AT91 USB HID Driver Implementation

1. Introduction

The **Human Interface Devices** (HID) class extends the USB specification in order to provide a standard way of handling devices manipulated by humans. This includes common computer devices such as keyboards, mice and joysticks, as well as electronic device controllers (e.g., VCR remote) and generic controls (e.g., knobs, switches).

The HID class also encompasses devices which do not require human interaction but provide HID-compatible data, like a thermometer. This flexibility makes it possible to enable **generic communications** between an HID device and a host system, in a very simple way.

HID also allows several functionalities to be multiplexed on the same endpoint. This makes it possible to have a device perform several tasks (e.g. keyboard + mouse + communication) while still only using a single endpoint. This means a device using HID can be very versatile.

This application note describes how to implement an HID driver with the **AT91 USB Device Framework** provided by Atmel® for use with its AT91 ARM® Thumb® based microcontrollers. First, generic information about HID-specific definitions and requirements is given. This document then details how to use the HID class to create a **mouse**, a **keyboard** and a **generic communication** device.

2. Related Documents

[1] Device Class Definitions for Human Interface Devices (HID), Version 1.11, June 27, 2001.

[2] HID Usage Tables, Version 1.12, January 21, 2005.

[2] Atmel Corp., AT91 USB Device Framework, 2006.

AT91 ARM Thumb Microcontrollers

Application Note

6273B–ATARM–29-Jun-09

3. Human Interface Device Class Basics

This section gives generic details on the HID class, including its purpose, architecture and how it is supported by various operating systems.

3.1 Purpose

The HID class has been specifically designed for **Human Interface Devices**, i.e., devices which are manipulated by humans to control a computer or an electronic device. This includes common peripherals such as a keyboard, a mouse or a joystick, as well as many other interfaces: remote controllers, switches, buttons, dedicated game controls, and so on.

It is also possible to use the HID class for devices which do not require human interaction, but still deliver information in a similar format. For example, devices like a thermometer or a battery indicator are supported.

In addition, the HID class also makes it possible to not only receive data from devices but also to send commands to them. Indeed, many devices offer some kind of display to give back information to the user, e.g., the LEDs on a keyboard.

Finally, since it is quite simple to send and receive data using the HID class, it can be used as a generic means of communication between a device and a host. This is made possible because of the very flexible framework defined in the HID specification.

In this document, three uses of the HID class will be detailed step-by-step, each showing one particular feature of the class. The first example shows the interaction with a simple mouse. In the second example, a keyboard is implemented to demonstrate the possibility to send data to a peripheral. The last example explains how to use HID as a simple two-way communication channel.

3.2 Architecture

3.2.1 Interfaces

An HID device only needs **one interface descriptor**. It should have the HID interface class code in its *bInterfaceClass* field. There are special subclass and protocol codes to specify if the HID device is a mouse or a keyboard, and must be supported by the BIOS. In such a case, the interface must be declared as a Boot Interface, and the type of the device (mouse or keyboard) must be given in the *bInterfaceProtocol* field.

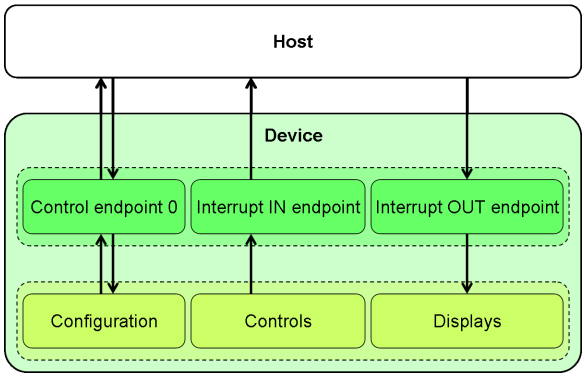
3.2.2 Endpoints

Up to three endpoints can be used with an HID interface. The first two are the default Control endpoint 0, as well as an Interrupt IN endpoint. They are mandatory and shall always be declared. An optional Interrupt OUT endpoint can be added as well.

Endpoint 0 is used for class-specific requests, as well as receiving data from the host if no Interrupt OUT endpoint has been defined. In addition, the host can also explicitly request or send report data through this endpoint.

The Interrupt IN and OUT pipes are used for sending asynchronous data to the host, and to receive low-latency information.

Figure 3-1. HID Class Driver Architecture



3.2.3 Class-Specific Descriptors

There are three class-specific descriptors defined in the *HID specification 1.11*: the HID descriptor, the report descriptor and the physical descriptor.

3.2.3.1 HID Descriptor

The **HID descriptor** gives information about the HID specification revision used, the country for which a device is localized, and lists the number of class-specific descriptors, including their length and type. The format is described in Table 3-1.

Table 3-1. HID Descriptor Format

Field	Size (bytes)	Description
bLength	1	Total length of the HID descriptor
bDescriptorType	1	HID descriptor type (21h)
bcdHID	2	HID specification release number in Binary Coded Decimal (BCD) format.
bCountryCode	1	Code of the country for which the device is localized. Should be 0 if the device is not localized.
bNumDescriptors	1	Number of class-specific descriptors used by the device.
bDescriptorType	1	Type of the first class-specific descriptor.
bDescriptorLength	1	Total length of the first class-specific descriptor.
[bDescriptorType]	1	Type of the second class-specific descriptor.
[bDescriptorLength]	1	Total length of the second class-specific descriptor.
...		

There is always at least one Report descriptor for an HID device, so the corresponding fields must be present in the HID descriptor. If other descriptors are defined, they must also be described here.

3.2.3.2 Report Descriptor

A HID device must have at least one **Report descriptor**. It defines the type of data manipulated by the device, which is referred to as *report*. When the device wants to notify that the cursor has moved, for example, it sends the corresponding report in the format previously defined in the Report descriptor.

This descriptor is quite different from others, as it does not have a fixed table of values. Instead, it is made up of a variable number of items, which collectively identify the information that a host can expect from or send to the device.

There are five categories of items:

- **Input** items, which define the format and type of the data sent by the device.
- **Output** items, which define the format and type of the data expected by the device
- **Feature** items, which define data sent to or received from the device, and not intended for the end user, such as configuration parameters.
- **Collection** items, which identify a set of data as related to the same group.
- **End Collection** items, which close other Collection items.

Usually, a Report descriptor defines only one use (report) for a device, e.g., a mouse. However, it is possible to declare several reports to perform different tasks, e.g., mouse & keyboard. This is done by assigning a different **Report ID** to each report; this makes it possible for the device to send both reports through the same Interrupt endpoint, while still permitting the host to correctly identify the data. Using only a single endpoint for several functionalities is very powerful, as the remaining ones can then be used by other interfaces (CDC, MSD, etc.) for an even more versatile device.

More details about Report descriptors will be given in the implementation examples. For more information about the possible items, tags and values, please refer to the *HID specification 1.11*.

3.2.3.3 Physical Descriptor

A **Physical descriptor** can be used to give information about which human body part is used to activate a particular control. While this is a useless piece of information for most devices, it can be relevant for complex devices which provide many similar controls. In such a case, a Physical descriptor allows an application to assign its functionalities more appropriately; for example, a game controller often has a large number of buttons, with some of them more accessible than the others. Those buttons would be assigned the most useful actions.

Since physical descriptors are not used very often, and are not useful in the case studies described in this document, they will not be discussed further.

3.2.4 Class-specific Requests

3.2.4.1 GetDescriptor

While **GET_DESCRIPTOR** is a standard request (defined in the *USB specification 2.0*), new descriptor type values have been added for the HID class. They make it possible for the host to request the HID descriptor, Report descriptor and Physical descriptors used by the device.

When requesting a HID-specific descriptor, the *wIndex* field of the request must be set to the HID interface number. For standard requests, this field is either set to 0 or, for String descriptors, to the index of the language ID used.

3.2.4.2 SetDescriptor

Similarly, **SET_DESCRIPTOR** is a standard request with added HID-specific values. It is used by the host to change the HID descriptors of a device. This is an optional request, and has few practical uses.

3.2.4.3 GetReport

The host can explicitly ask the device for a report by using the **GET_REPORT** request. However, it should be used primarily to get the state of feature items and absolute values at initialization time, not for consistent device polling.

The requested report is identified either by its Report ID (if they are used), and/or by its type (Input, Output or Feature).

Please note that a GET_REPORT request is different from a GET_DESCRIPTOR request for the Report descriptor. The latter returns the whole Report descriptor, i.e., all the items declared. The former returns the data defined by this descriptor.

3.2.4.4 SetReport

SET_REPORT is similar to GET_REPORT, except this request is used for changing the state of a report, instead of simply retrieving it.

For an Input report, this request can either be considered meaningless, or can be used to reset the current status of a control. For example, it could be used to calibrate the neutral position of a joystick.

3.2.4.5 SetIdle

This request is used to specify the minimum amount of time, called **Idle rate**, that a device must wait before transmitting a report if its state has not changed. This means the device must NAK all polls on its Interrupt IN endpoint until the report state changes, or the guarding period expires.

The SET_IDLE command can either be issued for a particular duration, or for an undefined period of time. The upper byte of the *wValue* field is used to specify this duration. In addition, if the device generates multiple reports, the Report ID of the target report to affect can be specified in the lower byte.

In practice, this request is often used with a keyboard to put a small delay before a key is repeated continuously. For a mouse, it must be set to 0, meaning that the device should never report an unchanged state.

3.2.4.6 GetIdle

The GET_IDLE request is issued by the host to retrieve the current Idle rate of the device. A particular Report can be specified with its Report ID.

3.2.4.7 GetProtocol

This request returns the protocol currently used by the device. This can either be the Report protocol (*wValue* set to 0) or the Boot protocol (*wValue* set to 1). Since a device supporting the Boot protocol can operate differently depending on which mode it is in, the host system can retrieve or change this mode with the GET_PROTOCOL and SET_PROTOCOL requests.

This request is only need for devices supporting the Boot protocol.

3.2.4.8 SetProtocol

Whenever an HID device starts up, it should use the Report protocol by default. However, the host driver shall still use the SET_PROTOCOL request to specify if the device should use the Report protocol or the Boot protocol.

This request is only need for devices supporting the Boot protocol.

3.3 Host Drivers

Most operating systems provide a generic HID driver which automatically handles standard devices, such as keyboard, mice and joystick. In addition, the driver can also be used by the application to easily access custom and vendor-specific devices.

4. HID Mouse

This section describes how to implement a mouse device using the HID class and the AT91 USB Device Framework. For more information about the framework, please refer to the *AT91 USB Device Framework* application note; details about the USB and the HID class can be found in the *USB specification 2.0* and the *HID specification 1.11* documents, respectively.

4.1 Purpose

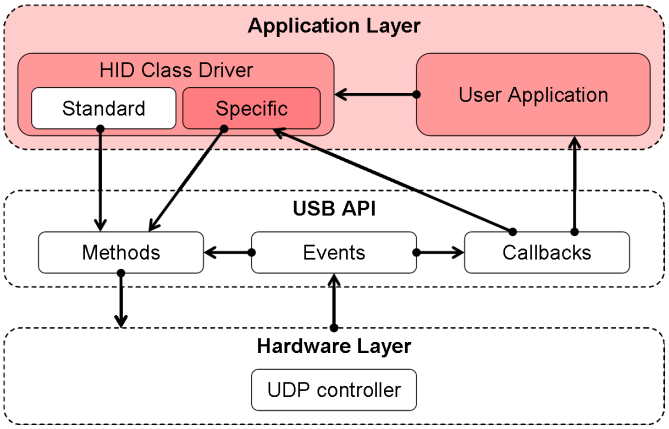
Along with the keyboard, the mouse is the principal way of interaction with a computer system. Using HID for a mouse is both a low-cost and highly portable way for such a device. In addition, the class driver design is the simplest possible one. Conversely to a PS/2 mouse, an HID mouse can also be plugged or unplugged dynamically and be immediately operational.

Other devices can also identify themselves as an HID mouse. For example, sensors and triggers targeted at people with disabilities can be affected the same functionalities as a mouse, e.g., left click, right click or cursor movement. This is again a very cost-effective, portable and simple way of implementing such a device.

4.2 Architecture

The AT91 USB Device Framework offered by Atmel makes it easy to create USB class drivers. The example software described in the current chapter is based on this framework. Figure 4-1 shows the application architecture.

Figure 4-1. Application Architecture Using the AT91 USB Device Framework



4.3 Descriptors

An HID device specifies its functionalities by using both standard descriptors (such as the device, configuration and interface descriptors), and several HID-specific descriptors such as the HID descriptor and the Report descriptor. All these descriptors and their values will now be detailed.

4.3.1 Device Descriptor

The **Device descriptor** of an HID device is very basic, since the HID class code is only specified at the Interface level. Thus, it only contains standard values, as shown below:

```
// HID Device Descriptor
const USBDeviceDescriptor sDevice = {

    sizeof(USBDeviceDescriptor), // Size of this descriptor
    USBGenericDescriptor_DEVICE, // DEVICE descriptor type
    USBDeviceDescriptor_USB2_00, // USB 2.0 specification
    HIDDeviceDescriptor_CLASS, // Class is in the interface descriptor.
    HIDDeviceDescriptor_SUBCLASS, // No device subclass code
    HIDDeviceDescriptor_PROTOCOL, // No device protocol code
    BOARD_USB_ENDPOINTS_MAXPACKETSIZE(0), // Maximum packet size for EP0
    HIDMouseDriverDescriptors_VENDORID, // ATMEL Vendor ID
    HIDMouseDriverDescriptors_PRODUCTID, // Product ID = 0x0001
    HIDMouseDriverDescriptors_RELEASE, // Device release number 0.01
    0x01, // Index of manufacturer description
    0x02, // Index of product description
    0x03, // Index of serial number description
    0x01 // One possible configuration
};
```

Note that the Vendor ID is a special value attributed by the USB-IF organization. The product ID can be chosen freely by the vendor.

4.3.2 Configuration Descriptor

Since one interface is required by the HID specification, this must be specified in the **Configuration descriptor**. There is no other value of interest to put here.

```
// HID Configuration Descriptor
{
    sizeof(USBConfigurationDescriptor), // Size of this descriptor
    USBGenericDescriptor_CONFIGURATION, // CONFIGURATION descriptor type
    sizeof(HIDDMouseDriverConfigurationDescriptors), // Total length of data
    0x01, // One interface is used by this configuration
    0x01, // Value 0x01 is used to select this configuration
    0x00, / No string is used to describe this configuration
    BOARD_USB_BMATTRIBUTES, // Device is self-powered, no remote wakeup
    USBConfigurationDescriptor_POWER(100) // maximum power consumption
}
```

When the Configuration descriptor is requested by the host (by using the GET_DESCRIPTOR command), the device must also sent all the related descriptors, i.e. Interface, Endpoint and Class-Specific descriptors. It is convenient to create a single structure to hold all this data, for sending everything in one chunk. In the example software, a `HIDDMouseDriverConfiguration-Descriptors` structure has been declared for that.

4.3.3 HID Class Interface Descriptor

The only **Interface descriptor** is for the HID Class Interface, in this example. It should specify the HID Class code (03h). In addition, if the mouse device supports the Boot protocol (which means it can be recognized directly by the BIOS), this should be notified by setting the *bInterfaceSubclass* and b*InterfaceProtocol* to 01h and 02h (respectively). There are only a few additional requirements for a Boot-enabled mouse, so they will be implemented in the present case study.

Furthermore, a mouse device only needs to send report data to the host. This means only the Interrupt IN endpoint is needed, so the *bNumEndpoints* field will have to be set to 1. This interface also uses the default Control endpoint, but this is not taken into account here.

Here is the whole Interface descriptor:

```
// HID Class Interface Descriptor
{
    sizeof(USBInterfaceDescriptor), // Size of this descriptor
    USBGenericDescriptor_INTERFACE, // INTERFACE Descriptor Type
    0x00, // Interface 0
    0x00, // No alternate settings
    0x01, // One endpoint used
    HIDInterfaceDescriptor_CLASS, // HID class code
    HIDInterfaceDescriptor_SUBCLASS_NONE, // Boot protocol is not supported
    HIDInterfaceDescriptor_PROTOCOL_MOUSE, // This is a mouse device
    0x00 // No associated string descriptor
}
```

4.3.4 HID Descriptor

An Interface descriptor can be followed by a set of class-specific descriptors. For the HID class, there are at least two of those: the HID descriptor and a Report descriptor. The first one to be transmitted must always be the HID descriptor, since it gives information about the number and types of the other defined descriptors (see Section 4.3.4 on page 8).

The software example for the mouse is only going to feature one Report and no Physical descriptor. The latter is useless for a mouse device, since it only exhibits a few buttons. Also, since the only requirement for a mouse is to send the displacement of the cursor and the state of the buttons, a single Report is sufficient. Note that at this point, the total length of the Report descriptor must already be known.

Finally, it is also useless for a mouse to be localized, since it does not deliver country-dependant information. This means the *bCountryCode* field can be safely set to 00h.

```
// HID-Specific Descriptor
{
    sizeof(HIDDescriptor), // Size of this descriptor in bytes (9)
    HIDGenericDescriptor_HID, // HID descriptor type
    HIDDescriptor_HID1_11, // HID Class Specification 1.11
    0x00, // No country code
    0x01, // 1 HID class descriptor
    HIDGenericDescriptor_REPORT, // First HID-specific descriptor type
    sizeof(hiddReportDescriptor) // Total length of first descriptor
```

```
}
```

4.3.5 Report Descriptor

The **Report descriptor** is used to specify the data sent or received by the device (see Section 3.2.3.2 on page 3). For a mouse device, there are two different types of information to transmit, and none to receive.

A mouse is used both for moving the cursor and clicking on buttons. Thus, the two pieces of data that an HID mouse must send is the mouse movement and the buttons state. Usually, the displacement is specified in the **relative** format, which means that the data sent by the mouse reflects its movement regarding its previous position. In the absolute format, it would be necessary to indicate the exact position of the cursor on the whole screen.

The format of a Report descriptor is a bit more complex than standard descriptors. Indeed, there is no fixed format for a Report descriptor, it all depends on the data described. However, there are items and tags which are almost always present. Basically, the descriptor is a byte array; each byte is either an item, or its associated data. Items are defined in the *HID specification 1.11*, and each one of them has a particular code. In addition, the item value must reflect the number of bytes associated with the item, which is one most of the time.

A Report descriptor starts with a **Global Usage Page** item:

```
const unsigned char hiddReportDescriptor[] = {
   HIDReport_GLOBAL_USAGEPAGE + 1, HIDGenericDesktop_PAGEID,
```

This item is used to specify the global functionality of the device. In this example, the **Generic Desktop** usage page is defined. This page encompasses mouse, keyboard, joystick and tablet devices. For more information about the available usage pages, please refer to the *HID Usage Table 1.12*.

A **Local Usage** item is then used to give more details about the device functionality, in the context of the previously defined usage page. Here, the usage value provided is the mouse one:

```
HIDReport_LOCAL_USAGE + 1, HIDGenericDesktop_MOUSE,
```

According to the *HID Usage Tables* document, the mouse usage is intended to be included in a collection of type **Application**. Collection are used to group data together; Application collections denote data which some applications may be familiar with.

```
HIDReport_COLLECTION + 1, HIDReport_COLLECTION_APPLICATION,
```

Since the data delivered by a mouse device is collected at a single geometric point, it must now be included in a **Physical collection**. Each collection must have an usage associated with it, so this collection will be defined as a **Pointer**:

```
HIDReport_LOCAL_USAGE + 1, HIDGenericDesktop_POINTER,
HIDReport_COLLECTION + 1, HIDReport_COLLECTION_PHYSICAL,
```

Now, the actual data sent by the device will be described. The first byte of the report will contain the state of the mouse buttons, in the form of a bitmap value. In this example, three buttons are defined, so the first three bits will either be 0 (button not pressed) or 1 (button pressed). The following items are used to define this behavior:

```
HIDReport_GLOBAL_USAGEPAGE + 1, HIDButton_PAGEID,
HIDReport_GLOBAL_REPORTCOUNT + 1, 3,
HIDReport_GLOBAL_REPORTSIZE + 1, 1,
HIDReport_LOCAL_USAGEMINIMUM + 1, 1,
HIDReport_LOCAL_USAGEMAXIMUM + 1, 3,
```

```
HIDReport_GLOBAL_LOGICALMINIMUM + 1, 0,
HIDReport_GLOBAL_LOGICALMAXIMUM + 1, 1,
HIDReport_INPUT + 1, HIDReport_VARIABLE,
```

First, the usage page is changed to *Buttons*. The *Local Usage Minimum* and *Maximum* values specify the range of buttons ID defined; in this case, button 1 to button 3 are used. The different values returned for each button is expressed using *Global Logical Minimum* and *Maximum*: it is either 0 or 1 here. Finally, the number of reports (one for each button) is specified, along with each report size (one bit). The last item creates an Input report with the previously defined parameter. The data must be specified as *Variable* because the field contains the state of several controls.

Since the value returned for the buttons is smaller than one byte, it needs to be padded with an extra 5 bits. This is done by adding one 5-bit Input report with a constant value:

```
HIDReport_GLOBAL_REPORTCOUNT + 1, 1,
HIDReport_GLOBAL_REPORTSIZE + 1, 5,
HIDReport_INPUT + 1, HIDReport_CONSTANT,
```

The last step is to define the pointer functionality of the mouse. Basically, two bytes of data are returned: one for the displacement along the X (horizontal) axis, and one for the displacement along the Y (vertical) axis. The displacement value can go both ways (left/right, up/down), so each byte will be signed and return a value between -127 and +127. The convention defined in the HID specification is that a negative value moves the cursor left (for the X axis) or up (for Y), and a positive value right or down.

```
HIDReport_GLOBAL_USAGEPAGE + 1, HIDGenericDesktop_PAGEID,
HIDReport_GLOBAL_REPORTSIZE + 1, 8,
HIDReport_GLOBAL_REPORTCOUNT + 1, 2,
HIDReport_LOCAL_USAGE + 1, HIDGenericDesktop_X,
HIDReport_LOCAL_USAGE + 1, HIDGenericDesktop_Y,
HIDReport_GLOBAL_LOGICALMINIMUM + 1, (unsigned char) -127,
HIDReport_GLOBAL_LOGICALMAXIMUM + 1, 127,
HIDReport_INPUT + 1, HIDReport_VARIABLE | HID_RELATIVE,
```

Note that the usage page must be switched back to *Generic Desktop*, because the X and Y-axis functionalities are defined in it. The Input report must be relative, as specified earlier in this document.

Finally, the two collections that were previously started must be closed using two **End Collection** items:

```
        HIDReport_ENDCOLLECTION,
    HIDReport_ENDCOLLECTION
};
```

This Report descriptor defined here enables the device to send a three-bytes report with the following data:

Table 4-1. Mouse Report Data

Field	Length	Description
bmButtons	3 bits of 1byte	Bitmap state of the three mouse buttons.
bX	1 byte	Pointer displacement along the X axis.
bY	1 bytes	Pointer displacement along the Y axis.

It is useful to actually declare the corresponding structure, to easily modify, send and receive report values. This has been done in the example program provided with this document: the structure is named *HIDDMouseInputReport*.

As a side note, the Report descriptor defined in this section is compliant with the Boot protocol, as highlighted by Appendix B of the *HID specification 1.11*.

4.3.6 Physical Descriptor

A Physical descriptor is useless for a mouse device, so there will not be any defined in this example.

4.3.7 Endpoint Descriptor

Since it has be specified that the HID interface uses one endpoint, the corresponding **Endpoint** descriptor must now be defined. As mentioned previously, this is an Interrupt IN endpoint. The USB controllers of AT91SAM chips all support the Interrupt type at any available address, so this endpoint will be given address 01h.

Additionally, an Interrupt endpoint maximum packet size should be as small as possible. The host must reserve a minimum amount of bandwidth which depends on this value. Defining a small value minimizes the loss of bandwidth, but is only possible when the data size is known. In this case, it will always be 3 bytes, so *wMaxPacketSize* can be set accordingly.

Finally, since a mouse device response latency is not extremely critical, it can be safely set to a high value. In this example, the endpoint is polled every 10 ms.

```
// Endpoint Descriptor
    {
        sizeof(USBEndpointDescriptor), // Size of this descriptor
        USBGenericDescriptor_ENDPOINT, // ENDPOINT descriptor type
        0x01 | HIDDKeyboardDriverDescriptors_INTERRUPTIN, // EP 1, IN
        USBEndpointDescriptor_INTERRUPT, // Endpoint type interrupt
        sizeof(HIDDMouseInputReport), // Endpoint maximum packet size (3)
        0x0A // Interval for polling: 10ms
}
```

4.3.8 String Descriptors

Several descriptors can be commented with a String descriptor. The latter are completely optional and do not influence the detection of the device by the operating system. Whether or not to include them is entirely up to the programmer.

4.4 Class-Specific Requests

A number of HID-only requests are defined in the corresponding specification. They have already been described in Section 3.2.4 on page 4. This section details their implementation regarding the current example of a HID mouse device.

A driver request handler should first differentiate between class-specific and standard requests using the corresponding bits in the *bmRequestType* field. In most cases, standard requests can be immediately forwarded to the standard request handler method; class-specific methods must be decoded and treated by the custom handler.

4.4.1 GetDescriptor

Three values have been added by the HID specification for the **GET_DESCRIPTOR** request. The high byte of the *wValue* field contains the type of the requested descriptor; in addition to the standard types, the HID specification adds the **HID descriptor** (21h), the **Report descriptor** (22h) and the **Physical descriptor** (23h) types.

There is no particular action to perform besides sending the descriptor. This can be done by using the *USBD_Write* method, after the requested descriptor has been identified:

```
//----------------------
case USBGenericRequest_GETDESCRIPTOR:
//----------------------
{
    unsigned char type = USBGetDescriptorRequest_GetDescriptorType(request);
    unsigned char length = USBGenericRequest_GetLength(request);
    switch(type) {
        case HIDGenericDescriptor_REPORT:
            // Adjust length and send report descriptor
            if (length > HIDDMouseDriverDescriptors_REPORTSIZE) {
                length = HIDDMouseDriverDescriptors_REPORTSIZE;
            }
            USBD_Write(0, &hiddReportDescriptor, length, 0, 0);
            break;
        case HIDGenericDescriptor_HID:
            if (USBD_IsHighSpeed()) {
                pConfiguration =
                    hiddMouseDriver.usbdDriver.pDescriptors->pHsConfiguration;
            }
            else {
                pConfiguration =
                    hiddMouseDriver.usbdDriver.pDescriptors->pFsConfiguration;
            }
            // Parse the device configuration to get the HID descriptor
            USBConfigurationDescriptor_Parse(pConfiguration, 0, 0,
                                (USBGenericDescriptor **) &hidDescriptor);
            // Adjust length and send HID descriptor
            if (length > sizeof(HIDDescriptor)) {
                length = sizeof(HIDDescriptor);
            }
            USBD_Write(0, hidDescriptor, length, 0, 0);
            break;
        default:
            USBDDriver_RequestHandler(&(hiddMouseDriver.usbdDriver),request);
    }
}
```

A slight complexity of the GET_DESCRIPTOR and SET_DESCRIPTOR requests is that those are standard requests, but the standard request handler (*USBDDriver_RequestHandler*) must not always be called to treat them (since they may refer to HID descriptors). The solution is to

first identify GET/SET_DESCRIPTOR requests, treat the HID-specific cases and, finally, forward any other request to the standard handler.

In this case, a GET_DESCRIPTOR request for the Physical descriptor is first forwarded to the standard handler, and STALLed there because it is not recognized. This is done because the device does not have any Physical descriptors, and thus, does not need to handle the associated request.

4.4.2 SetDescriptor

This request is optional and is never issued by most hosts. It is not implemented in this example.

4.4.3 GetReport

When a **GET_REPORT** request is received, the current status of the device Report must be returned. It will be assumed here that the current report is held in a structure and is accessible:

```
//------------------
case HIDGenericRequest_GETREPORT:
//------------------

if (length <= HIDDTransferDriver_REPORTSIZE && type ==
HIDReportRequest_INPUT) {

USBD_Write(0,
hiddTransferDriver.oReportBuf,
length,
HIDDTransferDriver_ReportSent,
0);
}
else {

USBD_Stall(0);
}
break;
```

Note that since a Report ID is not defined in the Report descriptor, only the Report Type value defined in the *wValue* field is meaningful. It should be parsed to check if it is Input, but in practice, the host should never request a Report which does not exist.

4.4.4 SetReport

Since a mouse device has no Output report, the only usage of the **SET_REPORT** request is to initialize the state of the Input report. According to the HID specification, this is completely optional; the data still has to be read from the USB, but can be safely discard:

```
//------------------
case HIDGenericRequest_SETREPORT:
//------------------

if (length <= HIDDTransferDriver_REPORTSIZE && type ==
HIDReportRequest_OUTPUT) {

USBD_Read(0,
```

```
hiddTransferDriver.iReportBuf,
length,
HIDDTransferDriver_ReportReceived,
0); // No argument to the callback function } else {

USBD_Stall(0);
}
break;
```

4.4.5 SetIdle

The **SET_IDLE** request is used to change the Idle rate of a HID device. This is the minimum amount of time before an unchanged report can be transmitted again.

A mouse device should never send a report indicating an unchanged state. Usually, a SET_IDLE command with 0 as a parameter, indicating that the Idle rate is indefinite, is sent by the host.

In practice, it is not necessary to perform any action, apart from sending a zero-length packet to acknowledge it. The main application however has to make sure that only new reports are sent by the device.

```
//-----------------
case HIDGenericRequest_SETIDLE:
//-----------------
    hiddMouseDriver.inputReportIdleRate = idleRate;
    USBD_Write(0, 0, 0, 0, 0);
    break;
```

Since the next request, GET_IDLE, needs to send back the current Idle rate, it needs to be stored in a variable; in the example, this is done at the class driver level (*HIDDMouseDriver* structure).

4.4.6 GetIdle

The only necessary operation for this request is to send the previously saved Idle rate. This is done by calling the *USBD_Write* method with the one-byte variable as its parameter:

```
//-----------------
case HIDGenericRequest_GETIDLE:
//-----------------
    USBD_Write(0, &(hiddMouseDriver.inputReportIdleRate), 1, 0, 0)
    break;
```

4.4.7 GetProtocol

When a HID device implements the Boot protocol, the host can request the current mode of operation by using the **GET_PROTOCOL** command. If the device is currently using the Boot protocol, then it must return 0 to the host, and 1 otherwise.

For the current example, the operation is similar whether in Boot protocol mode or not. However, to be consistent between the SET and GET_PROTOCOL requests, the current operating mode should still be stored. The *inputProtocol* field of the *HIDDMouseDriver* structure can be used to do this in a efficient way:

```
//--------------------
```

```
case HIDGenericRequest_GETPROTOCOL:
//--------------------
USBD_Write(0, &hiddMouseDriver.inputProtocol, 1, 0, 0);
   break;
```

inputProtocol is used as a bitmap field, and the data is sent using the *wData* field to have a persistent storage variable (needed by the *USBD_Write* method).

4.4.8 SetProtocol

Similarly to the previous request, the device should simply store the new mode given by the SET_PROTOCOL value in a variable, and acknowledge the request with a ZLP:

```
//--------------------
case HIDGenericRequest_SETPROTOCOL:
//--------------------
   hiddMouseDriver.inputProtocol = request->wValue;
   USBD_Write(0, 0, 0, 0, 0);
   break;
```

4.5 Main Application

The main function of the application has to perform two actions. The first one is to monitor the physical buttons and sensors of the device, to detect a movement or a pressed button.

In addition, the device must also report those changes, by sending the Report value through the Interrupt IN endpoint. This is done by using the *USBD_Write* function on endpoint 01h.

In the example software, the pointer is moved using the joystick present on the evaluation board. A left or right click can be performed by pushing either buttons 1 or 2.

4.6 Example Software Usage

4.6.1 File Architecture

The software example provided along with this application note is divided into four parts:

- **at91lib\usb\common\hid**: Folder for all general definitions for USB HID devices
- *HIDDescriptor.h*: header with definition of HID descriptor
- *HIDDeviceDescriptor.h*: header with definitions used in HID device descriptor
- *HIDGenericDescriptor.h*: header with definitions for using HID-specific descriptors
- *HIDGenericDesktop.h*: header with constants for using the HID generic desktop usage page
- *HIDInterfaceDescriptor.h*: header with definitions used in HID interface descriptor
- *HIDGenericRequest.h*: header with definition of constants for using HID-specific requests
- *HIDReport.h*: header with definitions used when declaring an HID report descriptor.
- *HIDReportRequest.c, HIDReportRequest.h*: methods and defintions to manipulate HID-specific GET_REPORT and SET_REPORT requests
- *HIDIdleRequest.c HIDIdleRrequest.h*: Methods and constants for manipulating HID-specific GET_IDLE and SET_IDLE requests
- *HIDButton.h*: definitions for the HID Buttons usage page
- **at91lib\usb\device\hid-mouse**: Folder for all source for USB HID mouse driver
- *HIDDMouseDriver.c, HIDDMouseDriver.h*: methods and definitions for HID mouse driver

- *HIDDMouseDriverDescriptors.c, HIDDMouseDriverDescriptors.h*: definition for HID mouse descritpros
- *HIDDMouseInputReport.c, HIDDMouseInputReport.h*: methods and defintions for HID mouse to handle input report

4.6.2 Compilation

The software is provided with a **Makefile** to build it. It requires the *GNU make* utility, which is available on www.GNU.org. Refer to the Atmel *AT91 USB Device Framework* application note for more information on general options and parameters of the Makefile.

To build the USB HID mouse example just run “make” in directory **usb-device-hid-mouse-project**, and two parameters may be assigned in command line, the CHIP= and BOARD=, the default value of these parameters are “at91sam7se512” and “at91sam7se-ek”:

```
make CHIP=at91sam7se512 BOARD=at91sam7se-ek
```

In this case, the resulting binary will be named *usb-device-hid-mouse-project-at91sam7se-ek-at91sam7se512-flash.bin* and will be located in the *usb-device-hid-mouse-project*/*bin* directory.

4.7 Using a Generic Host Driver

HID devices probably have the best native support for a USB class. As such, all operating systems can detect and use an HID mouse without any problem whatsoever.

5. HID Keyboard

This section describes how to implement a keyboard device using the HID class and the AT91 USB Device Framework. For more information on the framework, refer to the Atmel *AT91 USB Device Framework* application note; details about the USB and the HID class can be found in the *USB specification 2.0* and the *HID specification 1.11*, respectively.

Most topics have already been discussed in Section 4., but in greater detail. The reader is advised to read Section 4. first.

5.1 Purpose

The keyboard is the primary way of interaction with a computer. An HID keyboard has several advantages, e.g., it can be plugged in anytime and is immediately functional. In addition, new functionalities can be added by using the HID class: for example, it is easy to add an integrated pointing device (like a trackball) to an HID keyboard.

This example demonstrates how to not only send but also receive data using the HID class. Indeed, the computer needs to transmit the state of the LEDs to the keyboard.

5.2 Architecture

Please refer to Section 4.2 on page 6 for more information about the **HID driver architecture** used in this example, which is identical to the one used for the HID mouse.

5.3 Descriptors

5.3.1 Device Descriptor, Configuration Descriptor

Please refer to Section 4.3.1 on page 7 for more information about the Device descriptor of a HID device, and to Section 4.3.2 on page 7 for the Configuration descriptor.

5.3.2 HID Class Interface Descriptor

Since a keyboard device needs to transmit as well as receive data, two Interrupt (IN & OUT) endpoints are needed. This must be indicated in the **Interface descriptor**. Conversely to the mouse example, the Boot protocol is not implemented here, since there are more constraints on a keyboard device.

```
// HID Class Interface Descriptor
{
    sizeof(USBInterfaceDescriptor), // Size of this descriptor
    USBGenericDescriptor_INTERFACE, // INTERFACE Descriptor Type
    0x00, // Interface 0
    0x00, // No alternate settings
    0x02, // Two endpoints used
    HIDInterfaceDescriptor_CLASS, // Class HID (Human Interface Device)
    HIDInterfaceDescritpor_SUBCLASS_NONE, // No subclass
    HIDInterfaceDescritpor_PROTOCOL_NONE, // No protocol
    0x00 // No associated string descriptor
}
```

5.3.3 HID Descriptor

While a HID keyboard produces two different reports, one Input and one Output, only one Report descriptor can be used to describe them. Since having Physical descriptors is also useless for a keyboard, there will only be one HID class descriptor specified here.

For a keyboard, the *bCountryCode* field can be used to specify the language of the key caps. As this is optional, it is simply set to 00h in the example:

```
// HID-Specific Descriptor
{
    sizeof(HIDDescriptor), // Size of this descriptor in bytes (9)
    HIDGenericDescriptor_HID, // HID descriptor type
    HIDDescriptor_HID1_11, // HID Class Specification 1.11
    0x00, // No country code
    0x01, // 1 HID class descriptor
    HIDGenericDescriptor_REPORT, // First HID-specific descriptor type
    sizeof(hiddReportDescriptor) // Total length of first descriptor
}
```

5.3.4 Report Descriptor

Two current reports are defined in the **Report descriptor**. The first one is used to notify the host of which keys are pressed, with both modifier keys (alt, ctrl, etc.) and alphanumeric keys. The second report is necessary for the host to send the LED (num lock, caps lock, etc.) states.

The Report descriptor starts with the global device functionality, described with a **Usage Page** and a **Usage** items:

```
const unsigned char hiddReportDescriptor[] = {
    HIDReport_GLOBAL_USAGEPAGE + 1, HIDGenericDesktop_PAGEID,
    HIDReport_LOCAL_USAGE + 1, HIDGenericDesktop_KEYBOARD,
```

As in the mouse example, the *Generic Desktop* page is used. This time, the specific usage is the *Keyboard* one. An *Application collection* is then defined to group the reports together:

```
HIDReport_COLLECTION + 1, HIDReport_COLLECTION_APPLICATION,
```

The first report to be defined is the modifier keys. They are represented as a bitmap field, indicating whether or not each key is pressed. A single byte is used to map keys #224-231 defined in the *HID Usage Tables* document: LeftControl, LeftShift, LeftAlt, LeftGUI (e.g. Windows key), RightControl, RightShift, RightAlt and RightGUI. The *Keypad* usage page must be specified for this report, and since this is a bitmap value, the data is flagged as *Variable*:

```
HIDReport_GLOBAL_REPORTSIZE + 1, 1,
HIDReport_GLOBAL_REPORTCOUNT + 1, 8,
HIDReport_GLOBAL_USAGEPAGE + 1, HIDKeyboard_PAGEID,
HIDReport_LOCAL_USAGEMINIMUM + 1,
                  HIDDKeyboardDriverDescriptors_FIRSTMODIFIERKEY,
HIDReport_LOCAL_USAGEMAXIMUM + 1,
                  HIDDKeyboardDriverDescriptors_LASTMODIFIERKEY,
HIDReport_GLOBAL_LOGICALMINIMUM + 1, 0,
HIDReport_GLOBAL_LOGICALMAXIMUM + 1, 1,
HIDReport_INPUT + 1, HIDReport_VARIABLE,
```

Then, the actual alphanumeric key report is described. This is done by defining several bytes of data, one for each pressed key. In the example, up to three keys can be pressed at the same time (and detected) by the user. Once again, the usage page is set to *Keypad*. This time however, the data must be specified as an *Array*, since the same control (the keypad) produces several values:

```
HIDReport_GLOBAL_REPORTCOUNT + 1, 3,
HIDReport_GLOBAL_REPORTSIZE + 1, 8,
HIDReport_GLOBAL_LOGICALMINIMUM + 1,
                    HIDDKeyboardDriverDescriptors_FIRSTSTANDARDKEY,
HIDReport_GLOBAL_LOGICALMAXIMUM + 1,
                    HIDDKeyboardDriverDescriptors_LASTSTANDARDKEY,
HIDReport_GLOBAL_USAGEPAGE + 1, HIDKeyboard_PAGEID,
HIDReport_LOCAL_USAGEMINIMUM + 1,
                    HIDDKeyboardDriverDescriptors_FIRSTSTANDARDKEY,
HIDReport_LOCAL_USAGEMAXIMUM + 1,
                    HIDDKeyboardDriverDescriptors_LASTSTANDARDKEY,
HIDReport_INPUT + 1, 0, // Data Array
```

The LED array is finally defined, with the associated usage page. The Report descriptor is formatted in this order to avoid redefining unchanged *Global* items, in order to save memory. This time again, the LED status is reported as a bitmap field. Three LEDs are used here: Num Lock, Caps Lock and Scroll Lock (IDs 01h to 03h). It is important to note that this is an **Output** report:

```
HIDReport_GLOBAL_REPORTCOUNT + 1, 3,
HIDReport_GLOBAL_REPORTSIZE + 1, 1,
HIDReport_GLOBAL_USAGEPAGE + 1, HIDLeds_PAGEID,
HIDReport_GLOBAL_LOGICALMINIMUM + 1, 0,
HIDReport_GLOBAL_LOGICALMAXIMUM + 1, 1,
HIDReport_LOCAL_USAGEMINIMUM + 1, HIDLeds_NUMLOCK,
HIDReport_LOCAL_USAGEMAXIMUM + 1, HIDLeds_SCROLLLOCK,
HIDReport_OUTPUT + 1, HIDReport_VARIABLE,
```

Since the previous report only contains 3 bits, the data must be padded to a multiple of one byte. This is done by using constant Output data, as follows:

```
        HIDReport_GLOBAL_REPORTCOUNT + 1, 1,
        HIDReport_GLOBAL_REPORTSIZE + 1, 5,
        HIDReport_OUTPUT + 1, HIDReport_CONSTANT, // LED report padding
```

The last item, *End Collection*, is necessary to close the previously opened *Application Collection*.

```
        HID_MAIN_ENDCOLLECTION
    };
```

The Input and Output reports defined by this descriptor can be modeled by the following structures:

Table 5-1. HID Keyboard Input Report Structure

Field	Size	Description
bmModifierKeys	1 byte	Bitmap field reporting the state of the 8 modifier keys.
pressedKeys	3 byte	Pointer to pressed keys array.

Table 5-2. HID Keyboard Output Report Structure

Field	Size	Description
numLockStatus	1 bit	Status of the NumLock keyboard LED.
capsLockStatus	1 bit	Status of the CapsLock keyboard LED.
scrollLockStatus	1 bit	Status of the ScrollLock keyboard LED.
bPadding	5 bits	Padding data.

The two corresponding structures have been defined for these reports in the example software; they are named *HIDDKeyboardInputReport* and *HIDDKeyboardOutputReport*. An instance of each one of them is stored in a *HIDDKeyboardDriver* structure, which holds the standard class driver and HID keyboard-specific data.

5.3.5 Physical Descriptor

A Physical descriptor is useless for a keyboard device, so none are defined in this example.

5.3.6 Endpoint Descriptors

Following the Interface and HID-specific descriptors, the two necessary endpoints are defined. The parameters are similar as those defined in Section 4.3.7 on page 11, except the Interrupt OUT endpoint is given address 02h. The maximum packet size is also adjusted to 7 and 1 byte for the IN and OUT endpoints (respectively):

```
// Interrupt IN endpoint descriptor
{
sizeof(USBEndpointDescriptor),
USBGenericDescriptor_ENDPOINT,
USBEndpointDescriptor_ADDRESS(
USBEndpointDescriptor_IN,
HIDDKeyboardDriverDescriptors_INTERRUPTIN),
USBEndpointDescriptor_INTERRUPT,
```

```
sizeof(HIDDKeyboardInputReport),
HIDDKeyboardDriverDescriptors_INTERRUPTIN_POLLING
},
// Interrupt OUT endpoint descriptor
{
sizeof(USBEndpointDescriptor),
USBGenericDescriptor_ENDPOINT,
USBEndpointDescriptor_ADDRESS(
USBEndpointDescriptor_OUT,
HIDDKeyboardDriverDescriptors_INTERRUPTOUT),
USBEndpointDescriptor_INTERRUPT,
sizeof(HIDDKeyboardOutputReport),
HIDDKeyboardDriverDescriptors_INTERRUPTIN_POLLING
}
```

5.3.7 String Descriptors

Please refer to Section 4.3.8 on page 11 for more information on String descriptors.

5.4 Class-specific Requests

5.4.1 GetDescriptor, SetDescriptor

Those requests are handled in the same way as described in Section 4.4 on page 11. Refer to the corresponding sections for details.

5.4.2 GetReport

Since the HID keyboard defines two different reports, the Report Type value specified by this request (upper byte of the *wValue* field) must be examined to decide which report to send. If the type value is 01h, then the Input report must be returned; if it is 02h, the Output report is requested:

```
//------------------
case HIDGenericRequest_GETREPORT:
//------------------
    {
        unsigned char type = HIDReportRequest_GetReportType(request);
        unsigned short length = USBGenericRequest_GetLength(request);
        if (type == HIDReportRequest_INPUT) {
            // Adjust size and send report
            if (length > sizeof(HIDDKeyboardInputReport)) {
                length = sizeof(HIDDKeyboardInputReport);
            }
            USBD_Write(0, &(hiddKeyboardDriver.inputReport), length, 0, 0);
        }
        else if (type == HIDReportRequest_OUTPUT) {
            // Adjust size and send report
            if (length > sizeof(HIDDKeyboardOutputReport)) {
                length = sizeof(HIDDKeyboardOutputReport);
            }
```

```
            USBD_Write(0, &(hiddKeyboardDriver.outputReport), length, 0, 0);
        }
        else {
            // Unknown report type
            USB_Stall(0);
        }
    }
    break;
```

5.4.3 SetReport

For an HID keyboard, the **SET_REPORT** command can be sent by the host to change the state of the LEDs. Normally, the dedicated Interrupt OUT endpoint will be used for this; but in some cases, using the default Control endpoint can save some bandwidth on the host side.

Note that the SET_REPORT request can be directed at the Input report of the keyboard; in this case, it can be safely discarded, according to the HID specification. Normally, most host drivers only target the Output report. The Report Type value is stored in the upper byte of the *wValue* field.

The length of the data phase to follow is stored in the *wLength* field of the request. It should be equal to the total length of the Output report. If it is different, the report status must still be updated with the received data as best as possible.

When the reception of the new data is completed, some processing must be done to enable/disable the corresponding LEDs. This is done in the callback function passed as an argument to *USBD_Read*:

```
//------------------
case HIDGenericRequest_SETREPORT:
//------------------
    {
        unsigned char type = HIDReportRequest_GetReportType(request);
        unsigned short length = USBGenericRequest_GetLength(request);
        if (type == HIDReportRequest_OUTPUT) {
            if (length != sizeof(HIDDKeyboardOutputReport)) {
                USBD_Stall(0);
            }
            else {
                // Read the new report value
                USBD_Read(0, &(hiddKeyboardDriver.outputReport), length,
                        (TransferCallback) HIDDKeyboardDriver_ReportReceived,
                        0); // No argument to the callback function
            }
        }
        else {
            USB_Stall(0);
        }
    }
    break;
```

5.4.4 SetIdle

In this case study, the **SET_IDLE** request is used to set a delay before a key is repeated. This is common behavior on keyboard devices. Usually, this delay is set to about 500 ms by the host.

The only action here is to store the new Idle rate. The management of this setting must be done in the main function, since Interrupt IN reports are sent from there.

Refer to Section 4.4.5 on page 14 for the sample code associated with this request.

5.4.5 GetIdle

Refer to Section 4.4.6 on page 14 for more information on the **GET_IDLE** request.

5.4.6 GetProtocol, SetProtocol

This HID keyboard example does not support the Boot protocol, so there is no need to implement the SET_PROTOCOL and GET_PROTOCOL requests. This means they can be safely STALLed when received.

5.5 Main Application

Like the mouse example, the main program must perform two different operations. First, it has to monitor the physical inputs used as keys. In the example software, the buttons present on the evaluation boards are used to produce several modifier and alphanumeric keys.

Also, the main program is in charge of sending reports as they are modified, taking into account the Idle rate specified by the host. Idle rate management can be carried out by firing/resetting a timer once a new report is sent; if the timer expires, this means the Input report has not changed since. According to the HID specification, a single instance of the report must be sent in this case.

Finally, the HID specification also defines that if too many keys are pressed at the same time, the device should report an *ErrorRollOver* usage value (01h) in every byte of the key array. This has to be handled by the main application as well.

5.6 Example Software Usage

5.6.1 File Architecture

The software example provided along with this application note is divided into several groups:

- **at91lib\usb\common\hid**: Folder for all general definitions for USB HID devices
- *HIDDescriptor.h*: header with definition of HID descriptor
- *HIDDeviceDescriptor.h*: header with definitions used in HID device descriptor
- *HIDGenericDescriptor.h*: header with definitions for using HID-specific descriptors
- *HIDGenericDesktop.h*: header with constants for using the HID generic desktop usage page
- *HIDInterfaceDescriptor.h*: header with definitions used in HID interface descriptor
- *HIDLeds.h*: header with definition for the HID LEDs usage page
- *HIDGenericRequest.h*: header with definition of constants for using HID-specific requests
- *HIDReport.h*: header with definitions used when declaring an HID report descriptor.
- *HIDReportRequest.c, HIDReportRequest.h*: methods and defintions to manipulate HID-specific GET_REPORT and SET_REPORT requests

- *HIDIdleRequest.c HIDIdleRrequest.h*: Methods and constants for manipulating HID-specific GET_IDLE and SET_IDLE requests
- *HIDKeypad.c, HIDKeypad.h*: constants and methods for the HID keypad usage page
- **at91lib\usb\device\hid-keyboard**: Folder for all definitions for HID keyboard device driver
- *HIDDKeyboardDriver.c, HIDDKeyboardDriver.h*: Definition of methods for using a HID keyboard device driver
- *HIDDKeyboardDriverDescriptors.c, HIDDKeyboardDriverDescriptors.h*: Definitions of the descriptors required by the HID device keyboard driver.
- *HIDDKeyboardCallbacks.h*: Definitions of callbacks used by the HID keyboard device driver to notify the application of events
- *HIDDKeyboardCallbacks_LedsChanged.c*: source code to implement the callback for LED status change
- *HIDDKeyboardInputReport.c, HIDDKeyboardInputReport.h*: Class for manipulating HID keyboard input reports
- *HIDDKeyboardOutputReport.c, HIDDKeyboardOutputReport.h*: Definition of a class for manipulating HID keyboard output reports
- **usb-device-hid-keyboard-project**: Foler for main program of the keyboard example
- *main.c*: main program for the keyboard example

5.6.2 Compilation

The software is provided with a **Makefile** to build it. It requires the *GNU make* utility, which is available on www.GNU.org. Please refer to the *AT91 USB Device Framework* application note for more information on general options and parameters of the Makefile.

To build the HID keyboard example just run “make” in directory **usb-device-hid-keyboard-project**, and two parameters may be assigned in command line, the CHIP= and BOARD=, the default value of these parameters are “at91sam7s256” and “at91sam7s-ek”:

```
make CHIP=at91sam7se512 BOARD=at91sam7se-ek
```

In this case, the resulting binary is named *usb-device-hid-keyboard-project-at91sam7se-ek-at91sam7se512.bin* and is located in the *usb-device-hid-keyboard-project/bin* directory.

5.7 Using a Generic Host Driver

HID devices probably have the best native support for a USB class. As such, all operating systems are able to detect and use a HID keyboard without any problem whatsoever.

6. Generic Data Transfer Using HID

6.1 Purpose

While there are several classes dedicated to communicating over the USB, most of them are slightly complex to implement. In addition, they often require that a complex host-side driver be developed to be able to use the device.

The HID class can be used with as a very simple way of communication between a device and a host on the USB. This is so because most operating systems provide a generic HID driver, which makes it possible to use custom-defined Report descriptors. An application can thus use the standard API provided by the OS vendor.

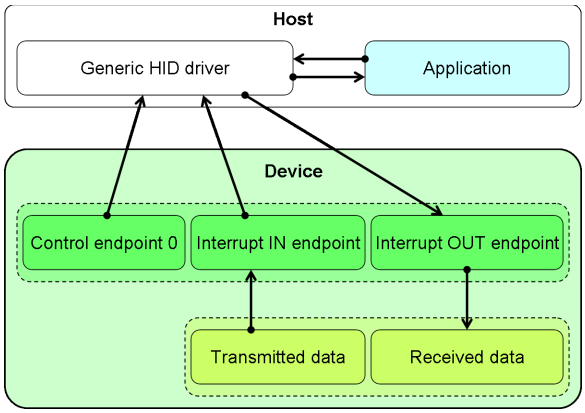
This case study describes the work needed on both sides (device and host) to use the HID class as a communication pipe. However, some topics have already been discussed in Section 4. and Section 5. of this application note, and are only treated briefly here.

6.2 Architecture

On the host side, a custom application must be developed to use the generic HID driver with the communication device. However, this is a simple task since the OS API should offer the necessary methods.

The HID driver on the device uses two Interrupt endpoints to receive and transmit data to the host. The default Control endpoint is used for the device enumeration.

Figure 6-1. Host and Device Architecture when Using HID as a Communication Medium



6.3 Descriptors

6.3.1 Device Descriptor, Configuration Descriptor

Refer to Section 4.3.1 on page 7 for more information about the Device descriptor of a HID device, and to Section 4.3.2 on page 7 for the Configuration descriptor.

6.3.2 HID Class Interface Descriptor, HID Descriptor

These two descriptors are similar to the ones described in Section 5.3.2 on page 17 and Section 5.3.3 on page 17. Like the HID keyboard, two Interrupt endpoints are defined, the Boot protocol is not used (it is meaningless here), and only one Report descriptor is used.

6.3.3 Report Descriptor

The **Report descriptor** is going to be entirely custom, since the *HID Usage Tables* document does not define any usage which may be appropriate for raw communication. To do that, the *Usage Page* and *Usage* items can be specified as vendor-specific:

```
const unsigned char hiddReportDescriptor[] = {
   HIDReport_GLOBAL_USAGEPAGE+2, 0xFF, 0xFF, // Vendor-defined
   HIDReport_LOCAL_USAGE+1, 0xFF, // Vendor-defined
```

Two one-byte reports, Input and Output, are defined in this descriptor. This makes it possible for the host and the device to easily exchange data. An *Application collection* is defined to group the two reports:

```
   HIDReport_COLLECTION+1, HIDReport_COLLECTION_APPLICATION,
```

The first report follows:

```
HIDReport_LOCAL_USAGE+1, 0xFF, // Vendor-defined usage
HIDReport_GLOBAL_LOGICALMINIMUM+1, (unsigned int) -128,
HIDReport_GLOBAL_LOGICALMAXIMUM+1, (unsigned int) 127,
HIDReport_GLOBAL_REPORTCOUNT+1, 1,
HIDReport_GLOBAL_REPORTSIZE+1, 8,
HIDReport_INPUT+1, 0, // No modifiers
```

The Output report is almost identical:

```
HIDReport_LOCAL_USAGE+1, 0xFF, // Vendor-defined usage
HIDReport_GLOBAL_LOGICALMINIMUM+1, (unsigned int) -128,
HIDReport_GLOBAL_LOGICALMAXIMUM+1, (unsigned int) 127,
HIDReport_GLOBAL_REPORTCOUNT+1, 1,
HIDReport_GLOBAL_REPORTSIZE+1, 8,
HIDReport_INPUT+1, 0, // No modifiers
```

Finally, the collection must be closed:

```
    HID_MAIN_END_COLLECTION
};
```

6.3.4 Physical Descriptor

A Physical descriptor is meaningless in this example, so none is defined.

6.3.5 Endpoint Descriptors

The two endpoint descriptors are similar to the ones defined in Section 5.3.6 on page 19, except that in order to maximize the data throughput, the maximum packet size can be set to 64 and the polling rate to 1ms, for both endpoints.

6.3.6 String Descriptors

Refer to Section 4.3.8 on page 11 for more information about String descriptors.

6.4 Class-specific Requests

6.4.1 GetDescriptor, SetDescriptor

These requests are handled in the same way as described in Section 4.4 on page 11; refer to the corresponding subsections for details.

6.4.2 GetReport, SetReport

The **GET_REPORT** and **SET_REPORT** requests can be used by the host when a low-latency transfer is not required. Normally, reports are exchanged over the two Interrupt endpoints.

In the current example, those two requests are not necessary, since it is more efficient to transfer the data using the Interrupt endpoints. Therefore, they need not be implemented, and the PC application makes sure that neither of them is ever sent.

6.4.3 SetIdle, GetIdle

As the two previous ones, these requests are not useful for a generic communication driver.

6.4.4 GetProtocol, SetProtocol

The Boot protocol is meaningless for this example, so these requests are not implemented.

6.5 Device-side Application

The main program of the device should simply use the *USBD_Write* and *USBD_Read* functions on the two Interrupt endpoints (endpoints 01h and 02h) to either send or receive data. Since these two methods are asynchronous, they can be called without blocking the program execution flow.

6.6 Host-side Application

This section explains how to program a PC application to communicate with the custom HID device driver described previously. This example is targeted at a Microsoft® Windows® platform; it uses the Windows Device Driver Kit (DDK) provided by Microsoft for developers.

The application is divided into three parts. The first part consists of detecting the connection or disconnection of the HID device designed in this case study. Once the device is connected, the program can then open it and start communicating with the device. Finally, once the application terminates, it must free the device resources that have been used.

6.6.1 Detecting the Device

There are two possibilities when the application starts: the device may be either connected or not. This means the program must first check if the device is present; if not, it will have to wait for its connection on the bus.

6.6.1.1 Finding a Connected Device

One way of checking that the device is connected is to look at the whole list of connected HID peripherals. The particular device developed in this example can be identified by its Vendor ID and Product ID. However, to get the Vendor ID and Product ID of a device, it must first be opened with the *CreateFile* function, which requires the name of the device.

The *SetupDiGetClassDevs* method of the DDK can be used to retrieve a set of device with particular properties. It can be used to retrieve all the devices which implement the HID class. To do that, the Globally Unique Identifier (GUID) or the HID class must be passed to the function. GUID are used in driver development to identify devices, interfaces or classes in a unique way. Getting the GUID of the HID class can be done by calling the *HidD_GetHidGuid* function.

Once a handle on the set of HID devices has been retrieved, the application must look for the correct device among the set. *SetupDiEnumDeviceInterfaces* is used to retrieve information about the interfaces of each device. Another function, *SetupDiGetDeviceInterfaceDetail*, is finally called to get a structure containing the name of the device.

The *CreateFile* method can now be invoked with the device name as its parameters. The returned handle can be exploited through a call to *HidD_GetAttributes*, which finally return the Vendor ID, Product ID and Serial number of the device. Those values must be checked to see if the current device is the one expected by the application. If it is not, then the application shall continue to examine all other connected devices; if this is the right one, the program closes all unneeded resources (list of HID devices, etc.) and can now use the device.

6.6.1.2 Handling Device Connection & Disconnection

In the event that the device was not found at program startup, the latter must register to be notified by the operating system of its connection. Also, even if the device was actually connected, it can still be brutally removed from the bus by the user; the application should be ready to handle this case.

When a device is plugged or unplugged, a WM_DEVICECHANGE message is issued to applications which are listening to it. The message parameters indicate if the device has been inserted, removed, and various other states.

By using the *RegisterDeviceNotification* with the appropriate parameters, a PC program can start listening to all events generated by HID devices. However, since it is not possible to request messages only from devices with particular Vendor ID and Product ID values, the application will have to check whether or not each message is relevant.

6.6.2 Communication

Once the device is connected and its handle has been retrieved by using *CreateFile*, the application can start communicating with it. Conversely to the previous subsection, this step is very straightforward: the *ReadFile* and *WriteFile* are used to read and send data to the device.

Note that when using Report IDs, the correct ID value must be prepended manually before each transfer, in the first byte of the read/write buffer. In addition, a single call to *ReadFile* only return one report, regardless of the data sent by the device. The bandwidth can be really cut down because of this. A workaround is to define a larger report (e.g. 63 bytes plus the report ID). However, keep in mind that a report must always contain the correct number of bytes, so a smaller report will have to be padded before transmission.

6.6.3 Freeing Resources

When the application has finished using the device, it must first free it using the *CloseHandle* method, and unregister the notification of the WM_DEVICECHANGE message.

7. Revision History

Table 7-1. Revision History

Document Reference	Date	Comments	Change Request Ref.
6273A	20-Oct-06	First issue.	
6273B	25-Jun-09	Section 4.6.2, Section 5.6.2: Need more informations on the nmake utility	3927

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com
www.atmel.com/AT91SAM

Technical Support
AT91SAM Support
Atmel techincal support

Sales Contacts
www.atmel.com/contacts/

Literature Requests
www.atmel.com/literature