# AVR309 USB 协议转换到 UART

## 特征

- 硬件实现**USB协议**
- 支持低速**(1.5Mbit/s) 并兼容 USB2.0**
- 可在非常小的AVR设备上运行，2K字节或以上。
- 只需很少的外部元件
- 使用一只电阻实现低速USB检测
- 电压分配/校准，带滤波（**Voltage divider/regulator, with filtering**）

**•实现功能:**

- **直接 I/O 引脚控制。**
- **USB 转换到 RS232**
- 擦写**EEPROM 记录**
- 用户可以象例子一样实现自己的功能**:**
- **USB 控制 TWI （$I^2C$）**
- **USB A/D 和 D/A 转换**
- **定制设备厂家名称（在PC端可见）。**
- PC端有完整的源代码及文档支持。
- **DLL 库功能**
- **Delphi 编写的演示程序**
- 通过DLL与设备进行通信的例子 **(Delphi, C++, Visual Basic)**

## 介绍

USB 界面变得非常流行，尤其因为终端用户可以方便地使用终端程序 (即插即用而无需重启)。作为开发者，无论如何，与使用 RS232 相比，在设备上实现 USB 困难得多。需要在 PC 端增加软件支持：设备驱动程序。因为这个原因，基于 RS232 的通信在设备厂商中仍然非常流行。这个界面有好的定制和完善的操作系统支持。但是最近 RS232 端口已经从标准 PC 界面中移除, 让给了 USB 端口。

当前将 USB 应用于外部设备有两个方法：

a) 使用微控制器硬件实现 USB 接口。这必须要知道 USB 怎样工作及写固件到微控制器。而且,还要在 PC 端写驱动程序（长期以来，操作系统没有提供标准的 USB 类）。缺点（对于小厂商及业余爱好者这是个非常不利的因素）是效率高的微控制器比较少，并且价格比简单的 RS232 微控制器高。

b) 第二个选择是在 USB 和其它接口之间做一些通常的转换。其它接口通常是 RS232，8 位数据总线，或 TWI 总线， 这个方法不必使用专用的固件，也不用知道 USB 怎样运作，不用写驱动程序，厂商会为整个解决方案提供一个驱动程序。缺点是整个系统的代价比较高，最终产品规模比较大。

在这个文档中当前的解决方案是使用低价格的微控制器，通过微控制器的固件仿真 USB 协议来实现 USB 接口。这个设计的难度在于获得足够的速度。USB 总线相当快：低速 - 1.5Mbit/s, 全速 - 12Mbit/s, 高速 - 480Mbit/s。常用的微控制器最高速度限制：AT89C2051 - 2MIPS = 24MHz/(12cycl/inst.), PIC16F84 - 5MIPS = 20MHz/(4cycl/inst.), AT90S23x3 - 10MIPS = 10MHz/(1cycl/inst)。虽然有很多高性能的微控制器，但是性价比不高，而且尺寸较大。基于上述原因， AT90S1200/AT90S23x3 便宜并且能够满足低速 USB 接口。这个方案并不适合于高速 USB。

# 操作理论

关于详细的USB物理通信可在网站www.usb.org上找到，该文档对于初学者来说非常复杂和困难（650页）
对于初学者来说《USB in a Nutshell》是一份非常好及非常简单的文档。《Making Sense of the USB Standard》 是*Craig Peacock* (Craig.Peacock@beyondlogic.org)编写的。这编文档可以在网站http://www.beyondlogic.org 或usb-in-a-nutshell.pdf找到，推荐阅读这编文档以理解USB是怎样工作的（仅30页）。
文档只在理解设备固件的范围内作解说。USB 物理界面由 4 条线组成：2 条供外设用的电源线(VCC 和 GND)，和 2 条信号线 (DATA+ 和 DATA-)。 电源线提供大约 5V 电压和最大 500mA 电流。可以从 Vcc 和 GND 给设备供电。信号线命名为 DATA+ 和 DATA- ，在主机和设备之间传递信息。这些线上的信号是 bi-directional 的。电平是不同的：当 DATA+ 为高电平时，DATA- 为低电平，但有些情况下 DATA+ 和 DATA- 处于相同的电平 (EOP – 包结束，空闲状态)。

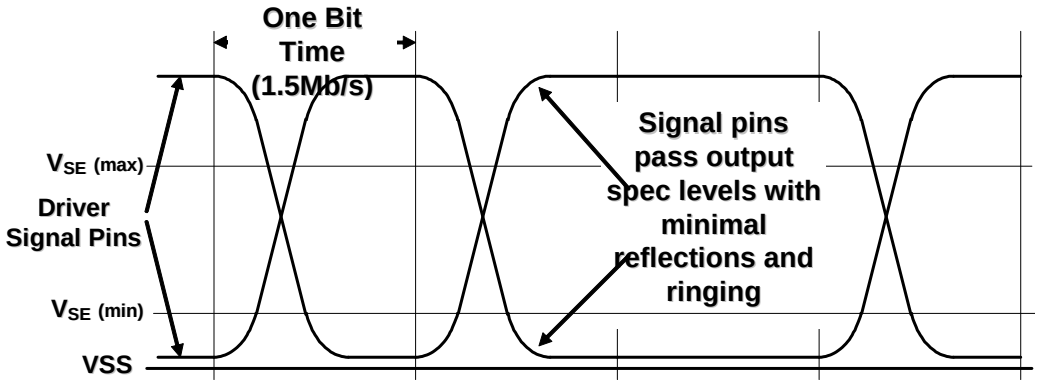
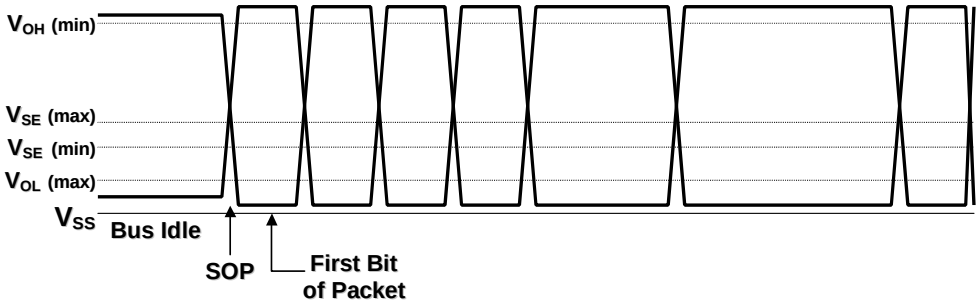



图1　低速驱动信号波形

所以，在固件实现USB驱动，必须要能够判断或处理这些信号。



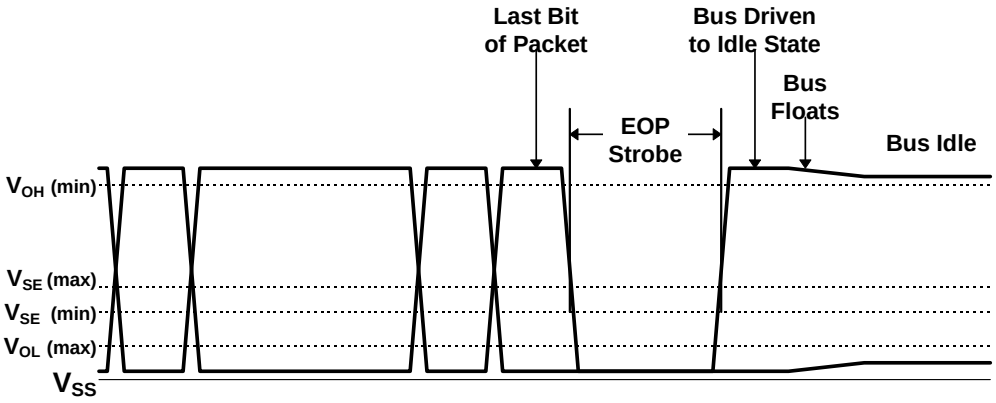



图2. 包事务处理电平

依照USB 标准，当USB主机提供的Vcc在4.4 - 5.25V时，信号线高电平必须在3.0-3.6V。 所以如果微控制器直接驱动USB 线，那么数据线必须通过电平转换以适应不同的电压标准。另一个解决方法是将主机提供的Vcc调节到 3.3V，并且微控制器在该电压运行。

USB 设备连接或断开的检测是基于USB线上的电阻感应。应用于低速USB设备（我们的方案中）时，必须在VCC及DATA-之间接一个1.5kohm 的上拉电阻(应用于高速USB设备时，这个电阻接到DATA+)。

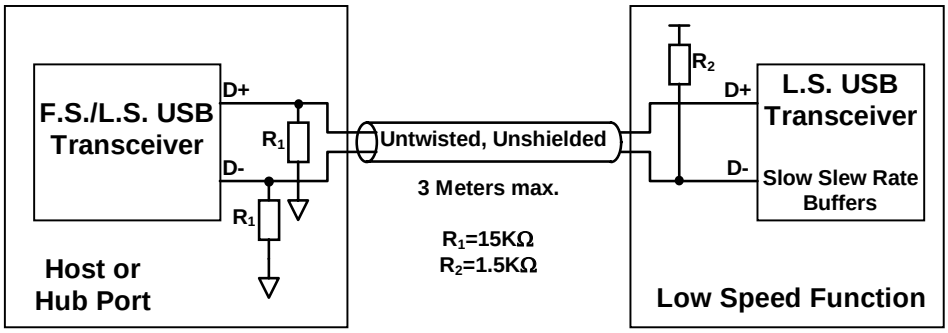



图1. 低速设备电缆和电阻连接

基于这个上拉，主机能够检测到连接到USB线上的新设备。

当主机检测到一个新设备时，能够使用与物理 USB 一致的协议与之通信。USB 协议，不象 UART,它是基于同步数据传输的。在通信中必须实现发送和接收的同步。因此，发送者必须在实际数据前加上一个小头部（同步块)。这个数据头是一个方波(101010)， 实际数据传送成功后，后续 2 个 0。

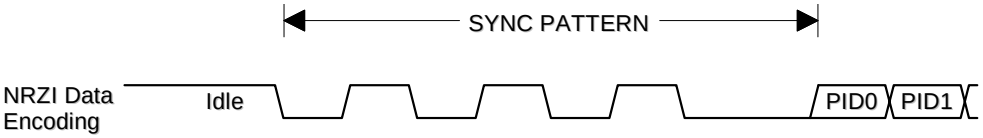



图2. 同步块

为了维护同步，对于全速设备USB 要求每1毫秒传送一个同步块，对于低速设备，要求两条信号线都拉低到0。在硬件实现USB接收中，由数字式PLL（锁相环）确保同步。在我们的实现中，我们必须从同步块中取得同步数据取样时间，等待2个0，接着开始接收数据。

接收USB数据必须确保数据发送和接收在任意时间内都是同步的,所以不充许在数据线上发送连续的0或1的数据流。USB 协议通过位填充确保同步。这意味首，数据线上出现6个连续的1或0之后，将会插入一个单一的（一位）变化。当USB线上的信号为NRZI码时，这意味着在逻辑数据流中连续6个逻辑1之后插入了一位0。

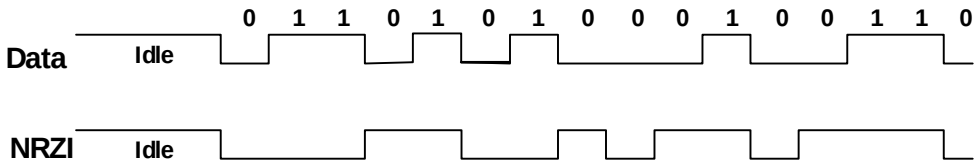



图 3. NRZI 数据编码

**Data Encoding Sequence:**

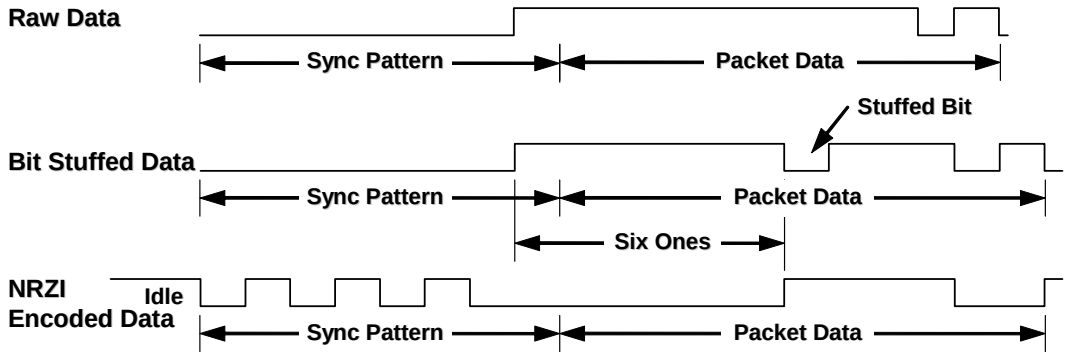



图 4. 位填充

数据传送结束通知使用EOP (包未端) 块。EOP 由2条数据线同时有2个0组成(物理的DATA+ 和 DATA-为低电平)。 EOP 之后跟一个短时的空闲状态 (最小为数据速率的2个周期)。之后，可以开始下一个传输。

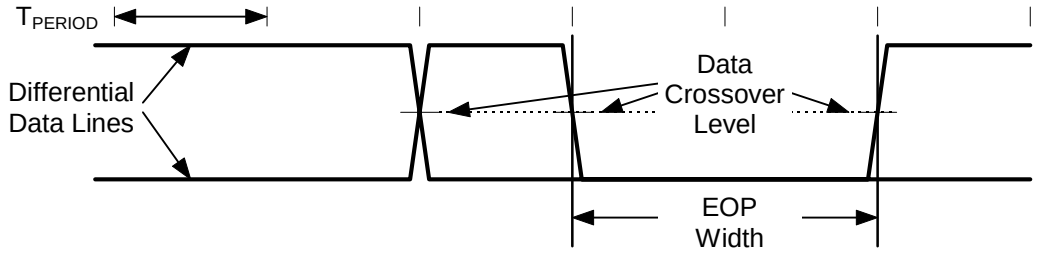



图 5. EOP 时间宽度

在同步块和EOP之间的数据是NRZI码，数据主机和USB设备之间的通信使用NRZI码。数据流封装为包，包括以下几个域：同步域 (SYNC 同步块)，包标签(PID)，地址域(ADDR)，终点域(ENDP)，数据域，CRC域 (CRC)。在数据传输中使用这些不同类型的域的解释见[2]。USB传递声明了4个类型：控制传递，中断传递，等时传递，批量传递。每个传递为不同的设备需求而设，关于它们的解释见[2]。

我们的设备使用控制传递。这种传递方式是专门用于设置设备，但也可以用于普通用途。每一个USB设备都必须实现控制传递，这种方式用于配置连接的设备 (从设备获得信息，设置设备地址,等)。关于传递控制的详细内容见[2] 和[1]。每个控制传输包括几个阶段：设置阶段，数据阶段和状态阶段。
数据分成小包在 USB 中传输，每个包只有几个字节。包的大小由设备决定，但受限于规范值。对于低速设备，包大小为 8 字节。在一次 USB 传输中，设备缓冲区必须能够接收 8 字节长的包+开始和结束域。基于硬件的 USB 接收器，传输的不同部分会自动解码，当完整的信息分配给确定的设备时，该设备会被通知。作为固件实现，当完整的信息被缓冲区接收之后，固件必须解码 USB 信息。这给我们提出了要求和限制：设备必须有一个能够接收完整的 USB 信息的缓冲区，及一个发送 USB 信息的缓冲区（准备发送的数据)，并管理信息的解码和检查。另外，更要求固件能够以快速和精确的同步来接收（物理引脚到缓冲区）和发送（缓冲区到物理引脚)。所有这些能力都受限于微控制器的资源（速度和程序/数据存储

容量），因此，固件必须加倍优化。同样原因，微控制器计算功率要非常接近最小要求，所有固件必须用 ASM 编写。

# 硬件实现

微控制器连接USB总线的原理图 见*图8和图9*是。这份原理图是用作USB转换到RS232的。 因此实现特定的功能来直接控制引脚和读写EEPROM 。

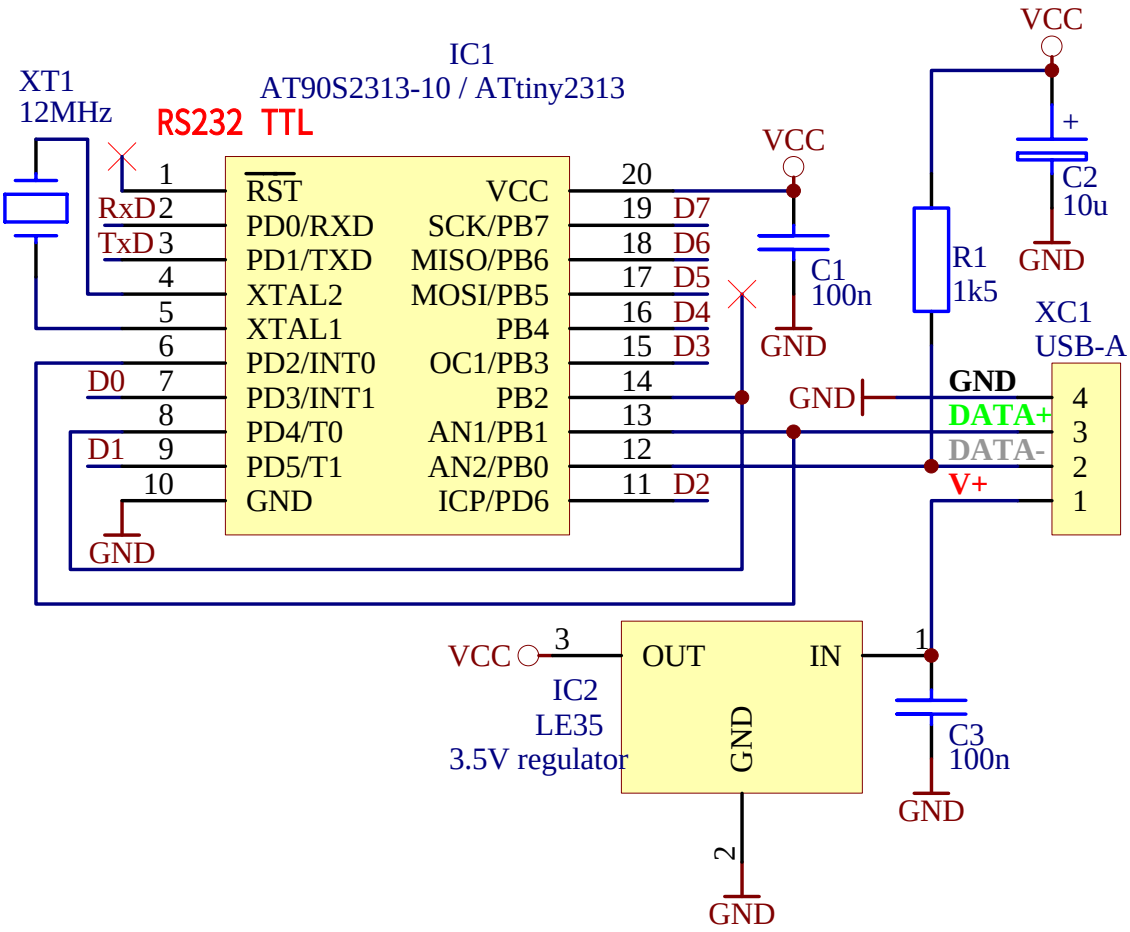



图8. 使用ATtiny2313的USB接口 (as USB to RS232 converter with 32 byte FIFO + 8-bit I/O control + 128 bytes EEPROM)

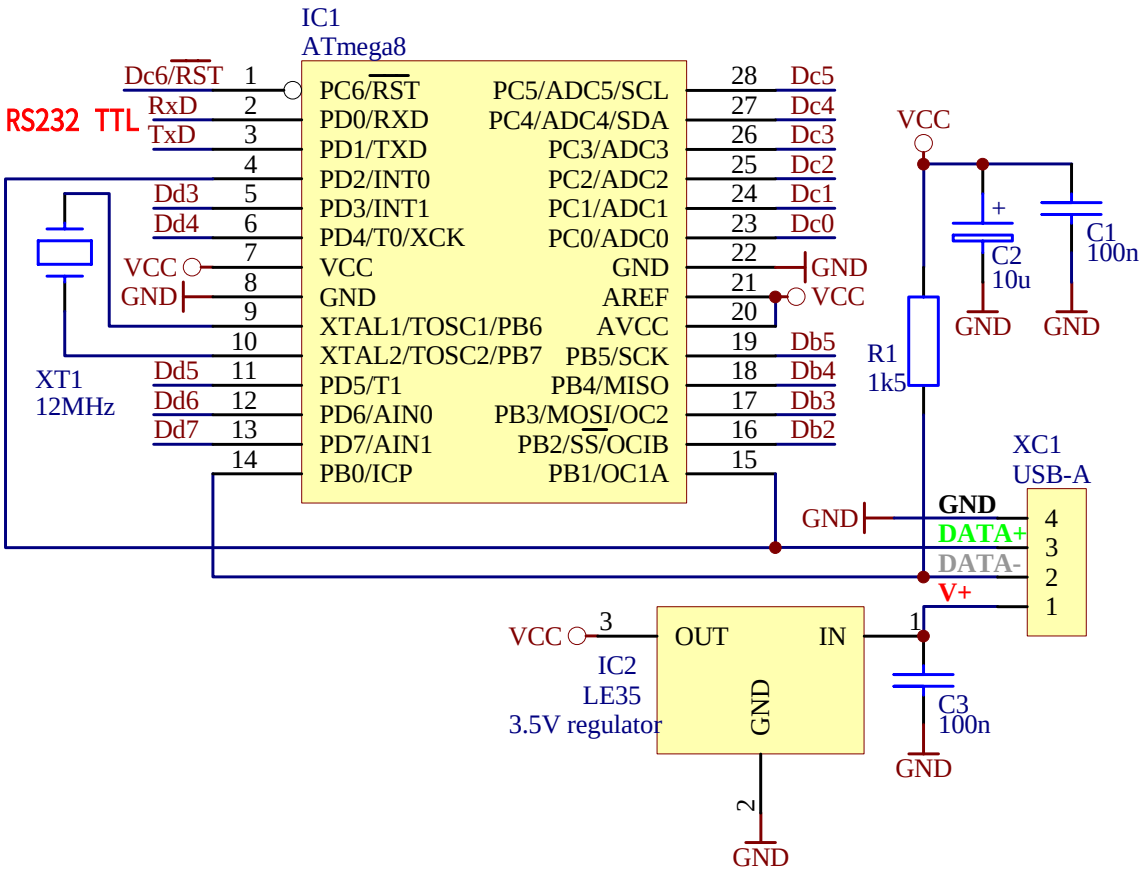



图9. 使用ATmega8的USB接口 (as USB to RS232 converter with 800 byte FIFO + EEPROM + I/O control + EEPROM)

USB 数据线，DATA- 和 DATA+连接到 AVR 的引脚 PB0 和 PB1。这个连接不能改变，因为制作固件使用 AVR 的一个技巧来快速接受信号：数据线上的信号是从 LSB（PB0）右移到进位的信号，接收寄存器从数据线收集位。PB1 用作输入信号，因为 8 脚应用的 AT90S2323 的这只引脚能够用作外部中断 INT0 (没有必要另外连接到 INT0 – AVR 的 8 脚版本是可能的最少引脚)。对于其它的 AVRs，如果我们想要使用不同的 AVR 微控制器，又不想改写固件的话，则需要从外部连接 DATA+到 INT0 引脚。

为了 USB 设备的正确连接和通信，AVR 用于低速 USB 设备时，必须连接一个 1.5k 的上接电阻到 DATA-。

另外的元件仅为微控制器的正常工作提供功能：晶振为时钟源，电容为电源滤波。

这几个小元件足够组成一个 USB 设备，能够通过 USB 界面与 PC 通信。这是一个简单便宜的解决方案。增加一些元件可以扩展设备的功能。如果要接收 IR 信号，可以加一个 TSOP1738 红外传感器，若想设备做成一个 USB 到 RS232 的转换器，则要加一块 MAX232 做 TTL 到 RS232 的电平转换。若想控制 LED 二极管或显示，可能直接或通过电阻连接到 I/O 引脚。

## 软件实现

固件实现所有的 USB 协议接收和解码。固件首先接收一个 USB 包的位流到内部缓冲区。通过外部中断 INT0 开始接收，必须注意同步块，在接收过程中，只检查包结束信号（只检查 EOP)。这是因为 USB

的数据传输速。成功接收后，固件进行解码和分析。首先，它判断包是否按照设备地址发送给设备的。每个 USB 传输都会发送一个地址以指出下一个传输是发送给哪个设备的。解码 USB 地址必须非常快，因为设备必须回答一个 ACK 包给 USB 主机，当它根据 USB 地址确认一个有效 USB 包时。所以，这是 USB 应答的临界段。

接收位流之后，我们能够从输入缓冲区得到位填充的 NRZI 编码队列。在解码过程中，我们首先删除位填充然后是 NRZI 编码。所有这些改变都在另一个缓冲区进行（接收缓冲区的复件），因此，在解码第一个包时可以接收新的包。在这里，解码速度不是很重要，因为设备延迟回答，但是，在解码过程中当主机要求回答时，设备必须立即回答 NAK，这样主机就知道设备没有准备好。因为这个原因，固件在解码过程中必须能够接收包，解码不管设备的事务是否结束，当有一些解码过程时，发送 NAK 包，主机会再次询问。因件同样解码 USB 主要事务和执行请求的动作（例如 :发送字符到 RS232 线并等待传送完成），并准备通信应答。在这过程中，设备被主机发送的包中断，通常 IN 包会从设备得到回答，对于这些包，设备必须回答 NAK 应答包。当回答就绪并且设备已完成请求动作时，在回答通过位流传送之前，回答必须加入 CRC 域，然后 NRZI 编码并进行位填充。现在，当主机要求回答时，我们可以依照 USB 规范传送这些位流到数据上（从同步块到 EOP）。

# 固件说明

下面我们介绍固件的主要部分。固件分为以下几块：中断例程，解码例程（NRZI 解码，位填充的删除/插入...），USB 接收，USB 发送，请求动作解码，执行请求的特定动作。

用户可以向固件增加自己的函数。在固件代码中可以找到一些怎样写指定功能的例子，用户可以扩展内建的函数来支持新设备。例如，依照内建函数，可以直接控制引脚以支持 TWI 功能。

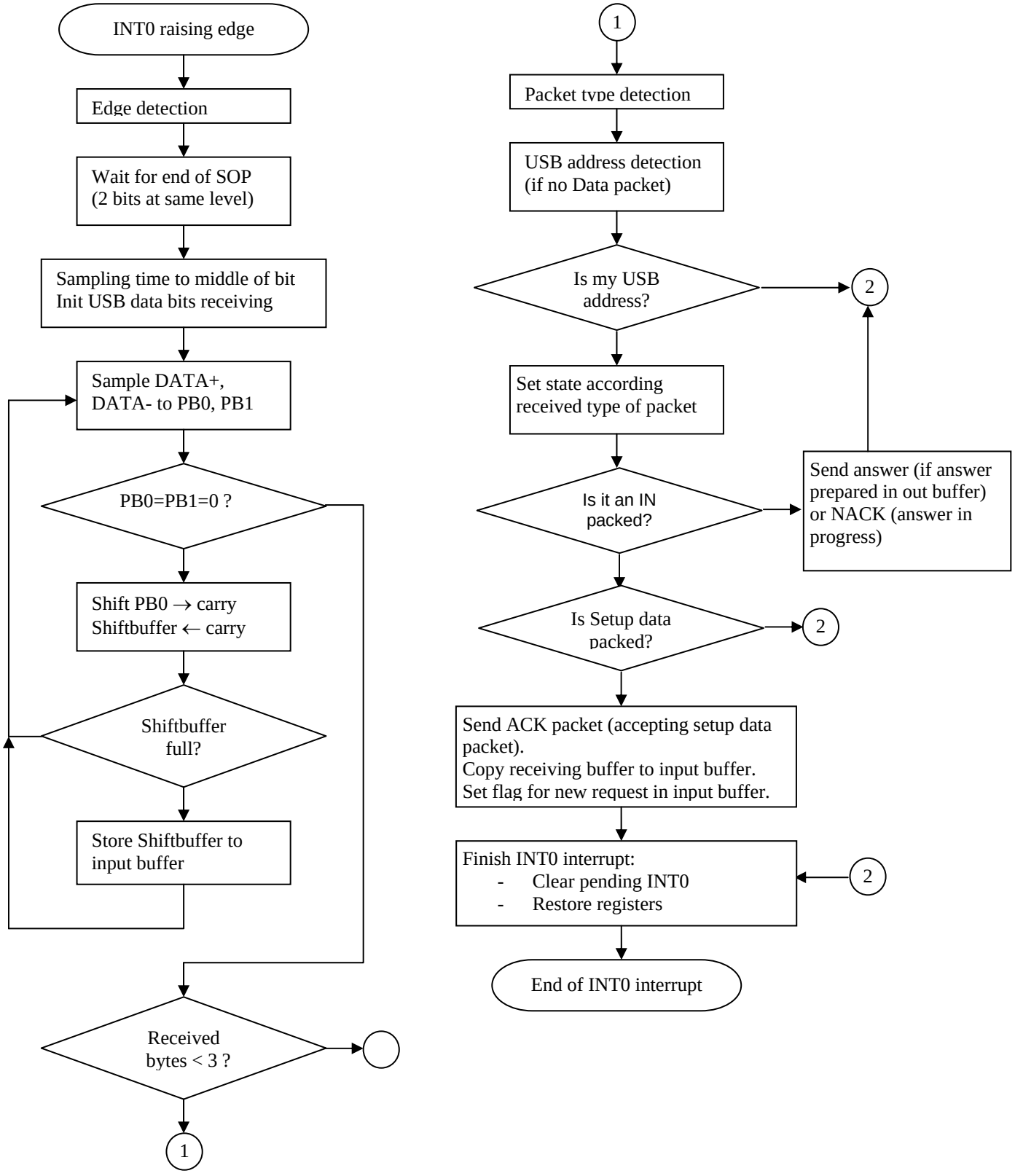



图6． 接收事务流程图

## *“EXT_INT0”中断服务例程*

固件运行时，外部中断 0 总是活动的。这个例程开始接收 USB 串行数据（另一个名字为“ USB 接收 ”）。当 INT0 引脚检测到一个上升沿时触发一个中断发生（一个上升沿标志 USB 包的同步块开始，见图 4）。接着激活 USB 接收例程。

首先，数据取样必须同步到位宽的中间，可以根据同步块 (方波信号)做到。因为位持续时间为 XTAL

时钟的 8 个周期，中断可以延迟（+/-4 个周期），同步块的边沿必须小心。同步块结束和数据位的开始可以依照图 4 的同步包的最后 2 个同电平的位来检测。

然后，开始数据取样。取样在位的中间进行。因为数据速率是 1.5Mbit/s(1.5MHz)，微控制器速度是 12MHz，用于数据位的取样只有 8 个周期，储存数据到缓冲区字节，移位缓冲区字节，检查是否接收整个字节，储存这些字节到 SRAM，检测 EOP。这可能是固件最重要的部分，每件事情都必须有准确的时间同步。当接收一个完整的 USB 包后，必须进行包解码。首先，必须快速检测包类型（SETUP, IN, OUT, DATA）和接收 USB 地址。快速解码必须在中断服务例程内，因为，在接收到 USB 包之后的应答要求非常快速（当接收到一个带设备地址的包时，设备必须应答一个 ACK 握手包，当包是发送给该设备时必须回答一个 NAK 包，且当没有准备好回答时）。

在接收例程的尾部（ACK/NAK 握手包发送之后），取样数据缓冲区的数据必须复制到另一个将要执行解码的缓冲区，这是为了空出接收缓冲区来接收新的包。

在接收包类型已解码和设置相应的标志值时，这个标志在主循环里检测，根据它的值，会执行相应的动作或准备好相应的回答，这里对微控制器的速度需求已不重要。

在所有固件例程里必须保证 INT0 有非常快的启动时间，因此，不充许禁止中断及在其它中断执行时也不充许（例如：串行线接收中断）INT0 中断必须充许。在 INT0 中断例程中快速接收是非常重要的，为了速度和精确的时间需要对固件进行优化。在中断例程中寄存器备份的优化是一个重要的问题。

## 主程序循环

主程序循环非常简单。只需要检查动作标志：当接收到一些数据时要执行什么动作。额外地，它也会检查，无论USB界面是否复位（长时间内两条数据线都处于低电平），如果这样，设备重新初始化当有一些事情要做时（动作标志活动），相应的动作会调用：解码包中的NRZI编码，删除位填充，在发送缓冲区中准备请求的回答（包括位填充和NRZI编码）。当一个标志激活时，回答也准备好发送。在接收例程回答IN包时物理输出缓冲区传送到USB线。

使用固件子程序的简要介绍

下面，简要介绍固件的例程和它们的目的。

***Reset:***

初始化 AVR 微控制器的资源：堆栈，串行线，USB 缓冲区，中断,,,

***Main:***

主循环。检查动作标志，如果标志置位，执行请求的动作。另外，这个例程检查数据线上的 USB 复位，如果这是一个产器则重新初始化 USB 微控制器界面。

***Int0Handler:***

这是 INT0 外部中断的中断服务例程。主要的接收/发送引擎，从 USB 数据线模仿，储存数据到缓冲，判断 USB 包的所有者（USB 地址），包确认，发送回答到 USB 主机。USB 引擎的主要基础。

***MyNewUSBAddress:***

当请求改变 USB 地址时由 INT0 接收例程调用，改变地址，并且为了在接收 USB 包时进行快速地址解码，它被编码为相等的 NRZI 编码。

***FinishReceiving:***

从 USB 接收包复制编码数据到解码包（为 NRZI 和位填充解码）

***USB reset:***

初始化 USB 界面到默认值（与开启电源后的状态一样）

***SendPreparedUSBAnswer:***

发送输出缓冲区的内容到 USB 线。传送过程执行 NRZI 编码和位填充。包以 EOP 结束。

***ToggleDATAPID:***

在 DATA0 和 DATA1 PID 之间绑定 DATAPID 包标识符。每个 USB 规范都要求在传送过程中绑定。

***ComposeZeroDATA1PIDAnswer:***

组成 0 回答的传送。0 回答不包含数据，在一些情况下回答，当设备上没有可用的额外数据时。

***InitACKBufffer:***

在 RAM 中生成包含 ACK 数据的缓冲区（ACK 握手包）。在应答时此缓冲区频繁地发送，因此它总是保存在存储器里。

***SendACK:***

发送 ACK 包到 USB 数据线。

***InitNAKBufffer:***

在 RAM 中生成包含 NAK 数据的缓冲区（NAK 握手包）。在应答时此缓冲区频繁地发送，因此它总是保存在存储器里。

***SendNAK:***

发送 NAK 包到 USB 数据线。

***ComposeSTALL:***

在 RAM 中生成包含 STALL 数据的缓冲区（STALL 握手包）。在应答时此缓冲区频繁地发送，因此它总是保存在存储器里。

***DecodeNRZI:***

执行 NRZI 解码。缓冲区从 USB 数据线得到的数据是 NRZI 编码的。这个例程从数据中移除 NRZI 编码。

***BitStuff:***

在接收 USB 数据中删除/添加位填充。位填充是主机硬件依照 USB 规范为了数据采样同步而添加的。这个例程去除接收数据中的位填充，发送数据时添加位填充。

***ShiftInsertBuffer:***

执行位填充增加时的辅助例程。添加一位到输出数据缓冲区会增加缓冲区的长度。缓冲区的剩余部分移出。

***ShiftDeleteBuffer:***

执行位填充删除时的辅助例程。从输出缓冲删除一位时会减少缓冲区长度。缓冲区的剩余部分移入。

***MirrorInBufferBytes:***

以字节为单位交换顺序，因为从 USB 线接收到缓冲区的数据是反序的（LSB/MSB）。

***CheckCRCIn:***

对接收的数据包进行 CRC 校验。USB 包添加 CRC 来检测数据的损坏。

***AddCRCOut:***

给输出数据包添加 CRC 域。依照 USB 规范提供的 USB 域计算 CRC。

***CheckCRC:***

检查和添加 CRC 的辅助例程。

***LoadDescriptorFromROM:***

从 ROM 加载数据到 USB 输出缓冲区（用于 USB 应答）。

***LoadDescriptorFromROMZeroInsert:***

从 ROM 加载数据到 USB 输出缓冲区（用于 USB 应答）但所有偶数字节都添加为 0。当一个字符串被作为 UNICODE 格式请求时（节约 ROM）。

***LoadDescriptorFromSRAM:***

从 RAM 加载数据到 USB 输出缓冲区（用于 USB 应答）。

***LoadDescriptorFromEEPROM:***

从 EEPROM 加载数据到 USB 输出缓冲区（用于 USB 应答）。

***LoadXXXDescriptor:***

选择回答的源位置：ROM, RAM 或 EEPROM。

***PrepareUSBOutAnswer:***

根据 USB 主机的请求在输出缓冲区准备 USB 应答，并执行请求的动作。添加位填充到应答中。

***PrepareUSBAnswer:***

执行请求动作和准备相应回答的主例程。此例程首先判断哪个动作要执行—从输入数据包得到函数编号—接着执行函数。函数参数定位于输入数据包。

例程分为两个大的部分：

- 标准请求
- 厂家指定的请求

标准请求是必须的，它在 USB 规范中描述（SET_ADDRESS, GET_DESCRIPTOR, …）。

厂家指定的请求要求能够获得厂家指定的数据（在 USB 控制传输中）。这个 AVR 设备与主机的通信使用 USB 控制传输。开发者可以添加他们自己的函数到这里来扩展设备的功能。源代码中不同文档的内建函数可以作为添加定制函数的模板使用。

标准 USB 函数（标准请求）：

***ComposeGET_STATUS:***

***ComposeCLEAR_FEATURE:***

***ComposeSET_FEATURE:***

***ComposeSET_ADDRESS:***

***ComposeGET_DESCRIPTOR:***

***ComposeSET_DESCRIPTOR:***

***ComposeGET_CONFIGURATION:***

***ComposeSET_CONFIGURATION;***

***ComposeGET_INTERFACE;***

***ComposeSET_INTERFACE:***

***ComposeSYNCH_FRAME;***

厂家 USB 函数（厂家请求）：

***DoSetInfraBufferEmpty:***

***DoGetInfraCode:***

***DoSetDataPortDirection:***

***DoGetDataPortDirection:***

***DoSetOutDataPort:***

***DoGetOutDataPort:***

***DoGetInDataPort:***

***DoEEPROMRead:***

***DoEEPROMWrite:***

***DoRS232Send:***

***DoRS232Read:***

***DoSetRS232Baud:***

***DoGetRS232Baud:***

***DoGetRS232Buffer:***

***DoSetRS232DataBits:***

***DoGetRS232DataBits:***

***DoSetRS232Parity:***

***DoGetRS232Parity:***

***DoSetRS232StopBits:***

***DoGetRS232StopBits:***

数据结构（USB 描述串)：

***DeviceDescriptor:***

***ConfigDescriptor:***

***LangIDStringDescriptor:***

***VendorStringDescriptor:***

***DevNameStringDescriptor:***

## USB 主机输入信息格式

基于上面的规定，我们的USB设备使用USB控制传输。USB规范定义这种传输类型使用的数据格式，介绍见usb-in-a-nutshell.pdf [2]第13页（控制传输)。这份文档详细解释了控制传输怎样工作，因此可

以找到我们的设备怎样与主机通信。这个AVR设备使用IN控制端点。在15页[2]有一个比较好的数据通信例子。主机与AVR设备的通信是依照这个例子来做的。

在实际的控制传输中，传输中DATA0/1域的格式会讨论到。控制传输定义一个请求在它的设置阶段，长8字节。它的格式在[2]第26页描述（设置包）。这里列表详细描述每个字节的含义。下列对我们的目的很重要。

标准设置包用来侦测和配置加电后的设备。这个包在*bmRequestType*域使用标准请求类型（位D6-D5=0）。在USB规范可以找到下一个域的含义（*bRequest, wValue, wIndex, wLength*）。在[2]第27~30页有关于它们的解释（标准请求）。

所有设置包都是 8 字节，应用和描述见下表。

| 偏移 | 域 | 大小<br>(字节) | 值 | 描述 |
|---|---|---|---|---|
| 0 | *bmRequestType* | 1 | 位映射 | 请求特性<br>D7 数据方向<br>0 = 主机到设备<br>1 = 设备到主机<br>D6..5 类型<br>0 = 标准<br>1 = Class<br>2 = 厂家<br>3 = 保留<br>D4..0 Recipient<br>0 = 设备<br>1 = 界面<br>2 = 端终<br>3 = 其它<br>4..31 = 保留 |
| 1 | *bRequest* | 1 | 值 | 指定请求 |
| 2 | *wValue* | 2 | 值 | 字，根据请求而变 |
| 4 | *wIndex* | 2 | 索引或偏移 | 字，根据请求而变—习惯用作一个索引或偏移 |
| 6 | *wLength* | 2 | 计数 | 如果是一个数据段指出传送的字节数 |

**表1 ： 标准设置包域（控制传输）**

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 00000000B<br>00000001B<br>00000010B | CLEAR_FEATURE | 特征选择 | 0<br>界面<br>端终 | 0 | 无 |
| 10000000B | GET_CONFIGURATION | 0 | 0 | 1 | 配置值 |

| | | | | | |
|---|---|---|---|---|---|
| 10000000B | GET_DESCRIPTOR | 描述类型或描述索引 | 0或代码ID | 描述长度 | 描述串 |
| 10000001B | GET_INTERFACE | 0 | 界面 | 1 | 交替界面 |
| 10000000B<br>10000001B<br>10000010B | GET_STATUS | 0 | 0<br>界面<br>端终 | 2 | 设备，界面或端终状态 |
| 00000000B | SET_ADDRESS | 设置地址 | 0 | 0 | 无 |
| 00000000B | SET_CONFIGURATION | 配置值 | 0 | 0 | 无 |
| 00000000B | SET_DESCRIPTOR | 描述类型或描述索引 | 0或代码ID | 描述串长度 | 描述串 |
| 00000000B<br>00000001B<br>00000010B | SET_FEATURE | 特征选择 | 0<br>界面<br>端终 | 0 | 无 |
| 00000001B | SET_INTERFACE | 交替设置 | 界面 | 0 | 无 |
| 10000010B | SYNCH_FRAME | 0 | 端终 | 2 | 帧号码 |

**表2：标准设备请求**

| bmRequestType | bRequest<br>(函数名) | bRequest<br>(函数号码) | wValue<br>(参数 1) | wIndex<br>(参数 2) | wLength | Data |
|---|---|---|---|---|---|---|
| 110xxxxxB | FNCNumberDoSetInfraBufferEmpty | 1 | 无 | 无 | 1 | 状态 |
| 110xxxxxB | FNCNumberDoGetInfraCode | 2 | 无 | 无 | 1 | 状态 |
| 110xxxxxB | FNCNumberDoSetDataPortDirection | 3 | DDRB<br>DDRC | DDRD<br>用户端口号 | 1 | 状态 |
| 110xxxxxB | FNCNumberDoGetDataPortDirection | 4 | 无 | 无 | 3 | DDRB<br>DDRC<br>DDRD |

| | | | | | | |
|---|---|---|---|---|---|---|
| 110xxxxxB | FNCNumberDoSetOutDataPort | 5 | PORTB<br>PORTC | PORTD<br>用户端口号 | 1 | 状态 |
| 110xxxxxB | FNCNumberDoGetOutDataPort | 6 | 无 | 无 | 3 | PORTB<br>PORTC<br>PORTD |
| 110xxxxxB | FNCNumberDoGetInDataPort | 7 | 无 | 无 | 3 | PINB<br>PINC<br>PIND |
| 110xxxxxB | FNCNumberDoEEPROMRead | 8 | 地址 | 无 | 长度 | EEPROM 字节 |
| 110xxxxxB | FNCNumberDoEEPROMWrite | 9 | 地址 | EEPROM 值 | 1 | 状态 |
| 110xxxxxB | FNCNumberDoRS232Send | 10 | RS232 字节值 | 无 | 1 | 状态 |
| 110xxxxxB | FNCNumberDoRS232Read | 11 | 无 | 无 | 2 | 状态 |
| 110xxxxxB | FNCNumberDoSetRS232Baud | 12 | 波特率低字节 Lo | 波特率高字节 Hi | 1 | 状态 |
| 110xxxxxB | FNCNumberDoGetRS232Baud | 13 | 无 | 无 | 2 | 速率 |
| 110xxxxxB | FNCNumberDoGetRS232Buffer | 14 | 无 | 无 | 长度 | RS232 字串来自 FIFO |
| 110xxxxxB | FNCNumberDoSetRS232DataBits | 15 | 数据位值 | 无 | 1 | 状态 |
| 110xxxxxB | FNCNumberDoGetRS232DataBits | 16 | 无 | 无 | 1 | 数据位值 |
| 110xxxxxB | FNCNumberDoSetRS232Parity | 17 | 奇偶位值 | 无 | 1 | 状态 |
| 110xxxxxB | FNCNumberDoGetRS232Parity | 18 | 无 | 无 | 1 | 奇偶位值 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 110xxxxxB | FNCNumberDoSetRS232StopBits | 19 | 停止位值 | 无 | 1 | 状态 |
| 110xxxxxB | FNCNumberDoGetRS232StopBits | 20 | 无 | 无 | 1 | 停止位值 |

**表3：固件函数调用使用的厂家设备请求**

注意：*FNCNumberXXX* 值定义在 DLL 中。

控制传输用于用户通信，在固件中用特定的函数实现，这很好。*bmRequestType*域（位D6-D5=2）用作厂家请求类型。在这个方案中，所有有效的域(*bRequest, wValue, wIndex*)可以根据设计者的目的进行修改。在我们的实现中，*bRequest*域用来表示函数序号，下一个域放函数的参数。第一个参数放在*wValue*，第二个放在*wIndex*。*wValue*域表示EEPROM地址，写到EEPROM的数据放在*wIndex*域。根据上述，我们得到下列的函数：

一个实现写EEPROM的例子。*bRequest* = 9选择函数序号。根据上述，我们得到下列的函数：*EEPROMWrite(Address, Value)*。

如果需要更多的用户功能，只需要增加需要的函数号码和函数体到固件中。可以从固件内建函数（见源代码）中吸取技巧。

USB主机总是使用IN控制传输与设备进行通信。主机使用上面定义的格式（函数号码及参数）发送8字节的IN数据包到设备，设备回答请求的数据。在某些场合，固件限制了应答数据的长度<=255字节，但主要的限制是在主机端的设备驱动程序。这个驱动支持8字节长的回答（厂家请求类型）。

## 定制固件

用户（设备开发者）为了扩展设备特性可以往固件增加新的函数。

在固件中准备了3个怎样增加用户函数的例子：DoUserFunctionX (X=0,1,2)。依照这些例子可以增加类似的扩展函数。函数内容依赖于设备的需求。

固件可以定制设备名—在计算机端显露的名称。这个名称定位于固件中，可以改为任意的字符串。但是，推荐这些名称总是使用USB PID（产品ID）+VID（厂家ID），以便系统能够正确确认。

对给定的设备类型，VID+PID必须是唯一的。因此建议如果设备改变功能，则要修改PID和/或VID。厂家ID依据USB设备的厂家并必须由USB组织指定（更多的信息见[1]）。每个厂家有自己的ID，因此这些值不能改为其它未指派的值。但是，产品ID是厂家为了识别不同的设备而选择的。这份应用说明设置VID=0x03EB，PID=0x21FF，这是Atmel的VID，不要在你的商业系统中使用这个VID。

# PC 软件

为了与设备通信，在PC端需要一些软件的支持。软件分为三层：

1) 设备驱动：与设备的底层通信，安装到操作系统（Windows98/ME/NT/XP）。
2) DLL库：卦装与设备驱动程序通信的函数。DLL简化了用户应用程序访问设备函数。它包含了一些设备和操作系统相关的函数（例如：线程、缓冲等）。
3) 用户应用程序：使用户与设备有一个友好的界面。只从DLL库调用函数。

## 设备驱动程序和安装文件

首先，我们连接USB设备到计算机的USB接口，操作系统会检测到设备并要求驱动文件。这会调用设备安装。安装过程需要但并不只是安装设备驱动程序，但是，安装脚本中有安装步骤的描述。

这份文档中设备描述的设备驱动程序是用Windows2000 DDK写的。USB驱动开发是基于DDK包含的一个例子—***IsoUsb***。根据我们的用途作了修改—AVR USB设备通信。在源代码中，IOCTL 通信作了部分扩展/增加，因为我们的设备通过这些IOCTL调用与计算机通信。为减少驱动代码，删除了没有用到的部分（读和写例程）。驱动程序名为“ AVR309.sys ”，它的工作是向USB设备发送命令（控制IN传输）。这个驱动可以在除WIN95外的所有WIN32平台上工作。

在安装设备时用到的安装脚本写在一个INF文件中。在INF文件中，各种不同的安装步骤都有描述。文件AVR309.inf 是用记事本建立的。操作系统安装时需要这个文件。在安装过程中，驱动文件会复制到系统中并需要系统的更改生效。对不同的应用软件，INF文件确保安装DLL库到系统路径。

设备安装需要三个文件：INF文件“ AVR309.inf ”,驱动程序“ AVR309.sys ”,DLL库“ AVR309.dll ”。

## DLL 库

DLL库与设备驱动程序通信并且所有的设备函数都在库中实现。这样，编写端终用户程序非常简单。DLL库确保对设备的访问是排外的（串行化设备访问），包含接收RS232数据的系统缓冲区，并为设备建立单独的系统线程来读RS232数据缓冲区。

DLL的串行化确保在给定时间内只有一个应用程序/线程能够与设备通信。这是必须的，因为可能会混合同时访问的不同程序的询问和回答。

接收RS232数据的系统缓冲区确保从设备的RS232线上接收的数据存储到一个所有的应用程序共用的缓冲区。这样，从设备接收到的数据会发送到所有应用程序。这里应用程序接收不完全的数据并不危险，因为其它的应用程序之前已经读取了数据。

对于所有的应用程序，只有一个系统线程，并周期性地向设备请求RS232数据。线程将接收到的数据存储到系统缓冲区。仅使用一个系统缓冲区的方案确保少占用CPU时间（与每个应用程序拥有自己的线程比较）并简化了存储数据到系统缓冲区。

所有设备函数都在DLL库中定义，并有一个友好的用户接口：不是函数号码和参数，而是整齐的函数名和参数。一些函数比较复杂，例如讯RS232缓冲区数据的函数。这里，终端用户开发者只使用DLL界面就可以快速地写应用程序。这里不需要学习底层的设备函数，DLL库硬件层从应用程序设计层分离出来了。

DLL函数接口在下面描述。描述使用3种常用的程序设计语言编写： Borland Delphi, C++ (Borland or Microsoft) and Visual Basic。这些函数的详细描述可以在文件AVR309_DLL_help.htm中找到。

## 终端用户应用程序

终端用户应用程序仅使用DLL库函数与设备通信。主要是为了有一个友好的GUI（图形用户界面）。

应用程序设计者使用DLL库写他们自己的应用程序。所有源代码都是开放的，在公布的项目中能找到例子。很多应用程序可以从使用这个例子开始写，可以使用几种编程语言(Delphi, C++, Visual Basic)。

这里包括了一个终端应用程序的例子名为“ AVR309demo.exe ”,这个软件仅是一个如何使用DLL库函数的例子。源代码可以作为其它应用程序的模板使用。

# UART 速率误差讨论

微控制器使用12MHz时钟，因为USB采样。但是使用这个时钟值产生的波特率相对标准波特率包含了一些误差。高的时钟会减少误差。产生的波特率最高可以接受的误差是4%：因为最大误差是半位的持续时间（0.5），包最大时间是12位=1开始位+8数据位+1奇偶位+2停止位。所以误差为：0.5/12*100%=4.1%。

DLL函数自动检查这个误差并设置微控制器的波特率如果误差小于4%时（对不支持的波特率返回错误）。

下面的表给出在12MHz时钟下相对标准波特率的误差。

| 标准波特率 | AVR波特主 | 误差 [%] |
|---|---|---|
| 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 |

**表 1: AVR UART 波特率误差 (12MHz 时钟)**

*Function prototype in "AVR309.dll" library:*

**Delphi:**

```
const
AVR309DLL= 'AVR309.dll';
//return values from AVR309DLL functions:
NO_ERROR = 0;
DEVICE_NOT_PRESENT = 1;
NO_DATA_AVAILABLE = 2;
INVALID_BAUDRATE = 3;
OVERRUN_ERROR = 4;
INVALID_DATABITS = 5;
INVALID_PARITY = 6;
INVALID_STOPBITS = 7;

function DoGetInfraCode(var TimeCodeDiagram:array of byte; var DiagramLength:integer):integer; stdcall external AVR309DLL name 'DoGetInfraCode';
function DoSetDataPortDirection(DirectionByte:byte):integer; stdcall external AVR309DLL name 'DoSetDataPortDirection';
function DoGetDataPortDirection(var DataDirectionByte:byte):integer; stdcall external AVR309DLL name 'DoGetDataPortDirection';
function DoSetOutDataPort(DataOutByte:byte):integer; stdcall external AVR309DLL name 'DoSetOutDataPort';
function DoGetOutDataPort(var DataOutByte:byte):integer; stdcall external AVR309DLL name 'DoGetOutDataPort';
function DoGetInDataPort(var DataInByte:byte):integer; stdcall external AVR309DLL name 'DoGetInDataPort';
function DoSetDataPortDirections(DirectionByteB, DirectionByteC, DirectionByteD,
```

```
UsedPorts:byte):integer; stdcall external AVR309DLL name 'DoSetDataPortDirections';
function DoGetDataPortDirections(var DataDirectionByteB, DirectionByteC, DirectionByteD,
UsedPorts:byte):integer; stdcall external AVR309DLL name 'DoGetDataPortDirections';
function DoSetOutDataPorts(DataOutByteB, DataOutByteC, DataOutByteD, UsedPorts:byte):integer;
stdcall external AVR309DLL name 'DoSetOutDataPorts';
function DoGetOutDataPorts(var DataOutByteB, DataOutByteC, DataOutByteD, UsedPorts:byte):integer;
stdcall external AVR309DLL name 'DoGetOutDataPorts';
function DoGetInDataPorts(var DataInByteB, DataInByteC, DataInByteD, UsedPorts:byte):integer; stdcall
external AVR309DLL name 'DoGetInDataPorts';

function DoEEPROMRead(Address:word; var DataInByte:byte):integer; stdcall external AVR309DLL name
'DoEEPROMRead';
function DoEEPROMWrite(Address:word; DataOutByte:byte):integer; stdcall external AVR309DLL name
'DoEEPROMWrite';
function DoRS232Send(DataOutByte:byte):integer; stdcall external AVR309DLL name 'DoRS232Send';
function DoRS232Read(var DataInByte:byte):integer; stdcall external AVR309DLL name 'DoRS232Read';
function DoSetRS232Baud(BaudRate:integer):integer; stdcall external AVR309DLL name
'DoSetRS232Baud';
function DoGetRS232Baud(var BaudRate:integer):integer; stdcall external AVR309DLL name
'DoGetRS232Baud';
function DoGetRS232Buffer(var RS232Buffer:array of byte; var RS232BufferLength:integer):integer;
stdcall external AVR309DLL name 'DoGetRS232Buffer';
function DoRS232BufferSend(var RS232Buffer:array of byte; var RS232BufferLength:integer):integer;
stdcall external AVR309DLL name 'DoRS232BufferSend';
function DoSetRS232DataBits(DataBits:byte):integer; stdcall external AVR309DLL name
'DoSetRS232DataBits';
function DoGetRS232DataBits(var DataBits:byte):integer; stdcall external AVR309DLL name
'DoGetRS232DataBits';
function DoSetRS232Parity(Parity:byte):integer; stdcall external AVR309DLL name 'DoSetRS232Parity';
function DoGetRS232Parity(var Parity:byte):integer; stdcall external AVR309DLL name 'DoGetRS232Parity';
function DoSetRS232StopBits(StopBits:byte):integer; stdcall external AVR309DLL name
'DoSetRS232StopBits';
function DoGetRS232StopBits(var StopBits:byte):integer; stdcall external AVR309DLL name
'DoGetRS232StopBits';
```

---

**C++ Builder / Microsoft Visual C++:**

```
#ifdef __cplusplus
extern "C" {
#endif

#define AVR309DLL "AVR309.dll";
//return values from AVR309DLL functions:
#define NO_ERROR 0;
#define DEVICE_NOT_PRESENT 1;
#define NO_DATA_AVAILABLE 2;
#define INVALID_BAUDRATE 3;
#define OVERRUN_ERROR 4;
#define INVALID_DATABITS 5;
#define INVALID_PARITY 6;
#define INVALID_STOPBITS 7;

int __stdcall DoGetInfraCode(uchar * TimeCodeDiagram, int DummyInt, int * DiagramLength);
int __stdcall DoSetDataPortDirection(uchar DirectionByte);
int __stdcall DoGetDataPortDirection(uchar * DataDirectionByte);
```

```
int __stdcall DoSetOutDataPort(uchar DataOutByte);
int __stdcall DoGetOutDataPort(uchar * DataOutByte);
int __stdcall DoGetInDataPort(uchar * DataInByte);
int __stdcall DoSetDataPortDirections(uchar DirectionByteB, uchar DirectionByte, uchar DirectionByte,
uchar UsedPorts);
int __stdcall DoGetDataPortDirections(uchar * DataDirectionByteB, uchar * DataDirectionByteC, uchar *
DataDirectionByteD, uchar * UsedPorts);
int __stdcall DoSetOutDataPorts(uchar DataOutByteB, uchar DataOutByteC, uchar DataOutByteD, uchar
UsedPorts);
int __stdcall DoGetOutDataPorts(uchar * DataOutByteB, uchar * DataOutByteC, uchar * DataOutByteD,
uchar * UsedPorts);
int __stdcall DoGetInDataPorts(uchar * DataInByteB, uchar * DataInByteC, uchar * DataInByteD, uchar
* UsedPorts);
int __stdcall DoEEPROMRead(ushort Address, uchar * DataInByte);
int __stdcall DoEEPROMWrite(ushort Address, uchar DataOutByte);
int __stdcall DoRS232Send(uchar DataOutByte);
int __stdcall DoRS232Read(uchar * DataInByte);
int __stdcall DoSetRS232Baud(int BaudRate);
int __stdcall DoGetRS232Baud(int * BaudRate);
int __stdcall DoGetRS232Buffer(uchar * RS232Buffer, int DummyInt, int * RS232BufferLength);
int __stdcall DoRS232BufferSend(uchar * RS232Buffer, int DummyInt, int * RS232BufferLength);
int __stdcall DoSetRS232DataBits(uchar DataBits);
int __stdcall DoGetRS232DataBits(uchar * DataBits);
int __stdcall DoSetRS232Parity(uchar Parity);
int __stdcall DoGetRS232Parity(uchar * Parity);
int __stdcall DoSetRS232StopBits(uchar StopBits);
int __stdcall DoGetRS232StopBits(uchar * StopBits);

#ifdef __cplusplus
}
#endif
```

---

**Visual Basic:**

```
Public Const AVR309DLL="AVR309.dll";
'return values from AVR309DLL functions:
Public Const NO_ERROR = 0;
Public Const DEVICE_NOT_PRESENT = 1;
Public Const NO_DATA_AVAILABLE = 2;
Public Const INVALID_BAUDRATE = 3;
Public Const OVERRUN_ERROR = 4;
Public Const INVALID_DATABITS = 5;
Public Const INVALID_PARITY = 6;
Public Const INVALID_STOPBITS = 7;
Public Declare Function DoGetInfraCode Lib "AVR309.dll" (ByRef TimeCodeDiagram As Any, ByVal
DummyInt As Long, ByRef DiagramLength As Long) As Long
Public Declare Function DoSetDataPortDirection Lib "AVR309.dll" (ByVal DirectionByte As Byte) As Long
Public Declare Function DoGetDataPortDirection Lib "AVR309.dll" (ByRef DataDirectionByte As Byte) As
Long
Public Declare Function DoSetOutDataPort Lib "AVR309.dll" (ByVal DataOutByte As Byte) As Long
Public Declare Function DoGetOutDataPort Lib "AVR309.dll" (ByRef DataOutByte As Byte) As Long
Public Declare Function DoGetInDataPort Lib "AVR309.dll" (ByRef DataInByte As Byte) As Long
Public Declare Function DoSetDataPortDirections Lib "AVR309.dll" (ByVal DirectionByteB As Byte, ByVal
DirectionByteC As Byte, ByVal DirectionByteD As Byte, ByVal UsedPorts As Byte) As Long
Public Declare Function DoGetDataPortDirections Lib "AVR309.dll" (ByRef DataDirectionByteB As Byte,
```

ByRef ***DataDirectionByteC*** **As Byte**, ByRef ***DataDirectionByteD*** **As Byte**, ByRef ***UsedPorts*** **As Byte**) **As Long**
Public Declare Function **DoSetOutDataPorts** Lib "AVR309.dll" (ByVal ***DataOutByteB*** **As Byte**, ByVal ***DataOutByteC*** **As Byte**, ByVal ***DataOutByteD*** **As Byte**, ByVal ***UsedPorts*** **As Byte**) **As Long**
Public Declare Function **DoGetOutDataPorts** Lib "AVR309.dll" (ByRef ***DataOutByteB*** **As Byte**, ByRef ***DataOutByteC*** **As Byte**, ByRef ***DataOutByteD*** **As Byte**, ByRef ***UsedPorts*** **As Byte**) **As Long**
Public Declare Function **DoGetInDataPorts** Lib "AVR309.dll" (ByRef ***DataInByteB*** **As Byte**, ByRef ***DataInByteC*** **As Byte**, ByRef ***DataInByteD*** **As Byte**, ByRef ***UsedPorts*** **As Byte**) **As Long**
Public Declare Function **DoEEPROMRead** Lib "AVR309.dll" (ByVal ***Address*** **As Word**, ByRef ***DataInByte*** **As Byte**) **As Long**
Public Declare Function **DoEEPROMWrite** Lib "AVR309.dll" (ByVal ***Address*** **As Word**, ByVal ***DataOutByte*** **As Byte**) **As Long**
Public Declare Function **DoRS232Send** Lib "AVR309.dll" (ByVal ***DataOutByte*** **As Byte**) **As Long**
Public Declare Function **DoRS232Read** Lib "AVR309.dll" (ByRef ***DataInByte*** **As Byte**) **As Long**
Public Declare Function **DoSetRS232Baud** Lib "AVR309.dll" (ByVal ***BaudRate*** **As Long**) **As Long**
Public Declare Function **DoGetRS232Baud** Lib "AVR309.dll" (ByRef ***BaudRate*** **As Long**) **As Long**
Public Declare Function **DoGetRS232Buffer** Lib "AVR309.dll" (ByRef ***RS232Buffer*** **As Any**, ByVal ***DummyInt*** **As Long**, ByRef ***RS232BufferLength*** **As Long**) **As Long**
Public Declare Function **DoRS232BufferSend** Lib "AVR309.dll" (ByRef ***RS232Buffer*** **As Any**, ByVal ***DummyInt*** **As Long**, ByRef ***RS232BufferLength*** **As Long**) **As Long**
Public Declare Function **DoSetRS232DataBits** Lib "AVR309.dll" (***DataBits*** **As Byte**) **As Long**
Public Declare Function **DoGetRS232DataBits** Lib "AVR309.dll" (ByRef ***DataBits*** **As Byte**) **As Long**
Public Declare Function **DoSetRS232Parity** Lib "AVR309.dll" (***Parity*** **As Byte**) **As Long**
Public Declare Function **DoGetRS232Parity** Lib "AVR309.dll" (ByRef ***Parity*** **As Byte**) **As Long**
Public Declare Function **DoSetRS232StopBits** Lib "AVR309.dll" (***StopBits*** **As Byte**) **As Long**
Public Declare Function **DoGetRS232StopBits** Lib "AVR309.dll" (ByRef ***StopBits*** **As Byte**) **As Long**

## 附录 A: 使用ATmega8 AVR 的固件源代码

使用ATmega8 和 ATtiny2313/AT90S2313 AVR 微控制器的固件源代码在AVR Studio 4下编写。 源代码在文件USBtoRS232.asm 或语法高亮的文件USBtoRS232asm.pdf中。

## 附录 B: *AVR309.dll* 界面

库 AVR309.dll 使用Delphi3编写，因此它的源代码基于Object Pascal 语言。DLL界面库 (Delphi, C/C++ and Visual Basic.) (接口函数) "AVR309.dll" 在文件AVR309_DLL_help.htm中描述。

## 附录C: DLL 库 *AVR309.dll* 源代码

Whole source code of AVR309.dll 库的全部源代码使用Delphi3编写， 可以在文件AVR309.dpr (Delphi3 项目)中找到。

## 附录 D: 终端应用程序例子- *AVR309USBdemo.exe* (包括源代码)

使用DLL库的终端应用程序例子AVR309USBdemo.exe. 这个例子程序的全部源代码是Delphi3 项目： AVR309USBdemo.dpr。

## 使用的文档和资源


http://www.cesko.host.sk - authors web pages and various projects

***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

***AVR related resources:***

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

[11] **doc0839.pdf** – AT90S2313 datasheet (URL http://www.atmel.com/atmel/acrobat/doc0839.pdf )

[12] **doc2486.pdf** – ATmega8 datasheet

[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)

***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

Author:

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