

PROTOCOL FOR DRIVERLESS USB HID RELIABLE DATA TRANSFER

Samuel Kosolapov

ORT Braude Academic College of Engineering, Karmiel, Israel

Abstract

The modern PC, instead of coming equipped with outdated serial and parallel ports, has USB ports. Their well-known advantages include high communication speed and hot swapping. However, in many cases, use of USB ports requires installation of software drivers on the target PC, where installation is not always possible because of security requirements. One possible solution is the driverless USB Human Interface Device (HID) protocol. The goal of this contribution is to evaluate the feasibility and reliability of the expandable and reusable driverless USB HID platform on PC and embedded PIC devices supporting USB HID data transfer. Two data transfer protocols were implemented, evaluated, and compared. The “master sends request – slave responds with message” protocol was found more reliable than the “blocking loop” protocol.

Key words: USB, HID, communication protocol

1. INTRODUCTION

Before the Windows 95 operating system (OS), the life of a hardware developer developing an embedded device designed to be controlled by PC software was simple. Typically, in those days, PCs had a number of parallel and serial ports that could be accessed directly using plain and simple *PortIn* and *PortOut* commands. Then, it was possible to connect the PC printer port to the dedicated port of an embedded device (for example, a PIC microcontroller) using a standard printer port cable (see Figure 1).

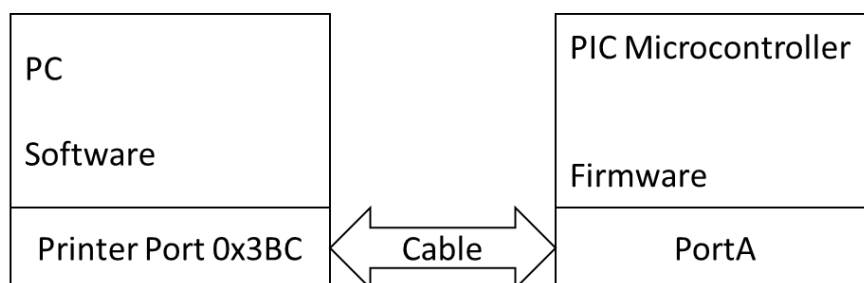


Fig. 1. Direct connection of the PortA of the embedded device (PIC Microcontroller) to the PC's printer port (having exemplary address 0x3BC)

Use of a serial port was even simpler, because a number of *ComPort* software libraries were available on the PC, whereas UART hardware and firmware were available for most microcontrollers.

In most new computers (especially portable computers), serial and parallel ports are absent or not activated. Instead, the modern PC has a number of USB ports, possessing well known advantages such as a high communication speed and hot swapping. However, generally speaking, use of USB ports requires operation of a special driver, which must be installed on the target computer. The requirement to install a special driver leads to a number of practical problems: modern software development tools still have no built-in means for working with most USB drivers; and for a typical corporate computer, the driver installation process can be executed by system administrator only. The USB protocol is sophisticated: the popular book by Axelson (2009) has more than 500 pages. As a result, many companies developed simplified versions of the USB protocols to provide an alternative to serial/parallel

ports (like virtual COM ports using USB hardware). Solutions based on virtual *ComPorts* are still very popular in PC desktop applications.

An alternative solution is use of the USB HID (Human Interface Device) protocol. Originally, this protocol was developed for keyboards, mice, and game controllers; hence, any computer is supposed to have the USB HID driver installed. And, thus, starting from Windows 95, the USB HID driver is a part of Windows OS; hence, no driver installation is needed to operate USB HID communication.

To check the feasibility and reliability of driverless USB HID data transfer between a "master" PC and a "slave" embedded device, a general purpose expandable platform was designed, built, and tested (Kosolapov, 2010). The embedded ("slave") side was implemented on the popular PIC18F4550 microcontroller having built-in hardware and firmware support for the USB HID protocol.

The following sections contain a description of the software elements of the minimalistic PC-PIC system: PC-side software, PIC-side firmware, communication data organization, the proprietary Data-on-Demand communication protocol between PC and PIC, and analysis of a different real-life situations affecting the reliability of communication between PC and PIC.

2. BASIC PC-PIC PLATFORM: USB HID HARDWARE, SOFTWARE, AND FIRMWARE

The basic elements of the PC-PIC expandable platform are presented in Figure 2. USB HID communication requires only two IO pins (for PIC18F4550, pins 23 and 24 are used). This means that all other pins on the microcontroller can be used to expand the platform by adding additional hardware according to the selected application. Sensors (for example, temperature sensors) can be connected to the analog inputs. Actuators (for example, motors and heaters) can be connected to the digital outputs. When relevant, digital input pins can be used.

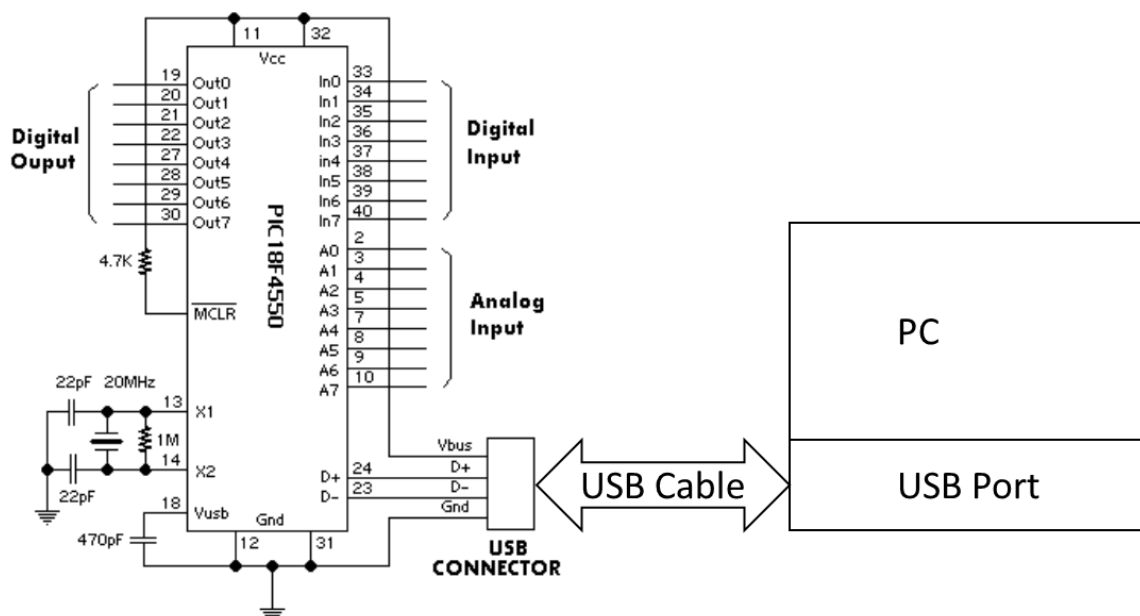


Fig. 2. Basic PC-PIC expandable platform (Look4tech 2010).

On the PIC side, the USB HID Library of the MikroC compiler governed USB HID communication with the host PC (MikroC Compiler 2016). During USB HID communication, the PIC used two buffers: Read Buffer and Write Buffer. The size of every buffer was set to 64 bytes. Actually, only four functions from the USB HID Library were used: *HID_Enable*, *HID_Read*, *HID_Write*, and *HID_Disable*. The *HID_Enable* function must be called before other functions from the USB HID Library. The *HID_Read*

function receives a message from the host PC and stores it in the Read Buffer. The function returns the number of bytes received from the host. Zero means that the function failed. The *HID_Write* function sends data from the Write Buffer to the host PC. If data transmission fails, 0 is returned. At the end of the communication with the host, the *HID_Disable* function must be called. The USB HID Library contains a number of other functions that use polling and interrupt approaches; however, use of these functions is not possible in every situation. Considering that the goal of this contribution is to propose an expandable platform, the interrupt and pooling functions of the USB HID library are not used in the following discussion.

The PC-side software was designed using Visual Studio as a desktop C# .NET application. ATMEL *AtUsbHid.dll* provided basic USB HID communication. Only three basic functions from the ATMEL library were used: *findHidDevice*, *readData*, and *writeData*. The *findHidDevice* function must be called first. A valid vendor ID (VID) and product ID (PID) must be provided. Default 0x1234 and 0x0001 values were used in this non-commercial R&D. The *writeData* function attempts to reads data from the PIC to the 64-byte read buffer of the PC, whereas the *writeData* function attempts to send data from the 64-byte write buffer of the PC to the PIC. All three functions return a Boolean “result”: “true” on success and “false” on failure.

3. PROTOCOL USING BLOCKING LOOPS

An example in the MikroC manual (MikroC Compiler 2016) utilizes a blocking while loop in an attempt to accept data from the PC (see block #2 in Figure 3). This leads to the following problem: the “too busy to listen to PC” PIC cannot do anything else even if the PC did not transmit anything; the while loop blocks any other PIC activity for an indefinite time. This means that the PIC cannot accept new data from sensors and cannot control actuators. Only when all 64 bytes from the PC are received, can the PIC process receive data, collect new data from relevant sensors, send commands to the actuators, provide calculations, and prepare data to be send back to the PC. The same example utilizes the blocking while loop to send data back to the PC when the data is ready to be sent (Block #4 in the block chart in Figure 3). However, this approach is problematic if the master PC is not ready to accept data from the PIC (for example, when the PC is executing other tasks). Data buffering in the PC USB buffer cannot be considered a reliable solution because in this case the PIC is blocked for an indefinite time, and, again, the “too busy to transmit data to PC” PIC cannot execute other tacks like collecting data from the sensors, controlling actuators, or making calculations. Worse, in this case, the PIC cannot receive and process new requests (commands and data) from the PC.

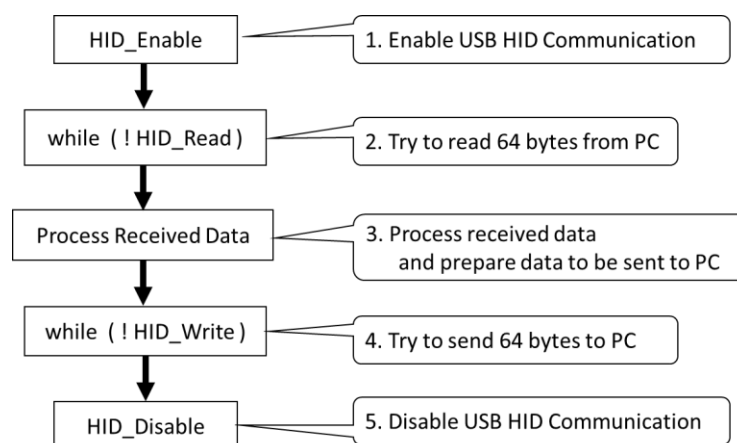


Fig. 3. Simplified block chart of USB HID Communication (PIC side) using blocking while loops. From the recommended example (MikroC Compiler 2016).

4. PROTOCOL MASTER SENDS REQUEST – SLAVE RESPONDS WITH MESSAGE

It is clear that use of blocking while loops cannot be considered as a reliable solution in many practical situations. Again, the use of interrupts to govern communication between PIC and PC would be ideal; but, unfortunately, the internal code of the USB HID library uses high priority interrupts, which makes use of additional interrupts problematic.

The goal of the protocol described in this section is to provide reliable data communication between PIC and PC. Reliability in this context does not mean that there are no errors during communication – this is unrealistic. Reliability here means that as PC software as PIC firmware will continue to operate even in case of errors during communication. In accordance with this approach, data loss during communication is not a “fatal error”, but a “legitimate event in real life”. However, the arrival of wrong or corrupted data must be considered a “fatal error”.

In an attempt to achieve the declared goals, a protocol with a strict discipline of communication was selected. In accordance with this protocol, it is the PC who initiates communication by sending a “request” to the PIC. The request may contain as plain commands as commands with data. For example: request to measure temperature by using (say) sensor #6. On request arrival, the PIC is expected to parse the request, prepare “the answer” (message), and send the message back to the PC.

In an attempt to provide an additional means of preventing the receipt of non-valid requests and messages, a special structure for request and message buffers was selected (see Table 1). Bytes with indexes from 0 to 60 can be used to transmit any type of requests, messages, commands, and data. A byte with the index 61 has a number of service options. Values in the range of {0..5} are reserved for reporting errors. For example, the PIC may set this byte to 1 indicating that the PIC is out of memory. Values in the range {6..255} are used to set the Message ID. Originally, when the PC sends a request to the PIC, the PC sets this number to, say, 6. In case of normal operation, it is expected that when the PIC prepares the “answer” to this request message, it sets this byte to the same number (6, in this case). In this way the PC validates that the received from PIC message is the answer to the specific request. It may so happen that the number of messages sent by PIC to PC is less than the number of requests sent by PC to PIC. In this case of data loss, it is the job of the PC-side software to analyze (by using the set of Message IDs used) what is missing, and to decide how to cope with this situation. For example, if some measurement is missing, the PC software may interpolate the missing value, modify the normal sequence of operations, request that the PIC provide additional data. A byte with index 62 is set by the PC to ‘M’. When the PIC prepares an answer, it sets this byte to ‘S’. This provides an indication that data were successfully processed by the PIC.

Table 1. Structure of the read and write 64-byte buffers

Structure of the 64-Byte Buffer		
Index	Meaning	Comment
0	Data	Any value in the range {0..255}
1	Data	Any value in the range {0..255}
...	Data	Any value in the range {0..255}
...	Data	Any value in the range {0..255}
60	Data	Any value in the range {0..255}
61	MessageID	Values 0..5 : reserved for errors reporting Values 6-255: Set by Master as Message ID Kind of “Registered Post Tracking Number”
62	NameByte	Is ‘M’ in the request message sent by “Master PC” to PIC Is ‘S’ in the answer message sent by PIC back to PC
63	Checksum	SUMMA of bytes with indexes from 0 to 62

The last byte (with index 63) is a summa of bytes with indexes from 0 to 62 – a standard way to provide CRC control ensuring that the arrived request or message is valid.

Figure 4 presents a simplified block chart of PC-side software, whereas Figure 5 presents a simplified block chart of PIC-side firmware.

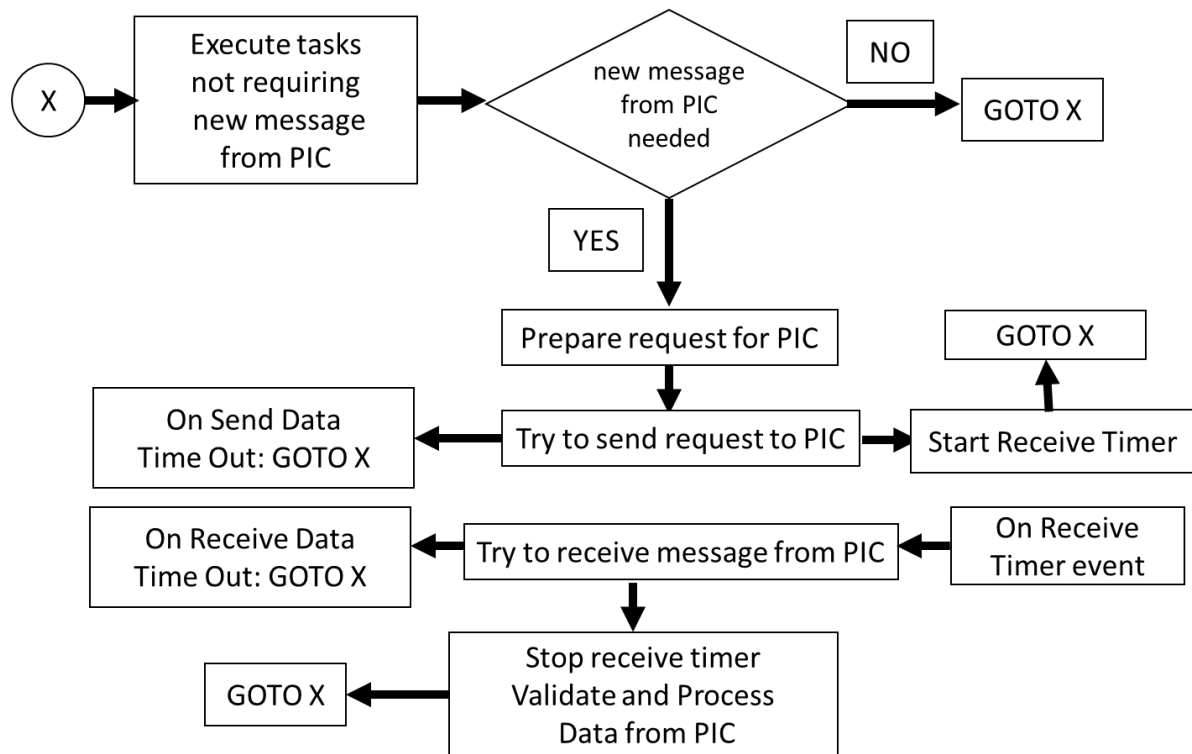


Fig. 4. Simplified block chart of PC-side software.

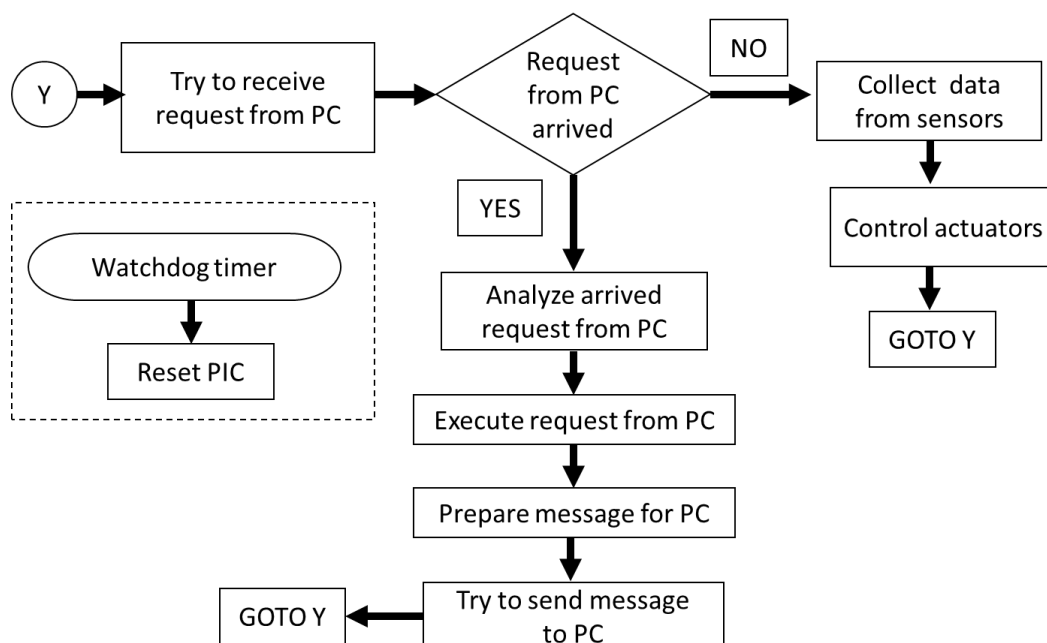


Fig 5. Simplified block chart of PIC-side firmware.

Use of the time-out timer in the PC-side software prevents the situation when the PC hangs because of communication problems. For PIC-side firmware, a time-out is implemented using a non-blocking “for loop” (see Figure 6 and Figure 7). This approach enables dedicating a limited number of PIC timer interrupts for more critical tasks (for example, for measurements with specified time intervals governed by a timer interrupt). Optimal values of the *numberOfReceiveAttempts* and *numberOfSendAttempts* parameters can be derived empirically for the specific application. Considering Figure 7, it can be seen, for example, that attempts to return data (a message) to the PC may fail for the selected value of *numberOfSendAttempts*. Again, it is the PC’s job to decide what to do when the answer to the specific request does not arrive to the PC. Despite the time-out approach ensuring that as PC as PIC does not hang during communication, the watchdog timer (see Figure 5) guarantees a PIC reset if the PIC stalls during measurements or calculations.

```
int TryToReceiveDataFromPC
{
    CleanReadBuffer
    for ( i=0; i< numberOfReceiveAttempts; i++) // Kind of Software Timer
    {
        numberOfBytesReceived = HID_Read();
        if (numberOfBytesReceived ==64 ) break; // Valid message arrived from PC
    }
    return numberOfBytesReceived; // use returned value as error indicator
}
```

Fig. 6. PIC-side firmware: Non-blocking “for loop” in attempt to receive data from PC.

```
voidt TryToSendDataToPC
{
    for ( i=0; i< numberOfSendAttempts; i++) // Kind of Software Timer
    {
        hidResult= HID_Write();
        if (hidResult==1 ) return; // Valid message send to PC
    }
}
```

Fig. 7. PIC-side firmware: Non-blocking “for loop” in an attempt to send data to PC.

5. RESULTS

Table 2, Table 3, and Table 4 summarizes data flow for a number of important real-life situations for the protocol described in Section 4. A logical analysis of the data flow in different situations reveals that as the request of PC to PIC as a response message of PIC to PC can be lost; however, as PC software as PIC firmware continues to operate without hanging. Practically, during the real-life tests of the data logging applications, all operations executed in accordance with Table 2 with a “normal sequence of operations”; that is, as PIC as PC was fast enough to execute the required measurements and data processing during the allocated time slot. To simulate “extremely long” operations, blocking “sleep” functions (for PC-side software) and “delay” functions (for PIC-side firmware) were inserted in different places of the PC software and PIC firmware. As a result, some requests and messages were lost; however, as PIC as PC was able to continue normal operation.

Table 2. Simplified data flow for the normal sequence of operations.

Normal sequence of operations			
Step	PC side		PIC side
1	Executes tasks not requiring data from PIC		Collects data from sensors Controls actuators. Periodically checks if request from PC arrives
3	Sends request to PIC	→	Accepts request from PC
4	Executes tasks not requiring data from PIC Periodically checks if message from PIC arrives		Executes request from PC Collects data from requested sensors Controls requested actuators)
4	Accepts message from PIC and process it	←	Sends message to PC
5	GOTO step 1		GOTO step 1

Table 3. Simplified data flow for the case when the PIC is busy.

Case: PIC is Busy			
Step	PC side		PIC side
1	Executes tasks not requiring data from PIC		Collects data from sensors Controls actuators. Periodically checks if request from PC arrives
3	Sends request to PIC	→	FAILED to Accept request from PC
4	Executes tasks not requiring data from PIC Periodically checks if message from PIC arrives		Collects data from sensors Controls actuators
4	Did not accept message from PIC Currently has nothing to process		Did not send message to PC
5	GOTO step 1		GOTO step 1

Table 4. Simplified data flow for the case when the PC is busy.

Case : PC is busy			
Step	PC side		PIC side
1	Executes tasks not requiring data from PIC		Collects data from sensors Controls actuators. Periodically checks if request from PC arrives
3	Sends request to PIC	→	Accepts request from PC
4	Executes tasks not requiring data from PIC Periodically checks if message from PIC arrives		Executes request from PC (Collects data from requested sensors & controls requested actuators)
4	Did not accept message from PIC (message from PIC is lost)	←	Sends message to PC
5	GOTO step 1		GOTO step 1

In accordance with the declared goals, loss of measurements is not a “fatal error” because the missed measurement points can be interpolated in case of data acquisition at fixed intervals. If data interpolation is not desirable, the missed data can be requested by the PC later; for example, when data from sensors are to be used to activate specific actuators depending on the sensor’s data.

6. SUMMARY

The protocol using blocking “while loops” was found inadequate for the real-life data logger application because as PIC as PC can hang in an unpredictable manner even in the simplest basic communication application without using sensors and/or actuators. An alternative protocol (PC sends request – PIC answers with message) was tested for a number of data loggers having multiple sensors and actuators connected to the PIC microcontroller communicating with PC using a driverless USB HID protocol. No hanging was detected during the real-life tests. The simulation of the “PIC is busy” case leads to the loss of the request from the PC, but as PIC as PC continues to operate and does not hang. The simulation of the “PC is busy” case leads to the loss of the message from the PIC, but as PIC as PC continues to operate and does not hang. More R&D is required to get a statistical analysis of the “PC sends request – PIC answers with message” protocol’s reliability.

REFERENCES

- Axelson, J., 2009, *USB Complete. The Developer’s Guide*, 4th ed., Lakeview Research, LLC, Madison.
- Kosolapov, S., 2010. *Reliability Evaluation of the Driverless USB-HID Platform for Data Transfer between Master PC and Slave Embedded Device*. The 6th ORT Braude Interdisciplinary Research Conference, Haifa, Israel, pp. 11.
- Look4tech, 2010, *USB Data Acquisition with PIC18F4550*, <<http://www.look4tech.com/page/2/>>.
- MikroC Compiler, 2016, *USB HID Library*, viewed 3 May 2016, <http://download.mikroe.com/documents/compilers/mikroc/pic/help/usb_hid_library.htm>.