Linux *Kernel Hacking* Free Course 2003

UNIVERSITA' DEGLI STUDI DI ROMA TorVergata Università

D. P. Bovet, M. Cesati, R. Spennato

Implementing an USB sensor device

The USB bus

- The Universal Serial Bus (USB) is a non-proprietary architecture
- It is the outcome of a consortium formed in 1994 by Compaq, Intel, Microsoft and NEC
- USB version 1.0 released in 1996
- USB version 1.1 (used in our project) released in 1998
- USB version 2.0 released in 2000

Main features (1)

- Hot-pluggable: I/O devices can be attached and detached while the PC is running
- Single connector type: all devices plug into the same socket type
- High performance: device transfer rate up to 1.5 Mbps for slow devices, up to 10 Mbps for fast devices (12 Mbps theoretical bandwidth)
- Up to 127 devices can be connected to a USB bus

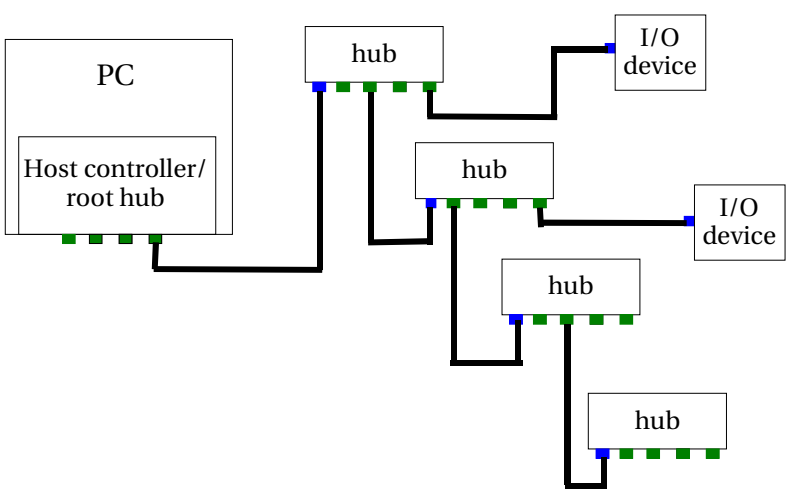
Main features (2)

- Power supplied by cable: many devices do not need an additional power supply
- Power management: devices automatically power down when not in use
- External to the PC: no need to open the PC to install new USB devices
- Old PCs can be upgraded to USB may making use of a PCI or PCMCIA card

Connecting devices to an USB bus (1)

- Tree-like architecture
- The *root hub* located inside the PC includes 2 or 4 USB ports
- Additional hubs, each of which includes one input line and four output lines, can be connected to the USB ports
- Devices can be nested five hub levels deep

Connecting devices to a USB bus (2)



The USB cable

- Four-wires cable
  - D+, D- used for serial transmission of bits
  - Vbus (5 v), Vss (ground) used for power transmission
- Maximum cable length of 5 meters (regular cables)
- Two different connectors: the rectangular one (green) plugs into a USB port, the square one (blue) plugs into the I/O device

Device detection

- If no device is plugged to a USB port, both D+ and D- are low
- When a device is plugged in, either D+ or D- raises above ground
- If it is D+, a fast device has plugged in
- If it is D-, a slow device has plugged in

Data transmission

- Data wires do not include a CLOCK signal: asynchronous communication at 12 MHz
- NonReturn to Zero Invert (NRZI) encoding: the status of the D+ and D- lines is inverted when:
  - Two consecutive 0s have been transmitted
  - The sequence 10 has been transmitted
- Bit stuffing of 0 after 6 consecutive 1s

How does it work (1)

- Master/slave architecture: the PC is the master, the I/O devices are the slaves polled by the master (one exception covered later)
- The tasks performed by the master are implemented by an *host controller* located inside the PC
- The tasks performed by the slaves (all USB devices, including the hubs) are implemented by *microcontrollers* located inside the I/O devices

How does it work (2)

- Data transmission along the USB cable is half duplex: either from the host controller to the microcontrollers, or from a microcontroller to the host controller
- Each device is assigned dynamically a 7-bit address (up to 128 devices on the same bus)

How does it work (3)

- No collision detection as in Ethernet: at each time, at most one device is allowed to send data to the host controller
- When a specific device is polled, its controller switches from receiving to transmitting mode and sends back the required data

The USB packet

- The basic transfer unit on the USB data bus is a *packet*
- Each packet consists of the following parts:
  - An 8-bit SYNC sequence
  - An 8-bit Packet Identifier (PID): 4 bits + 4 complemented bits
  - A sequence of 0 up to N 8-bit data characters
  - A 2-bit End-Of-Packet identifier: both D+ and D- low for two bit times

Packet types

- *Token packets:* SOF, SETUP, IN, OUT
- *Data packets*: DATA0, DATA1
- *Handshake packets*: ACK, NACK, STALL
- *Special*: PRE (used to establish a low-speed connection)
- Token packets are used to set up Data packets, which are acknowledged by Handshake packets

Frames (1)

- Used by the host controller to keep track of time
- One *frame* issued every millisecond
- Up to (1.0 ms/12 Mhz) = 12,000 bits = 1500 B can be packed within a frame
- The host controller schedules data transfers dynamically and tries to fill up the frame with data
- A data transfer cannot be split into several frames

Frames (2)

- Each frame starts with a Start-Of-Frame (SOF) token packet
- When the USB bus is idle, the only packets transmitted are the SOFs
- SOF packets, which are broadcasted to all USB devices, are the only ones that do not have a destination address

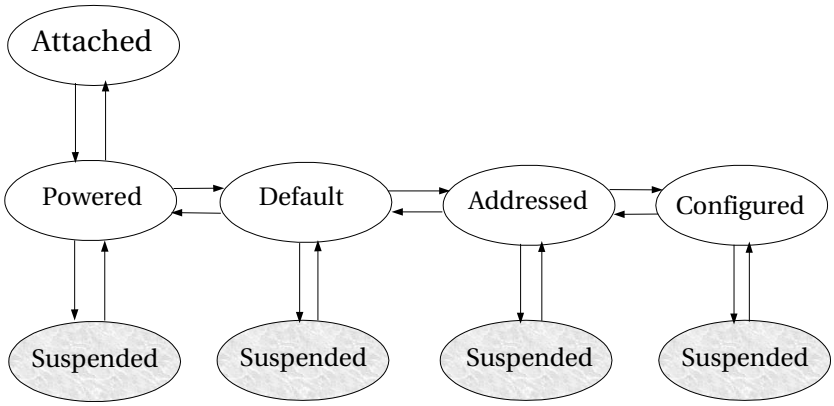
Frames (3)

- If an I/O device detects no bus activity (no SOF packet) for 3 ms, it assumes that the PC has powered down and it enters automatically into a *suspend state*
- An I/O device can send a remote wakeup signal to bring the PC out of its powered down state (a USB modem might want to wake up the PC when the phone rings)
- The *remote wake-up capability* must be predeclared in the device configuration descriptor

I/O device states

- The USB protocol distinguishes several I/O device states
  - Attached: hub signals a plug in on a port
  - Powered: hub configured and operational
  - Default: USB has received the device descriptors
  - Addressed: USB has assigned a 7-bit address to the device
  - Configured: USB has checked that the device is addressable
  - Suspended: device low-powered

I/O device state diagram



USB transactions (1)

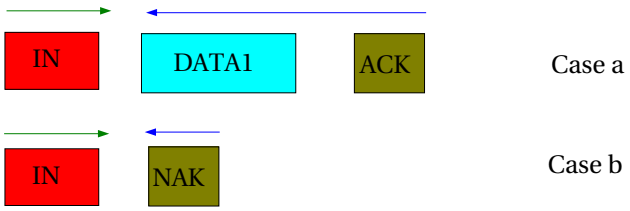
- A predefined sequence of packets used to define data transfer between the host controller and an I/O device
- Each *transaction* must be completed within the same frame
- USB specification defines four transaction types to handle different types of data transmitted along the bus
- The four transaction are handled differently both in terms of time accuracy and of quality accuracy

USB transactions (2)

- *Interrupt*: ack, transfer time guaranteed (mouse, keyboard)
- *Bulk*: ack, transfer time not guaranteed (printer, scanner)
- *Isochronous*: no ack, transfer time guaranteed (speakers, video)
- *Control*: control transfer divided into Setup, Data, and Status phases (sending a command to an I/O device)

An example of Interrupt transaction

- The host controller polls once every 10 frames (100 times a second) the USB mouse. Two cases may arise:
  - The mouse has fresh data in its buffer (case a)
  - The mouse has no fresh data (case b)



Building an USB device

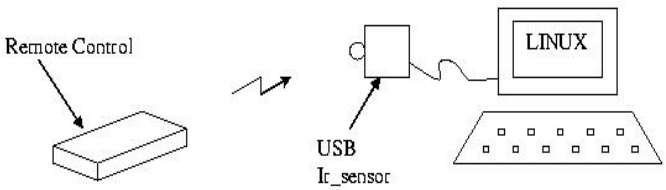
Our goal (1)

- The I/O device must comply with the USB 1.1 standard
- It must be able to pick up the signals sent by an infrared TV remote control and decode them
- The I/O device must be powered by the USB bus

Our goal (2)

- The decoding of the signals must be non ambiguous: each button pushed yields a different sequence of characters
- The characters received by our device will be read by a USB driver
- They may be used to by application programs in several way: for instance, to simulate sequences of chars issued by the keyboard

Our goal (3)



Hardware constraints

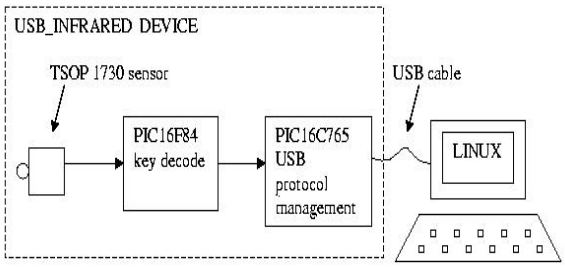
- The I/O device makes use of a microcontroller that runs the code implementing the USB protocol
- To save money, we selected a OTP (One Time Programming) to do this job
- A second simpler microprocessor that makes use of EEPROM decodes the signals coming out of the remote control
- This choice is based on economical reasons

An overview (1)

- The I/O device includes three main components:
  - PIC16C765 from Microchip (handling the USB protocol)
  - PIC16F84 from Microchip (decoding the signal coming out from the infrared sensor)
  - TSOP1730 from Vishay Telefunken (infrared sensor)

An overview (2)

- The PIC16F84 decodes the signal
- It sends the corresponding characters to the PIC16C765
- The PIC16C765 sends these data to the PC



The PIC16Cxxx family (1)

- All microcontrollers of the PIC16Cxxx family have the same core (CPU+ALU)
- They sport a Harvard architecture: two different memories and buses, one for the code and one for the data
- The two memories have different word sizes: 14 bit for the code memory, 8 bit for the data memory

The PIC16Cxxx family (2)

- The processors use 35 RISC-like instructions having the same length (14 bit)
- Almost all instructions are executed in a single clock cycle, conditional branches require 2 cycles
- Memory to memory transfers implemented using the W accumulator register

The PIC16Cxxx family (3)

- All PIC registers, but W, are mapped in RAM
- The memory that contains the code can be:
    - PROM: (One Time Programming)
    - EPROM: erasable by using ultraviolet radiations
    - EEPROM: electrically erasable, flash technology

The PIC16Cxxx family (4)

- Several I/O can be included into the chip:
    - Simple I/O lines
    - A/D converters
    - USB controllers
    - I^2C interface
- All PICs include an 8-bit TIMER

The PIC16Cxxx family (5)

- The ~MCLR pin selects the PIC mode, depending on the voltage value:
- 5V (Vcc): firmware execution
- 0V (GND): PIC reset
- 12V - 14V: firmware programming
- Interrupts detected by the PIC are handled by an interrupt handler located at 0004h (interrupt vector)
- When the PIC is reset, it starts executing the instruction located at 0000h (reset vector)

The PIC16Cxxx family (6)

- Each PIC16Cxxx has an initialization string stored in the program memory at location 2007h
- This string can be set up in order to specify:
    - The type of external oscillator used
    - The type of code protection
    - The watchdog timer value
    - The power-up timer value

The PIC16Cxxx family (7)

- In oder to work, the PIC must be connected to an external oscillator, in ur case we'll use a 4Mhz quartz crystal for the PIC16F84 and a 6Mhz one for the PIC16C765
- The CPU clock rate is one fourth of the oscillator's frequency (respectively 1MHz and 1.5 MHz)
- The power supply is 5V

The PIC16F84 (1)

- The PIC16F84 used by us is contained in an 18-pins PDIP
- This PIC is identical to the PIC16C84, but it offers a better code protection
- The size of the EEPROM code memory is 1K x 14bit
- The size of the EEPROM data memory is 64 byte x 8 bit
- The data RAM size is 68 Byte
- The stack depth is limited to 8

The PIC16F84 (2)

- Includes 13 I/O lines and a timer
- The I/O lines are splitted in two ports called A and B. PORTA has 5 lines, PORTB has 8 lines
- The lines are handled by means of 4 registers:
  - TRISA, TRISB: to determine the direction of an I/O line
  - PORTA, PORTB: to read or write data on the line
- The 8 bit.timer with prescaler will be used to decode the signal coming from the sensor

The PIC16C765 (1)

- The PIC16C765 used by us is contained in a 40-pins PDIP
- The size of the EPROM code memory is 8K x 14bit
- The data RAM size is 256 byte
- The stack depth is limited to 8
- This PIC has 33 I/O lines divided into 5 ports: A, B, C, D, E
- Includes a 5-channels A/D converter having an 8-bit resolution

The PIC16C765 (2)

- Three pins of the chip are used by USB
- The chip includes a USB controller conforming to the USB 1.1 protocol
- This controller is only able to handle slow devices (1.5Mb/s)
- Two pins, D+ e D-, are used for the data line
- The third pin, V_{USB}, is set at 3.3V. It must be connected to the D- pin by means of a 1.5KΩ resistor

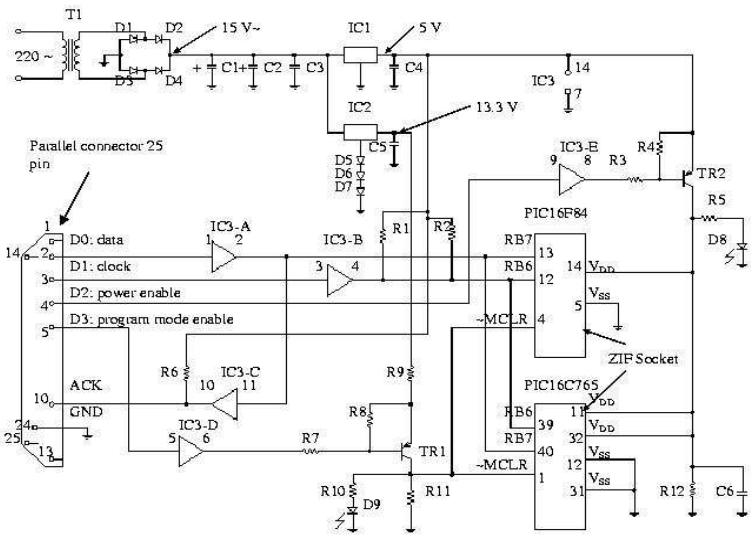
The PIC Programmer (1)

- A hardware/software interface used to store firmware into the chip
- Software: IC-Prog 1.05A written by Bonny Gijzen and working under Windows 9x
- Hardware: we have built a circuit, similar to the PIC programmer of David Tait, but adapted for the PIC16C765, that connects to the parallel port

The PIC Programmer (2)

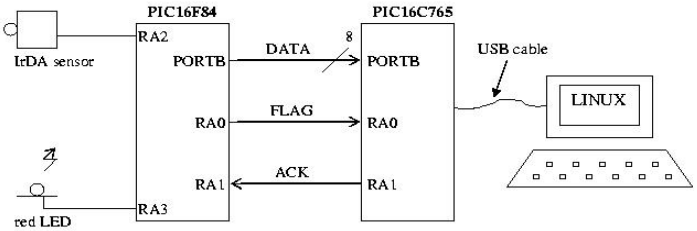
- PIC programing takes place serially, one bit at a time
- The programming protocol makes use of a series of commands to be issued to the PIC; the *comand form* is 24-bit long, (8 bits of code, 16 bits of data)
- The PIC pins used are RB6 (clock input), RB7 (I/O data), ~MCLR (*program mode)*

The PIC Programmer (3)

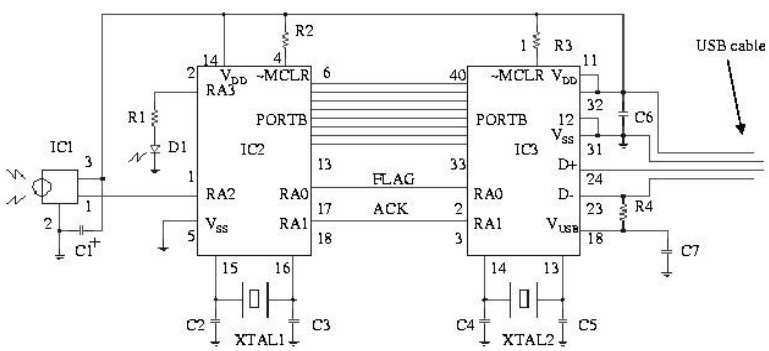


The USB Infrared device (1)

- Once it has decoded the signal coming from the sensor, the PIC16F84 chip sends the corresponding code to the PIC16C765 chip using a handshake protocol
- The led notifies the user that the signal has been received



The USB Infrared device (2)

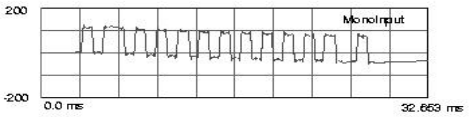


The TV remote control (1)

- We have used a common remote control
- Reverse engineering has been performed in order to understand the wave shapes associated with the keys
- We used the SftOsc v1.0 software oscilloscope written by Toshio Iwata for Windows 9x
- We discovered that each key yields 2 alternating codes that differ in the initial part
- According to the conversion rules set up by us, the code can be 13 or 14 bit long

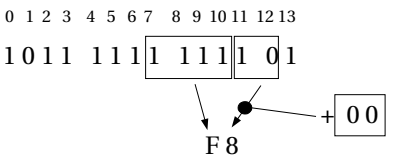
The TV remote control (2)

- Rules derived empirically to get the bits corresponding to wave shapes:
  - Long interval high or low equal to '0'
  - Short high interval equal to '1'
  - Short low interval is ignored
- Wave shape associated with key 1 of remote control (Bits: 1011 1111 1111 01 -> BFF4h):



The TV remote control (3)

- To get the code, we use the following technique (example refers to key '1'): discard bits 0-5 and bits 12-13, add two 0 padding bits):



- The code sent by the PIC16F84 is 'F8'

The firmware of the PIC16F84 (1)

- The MPLAB environment, which also includes a PIC simulator, has been used to develop the firmware for the two PICs
- Some remarks:
  - The duration of a short interval is 0.8 ms, that of a long one is 1.6 ms
  - The time interval between two sequences of signals is 88 ms
- In order to determine the step (and thus to identify the low and high intervals), we implement something similar to a clock inside the PIC

The firmware of the PIC16F84 (2)

- We use the TICK variable that is incremented every 0.62 ms by the interrupt routine tied to the timer overflow TMR0 of the PIC
- The PIC timer is programmed to yield interrupts at a 250Khz frequency (the 1Mhz CPU clock rate divided by the prescaler 4) and to start counting from 100 (TMR0=100)
- In formulas:
  - 250Khz / (256-100) = 1602 Hz ---> 0.62ms

The firmware of the PIC16F84 (3)

- Interrupt routine:

```
INTERRUPT_HANDLER
   incf tick,1  ;0,62 ms elapsed

   movlw TIME_BASE   ;initialize the timer
   movwf TMR0   ;timer

   bcf INTCON,2 ;clear the timer interrupt flag
   retfie
```

- Synchronize with the first step of the sequence using the SYNCHRO routine, which waits until 5 ms of “silence” occur
- “silence” means no spike and no step, if this does not happen, reset the timer and wait 5 ms

The firmware of the PIC16F84 (4)

- The RA2 line connected to the sensor device is polled to detect a rising or declining step
- Pseudo code of this routine:

```
Capture the raising step
   analyze the bottom interval (control spike,
                write data into buffer)
Capture the declining step
   analyze the top interval (control spike, write
                data into buffer)
Jump: Capture the raising step
```

- Another routine transmits the code associated with the key to the PIC16C765 using a handshake protocol

The firmware of the PIC16C765 (1)

- Microchip offers a library to support USB. Several functions of the library are interrupt-driven
- The firmware includes the following files:
  - usb_main.asm : main program
  - usb_ch9.asm : functions to handle device enumeration and USB bus
  - hidclass.asm : HID-specific functions
  - descript.asm : includes the descriptors
  - usb_defs.asm : macro definitions

The firmware of the PIC16C765 (2)

- We used the following functions and macros:
  - InitUSB: invoked by the main program enable the USB controller, to set the V_{USB} pin to 3.3V and to start the enumeration process. This function must be invoked at least16 µs after the PIC has been powered on, so that the SIE is properly initialized
  - ServiceUSBInt: invokes the USB handling routine, included in the main interrupt handler

The firmware of the PIC16C765 (3)

- ConfiguredUSB: (macro) blocks the firmware execution by polling bits of the status register, until the I/O device has been configured
- PutEP1:sends data to the host through endpoint 1. This function receives as parameters the address of the data block (indirect addressing by means of registers FSR/IRP), and the block size (register W). If the function succeeds, it sets the Carry Flag in the status register and it returns in W the number of bytes copied in the transmission buffer

The firmware of the PIC16C765 (4)

- This is the firmware main loop that polls the RA0 line waiting for data:

```
Loop
   pagesel Receive
   btfsc PORTA,FLAG ; FLAG=1?
   call Receive
   pagesel Loop
   goto Loop
```

- The Receive function gets data from the PIC16F84 through a handshake protocol and sends them to the host using the PutEP1 function