriteFile
(
    hci_handle,
    buffer,
    length,
    &written,
    0
);
```

Reading ACL Data

To read ACL data, you do a ReadFile on the handle obtained from a CreateFile call. This takes a buffer of length bytes long, and returns the number of bytes read in written. This is a non-blocking call, so if no bytes are currently available, then it will return immediately.

Example:

```
ReadFile
(
    hci_handle,
    buffer,
    length,
    &written,
    0
);
```

Reading Device Descriptor

Once you open a device, you can get its device descriptor. This is a 16 byte block of data, containing the following fields.

```
UCHAR type;
UCHAR length;
USHORT bcdUSB;
UCHAR bDeviceClass;
UCHAR bDeviceSubClass;
UCHAR bDeviceProtocol;
UCHAR bMaxPacketSize0;
USHORT idVendor;
USHORT idProduct;
USHORT bcdDevice;
UCHAR iManufacturer;
UCHAR iProduct;
UCHAR iSerialNumber;
UCHAR bNumConfigurations;
```

Example:

```
status = DeviceIoControl
(
    hci_handle,
    IOCTL_CSRBC01_GET_DEVICE_DESCRIPTOR,
    0,
    0,
    buffer,
    18,
    &written,
    0
);
```

Resetting the Device

If you have a connection, you can reset the device at any point, by sending a IOCTL_CSRBC01_RESET_DEVICE control code.

Example:

```
status = DeviceIoControl
(
    hci_handle,
    IOCTL_CSRBC01_RESET_DEVICE,
    0,
    0,
    0,
    0,
    &written,
    0
);
```

Sending SCO Data

This is unsupported.

Receiving SCO Data

This is unsupported.

Sending a Control Transfer

To send a Control Transfer Command to the CSR BC01 USB device, you first need to open the device, and then issue a DeviceIoControl IOCTL_CSRBC01_SEND_CONTROL_TRANSFER. This takes a buffer and the length of data to send to the device. This call will return with the actual number of bytes written.

The buffer has a defined data format which mirrors the USB request itself.

Buffer[0] = bmRequestType
Buffer[1] = bRequest
Buffer[2,3] = value
Buffer[4,5] = index

The actual data to be sent will start at buffer[6]. The length field passed into DeviceIoControl should be the actual data + 6 bytes (for the defined format).

Any data that is received will be placed into in_buffer, which has in_length number of bytes.

Example:

```
status = DeviceIoControl
(
    hci_handle,
    IOCTL_CSRBC01_SEND_CONTROL_TRANSFER,
    buffer,
    length,
    in_buffer,
    in_length,
    &written,
    0
);
```

High Speed Operation

To allow the high speed operation, and minimal CPU usage, it is advantageous to have a blocking call to get_hci_event, that if no event is available, then it blocks until an event arrives. The driver allows this by having two more DeviceIoControl IOCTL… To use these calls, you should however open the driver in OVERLAPPED mode. This is so that if you block on an hci_event, you can still send and receive ACL data.

Opening an Overlapped Device

To open a device in OVERLAPPED mode, you first call CreateFile with the FILE_FLAG_OVERLAPPED flag. This works on Win98 and Win2000.

Example

```
hci_handle = CreateFile
(
    “\\\\.\\CSR0”,
    GENERIC_READ | GENERIC_WRITE,
    0,
    0,
    OPEN_EXISTING,
    FILE_FLAG_OVERLAPPED,
    0
);
```

Blocking Events

When you think that an event is not going to be available for a long time, you call the DeviceIoControl IOCTL_CSRBC01_BLOCK_HCI_EVENT. This will not complete until an event is ready, which you can then read normally as above in the “Reading an HCI Event” section.

In the example, we have an overlapped structure, which we set the hEvent value equal to previously created EVENT. This can be done with CreateEvent (0, 0, 0, 0). After the DeviceIoControl exits, the request is still pending, so the easiest way is to WaitForSingleObject on the event you had in the overlapped structure. This will block this thread until the event is signalled. After an HCI event arrives, it signals the event, continues processing the thread, and finally GetOverlappedResult will find the status of the command.

Example

```
OVERLAPPED overlapped = { 0, 0, 0, 0 };

    overlapped.hEvent = wait_for_event;

DeviceIoControl
(
    hci_handle,
    IOCTL_CSRBC01_BLOCK_HCI_EVENT,
    0,
    0,
    0,
    0,
    &written,
    &overlapped
);

WaitForSingleObject (wait_for_event, -1);

status = GetOverlappedResult (handle, &overlappped, &written, FALSE);
```

Blocking Data

Blocking data is very similar to blocking events, except we use IOCTL_CSRBC01_BLOCK_HCI_DATA.

Example

```
OVERLAPPED overlapped = { 0, 0, 0, 0 };

    overlapped.hEvent = wait_for_data;

DeviceIoControl
(
    hci_handle,
    IOCTL_CSRBC01_BLOCK_HCI_DATA,
    0,
    0,
    0,
    0,
    &written,
    &overlapped
);

WaitForSingleObject (wait_for_data, -1);

status = GetOverlappedResult (handle, &overlappped, &written, FALSE);
```

Other Concerns about Overlapped Devices

The major problem with overlapped devices, is that on Windows 98, you must supply an OVERLAPPED structure to ever call into the driver that can take one. This includes sending HCI Commands and HCI Data. You should also have a separate EVENT for each HCI Command or HCI Data you send in parallel.

Example

```
OVERLAPPED overlapped = { 0, 0, 0, 0 };

    overlapped.hEvent = wait_for_command;

DeviceIoControl
(
    hci_handle,
    IOCTL_CSRBC01_SEND_HCI_COMMAND,
    buffer,
    length,
    0,
    0,
    &written,
    &overlapped
);

WaitForSingleObject (wait_for_command, -1);

status = GetOverlappedResult (handle, &overlappped, &written, FALSE);
```

Full Example

Following is the implementation of a full class which implements a simple interface to a USB device.

Class Definition

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////
//// Copyright Cambridge Silicon Radio 2000
////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

class USBStack
{

private:

    HANDLE  handle;
    HANDLE  wait_for_event;
    HANDLE  wait_for_data;
    HANDLE  wait_for_acl;
    HANDLE  wait_for_command;

    int     event_paused;
    int     data_paused;

    char    *event_buffer;
    char    *data_buffer;

    HANDLE  monitor_events_thread_handle;
    HANDLE  monitor_data_thread_handle;

    void    (*event_func) (void *data, unsigned long length, void *context);
    void    (*data_func) (void *data, unsigned long length, void *context);

    void    *event_func_context;
    void    *data_func_context;

    int USBStack::get_hci_event (void *buffer, int length);
    static DWORD WINAPI USBStack::monitor_events_thread (LPVOID arg);

    int USBStack::get_hci_data (void *buffer, int length);
    static DWORD WINAPI USBStack::monitor_data_thread (LPVOID arg);

    void monitor_events (void);
    void monitor_data (void);

public:

    USBStack ();
    ~USBStack ();

    bool open_connection (char *device_name);
    void close_connection (void);

    bool is_open (void);

    int send_hci_command (void *data, unsigned long length);
    int send_acl_data (void *data, unsigned long length);

    void set_event_callback (void (*func) (void *, unsigned long, void *), void *context);
    void set_data_callback (void (*func) (void *, unsigned long, void *), void *context);

};
```

Example Class

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////
//// Copyright Cambridge Silicon Radio 2000
////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

#include <windows.h>
#include <winioctl.h>
#include <stdio.h>

#include "csrbc01_usb.h"

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

USBStack::USBStack ()
{
    handle = INVALID_HANDLE_VALUE;
    wait_for_event = CreateEvent (0, 0, 0, 0);
    wait_for_data = CreateEvent (0, 0, 0, 0);
    wait_for_acl = CreateEvent (0, 0, 0, 0);
    wait_for_command = CreateEvent (0, 0, 0, 0);

    event_paused = 0;
    data_paused = 0;

    event_buffer = new char[256 + 4];
    data_buffer = new char[65536 + 8];

    event_func = 0;
    data_func = 0;

    event_func_context = 0;
    data_func_context = 0;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

USBStack::~USBStack ()
{
    delete event_buffer;
    delete data_buffer;

    CloseHandle (wait_for_event);
    CloseHandle (wait_for_data);

    CloseHandle (wait_for_acl);
    CloseHandle (wait_for_command);
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

bool USBStack::open_connection (char *device_name)
{
    handle = CreateFile
    (
        device_name,
        GENERIC_READ | GENERIC_WRITE,
        0,
        0,
        OPEN_EXISTING,
        FILE_FLAG_OVERLAPPED,
        0
```

```
    );

    if (handle == INVALID_HANDLE_VALUE)
    {
        return false;
    }

    monitor_events ();
    monitor_data ();

    return true;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

bool USBStack::is_open (void)
{
    return (handle != INVALID_HANDLE_VALUE);
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

void USBStack::close_connection (void)
{
    TerminateThread (monitor_events_thread_handle, 0);
    TerminateThread (monitor_data_thread_handle, 0);

    if (handle != INVALID_HANDLE_VALUE)
    {
        CloseHandle (handle);
    }

    handle = INVALID_HANDLE_VALUE;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

int USBStack::send_hci_command (void *buffer, unsigned long length)
{
    int
        status;

    unsigned long
        written;

    static OVERLAPPED
        overlapped = { 0, 0, 0, 0, 0 };

    if (handle != INVALID_HANDLE_VALUE)
    {
        overlapped.hEvent = wait_for_command;

        status = DeviceIoControl
        (
            handle,
            IOCTL_CSRBC01_SEND_HCI_COMMAND,
            buffer,
            length,
            0,
            0,
            &written,
            &overlapped
        );

        if ((!status) && (GetLastError () == ERROR_IO_PENDING))
        {
            WaitForSingleObject (wait_for_command, -1);

            written = 0;
```

```
        }
        else if (!status)
        {
            close_connection ();

            return 0;
        }

        status = GetOverlappedResult (handle, &overlapped, &written, FALSE);

        if (status == 0)
        {
            written = 0;
        }
    }
    else
    {
        written = 0;
    }

    return written;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

int USBStack::send_acl_data (void *data, unsigned long length)
{
    unsigned int
        status;

    unsigned long
        written;

    static OVERLAPPED
        overlapped = { 0, 0, 0, 0, 0 };

    if (handle != INVALID_HANDLE_VALUE)
    {
        overlapped.hEvent = wait_for_acl;

        status = WriteFile (handle, data, length, &written, &overlapped);

        if ((!status) && (GetLastError () == ERROR_IO_PENDING))
        {
            WaitForSingleObject (wait_for_acl, -1);

            written = 0;
        }
        else if (!status)
        {
            close_connection ();

            return 0;
        }

        status = GetOverlappedResult (handle, &overlapped, &written, FALSE);

        if (status == 0)
        {
            written = 0;
        }
    }
    else
    {
        written = 0;
    }

    return written;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

```
int USBStack::get_hci_event (void *buffer, int length)
{
    int
        status;

    unsigned long
        written;

    static OVERLAPPED
        overlapped = { 0, 0, 0, 0, 0 };

    if (handle != INVALID_HANDLE_VALUE)
    {
        if (event_paused > 64)
        {
            event_paused = 64;

            overlapped.hEvent = wait_for_event;

            status = DeviceIoControl
            (
                handle,
                IOCTL_CSRBC01_BLOCK_HCI_EVENT,
                0,
                0,
                0,
                0,
                &written,
                &overlapped
            );

            if ((!status) && (GetLastError () == ERROR_IO_PENDING))
            {
                WaitForSingleObject (wait_for_event, -1);

                written = 0;
            }
            else if ((!status) && (GetLastError () == ERROR_INVALID_PARAMETER))
            {
            }
            else if (!status)
            {
                close_connection ();

                return 0;
            }
        }

        overlapped.hEvent = wait_for_event;

        status = DeviceIoControl
        (
            handle,
            IOCTL_CSRBC01_GET_HCI_EVENT,
            0,
            0,
            buffer,
            length,
            &written,
            &overlapped
        );

        if ((!status) && (GetLastError () == ERROR_IO_PENDING))
        {
            WaitForSingleObject (wait_for_event, -1);

            status = GetOverlappedResult (handle, &overlapped, &written, FALSE);
        }
        else if (!status)
        {
            close_connection ();

            return 0;
        }
```

```
        if (status == 0)
        {
            written = 0;
        }
    }
    else
    {
        written = 0;
    }

    if (written == 0)
    {
        event_paused ++;

        if (event_paused > 50)
        {
            Sleep (event_paused - 50);
        }
    }
    else
    {
        event_paused = 0;
    }

    return written;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

DWORD WINAPI USBStack::monitor_events_thread (LPVOID arg)
{
    USBStack
        *stack;

    int
        offset,
        length,
        size;

    stack = (USBStack *) arg;

    while (1)
    {
        offset = 0;

        size = stack->get_hci_event (stack->event_buffer, 16);

        while (size)
        {
            length = stack->event_buffer[1] & 0xff;
            offset = 0;

            if (length + 2 != size)
            {
                offset += size;

                while (offset < length + 2)
                {
                    size = stack->get_hci_event (&stack->event_buffer[offset], 16);

                    offset += size;
                }

            }

            if (stack->event_func)
            {
                stack->event_func (stack->event_buffer, length + 2, stack->event_func_context);
            }

            size = stack->get_hci_event (stack->event_buffer, 16);
        }
```

```
    }


    return 0;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

void USBStack::monitor_events (void)
{
    DWORD
        thread_id;

    monitor_events_thread_handle =
        CreateThread (0, 0, monitor_events_thread, this, 0, &thread_id);
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

int USBStack::get_hci_data (void *buffer, int length)
{
    int
        status;

    unsigned long
        written;

    static OVERLAPPED
        overlapped = { 0, 0, 0, 0, 0 };

    if (handle != INVALID_HANDLE_VALUE)
    {
        if (data_paused > 64)
        {
            data_paused = 64;

            overlapped.hEvent = wait_for_data;

            status = DeviceIoControl
            (
                handle,
                IOCTL_CSRBC01_BLOCK_HCI_DATA,
                0,
                0,
                0,
                0,
                &written,
                &overlapped
            );

            if ((!status) && (GetLastError () == ERROR_IO_PENDING))
            {
                WaitForSingleObject (wait_for_data, -1);
            }
            else if ((!status) && (GetLastError () == ERROR_INVALID_PARAMETER))
            {
            }
            else if (!status)
            {
                close_connection ();
            }
        }

        overlapped.hEvent = wait_for_data;

        status = ReadFile (handle, buffer, length, &written, &overlapped);

        if ((!status) && (GetLastError () == ERROR_IO_PENDING))
        {
            WaitForSingleObject (wait_for_data, -1);
```

```
            status = GetOverlappedResult (handle, &overlapped, &written, FALSE);
        }
        else if (!status)
        {
            close_connection ();
        }

        if (status == 0)
        {
            written = 0;
        }
    }
    else
    {
        written = 0;
    }

    if (written == 0)
    {
        data_paused ++;

        if (data_paused > 50)
        {
            Sleep (data_paused - 50);
        }
    }
    else
    {
        data_paused = 0;
    }

    return written;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

DWORD WINAPI USBStack::monitor_data_thread (LPVOID arg)
{
    USBStack
        *stack;

    char
        *data_buffer;

    int
        offset,
        length,
        size;

    data_buffer = new char[65536 + 8];

    stack = (USBStack *) arg;

    while (1)
    {
        offset = 0;

        size = stack->get_hci_data (data_buffer, 4096);

        while (size)
        {
            length = (*((short*)&data_buffer[2])) & 0xffff;

            offset = 0;

            if (length + 4 != size)
            {
                offset += size;

                while (offset < length + 4)
                {
                    size = stack->get_hci_data (&data_buffer[offset], 64);
```

```
                offset += size;
            }

        }

        if (stack->data_func)
        {
            stack->data_func (data_buffer, length + 4, stack->data_func_context);
        }

        size = stack->get_hci_data (data_buffer, 64);
    }
}

    return 0;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

void USBStack::monitor_data (void)
{
    DWORD
        thread_id;

    monitor_data_thread_handle =
        CreateThread (0, 0, USBStack::monitor_data_thread, this, 0, &thread_id);
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

void USBStack::set_event_callback (void (*func)(void *, unsigned long, void *), void *context)
{
    event_func = func;
    event_func_context = context;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

void USBStack::set_data_callback (void (*func)(void *, unsigned long, void *), void *context)
{
    data_func = func;
    data_func_context = context;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```