Messbox mit USB-Kommunikation

1. Allgemeines

In der ersten Version wird der USB-Anschluss des Moduls nur zur Stromversorgung genutzt, die Daten werden über RS232 übertragen.

Dies soll in der neuen Firmware-Version geändert werden. Dann ist nur noch ein USB-Kabel als Verbindung zum PC erfordert.

Zum Datenaustausch mit dem PC ist kein besonderer Treiber erforderlich, da das Gerät als HID (Human Interface Device) angemeldet wird.
Allerdings muss auf dem PC eine Kommunikationssoftware installiert sein.
In diesem Artikel wird eine einfache Lösung in Python vorgestellt.

Im Prinzip sollen die alten Kommandos beibehalten werden, mit einigen leichten Änderungen.

Mit USB verfügbare Kommandos (Stand Januar 2015):

```
? = Liste aller Kommandos

0 = start 1s
1 = start 100ms
2 = start 10ms          (geändert gegenüber der alten Version)
4 = start 4ms           (geändert gegenüber der alten Version)

x = stop (exit)

G<period>  = set generator period to <period>

+ = Set trigger output
- = Reset trigger output

v = firmware version
```

2. Änderungen an der Bascom Firmware

Die Firmware basiert auf der usb_lib_hid_v1_3.bas von Guy Weiler + JC Feltes, zu beziehen hier:

http://staff.ltam.lu/feljc/school/asserlogger/USB/ usb_lib_hid_v1_3.bas
Diese wird im BASCOM-Quelltext inkludiert, im Hauptprogramm sind nur wenige Modifikationen nötig damit der Mikrocontroller als HID-Gerät funktioniert.

Achtung!

- Die Version 1.2 der USB-Library hat einen Bug, so dass keine Kommandos mit mehr als 1 Byte empfangen werden können.
- Die Firmware muß mit einer neuen Bascom Version kompiliert werden! Version 2.7.0.3 geht z.B. nicht, 2.7.0.6 ist OK. Typische Fehlermeldung in Linux mit

```
tail -f /var/log/syslog
Jan 23 09:13:47 JCF-PC kernel: [ 1248.113683] usb 5-2: input irq status -75 received
```

Weitere Erklärungen zur USB Library finden sich auf weigu.lu:
www.weigu.lu/b/usb

2.1. Grundlegendes zur USB Library

Die gesamte Arbeit zum USB-Bus wird mittels Interrupt-Service-Routinen abgewickelt. Alle dazu benötigten USB-Routinen und Unterprogramme befinden sich in dieser USB-Bibliothek.

Im Hauptprogramm muss gleich am Anfang diese Bibliothek inkludiert werden, damit globale Variablen zur Verfügung stehen.

Kurz vor der Hauptschleife soll dann das Unterprogramm Usb_init_device aufgerufen werden, das auch die Interrupts global freischaltet.

Ausser dem für die Deskriptoren zuständigen EP0 werden 2 Endpunkte definiert:

- EP1 (IN) schickt Daten zum PC
- EP2 (OUT) empfängt Daten vom PC

(IN und OUT entsprechen der Datenrichtung wie sie vom PC aus gesehen wird)

Für beide Endpunkte steht ein 64-Byte grosser Buffer zur Verfuegung.

Beim **Endpunkt 1 (IN)** fragt der PC Daten an.

Durch Setzen eines Flags (EP1_FLAG) im Hauptprogramm wird dem Endpunkt mitgeteilt, wenn neue Daten vorliegen. Die Anzahl der vorhandenen Bytes steht dann in der Variablen EP1_CNT. Die Daten selbst werden im Buffer (EP1_BUF) abgelegt.

Über den **Endpunkt 2 (OUT)** sendet der PC Daten.

Die Hauptschleife im Hauptprogramm erkennt an einem Flag (EP2_FLAG) wenn Daten vorliegen. Die Daten liegen im Buffer (EP2_BUF). Die Variable EP2_CNT gibt an wie viele Bytes gesendet wurden.

Im Hauptprogramm müssen die Konstanten Mouse und Keyboard definiert sein.

Mit ihnen kann ausgewaehlt werden ob eine Computermaus emuliert werden soll (MOUSE = 1) oder eine Tastatur (Keyboard = 1). In unserem Fall sind beide null gesetzt.

2.2. Änderungen im Hauptprogramm

Im Hauptprogramm wird initialisiert:

```
On Usb_gen Usb_isr_general
On Usb_com Usb_isr_communication

' For usb_lib_hid_v1_3.bas, the constants Mouse and Keyboard must be set:
Const Mouse = 0                                         'No mouse emulation
Const Keyboard = 0                                      'no keyboard remulation

'Include usb library without executing code (jump over)
Goto Init
$include "lib\usb_lib_hid_v1_3.bas"
```

```
Init:
Dim Usbstring As String * 63                                    'String var for USB operations
Dim Usbstringbuf As String * 63 At Ep1_buf Overlay              'Buffer for USB transfer to PC
'' these strings are dimensioned for 63 characters as strings end with chr(0)
'' maybe 64 wouls also be OK?

Gosub Usb_init_device                                           'activate USB
```

Hier wird auch die USB-Library usb_lib_hid_v1_3.bas eingebunden.
Damit der in ihr enthaltene Code nur eingebunden, aber nicht ausgeführt wird, wird er mit goto... übersprungen.

Die Variable Usbstring ist ein 64-Bit Buffer für die zu sendenden USB-Daten.

2.3. Empfang von Kommandos über USB

In der Hauptschleife wird geschaut ob wir ein Kommando vom PC empfangen:

```
Do
      'If new time interval: calculate voltages and print values
      ' (print always if switch activated or print only if sequence started, until stop)
      If Newtime = 1 Then

         Portd.6 = 1                                            'LED ON
         Gosub Calculatevoltages
         If Printvaluesflag = 1 Then Gosub Printvalues
         Portd.6 = 0                                            'LED OFF
         Newtime = 0
      End If

      'command over USB ?
      If Ep2_flag = 1 Then
            Ok = Ep2_buf(1)
            Ep2_flag = 1
      End If

      'user action by serial interrupt -> menu
       If Ok > 0 Then Gosub Menu

Loop
```

Wenn etwas über USB empfangen wurde, steht es in der String-Variablen ok.
Im Unterprogramm Menu wird dann die entsprechende Aktion ausgeführt. Hierbei ist es egal ob der Empfang des Kommandos über RS232 oder USB erfolgte.

Mit dieser kleinen Erweiterung funktionieren schon die Befehle ?, 0, 1, x, +, -, die Daten werden aber weiter nur über RS232 ausgegeben.
Für den Empfang von Kommandos mit mehreren Bytes sind noch einige Änderungen nötig, siehe Quellcode.

Da die Datenrate über USB weit höher ist als über RS232 wurden noch die Kommandos 2 und 4 vorgesehen, mit denen die Abtastzeit auf 10ms bzw. 4ms festgelegt wird. Im Gegensatz zur alten Firmware werden die Daten aller 4 Kanäle auch bei diesen kurzen Abtastzeiten ausgegeben.

Eine Untersuchung des Zeitbedarfs der verschiedenen Unterprogramme zeigte, dass vor allem die Wandlung der Zahlenwerte in Strings recht viel Zeit (>2ms) kostet, während die Berechnungen ca. 700us brauchen. Auf eine weitere Optimierung wurde verzichtet, da ein Zeitraster von 4ms für die meisten regelungstechnischen Vorgänge völlig ausreicht. Bei schnelleren Vorgängen würde man sowieso ein Oszilloskop verwenden.

2.4. Senden der Daten über USB

Hierzu müssen die Daten als String in den Puffer **EP1_buf** kopiert werden. Dies geht automatisch durch Setzen der Variablen Usbstringbuf da diese als Overlay an gleicher Stelle definiert ist.

Anschliessend wird mit

```
       Ep1_cnt = 64
      Ep1_flag = 1
```

der USB-Library mitgeteilt, dass der Puffer gefüllt und die Daten bereit zum Absenden sind. (Eigentlich sollte man besser sagen, die Daten sind bereit zum Abholen durch den PC. Dieser prüft beim HID-Modus jede Millisekunde, ob neue Daten verfügbar sind.)

Um das Umschreiben der Firmware zu vereinfachen, wurde eine Funktion USB_print geschrieben, die Strings über USB und RS232 ausgeben kann:

```
Sub Usb_print(s)
      'output s (max. 64 bytes) to USB and RS232

      ' -> USB
      ' Set number of bytes and flag,the rest is done by the library
      Usbstringbuf = S
      Ep1_cnt = 64
      Ep1_flag = 1

      ' -> RS232
      Print S

      ' to be sure that PC gets USB message, wait 1ms
      ''' this could be done in a better way depending on the size of the message
      Waitms 1
      ''' eventually not necessary if it is sure that s is not updated too fast
      '' but this would be the case if multiple Usb_print ... occur one after the other
End Sub
```

Diese Funktion kann überall da wo es nicht zeitkritisch ist die Print-Funktion ersetzen.
Sie muss zuvor am Programmbeginn deklariert werden:

```
Declare Sub Usb_print(byval S As String * 63)
```

Nun können alle Print-Befehle durch USB_Print ersetzt werden. Nun wird auch die Ausgabe der Daten im 1s- und 100ms- Raster funktionieren, eine entsprechende Kommunikationssoftware vorausgesetzt.

2.5. Timing

Bei der alten Firmware wurden nur im Zeitraster 1s und 100ms die Daten im Klartext ausgegeben. Bei 10ms und 1ms wurde nur ein Kanal als Word-Werte in einem Array gespeichert und dann nach der Messung ausgegeben. Um die Firmware zu vereinfachen, wird auf das 1ms-Raster verzichtet. Dieser Nachteil wird dadurch kompensiert, dass alle 4 Kanäle laufend ausgegeben werden können.

Da USB schneller als RS232 ist, sind nun auch kürzere Zeitintervalle zwischen den Messungen möglich, mit Übertagung aller 4 Kanäle im Klartext.

Problemlos funktioniert die Übertragung im 10ms-Raster (wenn nur via USB übertragen wird, denn RS232 frisst zuviel Zeit. Hier bringt die USB_Print-Funktion also nichts).

Leider ist ein 1ms-Raster nicht zu schaffen, denn die Firmware ist dafür nicht schnell genug. Um das Timing zu untersuchen, wurde jeweils am Beginn eines zeitkritischen Blocks der fürs Debuggen reservierte Port D.7 gesetzt und am Ende wieder rückgesetzt. mit dem Oszilloskop kann dann bequem die Zeit für diesen Block gemessen werden.

Dabei stellte sich heraus, dass die Berechnungen mit ein wenig Code-Optimierung ca. 0.7ms beanspruchen. Wesentlich länger braucht die Konvertierung in einen String, insgesamt ca. 2ms. Die Zeiten sind nicht konstant, sie hängen von den gerade anliegenden Werten ab.
Bei dieser Messung hilft der Envelope-Modus des Oszilloskops, denn man sieht sehr gut die längste vorkommende Dauer der Impulse.

Das Ganze braucht also ca. 2.7ms, also erscheint als kürzestes Zeitraster 3ms praktikabel.

Versuche haben aber ergeben, dass dies noch zu knapp ist, wahrscheinlich weil die Hauptschleife und das Interrupt-Timing im Prinzip asynchron ablaufen.

So wurde ein kleinstes Raster von 4ms festgelegt.

3. PC Software in Python

Die folgenden Überlegungen gelten hauptsächlich für Linux, das meiste lässt sich aber auf Windows als Betriebssystem übertragen.

3.1. pyUSB

Für die beschriebenen Programme wird die neue Version 1.0 von pyUSB benötigt.

Einfacher Test in Python:

```
import usb.core
```

muss ohne Fehlermeldung funktionieren.
Wenn nur import usb funktioniert, ist wahrscheinlich nur die alte Version von pyUSB installiert.

a) Installation von pyUSB in Linux:

- py USB herunterladen von http://sourceforge.net/projects/pyusb
- Entpacken des ZIP-Archivs in einen beliebigen Folder
- In diesem Folder dann Terminal starten und:

```
sudo python setup.py install
```

Test der Installation:
In Python: import usb.core muß ohne Fehlermeldung funktionieren

b) Linux: Verbindung auch ohne Root-Rechte

Hardware-Verbindungen dürfen in Linux nur mit den nötigen Rechten zustande kommen.
Ohne weitere Vorkehrungen wird ein Python-Programm diese Fehlermeldung liefern:

```
usb.core.USBError: [Errno 13] Access denied (insufficient permissions)
```

Dies kann man vermeiden indem man

- das Programm mit Root-Rechten ausführt:

```
sudo python myprogram.py
```

- oder eine Regel definiert die dem Benutzer die nötigen Rechte gibt.
  Diese Methode ist vorzuziehen.

Um die Regel zu definieren wird (mit root-Rechten!) im Verzeichnis `/etc/udev/rules.d` eine Datei mit der Namensendung "`.rules`" erstellt werden, z.B. `15-avrhid.rules`

Existiert schon eine ähnliche Datei, kann der nachfolgende Text auch dort eingefügt werden.

In die Datei kommt dieser Text:

```
SUBSYSTEMS=="usb", ATTRS{idVendor}=="03eb", ATTRS{idProduct}=="0002", GROUP="plugdev", MODE="0666"
```

Hiermit erhält ein Benutzer, der Mitglied der Gruppe "plugdev" ist, die Rechte auf das Gerät mit Vendor-ID 3EB und Attribut 2 zuzugreifen. (die Vendor ID ist die von atmel, und das Attribut 2 ist in der Firmware definiert.

Nun muss der Benutzer der Gruppe plugdev hinzugefügt werden, wenn er nicht schon dort Mitglied ist.

Mit
```
groups jcf
```
erhält man die Information, in welcher Gruppe der Benutzer jcf ist.

Wenn die Gruppe plugdev hier nicht auftaucht, kann der Benutzer hinzugefügt werden:
```
sudo adduser jcf plugdev
```

Damit die neu erstellte Regel nicht erst nach dem nächsten Booten sondern sofort wirksam wird, muss sie getriggert werden:
```
sudo udevadm trigger
```
Eine Beschreibung der ganzen Vorgehensweise findet man z.B. hier:
http://www.tincantools.com/wiki/Accessing_Devices_without_Sudo

Nun müsste die Software zur Verbindung mit der Messbox in Linux auch ohne Root-Rechte laufen.

c) Test der Verbindung mit dem Gerät

In Python:

```
import usb.core
dev = usb.core.find(idVendor = 0x3EB, idProduct = 2)
```

darf keinen Fehler melden.

d) Test von der Betriebssystemebene

Im Terminal:
```
lsusb –vv
```
gibt detaillierte Informationen über alle angeschlossenen USB-Geräte
inklusive Infos über Endpunkte, Konfiguration usw.

```
.....
Bus 005 Device 012: ID 03eb:0002 Atmel Corp.
Device Descriptor:
  ...
  idVendor           0x03eb Atmel Corp.
  idProduct          0x0002
  bcdDevice            0.01
  iManufacturer           1 FELJC WEIGU.LU
  iProduct                2 HID-DEVICE
  iSerial                 3 WH12b
  ...
    bmAttributes         0x80
      (Bus Powered)
    MaxPower              500mA
    Interface Descriptor:
      ...
      bInterfaceClass         3 Human Interface Device
      ...
        bEndpointAddress     0x81  EP 1 IN
        bmAttributes            3
          Transfer Type            Interrupt
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               1
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x02  EP 2 OUT
        bmAttributes            3
          Transfer Type            Interrupt
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               1
Device Status:     0x0000
  (Bus Powered)
```

e) Empfang der Daten

Einfaches Test-Programm:

```
Vendor_ID = 0x3EB #(ATMEL)
Product_ID = 0x02 #usb_small_lib "hid")

import usb.core, time


dev = usb.core.find(idVendor = Vendor_ID,idProduct = Product_ID)

if dev is None:
    print "device not found"

else:
    # only linux: uncomment if device not free (error resource busy)
    interface = 0
    if dev.is_kernel_driver_active(interface) is True:
        dev.detach_kernel_driver(interface)
        usb.util.claim_interface(dev, interface)
        dev = usb.core.find(idVendor = Vendor_ID,idProduct = Product_ID)

    dev.set_configuration()

    # communication code
    while True:
        res = dev.read(0x81, 64) # read 64 Byte (ADC) from endpoint1 (0x81)

        if len(res):
            s= res.tostring()
            s=s.split(chr(0))       # USB string is delimi1ted by chr(0)
            print( s[0] )           # print everything before chr(0)
```

Dieses Programm zeigt die Daten genau wie das RS232-Terminal, wenn die Messbox sendet.

Zunächst wird usb.core.find benutzt um das Gerät zu finden.
Der Linux-spezifische Code dient dazu, das Gerät freizugeben, wenn es von einem anderen Programm benutzt werden sollte. Dieser Abschnitt ist für einen ersten Test nicht erfordert, er sollte eingefügt werden wenn man die Meldung "resource busy" erhält.

Dann wird die (einzige) Konfiguration gesetzt. Dieser Schritt scheint nicht unbedingt nötig zu sein.

Anschliessend wird in einer Endlosschleife der Endpunkt 1 an der Adresse 0x81 abgefragt. Wenn Daten vorliegen, werden sie in einen String umgewandelt. Es kann sein, dass von vorherigen Ausgaben noch "Müll" im Puffer zu finden ist. Dieser wird abgetrennt, indem nur das berücksichtigt wird was links vom String-Endezeichen 0x00 steht.

Wegen der Endlosschleife muss das Programm mit <Ctrl>C abgebrochen werden.

f) Senden von Kommandos (und Empfangen von Daten)

Die Hürde bei einem Konsolenprogramm ist das Reagieren auf Tastendruck ohne Blockierung des Hauptprogramms. In einer GUI hat man dieses Problem nicht.

Es soll trotzdem zunächst ein einfaches Konsolenprogramm beschrieben werden.
Was wie eine Banalität aussieht, nämlich das Abfragen einer Taste ohne Blockierung, ist in Python erstaunlich schwierig. Braucht das denn niemand? Scheinbar doch, wie eine

Internetrecherche zeigt. Funktionierende Lösungen sind aber schwer zu finden.
Das Modul Curses würde es bieten, aber es existiert nur für Linux.
In Windows gibt es Mscvrt, aber dieses gibt es für Linux nicht.

Unter
http://stackoverflow.com/questions/2408560/python-nonblocking-console-input
habe ich nach langer Suche und vielen Tests eine Lösung für dieses Problem gefunden.
Diese habe ich unter Linux getestet, sie sollte aber auch in Windows funktionieren.

Diese Lösung wird als leicht modifizierte Variante in das Programm eingebaut:

```
# This is for keyboard reaction, nonblocking
import keypress_nonblocking as kp
kb = kp.KBHit()
```

Das Modul keypress.py gibt es hier:
http://staff.ltam.lu/feljc/school/asserlogger/USB/ keypress.py

In der While-Schleife des Empfangsprogramms wird nun erweitert:

```
# communication code
   while True:

       # receive
       res = dev.read(0x81, 64)    # read 64 Byte (ADC) from endpoint1 (0x81)
       if len(res):                # and print
           s=res.tostring()
           s=s.split(chr(0))       # USB string is delimi1ted by chr(0)
           print( s[0] )           # print everything before chr(0)


       # if keyboard input, send command
       if kb.kbhit():
            c = kb.getch()
            dev.write(0x02, c)      # write do device


           # quit with <ESC>
           if ord(c) == 27:
               break

   # end
   kb.set_normal_term()
   print ("# END")
```

Wenn kb.hit() einen Tastendruck registriert, wird das entsprechende Zeichen mit kb.getch() gelesen und mit dev.write an das USB-Gerät geschickt.
So können einzelne Bytes gesendet werden, wie es für die Kommandos erfordert ist.

Längere Strings erfordern etwas mehr Aufwand als hier beschrieben, siehe Quellcode von Firmware und USB-Terminal.

Da nun eine Tastenabfrage im Programm möglich ist, kann das Programm auf die sanfte Art durch Eingeben von <Esc> verlassen werden.
Vorher wird mit kb.set_normal_term() wieder auf normales Verhalten des Kommando-Terminals umgeschaltet.