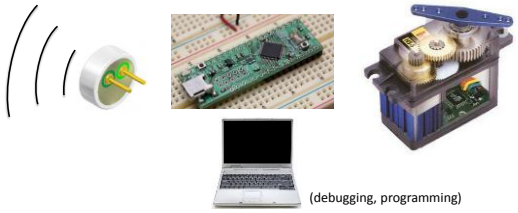
ETH Course 402-0248-00L: Electronics for Physicists II (Digital)

- **1: Setup uC tools, introduction**
- **2: Solder SMD AVR32 board**
- **3: Build application around AVR32 – finish today**
- **4: Design your own PCB schematic**
- **5: Place and route your PCB**
- **6: Start logic design with FPGAs**

Exercise 3: "Sound volume robot"

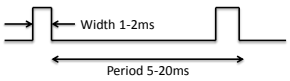
- measures sound volume and moves arm to indicate loudness
- microphone -> preamp -> ADC -> uC -> PWM output

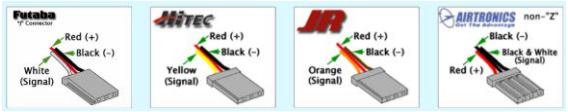


(debugging, programming)

"RC" servos (Radio-Control Servo-Motors)

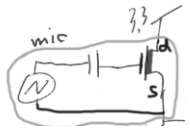
- Position controlled – Servo has internal position measurement and controller
- Rotation angle 120 degrees
- Pulse width from 1-2ms sets desired position
- Pulses must be sent at frequency 50-200Hz
- Pulse height >2V

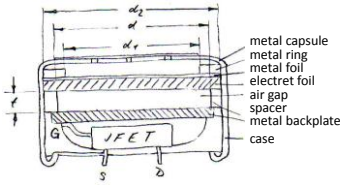


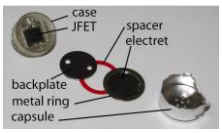


Electret Microphone

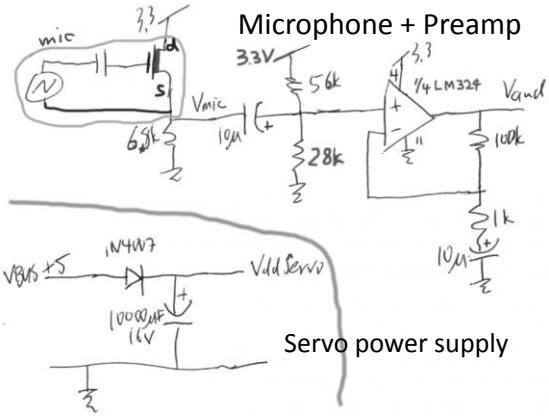
- Cheap (< 1$)
- Electret material, no polarization voltage is required
- Low-noise JFET buffer
- Metal foil is connected to source of the JFET through metal capsule







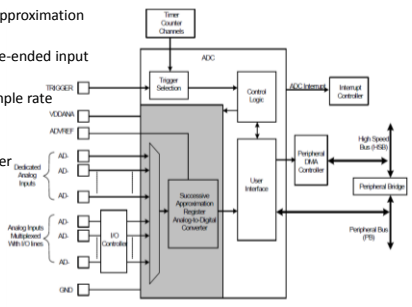
Microphone + Preamp

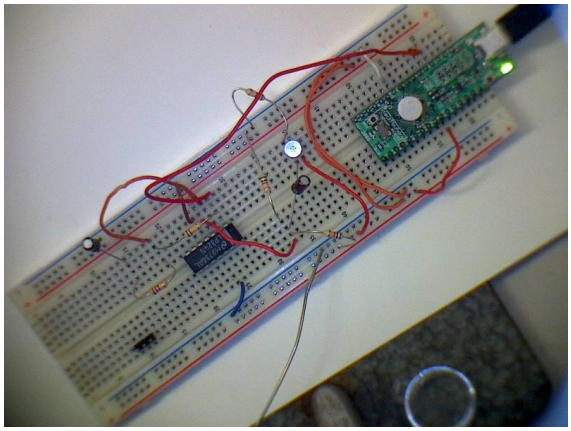


Servo power supply

AVR32 Analog to Digital converter

- 10-bit Successive approximation register (SAR) type
- 6 multiplexed single-ended input channels
- Max combined sample rate 384ks/s
- External trigger
- Hardware sequencer
- Peripheral DMA

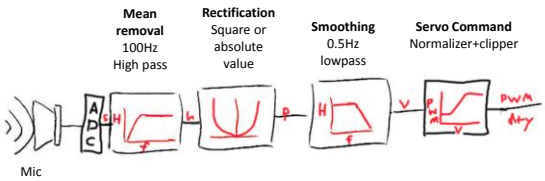




- Fixed-point digital signal processing pipeline
- Using timer interrupts for regular ADC sampling intervals

Signal processing pipeline

produces servo position corresponding to average sound volume



Some more about ADCs

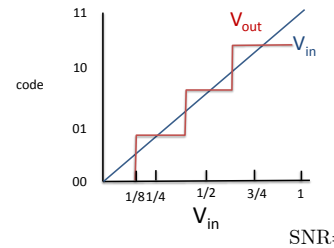
High resolution Low speed and power	Medium resolution Medium power	Low resolution but fast and hot
Single slope (imprecise)	SAR (good tradeoffs, most uC)	Flash (video rate, oscilloscopes)
Dual slope (precise but very slow)	Algorithmic (ΣΔ)	2-step

ADC specifications

INL	Integral nonlinearity	Max absolute sample deviation in bits
DNL	Differential nonlinearity	Max possible step size variation in bits
Sample rate		
Latency	In samples	How long in samples it takes for a conversion (can be >>1 for pipelined converter)
Reference voltage	Volts	Minimum resolution

"Quantization noise"

2-bit converter



$$V_Q = V_{out} - V_{in}$$

$$\overline{V_Q^2} = \frac{V_{LSB}^2}{12}$$

$$V_{QRMS} = \frac{V_{LSB}}{3.5}$$

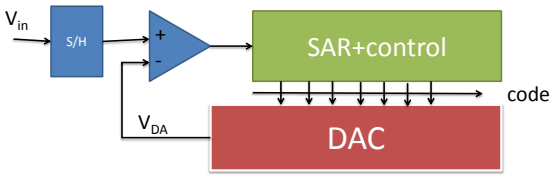
Max possible SNR? (Signal power/Noise power).
For uniformly distributed signal like a sawtooth, we get

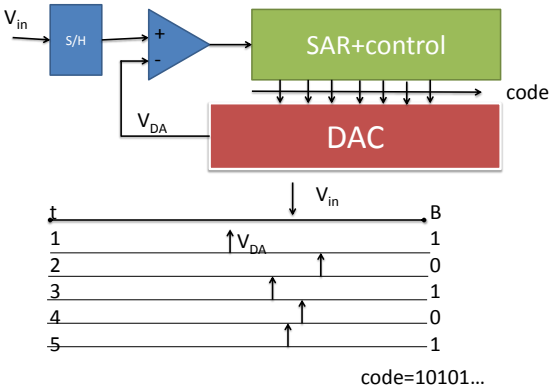
$$\mathrm{SNR} = \frac{\left(\frac{V_{REF}}{12}\right)^2}{\frac{V_{LSB}}{12}} = 2^N$$

$$= 20\log_{10} 2^N \,\mathrm{dB} = 6N \,\mathrm{dB}$$

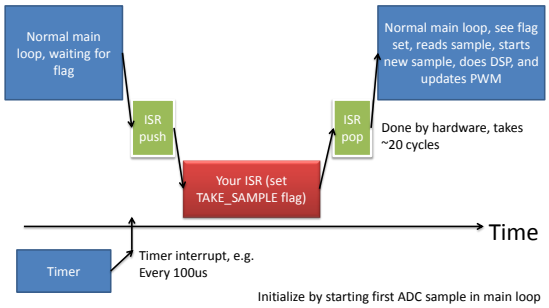
e.g. for N=10, SNR=60dB

Successive Approximate Register (SAR) ADC





Using timer interrupts for regular ADC sampling intervals in an Interrupt Service Routine (ISR)



ISR

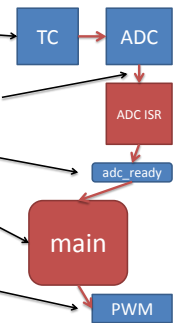
```
static void tc_irq(void) {
   // Increment the counter, which is also used to
   determine servo updates
   tc_tick++;

   // set a flag to tell main loop to take a sample
   takeSampleNow = TRUE;

   // Clear the interrupt flag. This is a side effect
   of reading the TC SR.
   tc_read_sr(EXAMPLE_TC, TC_CHANNEL);

   // Toggle a GPIO pin (this pin is used as a regular
   GPIO pin).
   gpio_local_tgl_gpio_pin(AVR32_PIN_PA10); // debug,
   should toggle at desired sample rate
}
```

Timer Counter (TC) setup

```
void init_tc(){
    static const tc_waveform_opt_t WAVEFORM_OPT = {// Timer/Counter Options for waveform generation.
    .channel  = TC_CHANNEL,                 // Channel selection.
    .wavsel   = TC_WAVEFORM_SEL_UP_MODE_RC_TRIGGER,// Waveform selection: Up mode with automatic
    trigger(reset) on RC compare.
    .tcclks   = TC_CLOCK_SOURCE_TC2         // Internal source clock 2, connected to
    fPBA/2=15.5MHz.
    };
    static const tc_interrupt_t TC_INTERRUPT = {//! Timer/counter interrupts.
    .cpcs  = 1,//! RC compare interrupt. Interrupt with counter reaching Reset Count value (RC)
    };
    volatile avr32_tc_t *tc = EXAMPLE_TC;
    Disable_global_interrupt();
    INTC_register_interrupt(&tc_irq, EXAMPLE_TC_IRQ, AVR32_INTC_INT0); // Register the RTC interrupt
    handler to the interrupt controller.
    Enable_global_interrupt();
    // Initialize the timer/counter.
    tc_init_waveform(tc, &WAVEFORM_OPT); // Initialize the timer/counter waveform.
    // Set the compare triggers for timer/counter (TC).
    // TC counter is 16-bits, with secondary clock TIMER_CLOCK2 = FPBA clock/2  = 33 MHz/2=15.5MHz.
    // Lowest possible freq is 15.5MHz/(2^16)=236Hz.
    // We want ADC sample rate of FADC Hz.  To get this, we load RC (Reset Counter) value so that
    // TC reaches RC value every 1/FADC s. Therefore we configure TC so that RC=FPBA/FADC.
    // E.g., to get FADC=10kHz, we need RC=15.5MHz/10000=1550.
    // The timer interrupt will then run at 10kHz (verified on scope).
    // The timer interrupt for debug toggles PA10 which will result in a square wave at 5kHz (verified)
    tc_write_rc(tc, TC_CHANNEL, (FPBA /2) / FADC); // Set RC value.
    tc_configure_interrupts(tc, TC_CHANNEL, &TC_INTERRUPT);
    tc_start(tc, TC_CHANNEL); // Start the timer/counter.
}
```

Alex Hungenberg tried the following hardware-driven approach

1. The Timer-Counter is set up to trigger the ADC at 40kHz.
2. When the ADC completes a conversion, it generates an ISR
3. The ISR sets a flag
4. The main loop sees the flag set, reads the ADC value, does DSP, and sets the servo PWM pulse width.



What's wrong with this ISR?

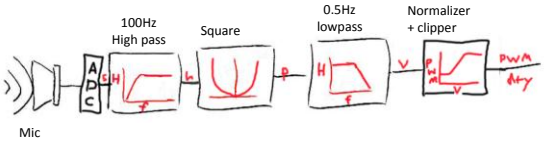
```
__attribute__((__interrupt__))
static void adc_int_handler(void) {
    adc_ready = 1;
//  volatile avr32_adc_t *adc = &AVR32_ADC;
//  update_pwm(*((unsigned long * )(&(adc->cdr0))));
}
```

ADC status register | 25.7.6 Name: | Status Register SR | End Of Conversion bit | 2 EOC2

- **EOCn: End of Conversion n**
  These bits are set when the corresponding conversion is complete.
  These bits are cleared when the corresponding CDR or LCDR registers are read.
  0: Corresponding analog channel (if implemented) is disabled, or the conversion is not finished.
  1: Corresponding analog channel (if implemented) is enabled and conversion is complete.

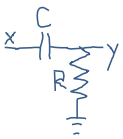
1. The ISR is triggered by the ADC.
2. Because Channel Data Register is not read in the ISR, the interrupt flag is not cleared.
3. **The interrupt is immediately re-triggered, so the main loop always sees the flag set.**

Fixed point signal processing pipeline



We need a digital low & high pass filters, like an RC or CR filter

A simple IIR high pass filter (discrete time)



$$\frac{y}{R} = C(\dot{x} - \dot{y})$$

$$RC\dot{y} + y = RC\dot{x}$$

$$\tau\dot{y} + y = \tau\dot{x}$$

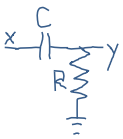
$$\tau\left(\frac{y_{t+\delta t} - y_t}{\delta t}\right) + y_t = \tau\left(\frac{x_{t+\delta t} - x_t}{\delta t}\right)$$

$$\alpha = \frac{\delta t}{\tau}$$

$$y_{t+\delta t} = y_t - \alpha y_t + x_{t+\delta t} - x_t$$

$$= (1-\alpha)y_t + x_{t+\delta t} - x_t$$

A simple IIR high pass digital filter
(fixed point, using binary shift operations)



$$y_{t+\delta t} = (1-\alpha)y_t + x_{t+\delta t} - x_t$$

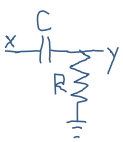
If $\alpha = \frac{1}{2^n}$, then

$$(1-\alpha)y_t = \frac{2^n - 1}{2^n}y_t = \left[\left(y_t \ll n\right) - 1\right] \gg n$$

$$y_{t+\delta t} = \left[\left(y_t \ll n\right) - 1\right] \gg n + \left(x_{t+\delta t} - x_t\right)$$

What is the time constant?

$$\alpha = \frac{\delta t}{\tau}$$



Suppose $\delta t = 100us$ (10kHz sample rate)

and $\alpha = 1/256$ (n=8).

Then

$\tau = 100us \times 256 = 25.6ms$

Corner frequency $f_{3dB} = \frac{1}{2\pi\tau} = 6.2Hz$

To filter with n times longer time constant, you can skip n samples

DSP code sample

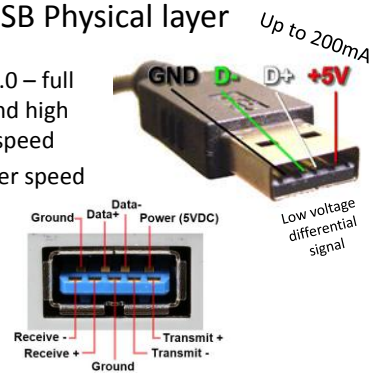
```
void device_task(void) {

if (takeSampleNow) {  // flag set in timer ISR
gpio_local_tgl_gpio_pin(AVR32_PIN_PA11); // debug
takeSampleNow=FALSE;
// signal processing
s16 adcval = (s16)get_adc_value(); // 0-1023=3.3v

if (initialized)
        audMean = ((adcval-audMean)>>NTAU1)+audMean; // TODO mix old and new value
else
        audMean = adcval; // init filter with first reading

// only update meanSq at TAU2 interval, so to produce effective time constant that
is TAU2 times tau of audMean filtering
if(dspCounter--==0){
    dspCounter=TAU2;
    long diff = adcval - audMean; // signed diff of sample from mean
    long sq = diff * diff; // square diff
    if (initialized)
        meanSq = ((sq-meanSq)>>NTAU1)+meanSq; // low pass square diff
    else
        meanSq = sq;
}
```

USB – Universal Serial Bus

- Physical layer
- User perspective (coder)
- Under the hood
  - Device side
  - Host side
- Achieving high performance

USB Physical layer

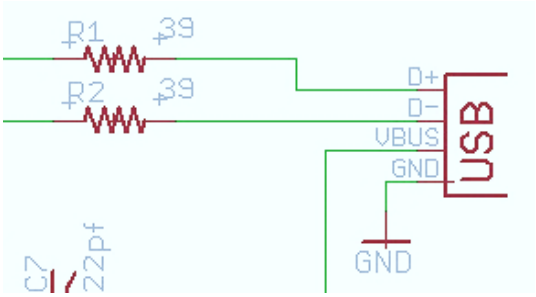
- Up to USB 2.0 – full (12Mbps) and high (480Mbps) speed
- USB 3.0 super speed (5Ggbs)



USB definitions

- IN means towards the host (the PC)
- OUT means towards the device (uC)

USB on PCB



USB user perspective (as coder)

Device side – Include USB driver

```
#include "usb_task.h"

#if USB_DEVICE_FEATURE == ENABLED
#include "usb_drv.h"
#include "usb_descriptors.h"
#include "usb_standard_request.h"
#include "device_task.h"
#endif
```

```
static U32 sof_cnt;

static U8 in_data_length;
static U8 in_buf[EP_SIZE_TEMP1];

static U8 out_data_length;
static U8 out_buf[EP_SIZE_TEMP2];

//!
//! @brief This function initializes
//!
void device_task_init(void) {
    sof_cnt = 0;
    in_data_length = 0;
    out_data_length = 0;

    Usb_enable_sof_interrupt();
//! This function increments the sof_cnt counter each time
//! the USB Start-of-Frame interrupt subroutine is executed (1 ms).
//! Useful to manage time delays
//!
void usb_sof_action(void) {
    gpio_local_tgl_gpio_pin(AVR32_PIN_PA10);
    sof_cnt++;
}
```

Main loop

```
int main() {

    usb_task_init();

#if USB_DEVICE_FEATURE == ENABLED
    device_task_init();
#endif
    while (TRUE) {
        usb_task();

#if USB_DEVICE_FEATURE == ENABLED
        device_task();
#endif
    }
```

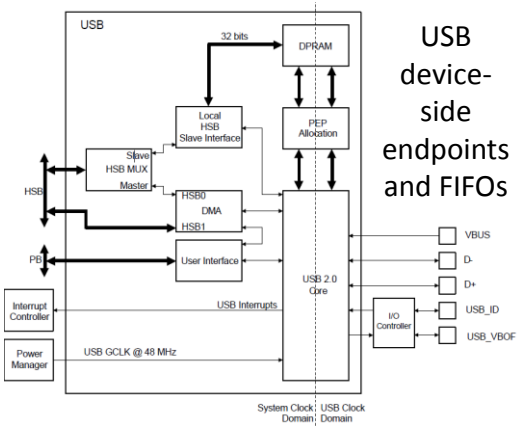
IN direction

```
void device_task(void) {
    // First, check the device enumeration state
    if (!Is_device_enumerated())
        return;

    // HERE STARTS THE USB DEVICE APPLICATIVE CODE
    // The example below just performs a loopback transmission/reception.
    // All data received with the OUT endpoint is stored in a RAM buffer and
    // sent back to the IN endpoint.


    // Load the IN endpoint with the contents of the RAM buffer
    if (Is_usb_in_ready(EP_TEMP_IN)) {
        gpio_local_tgl_gpio_pin(AVR32_PIN_PA11);

        // read ADC and store to buffer...:
        U16 adcval = get_adc_value();
        in_buf[0] = 0xDE;
        in_buf[1] = 0xAD;
        in_buf[2] = 0xFF & (adcval >> 8);
        in_buf[3] = 0xFF & (adcval >> 0);
        in_data_length = 4;

        Usb_reset_endpoint_fifo_access(EP_TEMP_IN);
        usb_write_ep_txpacket(EP_TEMP_IN, in_buf, in_data_length, NULL);
        in_data_length = 0;
        Usb_ack_in_ready_send(EP_TEMP_IN);

    }
```

OUT direction

```
// If we receive something in the OUT endpoint,
// just store it in the RAM buffer
if (Is_usb_out_received(EP_TEMP_OUT)) {
    gpio_local_tgl_gpio_pin(AVR32_PIN_PA12);

    Usb_reset_endpoint_fifo_access(EP_TEMP_OUT);
    out_data_length = Usb_byte_count(EP_TEMP_OUT);
    usb_read_ep_rxpacket(EP_TEMP_OUT, out_buf, out_data_length, NULL);
    Usb_ack_out_received_free(EP_TEMP_OUT);

    // update PWM:
    set_rgb(out_buf[1], out_buf[2], out_buf[3]);
}
```

USB device-side endpoints and FIFOs



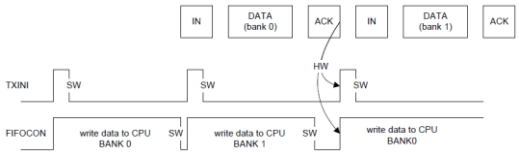
USB interrupts (you don't worry about them!)

Endpoints – multiple virtual channels

Pipe/Endpoint	Mnemonic	Max. Size	Max. Nb. Banks	DMA	Type
0	PEP0	64 bytes	1	N	Control
1	PEP1	64 bytes	2	Y	Isochronous/Bulk/Interr
2	PEP2	64 bytes	2	Y	Isochronous/Bulk/Interr
3	PEP3	64 bytes	2	Y	Isochronous/Bulk/Interr
4	PEP4	64 bytes	2	Y	Isochronous/Bulk/Interr
5	PEP5	256 bytes	2	Y	Isochronous/Bulk/Interr
6	PEP6	256 bytes	2	Y	Isochronous/Bulk/Interr

Can be double buffered

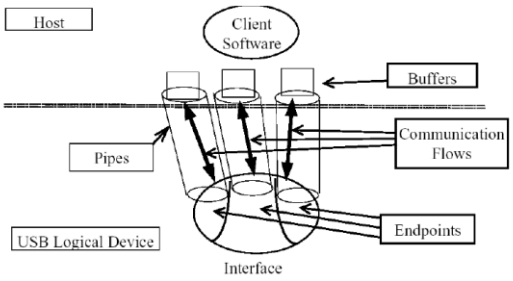
Double-buffered transfers can increase continuity



- When the bank is empty, TXINI and FIFOCON are set, what triggers an EPnINT interrupt if TXINE is one.
- The user acknowledges the interrupt by clearing TXINI.
- The user writes the data into the current bank by using the USB Pipe/Endpoint nFIFO Data virtual segment (see "USB Pipe/Endpoint n FIFO Data Register (USBFIFOnDATA)" on page 483), until all the data frame is written or the bank is full (in which case RWALL is cleared and the Byte Count (BYCT) field in UESTAn reaches the endpoint size).
- The user allows the controller to send the bank and switches to the next bank (if any) by clearing FIFOCON.

Host vs. Device

For the USBB in host mode, the term "pipe" is used instead of "endpoint" (used in device mode).
A host pipe corresponds to a device endpoint



Host side – using *pyusb*

```
#!/usr/bin/env python
import sys
import array
import usb
import colorsys
import time

busses = usb.busses()

VENDOR = 0x03eb
PRODUCT = 0x2300
IFACE = 0
EP_IN = 0x81
EP_OUT = 0x02

def get_device():
    for bus in busses:
        devices = bus.devices
        for dev in devices:
            if dev.idVendor == VENDOR and dev.idProduct == PRODUCT:
                return dev
    return None
```

```
def main():

    dev = get_device()
    try:
        dh = dev.open()
        dh.claimInterface(IFACE)

        rainbow(dh)

        dh.releaseInterface()
        del dh
        return 0
    except :
        print "no avr32 found?", sys.exc_info()
```

```
usbiocnt = 0L
def usbio(dh, r, g, b):
    global usbiocnt
    usbiocnt += 1

    dout = array.array('B', [0]*4)

    dout[0] = 0xFF & 0x00
    dout[1] = 0xFF & (r)
    dout[2] = 0xFF & (g)
    dout[3] = 0xFF & (b)

    dh.bulkWrite(EP_OUT, dout.tostring())

    #if usbiocnt % PWMperADC == 0:
    if 1:

        din = dh.bulkRead(EP_IN, 4)
        l = len(din)
        if l != 4:
            print "unexpected bulk read length: %d" % l
        else:
            if usbiocnt % PWMperADC == 0:
                adc( (din[2] << 8) + din[3] )
```

The key to high performance on host side: *Asynchronous* or *Overlapped* IO

- On the host side, an Input-Output (IO) thread manages the USB IO.
- Multiple buffers (which can be much larger than the device FIFO size) are submitted to the USB driver / host controller to be filled by the USB controller.

1. When a buffer is filled, the IO thread is notified asynchronously, which wakes it up.
2. The IO thread processes the buffer, and then gives it back to the controller. The IO thread then notifies the main user code that data is available, e.g. by writing to a software queue.

- That way, the user doesn't *block* waiting for data
- Our *pyusb* example doesn't do this yet

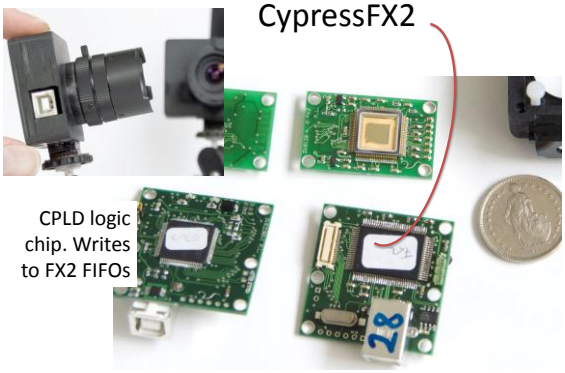
USB performance

- USB full speed (12Mbps): about 1MBps
- USB high speed (480Mbps): about 40MBps
- USB super speed (5Gbps): ??

ICs for USB

USB full speed	• Many uC. Also FTDI.
USB high speed	• CypressFX2
USB super speed	• CypressFX3

CypressFX2



CypressFX3

Cypress EZ-USB® FX3™ is the next-generation SuperSpeed USB 3.0 peripheral controller that enables developers to add USB 3.0 device functionality to any system.

EZ-USB FX3 has a fully configurable, General Programmable Interface (GPIF™ II) that can interface with any processor, ASIC, image sensor, or FPGA. GPIF™ II is an enhanced version of the original GPIF™ in FX2LP, Cypress's flagship USB 2.0 product. It provides easy and glue-less connectivity to popular industry interfaces such as

ftdichip.com

- uC UART – USB interface; looks like COM serial port on host side.
- Max speed is only 12Mbaud for the UART port unfortunately