AVR309: Software Universal Serial Bus (USB)

8-bit AVR® Microcontrollers

Application Note

PRELIMINARY

Features

- **USB (Universal Serial Bus) protocol implemented in firmware**
- **Supports Low Speed USB (1.5Mbit/s) in accordance with USB2.0**
- **Implementation runs on very small AVR devices, from 2kBytes and up**
- **Few external components required**
  - **One resistor for USB low speed detection**
  - **Voltage divider/regulator, with filtering**
- **Implemented functions:**
  - **Direct I/O pin control**
  - **USB to RS232 converter**
  - **EEPROM scratch register**
- **User can easily implement own functions as e.g.:**
  - **USB to TWI control**
  - **USB A/D and D/A converter**
- **Vendor customizable device identification name (visible from PC side)**
- **Full PC side support with source code and documentation**
  - **MS Windows USB drivers**
  - **DLL library functions**
  - **Demo application in Delphi**
- **Examples for developers on how to communicate with device through DLL (Delphi, C++, Visual Basic)**

Introduction

The Universal Serial Bus (USB) interface has become extremely popular, due to its simplicity for end user applications (Plug and Play without restart). For developers, however, USB implementation into end systems has been more complicated compared to e.g. RS232. In addition there is a need for device drivers as software support on the PC side. Because of this, RS232 based communication is still very popular among end systems manufacturers. This interface is well established and has good operating system support, but recently the physical RS232 port has been removed from the standard PC interface, giving ground to USB ports.

Implementation of USB into external devices can be done in two ways:

1. By using a microcontroller with a hardware implemented USB interface. It is necessary to know how USB works and write firmware into the microcontroller accordingly. Additionally, it is necessary to create a driver on the computer side, unless if the operating system includes standard USB classes. The main disadvantage is the lack of availability of this kind of microcontrollers and their high price compared to simple “RS232” microcontrollers.

Rev. 2556-AVR-07/05

2. The second option is to use some universal converter between USB and another interface. This other interface will usually be RS232, 8-bit data bus, or TWI bus. In this case there is no need for special firmware, it isn't even necessary to know how USB works, and no driver writing is required, as the converter vendor will offer one driver for the whole solution. The disadvantage is the higher price of the complete system, and the greater dimensions of the complete product.

The solution presented in this document is a USB implementation into a low-cost microcontroller through emulation of the USB protocol in the microcontroller firmware. The main challenge for this design was obtaining sufficient speed. The USB bus is quite fast: LowSpeed - 1.5Mbit/s, FullSpeed - 12Mbit/s, HighSpeed - 480Mbit/s. The AVR microcontrollers are fully capable of meeting the hard speed requirements of LowSpeed USB. The solution is however not recommended for higher USB speeds.

1 Theory of Operation

Extensive details regarding physical USB communication can be found at the website www.usb.org. This documentation is very complex and difficult for beginners. A very good and simple explanation for beginners can be found in the document "USB in a Nutshell. Making Sense of the USB Standard" written by Craig Peacock [2].

In this application note the explanation is limited in scope to understanding the device firmware. The USB physical interface consists of 4 wires: 2 for powering the external device (V_{CC} and GND), and 2 signal wires (DATA+ and DATA-). The power wires give approximately 5 volts and max. 500mA. The AVR can be supplied from the Vcc and GND. The signal wires named DATA+ and DATA- handle the communication between host (computer) and device. Signals on these wires are bi-directional. Voltage levels are differential: when DATA+ is at high level, DATA- is at low level, but there are some cases in which DATA+ and DATA- are at the same level, like EOP (end of packet).

Therefore, in the firmware driven USB implementation it is necessary to be able to sense or drive both these signals.

According to the USB standard the signal wires must be driven high between 3.0-3.6V, while the Vcc supported by the USB host is 4.4 - 5.25V. So if the microcontroller is powered directly from the USB lines, then the data lines must pass through a level converter to compensate for the different voltage levels. Another solution is to regulate the Vcc supported by the host down to 3.3V, and run the microcontroller at that voltage level.

Figure 1. Low Speed Driver Signal Waveforms

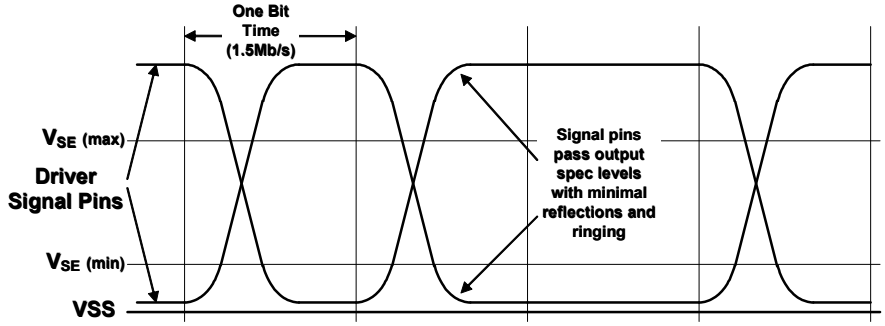


Figure 2. Packet Transaction Voltage Levels

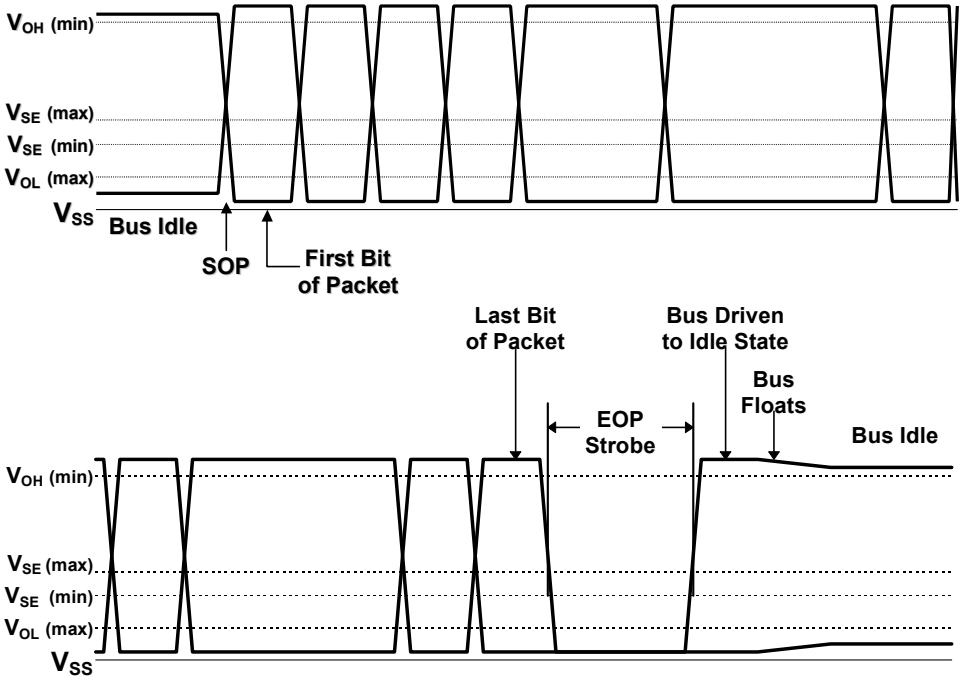
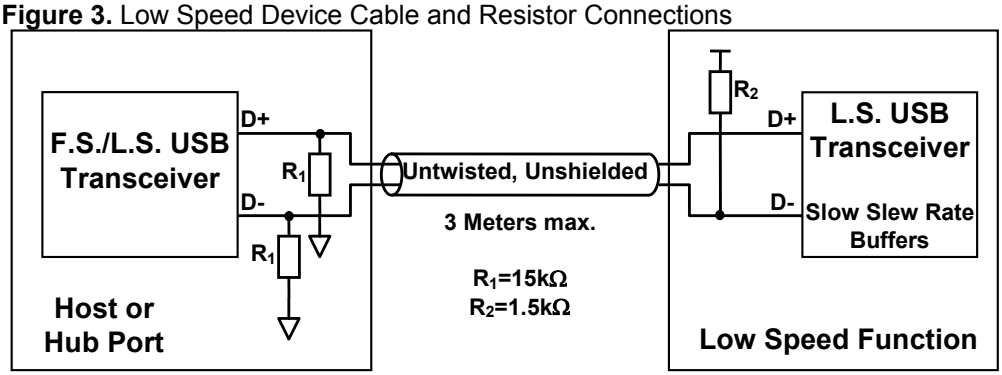


Figure 3. Low Speed Device Cable and Resistor Connections

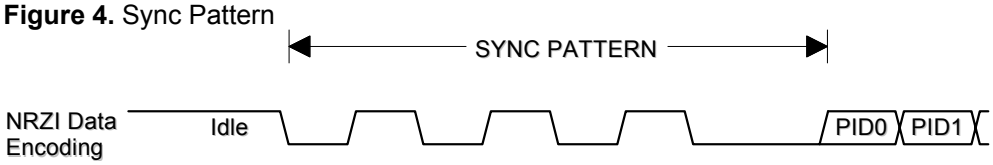


The USB device connection and disconnection is detected based on the impedance sensed on the USB line. For LowSpeed USB devices a 1.5k Ohm pull-up resistor between DATA- signal and Vcc is necessary. For FullSpeed devices, this resistor is connected to DATA+.

Based on this pull-up, the host computer will detect that a new device is connected to the USB line.

After the host detects a new device, it can start communicating with it in accordance with the physical USB protocol. The USB protocol, unlike UART, is based on synchronous data transfer. Synchronization of transmitter and receiver is necessary to carry out the communication. Because of this, the transmitter will transmit a small header as a sync pattern preceding the actual data. This header is a square wave (101010), succeeded by two zeros after which the actual data is transmitted.

Figure 4. Sync Pattern



In order to maintain synchronization, USB demands that this sync pattern is transmitted every millisecond in the case of full speed devices, or that both signal lines are pulled to zero every millisecond in the case of low speed devices. In hardware-implemented USB receivers, this synchronization is ensured by a digital PLL (phase locked loop). In this implementation, the data sampling time must be synchronized with the sync pattern, then wait for two zeros, and finally start receiving data.

Data reception on USB must satisfy the requirement that receiver and transmitter are in sync at all times. Therefore it is not permitted to send a stream of continuous zeros or ones on the data lines. The USB protocol ensures synchronization by bit stuffing. This means that, after 6 continuous ones or zeros on the data lines, one single change (one bit) is inserted. The signal on the USB lines are NRZI coded. In NRZI each '0' is represented, by a shift in the current signal level, and each '1' is represented by a hold of the current level. For the bit stuffing this means that one zero bit is inserted into the logical data stream after 6 contiguous logical ones.

Figure 5. NRZI Data Encoding

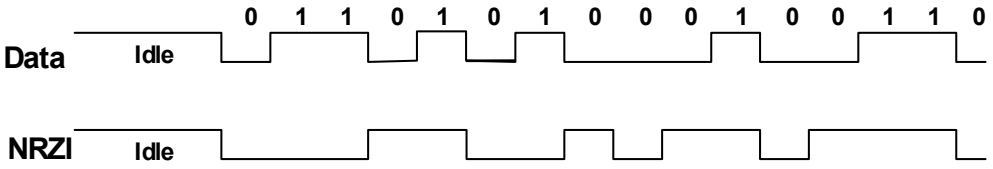


Figure 6. Bit Stuffing

Data Encoding Sequence:

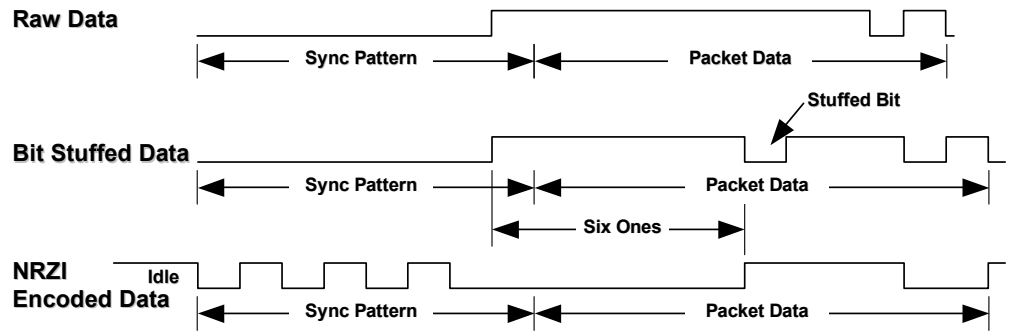
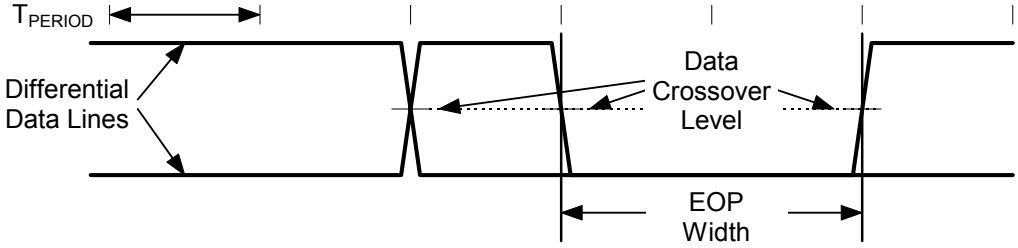


Figure 7. EOP Width Timing



Notification of end of data transfer is made by an end-of-packet (EOP) part. EOP consists of pulling both physical DATA+ and DATA- to the low voltage level. EOP is succeeded by a short time of idle state (min 2 periods of data rate). After this, the next transaction can be performed.

Data between sync pattern and EOP is NRZI coded communication between USB device and host. The data stream is composed of packets consisting of several fields: Sync field (sync pattern), PacketID (PID), Address field (ADDR), Endpoint field (ENDP), Data, and Cyclic redundancy check field (CRC). Usage of these fields in different types of data transfer is explained well in [2]. USB describes four types of transfer: Control Transfer, Interrupt Transfer, Isochronous Transfer, and Bulk Transfer. Each of these transfers is dedicated for different device requirements, and their explanations can be found in [2].

This implementation is using Control transfer. This transfer mode is dedicated for device settings, but can also be used for general purpose transfers. Implementation of Control transfer must exist on every USB device, as this mode is used for configuration when the device is connected for obtaining information from the device, setting device address, etc.. A description of the Control transfer and its contents can be found in [2] and [1]. Each Control transfer consists of several stages: Setup stage, Data stage and Status stage.

Data is in USB transferred in packets, with several bytes each. The packet size is determined by each device, but is limited by specification. For LowSpeed devices, packet size is limited to 8 byte. This 8 byte long packet + start and end field must be received into the device buffer in one USB transfer. In hardware-based USB receivers, the various parts of the transfer are automatically decoded, and the device is notified only when the entire message has been assigned to the particular device. In a firmware implementation, the USB message must be decoded by firmware after the entire message has been received into the buffer. This gives us the requirements and limitations: The device must have a buffer for storing the whole USB message length, another buffer for USB transmission (prepared data to transmit), and administration overhead with message decoding and checking. Additionally the firmware is required to perform fast and precise synchronous speed reception from physical pins to buffer and transmission from buffer to pins. All these capabilities are limited by the microcontroller's speed and program/data memory capacity, so the firmware must be carefully optimized. In some cases the microcontroller computation power is very close to the minimum requirements and therefore all firmware is written in assembly.

2 Hardware Implementation

A schematic diagram of the microcontroller connection to the USB bus is shown in Figure 8 and Figure 9. These schematics were made for the specific purpose of a USB to RS232 converter. There were also implemented specific functions as direct pin control and EEPROM read/write.

Figure 8. USB interface with ATtiny2313 as USB to RS232 converter with 32 byte FIFO + 8-bit I/O control + 128 bytes EEPROM

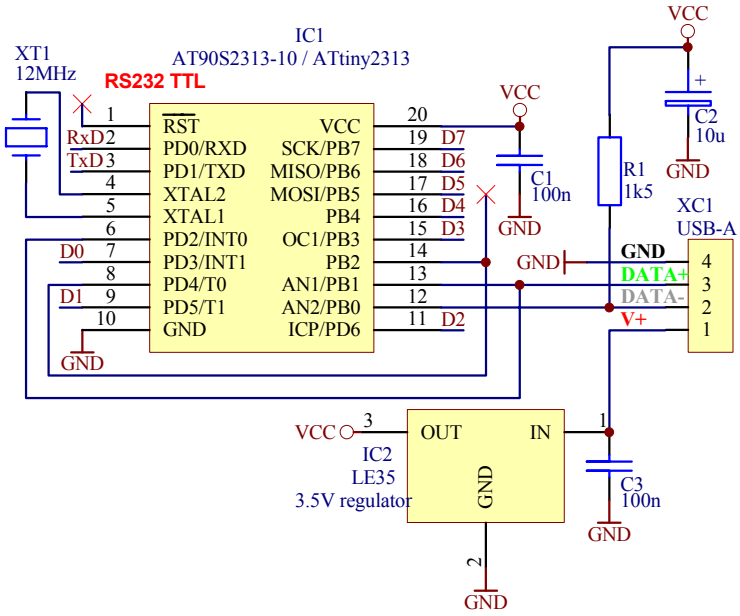
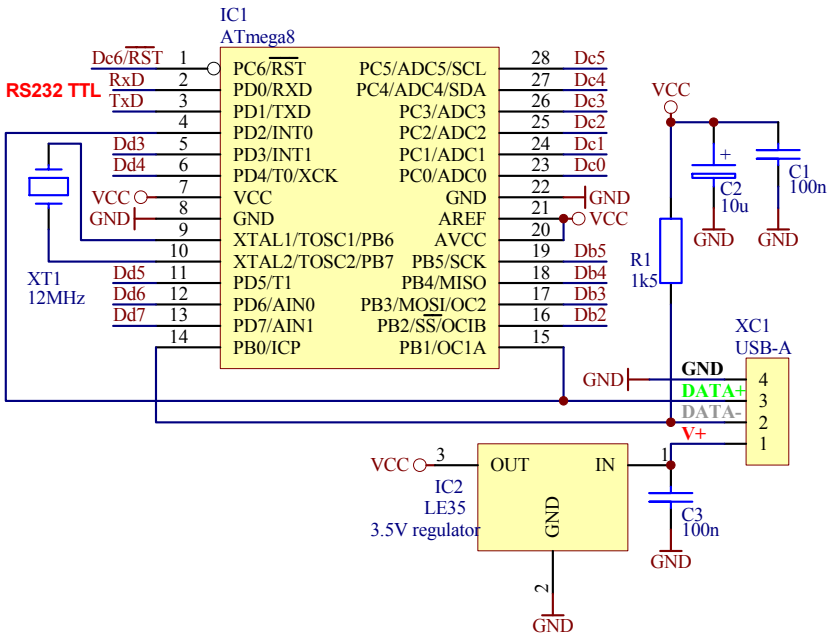


Figure 9. USB interface with ATmega8/48/88/168 as USB to RS232 converter with 800 byte FIFO + I/O control + 512bytes EEPROM



The USB data lines, DATA- and DATA+, are connected to pins PB0 and PB1 on the AVR. This connection cannot be changed because the firmware makes use of an AVR finesse for fast signal reception: The bit signal captured from the data lines is right shifted from LSB (PB0) to carry and then to the reception register, which collects the bits from the data lines. PB1 is used as input signal because on 8-pin AT90S2323 this pin can be used as external interrupt INT0 no additional connection to INT0 is necessary – the 8-pin version of the AVR is the smallest pin count available. On other AVRs, an external connection from DATA+ to the INT0 pin is necessary to ensure no firmware changes between different AVR microcontrollers.

For proper USB device connection and signaling, the AVR running as low speed USB device must have a 1.5k Ohm pull-up resistor on DATA-.

The Vcc supplied by the USB host may vary from 4.4V to 5.25V. This supply has to be regulated to 3.0 – 3.6V before connecting the 1.5k Ohm pull-up resistor and sourcing the AVR. Dimension a voltage regulator depending on the power load of the target system. The voltage regulator must be a low drop-out regulator. The schematics in Figure 8 and Figure 9 use a LE35 regulator with a nominal output voltage of 3.5V. But one can use any similar solution as long as the required properties are held. Even a very simple regulator based on a Zener diode could in some cases be used.

The other components provide functions for proper operation of the microcontroller only: Crystal as clock source, and capacitors for power supply filtering.

This small component count is sufficient to obtain a functional USB device, which can communicate with a computer through the USB interface. This is a very simple and inexpensive solution. Some additional components can be added to extend the device functions.

A TSOP1738 infrared sensor can be used to receive an IR signal. A MAX232 TTL to RS232 level converter should be added to make a USB to RS232 converter. To control LED diodes or display, they can be connect to I/O pins directly or through resistors.

3 Software Implementation

All USB protocol reception and decoding is performed at the firmware level. The firmware first receives a stream of USB bits in one USB packet into the internal buffer. Start of reception is based on the external interrupt INT0, which takes care of the sync pattern. During reception, only the end of packet signal is checked (EOP detection only). This is due to the extreme speed of the USB data transfer. After a successful reception, the firmware will decode the data packets and analyze them. First it checks if the packet is intended for this device according to its address. The address is transferred in every USB transaction and therefore the device will know if the next transferred data are dedicated to it. USB address decoding must be done very quickly, because the device must answer with an ACK handshake packet to the USB host if it recognizes a valid USB packet with the given USB address. Therefore, this is a critical part of the USB answer.

After the reception of this bit stream, we obtain an NRZI coded array of bits with bit stuffing in the input buffer. In the decoding process we first remove the bit stuffing and then the NRZI coding. All these changes are made in a second buffer (copy of the reception buffer). A new packet can be received while the first one is being decoded. At this point, decoding speed is not so important. This is because the device can delay the answer itself. If the host asks for an answer during decoding, the device

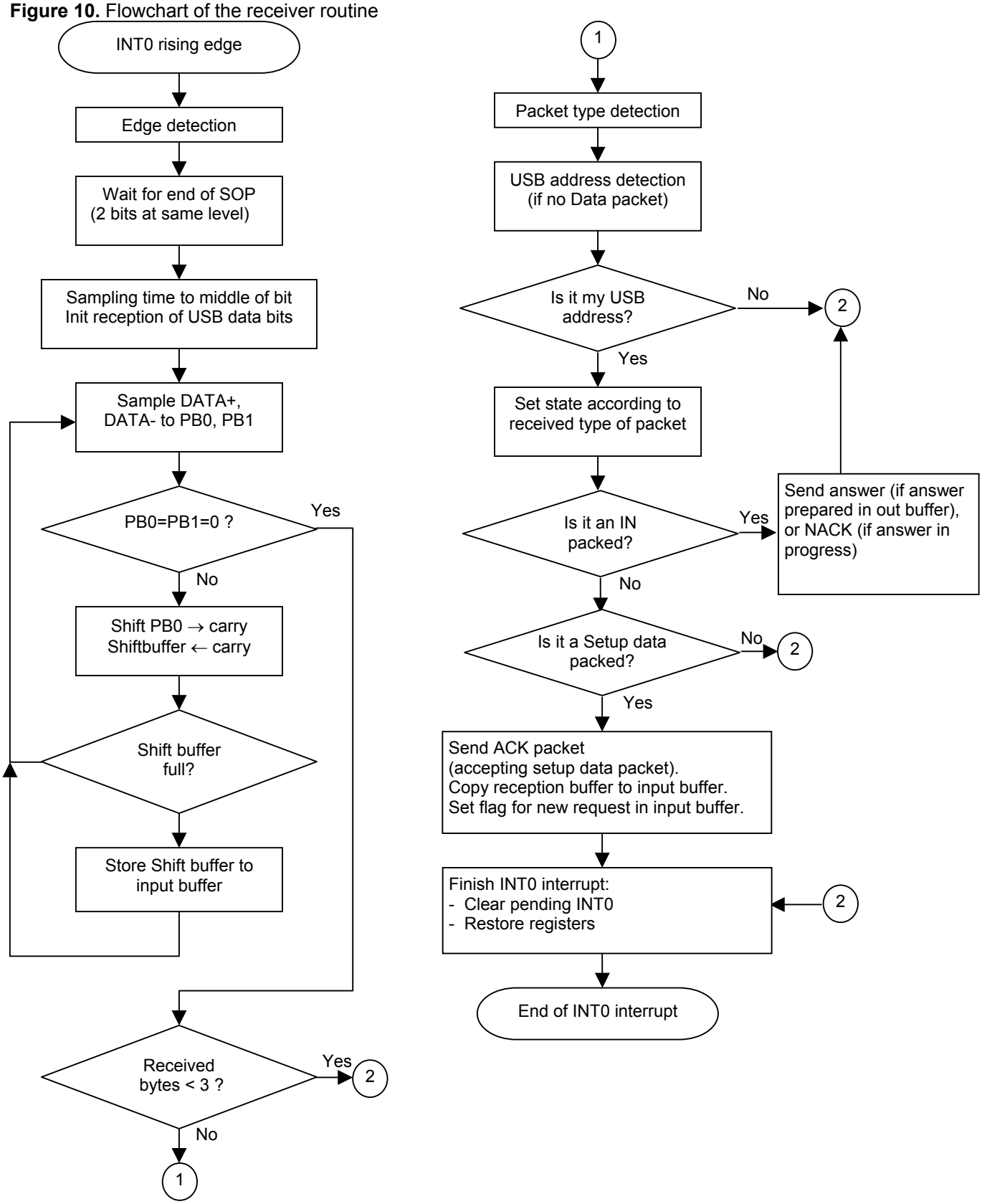
must answer immediately with NAK so that the host will understand it is not ready yet. Because of this, the firmware must be able to receive packets from the host during decoding, decode whether the transaction is intended for the device, and then send a NAK packet if there is some decoding in progress. The host will then ask again. The firmware also decodes the main USB transaction and performs the requested action; for instance, send char to RS232 line and wait for transmission complete, and prepares the corresponding answer. During this process the device will be interrupted by some packets from the host, usually IN packets to obtain an answer from the device. To these IN packets, the device must answer with NAK handshake packets. When the answer is ready, and the device has performed the required action, the answer must first go through a CRC field calculation and inclusion, then the NRZI coding, and then bit stuffing. Now, when the host requests an answer, this bit stream can be transferred to the data lines according to the USB specification.

3.1 Firmware description

In the following, the main parts of the firmware will be described. The firmware is divided into blocks: interrupt routines, decoding routines, USB reception, USB transmission, requested action decoding, and performing requested custom actions.

Users can add their own functions to the firmware. Some examples on how to make customer-specific functions can be found in the firmware code, and the user can write new device extensions according to the existing built-in functions. For example TWI support can be added according to the built-in function for direct pin control.

Figure 10. Flowchart of the receiver routine



3.2 "EXT_INT0" Interrupt Service Routine

The external interrupt 0 is active all the time while the firmware is running. This routine initiates the reception of the USB serial data. An external interrupt occurs on a rising edge on the INT0 pin, a rising edge marks the beginning of the sync pattern of a USB packet, se Figure 4. This activates the USB reception routine.

First, the data sampling must be synchronized to the middle of the bit width. This is done according to the sync pattern. Since the bit duration is only 8 cycles of the XTAL clock, and the interrupt occurrence could be delayed (+/- 4 cycles), the edge synchronization of the sync pattern must be performed carefully. End of sync pattern and begin of data bits are detected according to the last dual low level bits in the sync packet (see Figure 4).

After this, the actual data sampling is started. Sampling is performed in the middle of the bit. Because data rate is 1.5Mbit/s (1.5MHz) and the microcontroller speed is 12MHz, we have only 8 cycles at our disposal for data bit sampling, storing the result into the buffer byte, shifting the buffer byte, checking if the whole byte has been received, storing this byte into SRAM, and checking for EOP. This is perhaps the most crucial part of the firmware; everything must be done synchronously with exact timing. When a whole USB packet has been received, packet decoding must be performed. First, we must quickly determine the packet type (SETUP, IN, OUT, DATA) and received USB address. This fast decoding must be performed inside the interrupt service routine because an answer is required very quickly after receiving the USB packet (the device must answer with an ACK handshake packet when a packet with the device address has been received, and with NAK when the packet is for the device, but when no answer is currently ready).

At the end of the reception routine (after ACK/NAK handshake packet has been sent) the sampled data buffer must be copied into another buffer on which the decoding will be performed. This is in order to free the reception buffer to receive a new packet.

During reception the packet type is decoded and the corresponding flag value is set. This flag is tested in the main program loop, and according to its value the appropriate action will be taken and the corresponding answer will be prepared with no regard to microcontroller speed requirements.

The INT0 must be allowed to keep its very fast invocation time in all firmware routines, so no interrupt disabling is allowed and during other interrupts' execution (for example serial line receive interrupt) INT0 must be enabled. Fast reception in the INT0 interrupt routine is very important, and it is necessary to optimize the firmware for speed and exact timing. One important issue is register backup optimization in interrupt routines.

3.3 Main program loop

The main program loop is very simple. It is only required to check the action flag: what to do when some received data are present. In addition it checks whether the USB interface is reset (both data lines at low level for a long time) and, if it is, reinitializes the device. When there is something to do (i.e. action flag active), the corresponding action is called: decoding NRZI in packet, bit stuffing removal, and preparation of the requested answer in the transmit buffer (with bit stuffing and NRZI coding). Then one flag is activated to signal that the answer is prepared for sending.

Physical output buffer transmission to the USB lines is performed in the reception routine as answer to the IN packet.

3.4 Short description of firmware subroutines

In the following, the firmware subroutines and their purposes are described briefly.

3.4.1 Reset:

Initialization of the AVR microcontroller resources: stack, serial lines, USB buffers, interrupts.

3.4.2 Main:

The main program loop. Checks the action flag value and, if flag is set, performs the required action. Additionally, this routine checks for USB reset on data lines and reinitializes the USB microcontroller interface if this is the case.

3.4.3 Int0Handler:

The interrupt service routine for the INT0 external interrupt. Main reception/transmission engine; emulation from USB data lines. Storing data to buffer, decision of USB packet owners (USB address), packet recognition, sending answer to USB host. Basically the heart of the USB engine.

3.4.4 SetMyNewUSBAddresses:

Routine to change the USB address. The address is changed and is coded to its NRZI equivalent. This is done because the address decoding during USB packet reception, must be performed quickly.

3.4.5 FinishReceiving:

Copies coded raw data from USB reception packet to decoding packet (for NRZI and bit stuffing decoding).

3.4.6 USBreset:

Initializes USB interface to default values (as the state after power on).

3.4.7 SendPreparedUSBAnswer:

Sends prepared output buffer contents to USB lines. NRZI coding and bit stuffing is performed during transmission. Packet is ended with EOP.

3.4.8 ToggleDATAPID:

Toggles DATAPID packet identifier (PID) between DATA0 and DATA1 PID. This toggling is necessary during transmission as per the USB specification.

3.4.9 ComposeZeroDATA1PIDAnswer:

Composes zero answer for transmission. Zero answer contains no data and is used in some cases as answer when no additional data is available on device.

3.4.10 InitACKBufffer:

Initializes buffer in RAM with ACK data (ACK handshake packet). This buffer is frequently sent as answer so it is always kept ready in memory.

3.4.11 SendACK:

Transmits ACK packet to USB lines.

3.4.12 InitNAKBuffer:

Initializes buffer in RAM with NAK data (NAK handshake packet). This buffer is frequently sent as answer so it is always kept ready in memory.

3.4.13 SendNAK:

Transmits NAK packet to USB lines.

3.4.14 ComposeSTALL:

Initializes buffer in RAM with STALL data (STALL handshake packet). This buffer is frequently sent as answer so it is always kept ready in memory.

3.4.15 DecodeNRZI:

Performs NRZI decoding. Data from USB lines in buffer is NRZI coded. This routine removes the NRZI coding from the data.

3.4.16 BitStuff:

Removes/adds bit stuffing in received USB data. Bit stuffing is added by host hardware according to the USB specification to ensure synchronization in data sampling. This routine produces received data without bit stuffing or data to transmit with bit stuffing.

3.4.17 ShiftInsertBuffer:

Auxiliary routine for use when performing bit stuffing addition. Adds one bit to output data buffer and thus increases the buffer length. The remainder of the buffer is shifted out.

3.4.18 ShiftDeleteBuffer:

Auxiliary routine for use when performing bit stuffing removal. Removes one bit to output data buffer and thus decreases the buffer length. The remainder of the buffer is shifted in.

3.4.19 MirrorInBufferBytes:

Exchanges bit order in byte because data is received from USB lines to buffer in reverse order (LSB/MSB).

3.4.20 CheckCRCIn:

Performs CRC (cyclic redundancy check) on received data packet. CRC is added to USB packet to detect data corruption.

3.4.21 AddCRCOut:

Adds CRC field into output data packet. CRC is calculated according to the USB specification from given USB fields.

3.4.22 CheckCRC:

Auxiliary routine used in CRC checking and addition.

3.4.23 LoadDescriptorFromROM:

Loads data from ROM to USB output buffer (as USB answer).

3.4.24 LoadDescriptorFromROMZeroInsert:

Loads data from ROM to USB output buffer (as USB answer) but every even byte is added as zero. This is used when a string descriptor in UNICODE format is requested (ROM saving).

3.4.25 LoadDescriptorFromSRAM:

Loads data from RAM to USB output buffer (as USB answer).

3.4.26 LoadDescriptorFromEEPROM:

Loads data from data EEPROM to USB output buffer (as USB answer).

3.4.27 Load[X]Descriptor:

Performs selection for answer source location: ROM, RAM or EEPROM.

3.4.28 PrepareUSBOutAnswer:

Prepares USB answer to output buffer according to request by USB host, and performs the requested action. Adds bit stuffing to answer.

3.4.29 PrepareUSBAnswer:

Main routine for performing the required action and preparing the corresponding answer. The routine will first determine which action to perform – discover function number from received input data packet – and then perform the requested function. Function parameters are located in the input data packet.

The routine is divided into two parts:

- standard requests
- vendor specific requests

Standard requests are necessary and are described in USB specification (SET_ADDRESS, GET_DESCRIPTOR, …).

Vendor specific requests are requests that can obtain vendor specific data (in Control USB transfer). Control IN USB transfer is used for this AVR device to communicate with host. Developers can add their own functions here and in this manner extend the

device versatility. The various documented built-in functions in the source code can be used as templates on how to add custom functions.

3.5 Standard USB functions (Standard Requests)

ComposeGET_STATUS;

ComposeCLEAR_FEATURE;

ComposeSET_FEATURE;

ComposeSET_ADDRESS;

ComposeGET_DESCRIPTOR;

ComposeSET_DESCRIPTOR;

ComposeGET_CONFIGURATION;

ComposeSET_CONFIGURATION;

ComposeGET_INTERFACE;

ComposeSET_INTERFACE;

ComposeSYNCH_FRAME;

3.6 Vendor USB functions (Vendor requests)

DoSetInfraBufferEmpty;

DoGetInfraCode;

DoSetDataPortDirection;

DoGetDataPortDirection;

DoSetOutDataPort;

DoGetOutDataPort;

DoGetInDataPort;

DoEEPROMRead;

DoEEPROMWrite;

DoRS232Send;

DoRS232Read;

DoSetRS232Baud;

DoGetRS232Baud;

DoGetRS232Buffer;

DoSetRS232DataBits;

DoGetRS232DataBits;

DoSetRS232Parity;

DoGetRS232Parity;

DoSetRS232StopBits;

DoGetRS232StopBits;

3.7 Data structures (USB descriptors and strings)

DeviceDescriptor;

ConfigDescriptor;

LangIDStringDescriptor;

VendorStringDescriptor;

DevNameStringDescriptor;

3.8 Format of input message from USB host

As stated above, this USB device uses USB Control Transfer. This type of transfer uses a data format defined in the USB specification described in [2] on page 13 (Control Transfers). The document describes the details and explains how the control transfer works, and therefore also how this device communicates with the USB host. The AVR device is using control IN endpoint. A nice example of data communication can be found on page 15 of [2]. The communication between the host and the AVR device is done according to this example.

In addition to the actual control transfer, the format of the DATA0/1 field in the transfer is discussed. Control transfer defines in its setup stage a standard request, which is 8 bytes long. Its format is described on page 26 of [2] (The Setup Packet). There is a table with a description of the meaning of every byte. The following is important:

The standard setup packet is used for detection and configuration of the device after power on. This packet uses the Standard Type request in the bmRequestType field (bits D6-D5 = 0). All the following fields' (bRequest, wValue, wIndex, wLength) meanings can be found in the USB specification. Their explanation can be found on pages 27-30 in [2] (Standard Requests).

Every setup packet has eight bytes, used as described in Table 1.

Table 1. Standard setup packet fields (control transfer)

Offset	Field	Size	Value	Description

0	bmRequestType	1	Bit-map	Characteristics of request D7 Data xfer direction 0 = Host to device 1 = Device to host D6..5 Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4..0 Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	bRequest	1	Value	Specific request
2	wValue	2	Value	Word-sized field that varies according to request
4	wIndex	2	Index or Offset	Word sized field that varies according to request - typically used to pass an index or offset
6	wLength	2	Count	Number of bytes to transfer if there is a data phase

Table 2. Standard device requests

BmRequest-Type	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

Table 3. Vendor device requests used in firmware as functions calls.

BmRequest-Type	bRequest (function name)	bRequest (number)	wValue (param1)	wIndex (param2)	wLength	Data
110xxxxxB	FNCNumberDoSetInfraBufferEmpty	1	None	None	1	Status
110xxxxxB	FNCNumberDoGetInfraCode	2	None	None	1	Status
110xxxxxB	FNCNumberDoSetDataPortDirection	3	DDRB DDRC	DDRD usedports	1	Status
110xxxxxB	FNCNumberDoGetDataPortDirection	4	None	None	3	DDRB DDRC DDRD
110xxxxxB	FNCNumberDoSetOutDataPort	5	PORTB PORTC	PORTD usedports	1	Status
110xxxxxB	FNCNumberDoGetOutDataPort	6	None	None	3	PORTB PORTC PORTD
110xxxxxB	FNCNumberDoGetInDataPort	7	None	None	3	PINB PINC PIND
110xxxxxB	FNCNumberDoEEPROMRead	8	Address	None	Length	EEPROM bytes
110xxxxxB	FNCNumberDoEEPROMWrite	9	Address	EEPROM value	1	Status
110xxxxxB	FNCNumberDoRS232Send	10	RS232 byte value	None	1	Status
110xxxxxB	FNCNumberDoRS232Read	11	None	None	2	Status
110xxxxxB	FNCNumberDoSetRS232Baud	12	Baudrate Lo	Baudrate Hi	1	Status
110xxxxxB	FNCNumberDoGetRS232Baud	13	None	None	2	Baudrate
110xxxxxB	FNCNumberDoGetRS232Buffer	14	None	None	Length	RS232 bytes from FIFO
110xxxxxB	FNCNumberDoSetRS232DataBits	15	Databits value	None	1	Status
110xxxxxB	FNCNumberDoGetRS232DataBits	16	None	None	1	Databits value
110xxxxxB	FNCNumberDoSetRS232Parity	17	Parity value	None	1	Status
110xxxxxB	FNCNumberDoGetRS232Parity	18	None	None	1	Parity value
110xxxxxB	FNCNumberDoSetRS232StopBits	19	Stopbits value	None	1	Status
110xxxxxB	FNCNumberDoGetRS232StopBits	20	None	None	1	Stopbits value

The Control Transfer mode is used for the user communication, implemented as custom functions in the firmware. The Vendor Type request in the bmRequestType field (bits D6-D5 = 2) is used. Here all succeeding fields (bRequest, wValue, wIndex) can be modified according to the programmer's purposes. In our implementation, the bRequest field is used for the function number and the next fields are used for function parameters. The first parameter is in the wValue slot, the second at the wIndex location.

An example from the implementation is EEPROM writing. bRequest = 9 is chosen as function number. The wValue field is used for EEPROM address, and the value to write (EEPROM data) is in the wIndex field. According to this, we obtain the following function is obtained: EEPROMWrite(Address, Value).

If more user functions are required, it is enough to add function numbers and the body of the required function into the firmware. The technique can be extracted from the built-in functions in the firmware (see source code).

USB host also communicates with device with IN control transfers. Host sends an 8-byte IN data packet to the device in the format defined above (function number and parameters), and the device then answers with requested data. The length of the answered data is firmware limited in some cases up to 255 bytes, but the main limitation is on the device driver side on the host computer. The current driver supports 8-byte length answers in Vendor Type requests.

3.9 Firmware customization

Users can add new functions into the firmware and extend the device features.

In the firmware there are 3 examples on how to add user functions: DoUserFunctionX (X=0,1,2). Look at these examples to see how to add similar extended functions. The contents of the functions only depend on the device requirements.

The identification and device name presented to the computer side can be modified in the firmware This name is located in firmware as strings and can be changed to any string. However these names are recommended to be changed together with the USB PID (product ID) and VID (vendor ID) for correct recognition in target systems.

VID together with PID must be unique for a given device type. Therefore it is recommended that if a device functionality is changed, one should also modify the PID and/or VID. Vendor ID depends on the USB device vendor and must be assigned from the USB organization (see more information in [1]). Every vendor has its own ID and therefore this value cannot be changed to any unassigned value. But the product ID depends only on the vendor's choice, and the purpose of the PID is to recognize the different devices from the same vendor.

This application note is setup with VID 0x03EB and PID 0x21FF which is Atmel's VID. Do not use this VID in your target system.

3.10 PC software

In order to communicate with the device some software support on the PC side is needed. This software is divided into 3 levels:

1. Device driver: Used for low-level communication with the device and for installation into operating system (Windows98/ME/NT/XP).
2. DLL library: Used for encapsulation of device functions and communication with the device driver. The DLL simplifies the device function access from the user's application. It includes some device and operating system related functions (threads, buffers, etc.).
3. User application: Makes user interface for friendly communication between user and device. Uses function calls from DLL library only.

3.10.1 Device driver and installation files

The first time the USB device is connected to the computer USB port, the operating system will detect the device and request driver files. This is called device installation. For the installation process it is necessary not only to make the device driver, but also an installation script in which the installation steps are described.

The device driver for the device described in this document is made with Windows2000 DDK (Driver Development Kit). The development of the USB driver is based on one of the included examples in the DDK – IsoUsb. This driver has been modified for AVR USB device communication. In the original source code, parts have been extended/added around the IOCTL communications, because this device communicates with the computer through these IOCTL calls. To reduce the driver code size, unused parts have been removed. The name of the driver is “AVR309.sys” and it works as sender of commands to the USB device (Control IN transfers). The driver works on all 32-bit Windows versions except Win95.

An installation script written in an INF file is used during device installation. In this INF file the various installation steps are described. The file “AVR309.inf” was created using a text editor. This file is requested by the operating system during installation. During the installation process, the driver file is copied into the system and the required system changes are made. The INF file ensures installation of the DLL library to the system search path for easy reach from various applications.

Three files are necessary for device installation: INF file “AVR309.inf”, driver “AVR309.sys”, and DLL library “AVR309.dll”.

3.10.2 DLL library

The DLL library communicates with the device driver and all device functions are implemented in this library. This way the programming of end-user applications is simplified. The DLL library ensures exclusive access to the device (serializes device access), contains system buffer for RS232 data reception, and creates a single system thread for device RS232 data buffer reading.

Serialization in DLL ensures that only one application/thread will communicate with the device at any given time. This is necessary because of the possibility of mixing question and answer from various applications at the same time.

A system buffer for RS232 data reception ensures that the data received from the device’s RS232 line is stored into one buffer that is common to all applications. This way, data received by the device will be sent to all applications. There is no danger that an application will receive incomplete data because some other application has read some of the data before.

Only one system thread exists for all applications, and will periodically request device for RS232 data. The thread will then store received data into the system buffer. Only one system buffer solution ensures small CPU usage (in comparison to every application having their own thread) and simplifies storing data into the system buffer.

All device functions are defined in the DLL library, and they are exported in a user-friendly form: not as function number and parameters, but as tidy function names with parameters. Some functions are more complex internally, as the function for RS232 buffer data read. This way, developers of end-user applications can rapidly write application using only the DLL interface. There is no need to study the low-level

device functions, as the DLL library separates the application programmer level from the hardware level.

Declarations are written for the 3 mostly used programming languages: Borland Delphi, C++ (Borland or Microsoft) and Visual Basic. A detailed description of these functions can be found in the included help file AVR309_DLL_help.htm.

The DLL is written in Delphi, and all source code is included in this application note.

3.10.3 End user application

The end-user application will use functions from the DLL library to communicate with the device only. Its main purpose is to make a user-friendly graphical user interface (GUI).

Application programmers use the DLL library to write their own applications. An example can be found in the published project where all the source code is available. Many applications can be written using this example as starting point, and in several programming languages (Delphi, C++, and Visual Basic).

There is included an example of an end user application called “AVR309demo.exe”. This software is only meant as an example on how to use the functions from the DLL library. The included source code is written in Delphi, and can be used as a template for other applications.

3.10.4 UART speed error discussion

The microcontroller uses a 12MHz clock because of the USB sampling. But using this clock value has a minor disadvantage in that baud rate generation will contain small errors for the standard baud rates. However the high value of the clock minimizes this error. Absolute maximum error that could be accepted in baud rate generation is 4%: because maximum error is half a bit duration (0.5) and the maximum packet time is 12bits = 1start bit + 8data bits + 1parity bit + 2stop bits. Then the error is: 0.5/12*100% = 4.1%.

The functions in the DLL automatically check this error and set the baud rate of the microcontroller only if the error is below 4%, it also returns an error message in case of an unsupported baud rate. It is however recommended to never use an error larger than 2%.

Table 4 summarizes the errors of the standard baud rates when using a 12MHz clock.

Table 4. AVR UART baud rate errors (12MHz clock)

Standard baudrates	Baudrate in AVR	Error [%]
600	602	+0.33
1200	1204	+0.33
2400	2408	+0.33
4800	4808	+0.17
9600	9616	+0.17
19200	19230	+0.16
28800	28846	+0.16

38400	38462	+0.16
57600	57692	+0.16
115200	115384	+0.16

4 Used documentation and resources

4.1 USB related resources:

[1] http://www.usb.org - USB specification and another USB related resources

[2] usb-in-a-nutshell.pdf from http://www.beyondlogic.org/usbnutshell/usb-in-a-nutshell.pdf - very good and simple document how USB works

[3] http://www.beyondlogic.org - USB related resources

[4] enumeration.pdf - exact pictures how USB enumeration works

[5] http://mes.loyola.edu/faculty/phs/usb1.html

[6] http://www.mcu.cz - USB section (in Czech/Slovak language)

[7] crcdes.pdf – implementation CRC in USB

[8] USBspec1-1.pdf – USB 1.1 specification

[9] usb_20.pdf – USB 2.0 specification

4.2 AVR related resources:

[10] http://www.atmel.com/AVR - AVR 8-bit microcontrollers family

[11] doc2543.pdf – ATtiny2313 datasheet
http://www.atmel.com/dyn/products/product_card.asp?part_id=3229

[12] doc2545.pdf – ATmega48/88/168 datasheet
http://www.atmel.com/dyn/products/product_card.asp?part_id=3301

[13] avr910.pdf – AVR ISP programming

[14] http://www.avrfreaks.com - a lot of AVR resources and information

[15] AVR Studio 4 – debugging tool for the AVR family (from http://www.atmel.com)

[16] Simple LPT ISP programmer.
(http://www.hw.cz/products/lpt_isp_prog/index.html)

4.3 Driver related resources:

[17] http://www.beyondlogic.org - USB related resources about USB drivers

[18] http://www.cypress.com - fully documented USB driver (for USB thermometer)

[19] http://www.jungo.com - WinDriver and KernetDriver – easy to use USB drivers

[20] http://microsoft.com Microsoft Windows DDK – Driver Development Kit – tools for drivers writing

5 Appendix

The appendixes are included in a separate attachment file to this application note.

5.1 AVR firmware with source code

Firmware source code for the ATmega8 AVR microcontroller was written in AVR Studio 4. Source code can be found in the text file USBtoRS232.asm or in syntax highlighted form file USBtoRS232asm.pdf.

5.2 DLL library with source code

Library AVR309.dll was written in Delphi3, so its source code is based on Object Pascal language. Interfaces (Delphi, C/C++ and Visual Basic.) to the DLL library (exported functions) “AVR309.dll” are described in file AVR309_DLL_help.htm. Whole source code of AVR309.dll library written in Delphi3 can be found in file AVR309.dpr (Delphi3 project).

5.3 End user application example, with source code

The example using the DLL library as an end user application is AVR309USBdemo.exe. The source code is a Delphi3 application: AVR309USBdemo.dpr.

6 About the Author

Ing. Igor Cesko
Slovakia
www.cesko.host.sk
cesko@internet.sk

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

2556-AVR-07/05