AT91 USB CDC Driver Implementation

1. Introduction

The **Communication Device Class** (CDC) is a general-purpose way to enable all types of communications on the Universal Serial Bus (USB). This class makes it possible to connect **telecommunication devices** such as digital telephones or analog modems, as well as **networking devices** like ADSL or Cable modems.

While a CDC device enables the implementation of quite complex devices, it can also be used as a very simple method for communication on the USB. For example, a CDC device can appear as a **virtual COM port**, which greatly simplifies application programming on the host side.

The purpose of this document is to explain how to implement CDC on AT91 ARM® Thumb® based microcontrollers using the AT91 USB Framework offered by Atmel®. For this purpose, a sample implementation of a **USB to Serial converter** is described step-by-step.

2. Related Documents

[1] Universal Serial Bus Class Definitions for Communication Devices, Version 1.1, January 19, 1999.

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

AT91 ARM Thumb Microcontrollers

Application Note

6269A–ATARM–10-Oct-06

3. Communication Device Class Basics

This section gives some basic information about the Communication Device Class, such as when to use it and which drivers are available for it. Its architecture is also described.

3.1 Purpose

CDC is used to connect communication devices, such as modems (digital or analog), telephones or networking devices. Its generic framework supports a wide variety of physical layers (xDSL, ATM, etc.) and protocols.

In this document, CDC is used to implement a USB to a serial data converter. Serial ports (also known as COM or RS-232 ports) are still widely used, but most modern PCs (especially laptops) are now shipped without serial ports. A USB to serial converter can be used in this case to bridge a legacy RS-232 interface with a USB port.

3.2 Architecture

3.2.1 Interfaces

Two interfaces are defined by the *CDC specification 1.1*. The first one, the **Communication Class Interface**, is used for device management. This includes requests to manage the device state, its responses, as well as event notifications. This interface can also be optionally used for call management, i.e., setting up and terminating calls as well as managing their parameters.

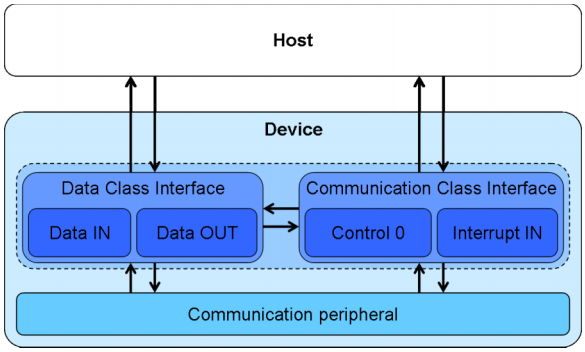
Another interface is defined for generic data transmissions. It is referred to as the **Data Class Interface**. It provides a means for a communication device to actually transfer data to and from the host. In addition, it also enables the multiplexing of data and commands on the same interface, through the use of wrappers.

3.2.2 Endpoints

The Communication Class Interface requires at least one endpoint, which is used for device management. Default control endpoint 0 is used for this task. Optionally, another endpoint can be dedicated to events notification. This will usually be an **Interrupt IN** endpoint.

For the Data Class Interface, endpoints must exist in pairs of the same type. This is necessary to allow both IN and OUT communication. Only the **Bulk** and **Isochronous** types can be used for these endpoints.

Figure 3-1. CDC Class Driver Architecture



3.2.3 Models

To account for the wide variety of existing communication devices, several **models** have been defined. They describe the requirements in term of interfaces, endpoints and requests that a device must fulfill to perform a particular role. Here is a list of models defined in the *CDC specification 1.1*, grouped by their intended functionality:

- POTS (Plain Old Telephone Service)
  - Direct Line Control Model
  - Datapump Model
  - Abstract Control Model
- Telephone
  - Telephone Control Model
- ISDN
  - Multi-Channel Model
  - USB CAPI Model
- Networking
  - Ethernet Networking Model
  - ATM Networking Control Model

Some of these models and their uses will be detailled further in this document, along with the corresponding implementation cases.

3.2.4 Class-Specific Descriptors

CDC-specific information is described using **Functional Descriptors**. They define various parameters of an interface, such as how the device handles call management, or model-specific attributes.

Since the CDC specification defines quite a number of functional descriptors, they are not detailed here. Instead, they are presented in the various case studies of this document in which they are used.

3.3 Host Drivers

Most Operating Systems (OS) now include generic drivers for a wide variety of USB classes. This makes developing a device simpler, since the host complexity is now handled by the OS. Manufacturers can thus concentrate on the device itself, not on developing specific host drivers.

Here is a brief list of the various CDC implementations supported by several OS:

- Windows®
  - Abstract Control Model
  - Remote NDIS
- Linux®
  - Abstract Control Model
  - Ethernet Model

4. USB to Serial Converter

This section describes the implementation of an USB to serial converter using the CDC class and the AT91 USB Framework. Refer to the Atmel document, literature no. 6263 for information

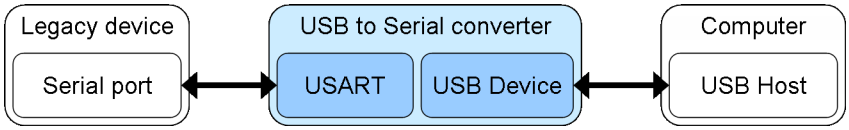
on the USB framework, and to the *CDC specification 1.1* and the *USB specification 2.0* for USB/CDC-related details.

4.1 Purpose

While the USB is increasingly used for a lot of new products, many legacy devices still use a basic RS-232 (also named serial or COM) port to connect to a PC. This kind of product is still widely available, but most computer manufacturers are starting to ship their machines without any COM port. A USB to serial converter can be used to add a **virtual COM port** to a computer, enabling the connection to a legacy RS-232 device.

A virtual COM port could also be used to provide a way for an USB device to connect to older PC applications. This is also a simple way to communicate through the USB, as no custom driver programming is needed.

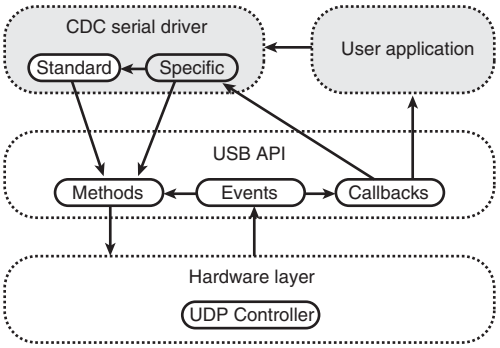
Figure 4-1. Bridging a Legacy Device and a PC with a USB to Serial Converter



4.2 Architecture

The AT91 USB Framework offers an API which makes it easy to build USB class drivers. The example software provided with this application note is based on this framework. Figure 4-2 shows the application architecture with the framework.

Figure 4-2. Software Architecture Using the AT91 USB Framework



4.3 Model

The CDC specification defines a model which suits this application perfectly: the **Abstract Control Model** (ACM). It implements the requests and notifications necessary to communicate with an RS-232 interface.

The Abstract Control Model requires two interfaces, one **Communication Class Interface** and one **Data Class Interface**. Each of them must have two associated endpoints. The former shall have one endpoint dedicated to device management (default Control endpoint 0) and one for events notification (additional Interrupt IN endpoint).

The Data Class Interface needs two endpoints through which to carry data to and from the host. Depending on the application, these endpoints can either be *Bulk* or *Isochronous*. In the case of

a USB to serial converter, using *Bulk* endpoints is probably more appropriate, since the reliability of the transmission is important and the data transfers are not time-critical.

4.4 Descriptors

The descriptors used by the USB to serial converter are mostly standard ones, i.e., defined in the *USB specification 2.0*. The following code examples thus use the structures described in the *AT91 USB Framework* application note.

For CDC-specific descriptors, new types are needed. Their implementation is trivial however, as they are fully described in the CDC specification. Only the values contained in each descriptor are detailed, but the list of the necessary structures for this example is found below:

- S_cdc_header_descriptor
- S_cdc_call_management_descriptor
- S_cdc_abstract_control_management_descriptor
- S_cdc_union_descriptor

4.4.1 Device Descriptor

The **Device Descriptor** must specify the value 02h, corresponding to the Communication Device Class, in its *bDeviceClass* field. This is necessary to have the host driver correctly enumerate the device: since a CDC device often displays more than one interface, they have to be logically grouped together and not considered separate functionalities.

No subclass codes or protocol codes are defined for the CDC.

Here is how the device descriptor looks when using the S_usb_device_descriptor structure of the AT91 USB Framework:

```
const S_usb_device_descriptor sDevice = {

   sizeof(S_usb_device_descriptor), // Size of this descriptor
   USB_DEVICE_DESCRIPTOR, // DEVICE Descriptor Type
   USB2_00, // USB 2.0 specification
   USB_CLASS_COMMUNICATION, // USB Communication class code
   0x00, // No device subclass code
   0x00, // No device protocol code
   USB_ENDPOINT0_MAXPACKETSIZE, // Maximum packet size for endpoint zero
   USB_VENDOR_ATMEL, // ATMEL Vendor ID
   SER_PRODUCT_ID, // Product ID (6119)
   0x0001, // 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
};
```

The Vendor ID and Product ID fields are used to determine which driver to use when the device is enumerated. The Vendor ID is provided by the USB-IF organization after registration; the product ID is completely vendor-specific. In the example implementation provided with this document, the Atmel vendor ID (03EBh) is used along with a custom product ID (6119h).

4.4.2 Configuration Descriptor

When requested by the host, the configuration descriptor is followed by interface, endpoint and class-specific descriptors. While the CDC specification does not define any special values for the configuration descriptor, a set of class-specific descriptors is provided. They are referred to as **Functional Descriptors**, and some of them have to be implemented.

```
// Standard configuration descriptor
{
    sizeof(S_usb_configuration_descriptor), // Size of this descriptor
    USB_CONFIGURATION_DESCRIPTOR, // CONFIGURATION descriptor type
    sizeof(S_ser_configuration_descriptor), // Total size
    0x02, // Two interfaces are used by this configuration
    0x01, // Value 0x01 is used to select this configuration
    0x00, // No string is used to describe this configuration
    USB_CONFIG_SELF_NOWAKEUP, // Device is self-powered, no wakeup support
    USB_POWER_MA(100) // Maximum power consumption of the device is 100mA
}
```

4.4.3 Communication Class Interface Descriptor

The first interface to follow the configuration descriptor should be the **Communication Class Interface** descriptor. It should specify the Communication Class Interface code (02h) in its *bInterfaceClass* field.

The *bInterfaceSubClass* value selects the CDC Model used by the interface. In this case, the code corresponding to the Abstract Control Model is 02h.

Finally, a protocol code can be supplied if needed. Since this is not necessary for the USB to serial converter, it can be left at 0x00.

```
// Communication class interface descriptor
{
    sizeof(S_usb_interface_descriptor), // Size of this descriptor
    USB_INTERFACE_DESCRIPTOR, // INTERFACE Descriptor Type
    0x00, // Interface 0
    0x00, // No alternate settings
    0x01, // One endpoint used
    CDC_INTERFACE_COMMUNICATION, // Communication interface class
    CDC_ABSTRACT_CONTROL_MODEL, // Abstract control model subclass
    0x00, // No protocol code
    0x00 // No associated string descriptor
}
```

While the Communication Class Interface uses two endpoints (one for device management and one for events notification), the interface descriptor should have its *bNumEndpoints* field set to 0x01: the default control endpoint 0 is not included in the count.

4.4.4 Functional Descriptors

Several **Functional Descriptors** must follow the communication class interface descriptor. They are necessary to define several attributes of the device. The functional descriptor structure contains the descriptor length, type and subtype, followed by functional information.

The *bDescriptorType* value is always equal to CS_INTERFACE (24h), since CDC-specific descriptors only apply to interfaces. The values for the other two fields, *bFunctionLength* and *bDescriptorSubType*, are function-dependent.

4.4.4.1 Header

The first functional descriptor must always be the **Header Functional Descriptor**. It is used to specify the CDC version on which the device is based (current version is 1.1):

```
// Header functional descriptor
{
    sizeof(S_cdc_header_descriptor), // Size of this descriptor in bytes
    CDC_CS_INTERFACE, // CS_INTERFACE descriptor type
    CDC_HEADER, // Header functional descriptor
    CDC1_10, // CDC version 1.10
}
```

4.4.4.2 Call Management

Next comes the **Call Management Functional Descriptor**. This one indicates how the device processes call management. If the device performs the call management duty itself, the first bit of the *bmCapabilities* field must be set to one. In addition, call management requests can be multiplexed over the data class interface instead of being sent on the Control endpoint 0, by setting the second bit. According to the CDC specification, a device using the Abstract Control Model should process call management itself, so bit D0 will be set. The last byte (*bDataInterface*) has no meaning here, since bit D1 of *bmCapabilities* is cleared.

```
// Call management functional descriptor
{
    sizeof(S_cdc_call_management_descriptor), // Size of this descriptor
    CDC_CS_INTERFACE, // CS_INTERFACE type
    CDC_CALL_MANAGEMENT, // Call management descriptor
    0x01, // Call management is handled by the device
    0x01 // Data interface is 0x01
}
```

4.4.4.3 Abstract Control Management

Since the USB to serial converter uses the Abstract Control Model, the corresponding functional descriptor (**Abstract Control Management Functional Descriptor**) must be transmitted to give more information on which requests/notifications are implemented by the device. For this example, the driver is going to support all optional requests/notification except *NetworkConnection*; thus, the *bmCapabilities* field value will be set to 07h.

```
// Abstract control management functional descriptor
{
    // Size of this descriptor in bytes
    sizeof(S_cdc_abstract_control_management_descriptor),
    // CS_INTERFACE descriptor type
    CDC_CS_INTERFACE,
    // Abstract control management functional descriptor
    CDC_ABSTRACT_CONTROL_MANAGEMENT,
    // Every request/notification except NetworkConnection supported
```

```
    0x07
}
```

4.4.4.4 *Union*

Finally, the **Union Functional Descriptor** makes it possible to group several interfaces into one global function. In this case, the Communication Class Interface will be set as the master interface of the group, with the Data Class Interface as slave 0:

```
// Union functional descriptor with one slave interface
{
    // Union functional descriptor
    {
        sizeof(S_cdc_union_descriptor)+1, // Size of this descriptor
        CDC_CS_INTERFACE, // CS_INTERFACE descriptor type
        CDC_UNION, // Union functional descriptor
        0x00, // Master interface is 0x00 (Communication class interface)
    }
    0x01 // First slave interface is 0x01
}
```

4.4.5 Notification Endpoint Descriptor

As said previously, the notification element used by the Abstract Control Model is an Interrupt IN endpoint. The user-defined attributes are the endpoint address and the polling rate.

When choosing endpoint addresses, the specificities of the USB controller should be taken into account. For example, on AT91SAM7S chips, the UDP has only four endpoints, one of which is used by the default Control endpoint 0. Since only the second and third endpoints have dual FIFO banks, it seems wiser to use them for the Data Class Interface and have the last one used for events notification.

Finally, the polling rate should be set depending on the interrupt source. In this case, this will be a USART. Since this is a fairly slow interface, the polling rate can be relatively low, meaning the *bInterval* value can be high.

Here is how the notification endpoint descriptor is declared in the example software:

```
// Notification endpoint descriptor
{
    sizeof(S_usb_endpoint_descriptor), // Size of this descriptor
    USB_ENDPOINT_DESCRIPTOR, // ENDPOINT descriptor type
    USB_ENDPOINT_IN | SER_NOTIFICATION, // IN endpoint, address = 0x03
    ENDPOINT_TYPE_INTERRUPT, // INTERRUPT endpoint type
    64, // Maximum packet size is 64 bytes
    0x10 // Endpoint polled every 10ms
}
```

4.4.6 Data Class Interface Descriptor

The **Data Class Interface Descriptor** follows the Communication Class Interface and its related functional and endpoint descriptors. The Data Class Interface itself will have two endpoint descriptors following it.

The Data Class Interface code is 0Ah, and there is no subclass code. Several protocol codes are available; in this example, none is used. However, the v.42bis protocol could be used to compress the data if supported by the legacy device.

```
// Data class interface descriptor
{
   sizeof(S_usb_interface_descriptor), // Size of this descriptor
   USB_INTERFACE_DESCRIPTOR, // INTERFACE descriptor type
   0x01, // Interface 0x01
   0x00, // No alternate settings
   0x02, // Two endpoints used
   CDC_INTERFACE_DATA, // Data class code
   0x00, // No subclass code
   0x00, // No protocol code
   0x00 // No description string
}
```

4.4.7 Data IN & OUT Endpoint Descriptors

The Data Class Interface requires two additional endpoints, so the corresponding **Endpoint Descriptors** must follow. It was decided previously that those endpoints would be *Bulk* IN/OUT, since it is more appropriate for this particular application.

Since addresses 00h and 03h are already taken by the default Control endpoint 0 and the Interrupt IN notification endpoint (respectively), the data OUT and IN endpoints will take addresses 01h and 02h.

Here are the two descriptors:

```
// Bulk-OUT endpoint descriptor
{
   sizeof(S_usb_endpoint_descriptor), // Size of this descriptor
   USB_ENDPOINT_DESCRIPTOR, // ENDPOINT descriptor type
   USB_ENDPOINT_OUT | SER_DATA_OUT, // OUT endpoint, address = 0x01
   ENDPOINT_TYPE_BULK, // Bulk endpoint
   64, // Endpoint size is 64 bytes
   0x00 // Must be 0x00 for full-speed bulk endpoints
}
// Bulk-IN endpoint descriptor
{
   sizeof(S_usb_endpoint_descriptor), // Size of this descriptor
   USB_ENDPOINT_DESCRIPTOR, // ENDPOINT descriptor type
   USB_ENDPOINT_IN | SER_DATA_IN, // IN endpoint, address = 0x02
   ENDPOINT_TYPE_BULK, // Bulk endpoint
   64, // Endpoint size is 64 bytes
   0x00 // Must be 0x00 for full-speed bulk endpoints
}
```

4.4.8 String Descriptors

Several descriptors (device, configuration, interface, etc.) can specify the index of a string descriptor to comment their use. These strings are completely user-defined and have no impact on the actual choice of driver made by the OS for the device.

4.5 Class-specific Requests

The CDC specification defines a set of **class-specific requests** for devices implementing the Abstract Control Model. This section details those requests, including their uses and implementation. Please refer to section 3.6.2.1 of the *CDC specification 1.1* for more information about the Abstract Control Model Serial Emulation and the associated requests and notifications.

4.5.1 SendEncapsulatedCommand, GetEncapsulatedResponse

4.5.1.1 Purpose

These two requests are used when a particular control protocol is used with the communication class interface. This is not the case for a virtual COM port, so they do not have to be implemented, even though they are supposed to be mandatory. In practice, they should never be received.

4.5.2 SetCommFeature, GetCommFeature, ClearCommFeature

4.5.2.1 Purpose

The **Set/Get/ClearCommFeature** requests are used to modify several attributes of the communication.

The first attribute is the currently used **Country Code**. Some devices perform differently, or have different legal restrictions depending on the country in which they operate. Therefore, a country code is necessary to identify the corresponding parameters. However, this is useless for a USB to serial converter, since it does not connect to a national or country-dependent network.

The **Abstract State** of the device can also be modified through the use of those requests. The first feature which can be altered is whether or not calls are multiplexed over the data class interface. Since it has already been specified in the call management functional descriptor (see **Section 4.4.4.2 on page 7**) that this is not supported in this example, it is not meaningful.

The **Idle** state of the device can be toggled using this request. When in idle state, a device shall not accept or send data to and from the host.

4.5.2.2 Implementation

When an incoming SETUP request is received, the request handler should first check whether it is class-specific or standard, and then look at its *bRequest* field.

If it is a **GET_COMM_FEATURE** request, then the handler should simply call *USB_Write* on endpoint 0 to send the two bytes of data expected by the host. Remember that the data buffer used for the *USB_Write*/*USB_Read* methods must be persistant, since they are asynchronous. The *S_std_class* structure has a *wData* attribute which can be used as such.

For **SET_COMM_FEATURE**, the handler function must call the *USB_Read* to retrieve the two bytes sent by the host. For the USB to serial converter, only the **Idle** setting is relevant, so the driver only stores this piece of information.

Since the data must be processed once the read operation is complete, a callback must be provided to *USB_Read*. This callback must have access to the received data, so it needs a pointer

to the data buffer; once again, the *wData* member of the *S_std_class* structure can be used to that end.

Finally, there is no need to read data when receiving a **CLEAR_COMM_FEATURE** request; the only action to perform is resetting the specified feature to its default state.

Remember that when the Idle state is active, the device should neither accept nor send data from or to the host. This will have to be taken into account when implementing the main program.

4.5.3 SetLineCoding, GetLineCoding

4.5.3.1 Purpose

These two requests are sent by the host to either modify or retrieve the configuration of the serial line, which includes:

- Baudrate
- Number of stop bits
- Parity check
- Number of data bits

When the terminal application (such as *HyperTerminal*) on the host (PC) side changes the setting of the COM port, a *SetLineCoding* request is sent with the new parameters. The host may also retrieve the current setting using *GetLineCoding*, not modifying them if they are correct.

4.5.3.2 Implementation

When a **SET_LINE_CODING** request is received, the device should first read the new parameters, which are held in a 7-byte structure described in the *CDC specification 1.1*, section 6.2.13. The device must then program the new parameters in the USART. Similarly to the *SetCommFeature* request, a callback must be provided to the *USB_Read* function (see Section 4.5.2.2 on page 10).

The code handling **GET_LINE_CODING** shall simply invoke the *USB_Write* function to send the current settings of the USART to the host.

There are two possible options for storing the current settings. The most obvious one is to store and retrieve them directly from the USART. This has the advantage of saving memory. But, since the parameters are unlikely to be stored in the same way as the CDC-defined structure, they will have to be parsed for each GET_LINE_CODING request. Another option is to store the received values in a dedicated member structure of the class driver, for easy access.

4.5.4 SetControlLineState

4.5.4.1 Purpose

This request is sent by the host to notify the device of two state changes. The first bit (D0) of the *wValue* field of the request indicates whether or not a terminal is connected to the virtual COM port. Bit D1 indicates that the USART should enable/disable its carrier signal to start/stop receiving and transmitting data.

In practice, the USB to serial converter should operate only when those two bits are set. Otherwise, it should not transmit or receive data.

4.5.4.2 *Implementation*

Since the SET_CONTROL_LINE_STATE request does not have a data payload, the device only has to acknowledge the request by sending a ZLP (zero-length packet), using the *USB_SendZLP0* method.

Before that, the *wValue* field should be parsed to retrieve the new control line state. A single boolean variable can be used to keep track of the connection state. If both the D0 and D1 bits are set, then the converter should operate normally, i.e., forward data between the USART and the USB host. Otherwise, it should stop its activity.

4.5.5 SendBreak

4.5.5.1 *Purpose*

The **SendBreak** request is used to instruct the device to transmit a break of the specified length on the RS-232 line. This signal is sometimes used to get the attention of the connected machine.

4.5.5.2 *Implementation*

The USART peripheral of AT91 chips supports the transmission of a break signal. Two bits in the USART control register are used to start and stop the break. In order to comply with the length specified in the *wValue* field of the request, a timer can be used.

When the SEND_BREAK request is received, the device should set the corresponding bit of the USART to start transmitting the break. It should also start the timer as well. The break should be stopped once the timer expires; this can be done by using an interrupt.

Do not forget to acknowledge the request (by sending a ZLP) just after starting the break condition (and not after the break sequence finishes).

4.6 Notifications

Notifications are sent by the device when an event, such as a serial line state change, has occurred. In this example, they are transmitted through a dedicated Interrupt IN endpoint. A special header must precede the data payload of each notification. This header has the same format of a SETUP request, so the *S_usb_request* structure defined in the AT91 USB framework can be used.

Note that the device should only send a notification when there is a state change, and not continuously. This does not really matter in practice, but only sending notifications sporadically will reduce the stress on the device.

4.6.1 NetworkConnection

4.6.1.1 *Purpose*

The *NetworkConnection* notification is used to tell the host whether the USB to serial converter is connected on its RS-232 side. In the case of a USART, there is no way get this information, unless coupled with an extra signal. Therefore, this notification is not supported by the USB to serial converter.

4.6.2 ResponseAvailable

4.6.2.1 Purpose

This notification allows the device to tell the host that a response is available. However, since it has already been mentioned that the *GetEncapsulatedResponse* request is not relevant to this case study (see Section 4.5.1.1 on page 10), this notification is useless.

4.6.3 SerialState

4.6.3.1 Purpose

This command acts as an interrupt register for the serial line. It notifies the host that the state of the device has changed, or that an event has occured. The following events are supported:

- Buffer overrun
- Parity error
- Framing error
- Ring signal detection mechanism state
- Break detection mechanism state
- Transmission carrier state
- Receiver carrier detection mechanism state

Several of these values are only used for modem device, namely ring signal detection, transmission carrier state and receiver carrier detection.

4.6.3.2 Implementation

This notification can be directly tied with the status register of the USART. Indeed, the latter contains all the required information. An interrupt can be used to notify the device when the USART state changes; the corresponding notification can then be sent to the host.

To send a SERIAL_STATE notification, the device should first transmit the corresponding notification header. As noted previously, it has a format identical to a SETUP request, so a *S_usb_request* instance can be used to store the necessary information. In the following example, a typedef has been defined to rename *S_usb_request* to *S_cdc_notification_header*.

```
S_cdc_notification_header sHeader = {

    CDC_NOTIFICATION_TYPE,
    CDC_NOTIFICATION_SERIAL_STATE,
    0,
    0,
    0
};
```

The actual state information is transmitted in two bytes. Only the first 7 bits of the first byte are significant; the others can be set to 0. The device should set the bits corresponding to the current USART state, and then send the data using *USB_Write*.

Depending on the size of the interrupt endpoint, the header and data will have either have to be transmitted in one chunk (size superior to 8 bytes), or separately. In the first case, the simplest is probably to define a *S_cdc_serial_state* structure to hold both the header and the data payload.

In the second case, the transmission can be done by two consecutive *USB_Write* calls (since the header is 8 bytes long); however, this may not be convenient, as the first transfer must be finished before the second one can start. This means that the second call must either be done in a callback function (invoked upon the first transfer completion), or in a loop verifying that the returned result code is *USB_STATUS_SUCCESS* (indicating that the endpoint is not locked anymore).

4.7 Main Application

The job of the main application is to **bridge** the USART and the USB. This means that data read from one end must be forwarded to the other end. This section describes several possibilities to do this.

4.7.1 USB Operation

Reading data coming from the host is done using the *USB_Read* function on the correct endpoint. Since this is an asynchronous function, it does not block the execution flow. This means that other actions (like reading data from the USART) can be performed while the transfer is going on. Whenever some data is sent by the host, the transfer terminates and the associated callback function is invoked. This callback can be programmed to forward the received data through the USART.

Likewise, the *USB_Write* function can be called as soon as there is data to transmit, again without block the program flow. However, there cannot be two write operations at the same time, so the program must check whether or not the last transfer is complete. This can be done by checking the result code of the *USB_Write* method. If USB_STATUS_LOCKED is returned, then there is already another operation in progress. The device will have to buffer the data retrieved from the USART until the endpoint becomes free again.

4.7.2 USART Operation

The USART peripheral present on AT91 chips can be used in two different ways. The classic way is to read and write one byte at a time in the correct registers to send and receive data.

A more powerful method is available on AT91SAM chips, by using the embedded Peripheral DMA Controller (PDC). The PDC can take care of transfers between the processor, memory and peripherals, thus freeing the processor to perform other tasks.

Since the focus of this application note is on the USB component, the USART usage will not be described further here.

4.8 Example Software Usage

4.8.1 File Architecture

In the example program provided with this application note, the actual driver is divided into three files:

- **cdc.h**: header file with generic CDC definitions
- **cdc_serial_driver.h**: header file with definitions for the USB to serial converter driver
- **cdc_serial_driver.c**: source file for the USB to serial converter driver

Having all generic CDC definitions in a separate file makes it possible to easily reuse it for other CDC-related drivers.

The main application, which uses the driver to bridge the USART and USB interfaces, is implemented in the **cdc_serial_main.c** file.

4.8.2 Compilation

The software is provided with a **Makefile** to build it. It requires the *nmake* utility, which is available in the Microsoft® Windows Driver Development Kit (DDK). Please refer to the *AT91 USB Framework* application note for more information on general options and parameters of the Makefile.

To build the USB to serial converter example, two options must be specified. The CLASS parameters must be set to “CDC”, and MODE must be equal to “SERIAL”:

```
nmake TARGET=AT91SAM7S64 BOARD=AT91SAM7SEK CLASS=CDC MODE=SERIAL
```

In this case, the resulting binary will be named *AT91SAM7SEK_CDC_SERIAL.bin* and will be located in the *bin/AT91SAM7S64/* directory.

4.9 Using a Generic Host Driver

Both Microsoft Windows and Linux offer a generic driver for using a USB to serial converter device. This section details the steps required to make use of them.

4.9.1 Windows

On Microsoft Windows, the standard USB serial driver is named **usbser.sys** and is part of the standard set of drivers. It has been available since Windows 98SE. However, conversely to other generic driver such as the one for Mass Storage Devices (MSD), *usbser.sys* is not automatically loaded when a CDC device is plugged in.

4.9.1.1 Writing a Windows Driver File

For Windows to recognize the device correctly, it is necessary to write a *.inf* file. The Windows **Driver Development Kit** (DDK) contains information on this topic. A basic driver, named *6119.inf* in the example software provided, will now be described. The driver file is made up of several sections.

The first section of the *.inf* file must be the **[Version]** section. It contains information about the driver version, provider, release data, and so on.

```
[Version]
Signature="$Chicago$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%ATMEL%
DriverVer=09/12/2006,1.1.1.1
```

The *Signature* attribute is mandatory and can be either “$Windows 95$”, “$Windows NT$” or “$Chicago$”, depending on which Windows version(s) the driver supports. “$Chicago$” is used to notify that every Windows version is supported. Since in this example, the USB to serial converter is a virtual COM port, the *Class* attribute should be equal to “Ports”. The value of *ClassGuid* depends on which class the device uses. The *Provider* value indicates that the string descriptor for the driver provider will be defined further, under the tag ATMEL. Finally, the last tag show the driver version and release date. For the version number, each digit is optional (except the first one), but must not be null if present.

Next come two sections, **[SourceDisksNames]** and **[SourceDisksFiles]**. They are used to specify the installation disks required and the location of each needed files on these disks.

```
[SourceDisksNames]
1="Windows Install CD"

[SourceDisksFiles]
usbser.sys=1
```

The first one lists the names of source disks on which the user can find missing files. Since the driver requires *usbser.sys*, present on the Windows install CD, it will have to be listed here. The disk ID must be a unique and non-null digit. The second section indicates on which disk each file can be found. In this case, *usbser.sys* can be found on disk #1 (which is “Windows Install CD”). Optionally, the exact path of the file on the CD can be specified.

The driver file must now specify where copied files will be stored, using the **[DestinationDirs]** section.

```
[DestinationDirs]
DefaultDestDir=12
```

The target directory must be identified by its ID, which is system-defined. The ID for the *drivers* directory is 12.

The **[Manufacturer]** section lists the possible manufacturers for all devices supported by this driver. In this case, the only supported device is an ATMEL one, so this will be the only value.

```
[Manufacturer]
%ATMEL%=AtmelMfg
```

The attribute must be a string tag; its value must be the name of the *Models* section in which all supported devices from this manufacturer will be listed. In this case, it will be named *AtmelMfg*, which is the next section.

Each *Models* section must list the hardware ID of each supported device. For USB devices, the hardware ID is made up of the Vendor ID, the Product ID and (optionally) the Device Release Number. Those values are extracted from the device descriptor provided during the enumeration phase.

```
[AtmelMfg]
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_6119
```

The attribute name is again a string tag, which will be used to describe the device. The value is comprised of both the device install section name (USBtoSer.Install) and the hardware ID. The hardware ID is the same as the one specified in Section 4.4.1 on page 5.

Now, the *.inf* file must detail the install section of each device previously listed. In this example, there is only one install section, named *USBtoSer.Install*:

```
[USBtoSer.Install]
CopyFiles=USBtoSer.CopyFiles
AddReg=USBtoSer.AddReg

[USBtoSer.CopyFiles]
usbser.sys,,,0x00000002

[USBtoSer.AddReg
```

```
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,usbser.sys

[USBtoSer.Install.Services]
AddService=usbser,0x00000002,USBtoSer.AddService

[USBtoSer.AddService]
DisplayName=%USBSer%
ServiceType=1r
StartType=3
ServiceBinary=%12%\usbser.sys
```

The install section is actually divided in five. In the first section, two other section names are specified: one for the list of files to copy, and one for the keys to add to the Windows registry. There is only one file to copy, usbser.sys; a flag (0x00000002) is used to specify that the user cannot skip copying it. The registry keys are needed to install the driver on older versions of Windows (such as Windows 98). For newer versions, the **[USBtoSer.Install.Services]** registers the needed kernel services; each service is actually listed in a section on its own.

Finally, the last section, [Strings], defines all the string constants used through this file:

```
[Strings]
ATMEL="ATMEL Corp."
USBtoSerialConverter="AT91 USB to Serial Converter"
USBSer="USB Serial Driver"
```

4.9.1.2 Using the Driver

When a new device is plugged in for the first time, Windows looks for an appropriate specific or generic driver to use it. If it does not find one, the user is asked what to do.

This is the case with the USB to serial converter, since there is no generic driver for it. To install the custom driver given in the previous section, Windows must be told where to look for it. This can be done by selecting the second option, “Install from a list or specific location”, when the driver installation wizards pops up. It will then ask for the directory where the driver is located. After that, it should recognize the “AT91 USB to Serial Converter” driver as an appropriate one and display it in the list.

During the installation, the wizard asks for the location of the *usbser.sys* file. If it is already installed on the system, it can be found in “*C:\Windows\System32\Drivers*”. Otherwise, it is present on the Windows installation CD.

Once the driver is installed properly, a new COM port is added to the system and can be used with *HyperTerminal*, for example.

4.9.2 Linux

Linux has two different generic drivers which are appropriate for a USB to serial converter. The first one is an Abstract Control Model driver designed for modem devices, and is simply named **acm**. The other one is a generic USB to serial driver named **usbserial**.

If the support for the *acm* driver has been compiled in the kernel, Linux will automatically load it. A new terminal device will be created under /dev/ttyACMx.

The *usbserial* driver must be loaded manually by using the modprobe command with the vendor ID and product ID values used by the device:

```
modprobe usbserial vendor=0x03EB product=0x6119
```

Once the driver is loaded, a new terminal entry appears and should be named /dev/ttyUSBx.

5. Revision History

Table 5-1.

Document Ref.	Comments	Change Request Ref.
6269A	First issue.	

Atmel Corporation

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

Regional Headquarters

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

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

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High-Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80

Literature Requests
www.atmel.com/literature