AT91 USB Integrated Circuit(s) Cards Interface Devices (CCID) Driver Implementation

1. Introduction

The Integrated Circuit(s) Cards Interface Devices (CCID) class extends the USB specification in order to provide a standard means of handling Integrated Circuit(s) Card (ICC) devices such as Smart Cards conforming to ISO/IEC 7816 specifications.

This application note describes how to implement a CCID driver with the **AT91 USB Framework** provided by Atmel® for use with its AT91 ARM® Thumb® based microcontrollers.

- First, generic information about CCID-specific definitions and requirements is given.
- This document then details how to use the CCID class to communicate with a Smart Card.

2. Related Documents

1. CCID Device Class: Smart Card, specification for Integrated Circuit(s) Cards Interface Devices, revision 1.1, April 22, 2005.
2. Identification cards, Integrated circuits cards, part 3: Cards with contacts: electrical interface and transmission protocols, Version 2.1c2, 2005-06-3. ISO/IEC FDIS 7816-3: 2005(E)
3. Identification cards, Integrated circuits cards, part 4: Organization, security and commands for interchange, 2004-10-14. ISO/IEC FDIS 7816-4: 2004(E)
4. Atmel Corp., AT91 USB Framework, 2006, lit ° 6263
5. USB specification 2.0: http://www.usb.org
6. Software ISO 7816 I/O Line Implementation, Atmel Application Note, lit° 1154
7. DWG Smart-Card Integrated Circuit(s) Card Interface Devices http://www.usb.org

AT91 ARM Thumb Microcontrollers

Application Note

6348A–ATARM–06-Jul-09

3. Abbreviations and Terms

APDU	Application Protocol Data Unit
ATR	Answer to Reset
CCID	Integrated Circuit(s) Cards Interface Devices
Cold RESET	The sequence starts with the ICC powered off.
ICC	Integrated Circuit(s) Card (Used interchangeably with Smart Card)
ICCD	Integrated Circuit(s) Card Devices conforming to this specification.Used interchangeably with USB-ICC.
Interface Device	Terminal communication device or machine to which the ICC is electrically connected during operation [ISO/IEC 7816-3].
Lc	Optional part of the body of a command APDU. Its size is 0, 1 or 3 bytes. The maximum number of bytes present in this body.
Le	Optional part of the body of a command APDU. Its size is 0, 1, 2, or 3 bytes. The maximum number of bytes expected in the data field of the response APDU.
P1, P2	INS parameter of a command header.
P3	INS parameter of a command header. P3 contains Lc or Le
RFU	Reserved for Future Use
TPDU	Transport Protocol Data Unit
USB-ICC	USB Integrated Circuit(s) Card. An ICC providing a USB interface [ISO/IEC 7816-12]. Used interchangeably with ICCD.
Warm RESET	The sequence starts with the ICC already powered

4. CCID Device Class

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

4.1 Purpose

The CCID class has been specifically designed for **Smart Card Devices**.

4.2 Architecture

4.2.1 Interfaces

A CCID device only needs one interface descriptor. It should have the CCID interface class code in its *bInterfaceClass* field. There are special subclass and protocol codes to specify.

4.2.2 Endpoints

The CCID requires three endpoints.

One endpoint BULK IN and one endpoint BULK OUT. They are mandatory and should always be declared.

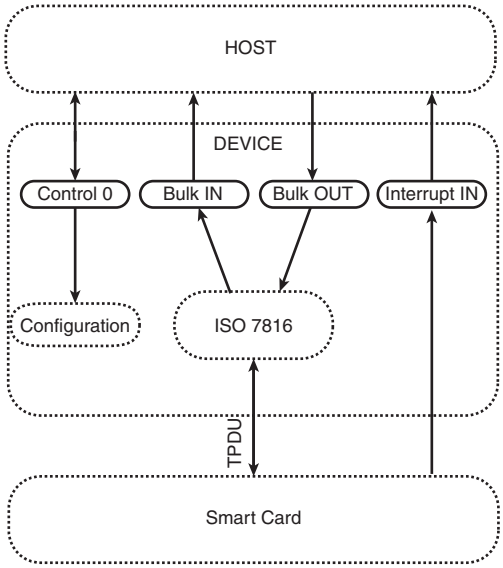
The interrupt pipe is mandatory for a CCID that supports ICC insertion/removal. It is optional for a CCID with ICCs that are always inserted and are not removable.

Endpoint 0 is used for class-specific requests. In addition, the host can also explicitly request or send report data through this endpoint.

The Bulk IN and OUT endpoints are used for sending data to the host, and to receive information.

The Interrupt endpoint is used for Insertion and Removal of the card, and in case of hardware error.

Figure 4-1. CCID Class Driver Architecture



4.2.3 Class-Specific Descriptors

The smart card device descriptor specifies certain device features or capabilities. Please refer to the CCID specifications for more information.

4.2.3.1 CCID Descriptor

The **CCID descriptor** gives information about the CCID 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 4-1.

Table 4-1. CCID Descriptor Format

Field	Size (bytes)	Description
bLength	1	Total length of the CCID descriptor
bDescriptorType	1	CCID descriptor type (21h)
bcdCCID	2	CCID specification release number in Binary Coded Decimal (BCD) format.
bMaxSlotIndex	1	Index of the highest available slot.
bVoltageSupport	1	What voltages the CCID can supply to its slots
dwProtocols	4	Supported protocol types

Table 4-1. CCID Descriptor Format

Field	Size (bytes)	Description
dwDefaultClock	4	Default ICC clock frequency in KHz
dwMaximumClock	4	Maximum supported ICC clock frequency in KHz
bNumClockSupported	4	Number of clock frequencies that are supported by the CCID
...		

4.2.4 Class-specific Requests

4.2.4.1 GetDescriptor

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

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

4.3 Host Drivers

Microsoft® class CCID driver is used to drive the CCID without any vendor-specific software. usbccid.sys can be found on the installation disk provided by Windows®, or can be downloaded from the Windows Update web site during the new device installation procedure.

Specific software is used to send the APDU command to the Smart Card.

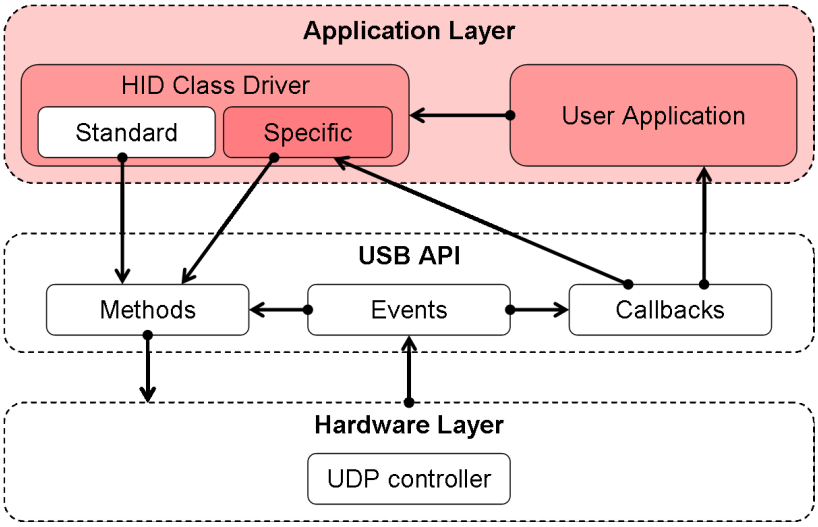
5. CCID Implementation

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

5.1 Architecture

The AT91 USB 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 5-1 shows the application architecture.

Figure 5-1. Application Architecture Using the AT91 USB Framework



5.2 Descriptors

5.2.1 Device Descriptor

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

```
static const USBDeviceDescriptor deviceDescriptor = {

    sizeof(USBDeviceDescriptor),
    USBGenericDescriptor_DEVICE,
    USBDeviceDescriptor_USB2_00,
    0,
    0,
    0,
    BOARD_USB_ENDPOINTS_MAXPACKETSIZE(0),
    CCIDDriverDescriptors_VENDORID,
    CCIDDriverDescriptors_PRODUCTID,
    CCIDDriverDescriptors_RELEASE,
    1, // Index of manufacturer description
    2, // Index of product description
    3, // Index of serial number description
    1  // 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.

5.2.2 Configuration Descriptor

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

```
{
    sizeof(USBConfigurationDescriptor),
    USBGenericDescriptor_CONFIGURATION,
    sizeof(CCIDDriverConfigurationDescriptors),
    1, // One interface in this configuration
    1, // This is configuration #1
    0, // No associated string descriptor
    BOARD_USB_BMATTRIBUTES,
    USBConfigurationDescriptor_POWER(100)
},
```

When the Configuration descriptor is requested by the host (by using the GET_DESCRIPTOR command), the device must also send 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 CCIDDriverConfigurationDescriptors structure has been declared for that purpose.

5.2.3 CCID Class Interface Descriptor

The interface descriptor is for the CCID Class Interface. It should specify the Smart Card Class code (0Bh).

A CCID device needs to send and receive data from the host. This means the CCID needs bulk IN, bulk OUT and Interrupt IN endpoints. So the *bNumEndpoints* field will have to be set to 3. This interface also uses the default Control endpoint, but this is not taken into account here.

Here is the whole Interface descriptor:

```
{
        sizeof(USBInterfaceDescriptor),
        USBGenericDescriptor_INTERFACE,
        0,                      // Interface 0
        0,                      // No alternate settings
        3,                      // uses bulk-IN, bulk-OUT and interrupt-IN
        SMART_CARD_DEVICE_CLASS,
        0,                      // Subclass code
        0,                      // bulk transfers optional interrupt-IN
        0                       // No associated string descriptor
},
```

5.2.4 CCID Descriptor

An Interface descriptor is followed by the CCID descriptor. The CCID descriptor gives information about the number and types of the other defined descriptors.

Example software for the Smart Card.

```
{
       sizeof(CCIDDescriptor), // bLength: Size of this descriptor in bytes
        CCID_DECRIPTOR_TYPE,    // bDescriptorType:Functional descriptor
type
```

```
        CCID1_10,               // bcdCCID: CCID version
        0,                  // bMaxSlotIndex: Value 0 indicates that one slot
is supported
        VOLTS_5_0,          // bVoltageSupport
        PROTOCOL_TO,        // dwProtocols
        3580,               // dwDefaultClock
        3580,               // dwMaxClock
        0,                  // bNumClockSupported
        9600,               // dwDataRate : 9600 bauds
        9600,               // dwMaxDataRate : 9600 bauds
        0,                  // bNumDataRatesSupported
        0xfe,               // dwMaxIFSD
        0,                  // dwSynchProtocols
        0,                  // dwMechanical
        //0x00010042,       // dwFeatures: Short APDU level exchanges
        CCID_FEATURES_AUTO_PCONF | CCID_FEATURES_AUTO_PNEGO |
CCID_FEATURES_EXC_TPDU,
       0x0000010F,      // dwMaxCCIDMessageLength: For extended APDU level
the value shall be between 261 + 10
        0xFF,               // bClassGetResponse: Echoes the class of the APDU
        0xFF,               // bClassEnvelope: Echoes the class of the APDU
        0,                  // wLcdLayout: no LCD
        0,                  // bPINSupport: No PIN
        1                   // bMaxCCIDBusySlot
    },
```

5.2.5 Physical Descriptor

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

5.2.6 Endpoint Descriptor

Since it has been specified that the CCID interface uses 3 endpoints, corresponding endpoint descriptors must now be defined. As mentioned previously, there are bulk OUT, bulk IN and Interrupt IN endpoints.

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

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 CCID device response latency is not extremely critical, it can be safely set to a high value. In this example, the endpoint is polled every 16 ms.

```
// Bulk-OUT endpoint descriptor
{
        sizeof(USBEndpointDescriptor),
```

```
        USBGenericDescriptor_ENDPOINT,
        USBEndpointDescriptor_ADDRESS( USBEndpointDescriptor_OUT,
CCID_EPT_DATA_OUT ),
        USBEndpointDescriptor_BULK,
        MIN(BOARD_USB_ENDPOINTS_MAXPACKETSIZE(CCID_EPT_DATA_OUT),
            USBEndpointDescriptor_MAXBULKSIZE_FS),
        0x00                              // Does not apply to Bulk endpoints
    },
// Bulk-IN endpoint descriptor
    {
        sizeof(USBEndpointDescriptor),
        USBGenericDescriptor_ENDPOINT,
        USBEndpointDescriptor_ADDRESS( USBEndpointDescriptor_IN,
CCID_EPT_DATA_IN ),
        USBEndpointDescriptor_BULK,
        MIN(BOARD_USB_ENDPOINTS_MAXPACKETSIZE(CCID_EPT_DATA_IN),
            USBEndpointDescriptor_MAXBULKSIZE_FS),
        0x00                              // Does not apply to Bulk endpoints
    },
    // Notification endpoint descriptor
    {
        sizeof(USBEndpointDescriptor),
        USBGenericDescriptor_ENDPOINT,
        USBEndpointDescriptor_ADDRESS( USBEndpointDescriptor_IN,
CCID_EPT_NOTIFICATION ),
        USBEndpointDescriptor_INTERRUPT,
        MIN(BOARD_USB_ENDPOINTS_MAXPACKETSIZE(CCID_EPT_NOTIFICATION),
            USBEndpointDescriptor_MAXINTERRUPTSIZE_FS),
        0x10
    }
```

5.2.7 String Descriptors

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

5.3 Class-Specific Requests

A number of CCID-only requests are defined in the corresponding specification. They have already been described in Section 4.2.4 on page 4. This section details their implementation regarding the current example of a CCID 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.

5.3.1 Get Descriptor

Three values have been added by the CCID 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 CCID specification adds the **ABORT** (01h), **GET_CLOCK_FREQUENCY** (02h) and **GET_DATA_RATES** (03h) types.

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 CCID descriptors). The solution is to first identify GET/SET_DESCRIPTOR requests, treat the CCID-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.

5.3.2 Set Descriptor

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

5.4 ISO7816

The ISO7816 software provided in this example is used to transform APDU commands to TPDU commands for the smart card.

The ISO7816 implemented is for the protocol T = 0 only.

The send and the receive of a character is made under polling.

In the ISO7816_Init file 3 pins of the card are defined. The user must change these pins according to the specific environment.

See paragraph "*ISO7816 Mode Overview"* in the corresponding Atmel product datasheet.

3 or 4 PIO pins are used, see Section 6.1 for the Pin connections:

PIN_ISO7816_RSTMC: for 7816_RST

PINS_ISO7816: which defines USART pin, CLOCK pin and the RSTMC pin

PIN_SMARTCARD_CONNECT: Smartcard detection pin (if present)

The driver is compliant with cases 1, 2, 3 of the ISO7816-4 specification.

5.4.1 USART Configuration

First, configure the USART in mode 7816, mode T=0, 1 stop bit, 8 chars, parity even.

Refer to the corresponding Atmel product datasheet, paragraph "*Baud Rate in ISO7816 Mode"* for more explanations.

The ISO7816 specification defines the bit rate with the following formula:

B = (Di / Fi) x f

where:

- B is the bit rate
- Di is the bit-rate adjustment factor

- Fi is the clock frequency division factor
- f is the ISO7816 clock frequency (Hz)

We use the most common: (Fi = 372, Di = 1).

The USART is programmed to operate in synchronous mode, so the selected clock is simply divided by the field CD in US_BRGR.

BaudRate = SelectedClock / CD

So, CD = SelectedClock / BaudRate

and BaudRate = FiDi x Baud= 372 x 9600 (in our case)

CD = 48MHz / (372 x 9600) = 13, to be programmed in US_BRGR

We use a transmitter timeguard of 5.

5.4.2 Cold Reset

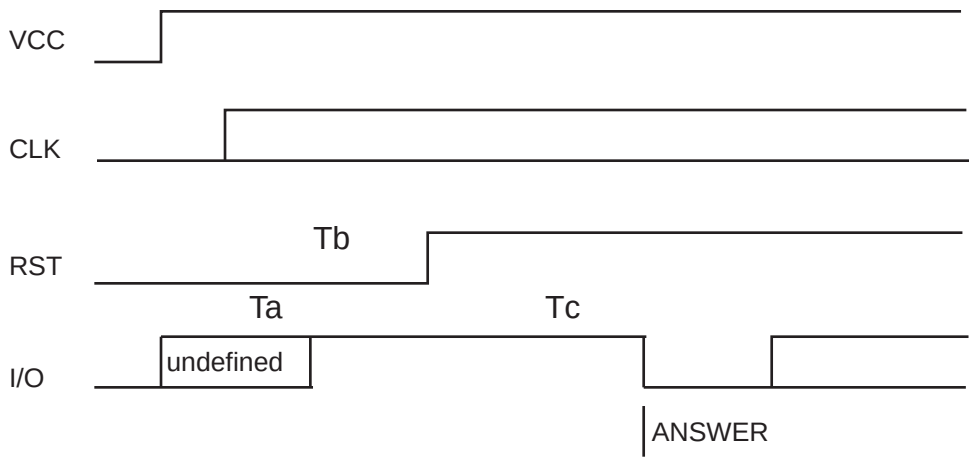
Activation by cold reset (see Reference [2]: ISO7816-3) needs a timer. For the cold reset, the device must wait a minimum of 400 smart card clock cycles.

It is needed to receive the ATR. See Reference [2]: ISO7816-3 paragraph 5.4.2.

The programmer needs to program the correct value in the US_BRGR (Baud Rate Generator Register) and US_FIDI (FI_DI_Ratio Register). The timeguard register is the US_TTGR (Transmitter Time-guard Register).

In order to initiate an interaction with a mechanically connected card, the interface device activates the electrical circuits according to a class of operating conditions. See: Reference [2]: ISO7816-3 paragraph 5.2.1 and 5.2.2 for more details.

Figure 5-2. Cold Reset



Ta = 200 / f

Tb = 400 / f

400 / f < Tc < 40 000 / f

There are two different modes for the smart card after a cold receive or a warm reset.

After a Cold Reset, the Smart Card answers by the “Answer to Reset”, and then goes in a specific mode or a negotiable mode. The only way to leave theses modes is to make a Warm Reset.

More details can be found in the ISO7816-3 specification.

5.4.3 Send and Receive Character

5.4.3.1 Send a character to the Smart Card

The function ISO7816_SendChar is used to send a character to the Smart Card.

The function waits for the USART to be ready to transmit and then transmits the character on the USART.

5.4.3.2 Receive a char from the Smart Card

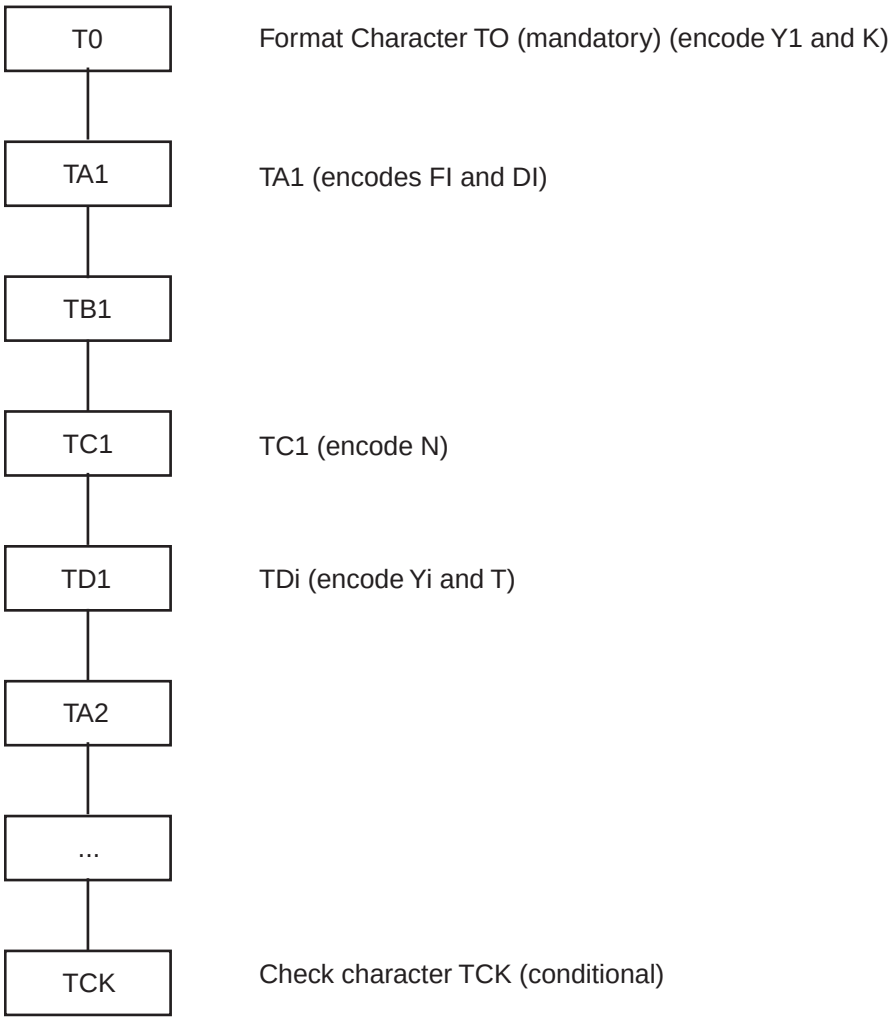
The function ISO7816_GetCharis used for receive a character from the Smart Card.

The function waits for the USART to be ready to receive, and then reads the character from the USART.

5.4.4 Answer To Reset

The card answers to any reset and the information exchange begins with the answer to the cold reset.

Figure 5-3. Answer To Reset (ATR):



Useful bits:

K : encode the number of historical bytes,

Y1: each bit set to 1 indicates the presence of a further interface byte

TA1 encodes:

- FI, the reference to a clock rate conversion factor over bits 8 to 5
- DI, the reference to a baud rate adjustment factor over bits 4 to 1

Useful for read and change the clock and the baud rate of the device according to the smart card possibility.
TC1 encodes N, the reference to compute the extra guard time over the eight bits.
TD1: If TD_i is present, then TA_{i+1}, TB_{i+1}, TC_{i+1} and TD_{i+1} are also present.
TCK: Check byte TCK. If only T=0 is indicated, possibly by default, then TCK is absent

5.4.5 APDU Commands

The APDU protocol, as specified in ISO 7816-4, is an application-level protocol between a smart card and a host application.

There are four structures of APDU commands:

- A command APDU in case 1 consists of one field: a header.
- A command APDU in case 2 consists of two consecutive fields: a header and a L_e field.
- A command APDU in case 3 consists of three consecutive fields: a header, a L_c field and a data field.
- A command APDU in case 4 consists of four consecutive fields: a header, a L_c field, a data field and a L_e field. A case 4 is a case 3 followed by a case 2.

5.4.6 APDU and TPDU Commands

APDU: Application Protocol Data Units

TPDU: Transmission Protocol Data Units

APDU are transmitted by the next-level protocol (the transport protocol) defined in ISO 7816-3. The data structures exchanged by a host and a card using the transport protocol are called transport protocol data units (TPDU).

The two transport protocols that are in primary use in smart card systems are the T=0 protocol and the T=1 protocol. The T=0 protocol is byte-oriented, which means that the smallest unit processed and transmitted by the protocol is a single byte. In this exemple, only T=0 protocol is used.

The different cases that follow correspond to different APDU and TPDU commands.

5.4.6.1 Case 1:

No data is transferred to or from the card

Command APDU: CLA INS P1 P2

Command TPDU: CLA INS P1 P2 {P3 set to '00'}

Response TPDU: SW1 SW2

5.4.6.2 Case 2

No data is transferred to the card, but data is returned from the card

Command APDU: CLA INS P1 P2 {L_e field = C(5)}

Command TPDU: CLA INS P1 P2 {P3 = C(5)}

Response: TPDU: N_a data bytes SW1 SW2

5.4.6.3 Case 3

Data is transferred to the card, but no data is returned from the card as a result of processing the command

Command APDU: CLA INS P1 P2 {L_c field = C(5)} N_c data bytes

Command TPDU: CLA INS P1 P2 {P3 = C(5)} N_c data bytes

Response TPDU: SW1 SW2

More explanation can be found in the specification of the ISO 7816-3.

All cases are treated in ISO7816_XfrBlockTPDU_T0 function.

5.5 Main Application

The main function of the application has to perform two actions: Enumeration and the APDU transfer command.

5.6 Example Software Usage

5.6.1 File Architecture

The software example associated with this application note is divided into six files:

- **cciddriver.c**: source file for the CCIDdriver
- **cciddriver.h**: header file with generic CCID definitions
- **cciddriverdescriptors.h**: header file for the CCID device descriptor
- **Iso7816_4.h**: header file for the ISO 816-4
- **Iso7816_4.c**: source file for the ISO7816 commands

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. 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 CCID example just run “make” in directory **usb-device-ccid-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-ccid-project-at91sam7se-ek-at91sam7se512-flash.bin* and will be located in the *usb-device-ccid-project/bin* directory.

5.7 Host-side Application

This section explains how to program a PC application to communicate with the custom CCID device driver described previously. This example is targeted at a Microsoft Windows platform.

On Microsoft Windows, the standard USB CCID driver is named usbccid.sys.

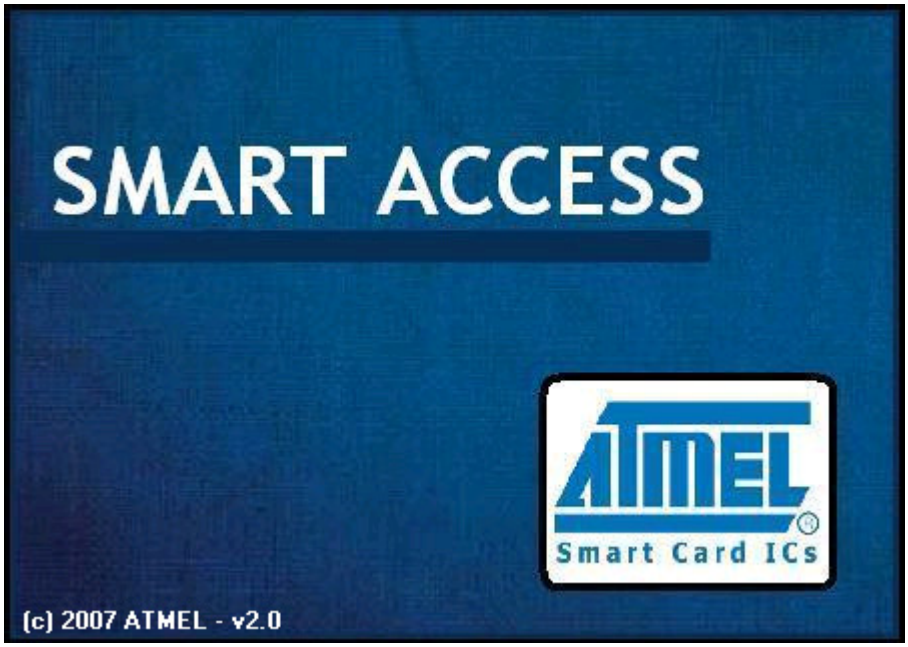
5.7.1 Using the Driver

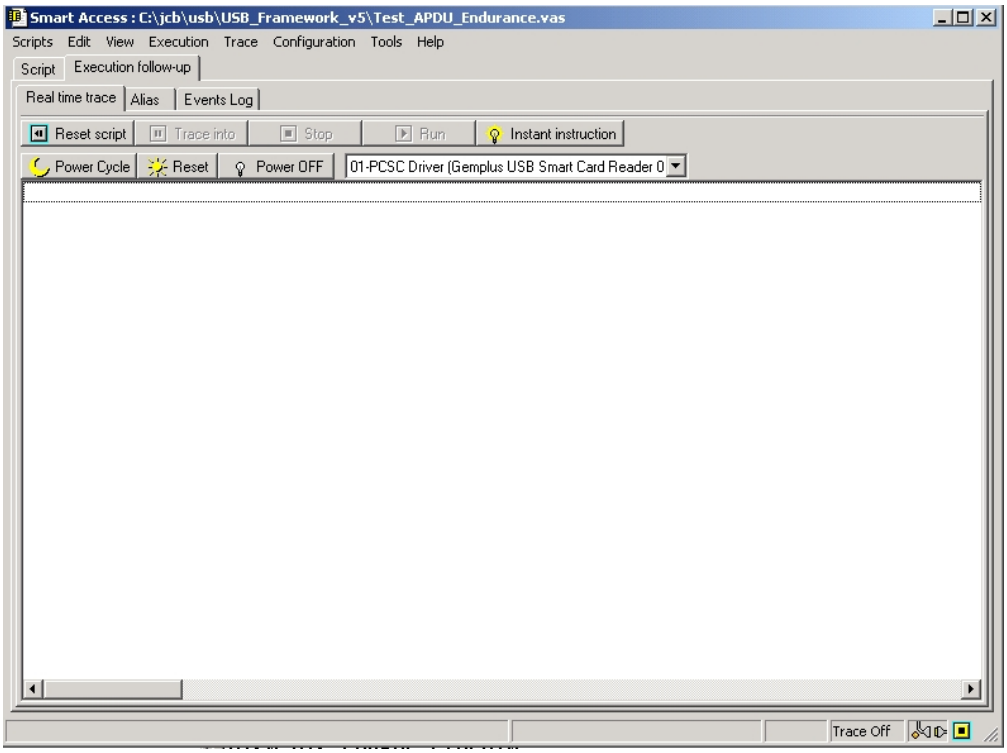
When a new device is plugged in for the first time, Windows looks for an appropriate specific or generic driver to use. If it does not find one, the user is asked what to do.

In this application the device is enumerated as a Smart Card Device implementing CCID class. The host uses the CCID device driver (usbccid.sys) as the functionnal driver.

5.7.2 Smart Access

Smart Access is software made by Atmel. (Contact an Atmel sales representative to find out more about this product.





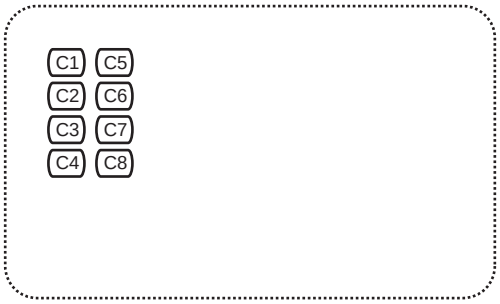
- Software Environment to Execute Smart Card ISO7816 Command Script Through Any Reader
- PC/SC or Transparent Readers Supported
- Meta-instructions Interpreter to Write Complex Validation Scripts
- Ergonomic Environment with Real-time
- Trace Window
- External Software Modules for More Flexibility

6. Hardware Requirement

On some Atmel boards, the Smart Card reader is not implemented. The user must use the RS232 in ISO mode and connect the card reader to it. The Smart Card has useful pins.

6.1 Pin Connection:

When the card is inserted into the reader, the contacts in the reader sit on the plates. According to ISO7816 standards the PIN connections are shown below:



C1: Vcc: 7816_3V5V

C5 : Gnd

C4 : RFU

C2 : Reset: 7816_RST

C6 : Vpp

C8 : RFU

C3 : Clock: 7816_CLK

C7 : 7816_IO

Another pin must be connected on the card reader for detecting insertion and removal: 7816_IRQ.

On Atmel's boards, all these pins can be easily connected with jumpers.

7816_RST is on PIO PIN_ISO7816_RSTMC

7816_CLK is on PIO PIN_USART0_SCK

7816_IO is on PIO PIN_USART0_TXD

7816_IRQ is on PIO PIN_SMARTCARD_CONNECT

Revision History

Doc. Rev	Date	Comments	Change Request Ref.
6348A	02-Jul-09	First issue	

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