USB support for Atmels AT90USB microcontroller
Enumeration (with Linux host) and basic data transfer (LibUSB) works fine now!
A simple example program for Linux console is included now.
Basic documentation should be available on my homepage, see http :// www.ssalewski.de/Misc.html
License: GPL−2
S. Salewski 2006/2007

USB Documentation:
H.J. Kelm, USB 2.0, FRANZIS
J. Axelson, USB Complete, Lakeview Research
www.beyondlogic.org/usbnutshell/usb−in−a−nutshell.pdf
Atmel: Datasheet for AT90USB
Atmel: AVR329, USB Firmware Architecture

Files :

AT90USB−Errata.txt	Some remarks about errors or unclear statements in AT90USB datasheet
com_def.h	Constants which are used by firmware and host application program
daq_dev.c	Function for data acquisition using the intern ADC (port ADC0)
daq_dev.h	
defines.h	Project specific constants
macros.h	Some general macros
Makefile	Makefile: type "make" to generate all binary files and "make dfu" to upload firmware
README	This file
ringbuffer.c	Simple ring buffer (FIFO) for buffering ADC data
ringbuffer.h	
SUDAP.c	Example firmware application : Data acquisition (port ADC0) and control of digital port B
SUDD.c	Example host application : Console based data acquisition and control of digital port B
usart_debug.c	Functions for debugging over serial port
usart_debug.h	
usart_drv.c	Access to serial port
usart_drv.h	
usb_api.c	This file contains constants and functions which need modifications
usb_api.h	to customize the software for a specific device
usb_drv.c	Basic USB functions for AT90USB controller
usb_drv.h	USB macros, basic low level register stuff
usb_isr.c	Interrupt service routines
usb_requests.c	USB (standard) requests , enumeration
usb_requests.h	
usb_spec.c	USB datastructures and constants
usb_spec.h	
gen_postscript.txt	Script for generating postscript from sources for printout using enscript
gen_pdf.tex	Small LaTeX file using listings package for pretty−printing sources
AT90USB−20070xyz.tar.gz	All source files in compressed form
AT90USB−20070xyz.zip	All source files in zip compressed form
sources.ps.gz	All source files preformated for printout (enscript)
sources.pdf	All source files preformated for printout (pdflatex)

```
// AT90USB/SUDAP.c
// Simple Usb Data Acquisition Program
// Sampling an analog voltage using the intern ADC of AT90USB chip and
// sending data to host (Linux PC) over USB in real time.
// Sampling interval and number of samples is selected with command line options.
// Additionally you can set output pins of digital port B.
// The main purpose of this tiny program is to demonstrate the functionality of my
// USB firmware written for Atmels AT90USB controller.
// Licence: GPL-2
// Build: gcc -l usb SUDAP.c
// Usage: see function usage() below.
// S. Salewski, 24-MAR-2007

#include <stdio.h>
#include <stdint.h>
#include <getopt.h>
#include <errno.h>
#include <usb.h>
#include "com_def.h"
#include "daq_dev.h"

#define TimeOut 2000 // ms

const char trstr [] = "Use 1ms, 500us, 200us, 100us, 50us or 20us for time resolution !";

static struct option long_options [] =
{
  {"timeres",    required_argument,    0, 't'},
  {"samples",    required_argument,    0, 's'},
  {"columns",    required_argument,    0, 'c'},
  {"portb",      required_argument,    0, 'p'},
  {"help",       no_argument,          0, 'h'},
  {0, 0, 0, 0}
};

static void
usage(void)
{
  puts("Usage: -t 100us -s 100 -c 1 -p 10001101");
  printf ("-t or --timeres: ");
  puts( trstr );
  puts("-s or --samples: number of samples (1 up to 65535)");
  puts("-c or --columns: number of columns for output, default is 32");
  puts("-p or --portb: output value for digital port b, use a 8-bit dual number");
}

static struct usb_device
*UsbFindDevice(void)
{
  struct usb_bus *bus;
  struct usb_device *dev;
  usb_init ();
  usb_find_busses ();
  usb_find_devices ();
  for (bus = usb_busses; bus; bus = bus->next)
  {
    for (dev = bus->devices; dev; dev = dev->next)
    {
```

```
      if ((dev->descriptor.idVendor == MyUSB_VendorID) && (dev->descriptor.idProduct == MyUSB_ProductID))
        return dev;
    }
  }
  return NULL;
}

// We use EP2 for DAQ
static void
dodaq(struct usb_dev_handle *handle, uint16_t timeres, uint16_t samples, int columns)
{
  unsigned char buf[EP2_FIFO_Size];
  unsigned char *b;
  unsigned int m;
  unsigned int c = 0;
  if (usb_control_msg(handle, USB_VendorRequestCode, UC_ADC_Read, (int) timeres, (int) samples, NULL, 0, TimeOut) ↩
      < 0)
    puts("USB_control_msg error!"); // should never occur
  while (samples)
  {
    if (samples > (EP2_FIFO_Size/2))
      m = (EP2_FIFO_Size/2);
    else
      m = samples;
    if (usb_bulk_read(handle, 2, (char*) buf, m * 2, TimeOut) < (m * 2))
      puts("USB_bulk_read error!"); // should never occur
    b = buf;
    samples -= m;
    while (m--)
    {
      unsigned int k;
      k = (unsigned int) *b++;
      k += (unsigned int) *b++ * 256;
      c++;
      if (c == columns)
      {
        c = 0;
        printf ("%u\n", k);
      }
      else
        printf ("%u ", k);
    }
  }
  if (c) putchar('\n');
}

int
main(int argc, char **argv)
{
  struct usb_device *dev;
  struct usb_dev_handle *handle;
  int port = -1;
  uint16_t samples = 0;
  uint16_t timeres = 0;
  int columns = 32;
  uint8_t status ;
  if (argc == 1)
  {
```

```
    usage();
    exit(EXIT_FAILURE);
  }
  while (1)
  {
    char * tail;
    long int l;
    int option_index = 0;
    int c;
    c = getopt_long(argc, argv, "t:s:p:c:h", long_options, &option_index);
    if (c == -1) break;
    switch (c)
    {
      case 'h':
        usage();
        exit(EXIT_FAILURE);
      case 'p':
        errno = 0;
        l = strtol(optarg, &tail, 2);
        if ((errno == 0) && (*tail == '\0') && (l >= 0) && (l < 256))
          port = (int) l;
        else
        {
          puts("Use a 8-bit binary number for setting port pins!");
          exit(EXIT_FAILURE);
        }
        break;
      case 'c':
        errno = 0;
        l = strtol(optarg, &tail, 0);
        if ((errno == 0) && (*tail == '\0') && (l > 0) && (l <= 256))
          columns = (int) l;
        else
        {
          puts("Use a positive integer <= 256 for number of columns!");
          exit(EXIT_FAILURE);
        }
        break;
      case 't':
        if ((optarg[0]=='1') && (optarg[1]=='m') && (optarg[2]=='s') && (optarg[3]==0))
          timeres = ADC1ms;
        else if ((optarg[0]=='5') && (optarg[1]=='0') && (optarg[2]=='0') && (optarg[3]=='u') && (optarg[4]=='s') ↵
            && (optarg[5]==0))
          timeres = ADC500us;
        else if ((optarg[0]=='2') && (optarg[1]=='0') && (optarg[2]=='0') && (optarg[3]=='u') && (optarg[4]=='s') ↵
            && (optarg[5]==0))
          timeres = ADC200us;
        else if ((optarg[0]=='1') && (optarg[1]=='0') && (optarg[2]=='0') && (optarg[3]=='u') && (optarg[4]=='s') ↵
            && (optarg[5]==0))
          timeres = ADC100us;
        else if ((optarg[0]=='5') && (optarg[1]=='0') && (optarg[2]=='u') && (optarg[3]=='s') && (optarg[4]==0))
          timeres = ADC50us;
        else if ((optarg[0]=='2') && (optarg[1]=='0') && (optarg[2]=='u') && (optarg[3]=='s') && (optarg[4]==0))
          timeres = ADC20us;
        else
        {
          puts( trstr );
          exit(EXIT_FAILURE);
```

```
        }
        break;
      case 's':
        errno = 0;
        l = strtol(optarg, &tail, 0);
        if ((errno == 0) && (*tail == '\0') && (l > 0) && (l <= MaxSamples))
          samples = (uint16_t) l;
        else
        {
          puts("Use 0 < samples < 65536!");
          exit(EXIT_FAILURE);
        }
        break;
      case '?':
        exit(EXIT_FAILURE);
        break;
      default:
        abort();
    }
  }
  if (optind < argc)
  {
    printf("Unsupported arguments: ");
    while (optind < argc)
      printf("%s ", argv[optind++]);
    putchar('\n');
    exit(EXIT_FAILURE);
  }
  if ((!samples) && (timeres))
  {
    puts("Required option: -s or --samples");
    exit(EXIT_FAILURE);
  }
  if ((samples) && (!timeres))
  {
    puts("Required option: -t or --timeres");
    exit(EXIT_FAILURE);
  }
  dev = UsbFindDevice();
  if (dev == NULL)
  {
    puts("USB device not found!");
    exit(EXIT_FAILURE);
  }
  handle = usb_open(dev);
  if (handle == NULL)
  {
    puts("USB device handle not found!");
    exit(EXIT_FAILURE);
  }
  if ( usb_claim_interface(handle, 0) < 0)
  {
    puts("Can not claim interface !");
    usb_close(handle);
    exit(EXIT_FAILURE);
  }
  if (port != -1)
  {
```

```
      status = (uint8_t) port;
      if (usb_bulk_write(handle, 3, (char*) &status, 1, TimeOut) < 1)
        puts("Can not set digital port B!");
    }
    if (samples && timeres)
    {
      dodaq(handle, timeres, samples, columns);
      usb_bulk_read(handle, 1, (char*) &status, 1, TimeOut);
      switch (status)
      {
        case (UsbDevStatusOK):
          break;
        case (UsbDevStatusDAQ_BufferOverflow):
          puts("DAQ Buffer Overflow!");
          break;
        case (UsbDevStatusDAQ_TimerOverflow):
          puts("DAQ Timer Overflow!");
          break;
        case (UsbDevStatusDAQ_ConversionNotFinished):
          puts("DAQ Conversion was not finished when reading result !");
          break;
        default:
          puts("Unknown error has occurred!");
      }
    }
     usb_release_interface (handle, 0);
    usb_close(handle);
    exit(EXIT_SUCCESS);
}
```

```
// AT90USB/SUDD.c
// Simple Usb Data-acquisition Device
// S. Salewski 23-MAR-2007
// Start minicom with 8N1 configuration to see debug messages
// The real work happens inside of interrupt-service-routines

#include <avr/io.h>
#include <avr/interrupt.h>
#include "usart_drv.h"
#include "usb_drv.h"

static void blinkforever (void);

int
main(void)
{
  cli();
  CLKPR = (1<<7);
  CLKPR = 0; // clock prescaler == 0, so we have 16 MHz mpu frequency with our 16 MHz crystal
  USART_Init();
  USART_WriteString("\r\n
      ----------------------------------------------------------------------\r\n");
  UsbDevLaunchDevice(false);
  // UsbDevWaitStartupFinished(); // no reason to wait
  blinkforever();
  return 0;
}

// a LED connected to PORTA0 will toggle to indicate the unused processing power
static void
blinkforever (void)
{
  int i = 0; // to suppress warning of avr-gcc
  DDRA |= (1<<DDA0);
  while (1)
  {
    while (i++);
    PORTA ^= (1<<PORTA0);
  }
}
```

```
// AT90USB/daq_dev.h
// Very simple data-acquisition device
// S. Salewski 19-MAR-2007

#ifndef _DAQ_DEV_H_
#define _DAQ_DEV_H_

#include <stdint.h>

// user commands
#define UC_ADC_Read   1

// sampling time
#define ADC20us       1
#define ADC50us       2
#define ADC100us      3
#define ADC200us      4
#define ADC500us      5
#define ADC1ms        6

extern uint8_t DAQ_Result;

void ProcessUserCommand(uint8_t command, uint16_t par1, uint16_t par2);

#endif
```

```
// AT90USB/daq_dev.c
// This file implements basic functionality of a simple data-acquisition device
// S. Salewski 23-MAR-2007

#include <avr/io.h>
#include <avr/interrupt.h>
#include "macros.h"
#include "usb_drv.h"
#include "usart_debug.h"
#include "ringbuffer.h"
#include "daq_dev.h"
#include "com_def.h"

uint8_t DAQ_Result;
static uint16_t SamplesToRead;

void
ProcessUserCommand(uint8_t command, uint16_t par1, uint16_t par2)
{
  uint8_t prescalerADC;
  uint8_t prescalerT0;
  uint8_t counterT0Max;
  switch (command)
  {
    case UC_ADC_Read:
      switch ((uint8_t) par1)
      {
        case ADC20us:
          prescalerADC = ((1<<ADPS2)); // mpu clock / 16
          prescalerT0 = (1<<CS01); // == 8
          counterT0Max = 40;
          break;
        case ADC50us:
          prescalerADC = ((1<<ADPS2) | (1<<ADPS0)); // mpu clock / 32
          prescalerT0 = (1<<CS01); // == 8
          counterT0Max = 100;
          break;
        case ADC100us:
          prescalerADC = ((1<<ADPS2) | (1<<ADPS1)); // mpu clock / 64
          prescalerT0 = (1<<CS01); // == 8
          counterT0Max = 200;
          break;
        case ADC200us:
          prescalerADC = ((1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)); // mpu clock / 128
          prescalerT0 = ((1<<CS01) | (1<<CS00)); // == 64
          counterT0Max = 50; // 400 / 8
          break;
        case ADC500us:
          prescalerADC = ((1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)); // mpu clock / 128
          prescalerT0 = ((1<<CS01) | (1<<CS00)); // == 64
          counterT0Max = 125;
          break;
        case ADC1ms:
          prescalerADC = ((1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)); // mpu clock / 128
          prescalerT0 = ((1<<CS01) | (1<<CS00)); // == 64
          counterT0Max = 250;
          break;
        default:
```

```
            Debug("DaqDev: Unsupported sampling rate!\r\n");
            return;
        }
        ADMUX = (1<<REFS1) | (1<<REFS0); // ADC0 single ended input, internal 2.56V reference with external capacitor on AREF pin
        ADCSRA = prescalerADC | (1<<ADEN) | (1<<ADSC); // set prescaler, enable ADC and start conversion
        TCCR0A = (1<<WGM01); // set timer registers: clear on compare match
        TCCR0B = prescalerT0; // set prescaler
        OCR0A = counterT0Max; // set output compare register
        SamplesToRead = par2;
        RB_Init();
        DAQ_Result = UsbDevStatusOK;
        prescalerT0 = 20;  // do a few conversions to warm up
        while (prescalerT0-- > 0)
        {
          while (ADCSRA & (1<<ADSC));
          ADCSRA |= (1<<ADSC);
        }
        TCNT0 = counterT0Max + 1; // set start value behind compare match -- we need some time to finish this USB-ISR
        TIMSK0 = (1<<OCIE0A); // activate compare match interrupt
        TIFR0 |= (1<<OCF0A); // clear compare match flag -- we will wait for next compare match
        break;
      default:
        Debug("DaqDev: Unsupported user command!\r\n");
    }
}

ISR(TIMER0_COMPA_vect)
{
  uint16_t w;
  // static uint8_t cnt;
  if (ADCSRA & (1<<ADSC)) // conversion finished?
    DAQ_Result = UsbDevStatusDAQ_ConversionNotFinished;
  w = ADCW;
  ADCSRA |= (1<<ADSC); // start next conversion as soon as possible
  if (SamplesToRead > 0)
  {
    if (RB_IsFull())
    {
      DAQ_Result = UsbDevStatusDAQ_BufferOverflow;
    }
    else
    {
      if (DAQ_Result == UsbDevStatusOK)
        RB_Write(w);
      else
        RB_Write(0xFFFF);
      SamplesToRead--;
    }
  }
  UsbDevSelectEndpoint(2);
  if (UsbDevTransmitterReady())
  {
    uint8_t i;
    if ((SamplesToRead == 0) && (RB_Entries == 0))
    {
      TIMSK0 = 0;
```

```
      i = 64;
    }
    else
      i = UsbDevGetByteCountLow();
    switch (i)
    {
      default:
        w = RB_Read();
        // w = cnt; dump counter to FIFO for testing
        UsbDevWriteByte(LSB(w));
        UsbDevWriteByte(MSB(w));
      case 62:
        if (RB_Entries > 0)
        {
          w = RB_Read();
          UsbDevWriteByte(LSB(w));
          UsbDevWriteByte(MSB(w));
        }
        break;
      case 64:
        UsbDevClearTransmitterReady();
        UsbDevSendInData();
    }
  }
  if (TIFR0 & (1<<OCF0A)) // is interrupt flag already set again?
  {
    DAQ_Result = UsbDevStatusDAQ_TimerOverflow;
  }
  // cnt = TCNT0; if there are too many instructions in this ISR, we may get timer-counter overflow
}
```

```
// AT90USB/usb_api.h
// Usb_api contains constants and functions which are project specific.
// You may modify this file (and usb_api.c) for your application.
// S. Salewski 21-MAR-2007

// The current version of this file is adapted to a simple Data-Acquisition-Device.
// Our DAQ device uses endpoints 0, 1, 2 and 3.
// Commands are send to our device by vendor-requests using endpoint 0.
// Endpoint 1 is configured as Bulk-IN-endpoint and used to transfer status information
// from device to the host. Whenever the host tries to read data from empty endpoint 1,
// a NAK-interrupt is generated, causing firmware to write status information to this ep.
// Endpoint 2 is configured an Bulk-IN-Endpoint and used to send DAQ-data to the host.
// DAQ operation starts when host sends a vendor request. Controlled by a periodically
// activated timer interrupt, daq data is put to the FIFO of endpoint 2.
// Endpoint 3 is configured as Bulk-OUT-Endpoint and used to send a single byte from the
// host to the device. This byte sets the pins of digital port B of our AT90USB.

#ifndef _USB_API_H_
#define _USB_API_H_

#include <stdint.h>
#include <stdbool.h>
#include "usb_spec.h"

// USB Vendor ID is assigned by www.usb.org
// Vendor- and product id is defined in com_def.h, because we need it in our host program too!
//#define MyUSB_VendorID      0x03eb // Atmel code
//#define MyUSB_ProductID     0x0001
#define MyUSB_DeviceBCD 0x0001

// modify these values for your application!
#define USB_NumConfigurations          1 // 1 or more
#define USB_MaxConfigurations          4 // our device can have multiple configuration with different number of ↩
    interfaces
#define USB_MaxInterfaces              4 // each interface of a configuration has a number of endpoints
#define USB_MaxStringDescriptorLength 22

// An interface can have multiple alternate settings, but number of endpoints of this interface is fix.
// To ensure that reallocation of endpoints with new FIFO size of one interface will not interfere with other ↩
    interfaces,
// only the interface with highest number should use multiple alternate settings with different endpoint FIFO ↩
    sizes

#define EP0_FIFO_Size 8 // 8, 16, 32 or 64 byte
// FIFO size of EP1 upto EP6 is defined in com_def.h

// These macros are called if an endpoint interrupt is triggered (if enabled)
// and may be used to fill (IN-Endpoint) or read (OUT-Endpoint) the FIFO.
#define UsbDevEP1IntAction() UsbDevFillEP1FIFO() // send status to host
#define UsbDevEP2IntAction() // EP2, used for DAQ data, is filled by timer-ISR
#define UsbDevEP3IntAction() UsbDevReadEP3FIFO() // read data byte for digital port B
#define UsbDevEP4IntAction()
#define UsbDevEP5IntAction()
#define UsbDevEP6IntAction()

// These functions provides the host with device specific USB descriptors during the enumeration process
void UsbGetDeviceDescriptor(USB_DeviceDescriptor *d);
bool UsbGetConfigurationDescriptor(USB_ConfigurationDescriptor *c, uint8_t confIndex);
```

```
bool UsbGetInterfaceDescriptor(USB_InterfaceDescriptor *i, uint8_t confIndex, uint8_t intIndex, uint8_t altSetting);
bool UsbGetEndpointDescriptor(USB_EndpointDescriptor *e, uint8_t confIndex, uint8_t intIndex, uint8_t altSetting, uint8_t endIndex);
void UsbGetStringDescriptor(char s[], uint8_t index);

// These functions allocate FIFO memory and setup all used endpoints
bool UsbDevSetConfiguration(uint8_t c);
bool UsbDevSetInterface(uint8_t conf, uint8_t inf, uint8_t as);

// User defined function, used in our application to start data acquisition
void UsbDevProcessVendorRequest(USB_DeviceRequest *req);

// These functions are called from within ISR and fill or read FIFO
void UsbDevFillEP1FIFO(void);
void UsbDevReadEP3FIFO(void);

#endif
```

```
// AT90USB/usb_api.c
// This file contains functions which are project specific and should be modified
// to customize the USB-driver.
// S. Salewski 21-MAR-2007

#include <stdint.h>
#include <stdbool.h>
#include "defines.h"
#include "com_def.h"
#include "usart_debug.h"
#include "usb_spec.h"
#include "usb_drv.h"
#include "daq_dev.h" // ProcessUserCommand(), DAQ_Result
#include "usb_api.h"
#include "usb_requests.h"

// The behavior of USB devices is specified by different descriptors defined by www.usb.org.
// In the simplest case a device has one Device-Descriptor, one Configuration-Descriptor,
// one Interface-Descriptor and a few Endpoint-Descriptors. For more advanced devices
// there may exist multiple Configuration- and Interface-Descriptors, each with multiple Endpoint-
// Descriptors. We may store all these structures in arrays or linked lists, which takes
// much storage in RAM and is not easy to handle.
// My currently preferred solution to handle the descriptors is to generate them on the fly if
// they are requested by the host. This is simple and consumes very few RAM.
// If a descriptor is needed during enumeration process, one of the functions
// of this file is called to initialize the structure according to specific parameters,
// and then the structure is passed to the USB host. The user has to modify the constants
// and the functions of this file for a specific device. For simple devices this is very
// easy -- for devices with multiple configurations / interfaces it is more demanding.
// You may add constants, variables or functions to this file, i.e. to remember the state of an ep.

// First let us write down the parameters of the endpoints which we want to use.
// We have to remember these facts during enumeration, when filling or reading the FIFO,
// and for the communication of the host with our device.

// The current version of this file is adapted to a simple Data-Acquisition-Device.
// Our DAQ device has only one configuration with one single interface.
// Numbers of configurations, interfaces, alternate settings and endpoints start with 0,
// configuration 0 indicates unconfigured (addressed) state, and endpoint 0 is the control endpoint.
// Our device use control endpoint 0 and data endpoints 1, 2 and 3 of interface 0.
// Commands are send to our device by vendor-requests using endpoint 0.
// Endpoint 1 is configured as Bulk-IN-endpoint and used to transfer status information
// from device to the host. Whenever the host tries to read data from empty endpoint 1,
// a NAK-interrupt is generated, causing firmware to write status information to this ep.
// Endpoint 2 is configured as Bulk-IN-Endpoint and used to send DAQ-data to the host.
// DAQ operation starts when host sends a vendor request. Controlled by a periodically
// activated timer interrupt, daq data is put to the FIFO of endpoint 2.
// Endpoint 3 is configured as Bulk-OUT-Endpoint and used to send a single byte from the
// host to the device. This byte sets the pins of digital port B of our AT90USB.

// To keep it simple: Only one configuration, only one interface.
// EP1: Bulk-IN, 8 bytes FIFO, one bank.
// EP2: Bulk-IN, 64 bytes FIFO, dual bank.
// EP3: Bulk-OUT, 8 bytes FIFO, one bank.

const uint8_t USB_Interfaces[USB_MaxConfigurations] = {1, 0, 0, 0}; // number of interfaces in each configuration
const uint8_t USB_MaxPower_2mA[USB_MaxConfigurations] = {50, 0, 0, 0}; // power consumption of each configuration
    in 2mA units
```

```
const uint8_t USB_AltSettings[USB_MaxConfigurations][USB_MaxInterfaces] =
        {{1, 0, 0, 0}, // number of alt. settings of interfaces of first configuration
         {0, 0, 0, 0}, // number of alt. settings of interfaces of second configuration
         {0, 0, 0, 0},
         {0, 0, 0, 0}};
const uint8_t USB_Endpoints[USB_MaxConfigurations][USB_MaxInterfaces] =
        {{3, 0, 0, 0}, // number of endpoints of interfaces of first configuration
         {0, 0, 0, 0}, // number of endpoints of interfaces of second configuration
         {0, 0, 0, 0},
         {0, 0, 0, 0}};

static void UsbDevDisableAndFreeEndpoint(uint8_t i);
static void UsbDevSetUnconfiguredState(void);

void
UsbGetDeviceDescriptor(USB_DeviceDescriptor *d)
{
  d->bLength = USB_DeviceDescriptorLength;
  d->bDescriptorType = USB_DeviceDescriptorType;
  d->bcdUSB = USB_Spec1_1;
  d->bDeviceClass = UsbNoDeviceClass;
  d->bDeviceSubClass = UsbNoDeviceSubClass;
  d->bDeviceProtocoll = UsbNoDeviceProtokoll;
  d->bMaxPacketSize0 = EP0_FIFO_Size;
  d->idVendor = MyUSB_VendorID;
  d->idProduct = MyUSB_ProductID;
  d->bcdDevice = MyUSB_DeviceBCD;
  d->iManufacturer = USB_ManufacturerStringIndex;
  d->iProduct = USB_ProductStringIndex;
  d->iSerialNumber = USB_SerialNumberStringIndex;
  d->bNumConfigurations = USB_NumConfigurations;
}

bool
UsbGetConfigurationDescriptor(USB_ConfigurationDescriptor *c, uint8_t confIndex)
{
  uint8_t i;
  if (confIndex >= USB_NumConfigurations) return false;
  c->bLength = USB_ConfigurationDescriptorLength;
  c->bDescriptorType = USB_ConfigurationDescriptorType;
  c->wTotalLength = USB_ConfigurationDescriptorLength;
  i = USB_Interfaces[confIndex];
  c->bNumInterfaces = i;
  while (i-- > 0)
    c->wTotalLength += (USB_InterfaceDescriptorLength+USB_EndpointDescriptorLength*USB_Endpoints[confIndex][i])*
        USB_AltSettings[confIndex][i];
  c->bConfigurationValue = UsbConfigurationValue(confIndex);
  c->iConfiguration = UsbNoDescriptionString; // no textual configuration description
  c->bmAttributes = UsbConfDesAttrBusPowered; // bus-powered, no remote wakeup
  c->MaxPower = USB_MaxPower_2mA[confIndex];
  return true;
}

bool
UsbGetInterfaceDescriptor( USB_InterfaceDescriptor *i, uint8_t confIndex, uint8_t intIndex, uint8_t altSetting )
{
  if ((confIndex>=USB_NumConfigurations)||(intIndex>=USB_Interfaces[confIndex])||( altSetting >=USB_AltSettings[
      confIndex][intIndex])) return false;
```

```
  i->bLength = USB_InterfaceDescriptorLength;
  i->bDescriptorType = USB_InterfaceDescriptorType;
  i->bInterfaceNumber = intIndex;
  i->bAlternateSetting = altSetting;
  i->bNumEndpoints = USB_Endpoints[confIndex][intIndex];
  i->bInterfaceClass = UsbNoInterfaceClass;
  i->bInterfaceSubClass = UsbNoInterfaceSubClass;
  i->bInterfaceProtocol = UsbNoInterfaceProtokoll;
  i->iInterface = UsbNoDescriptionString; // no textual interface description
  return true;
}

// Not used for EP0, so 1 <= endIndex <= USB_NumEndpoints <= 6
bool
UsbGetEndpointDescriptor(USB_EndpointDescriptor *e, uint8_t confIndex, uint8_t intIndex, uint8_t altSetting, ↵
    uint8_t endIndex)
{
  if ((confIndex>=USB_NumConfigurations)||(intIndex>=USB_Interfaces[confIndex])||(altSetting>=USB_AltSettings[↵
      confIndex][intIndex]) ||
      (endIndex>USB_Endpoints[confIndex][intIndex])) return false;
  // components identical for all of our endpoints
  e->bLength = USB_EndpointDescriptorLength;
  e->bDescriptorType = USB_EndpointDescriptorType;
  e->bmAttributes = USB_BulkTransfer;
  e->bInterval = 0; // bulk endpoint, no polling
  // components which differ
  switch (endIndex) // only endpoints for interface 0 in our application
  {
    case 1:
      e->bEndpointAddress = UsbInEndpointAdress(1);
      e->wMaxPacketSize = EP1_FIFO_Size;
    break;
    case 2:
      e->bEndpointAddress = UsbInEndpointAdress(2);
      e->wMaxPacketSize = EP2_FIFO_Size;
    break;
    case 3:
      e->bEndpointAddress = UsbOutEndpointAdress(3);
      e->wMaxPacketSize = EP3_FIFO_Size;
    break;
  }
  return true;
}

void
UsbGetStringDescriptor(char s[], uint8_t index)
{
  uint8_t i;
#if (USB_MaxStringDescriptorLength < 18)
#error USB_MaxStringDescriptorLength too small!
#endif
  i = USB_MaxStringDescriptorLength;
  while (i--) *s++ = '\0';
  s -= USB_MaxStringDescriptorLength;
  s[1] = USB_StringDescriptorType;
  switch (index)
  {
    case USB_LanguageDescriptorIndex: // == 0
```

```
      s[0] = 4;
      s[2] = 9; // two byte language code, only support for English
      s[3] = 4;
      break;
    case USB_ManufacturerStringIndex:
      s[2] = 'S';
      s[4] = 'A';
      s[6] = 'L';
      s[8] = 'E';
      s[10] = 'W';
      s[12] = 'S';
      s[14] = 'K';
      s[16] = 'I';
      s[0] = 18; // length of descriptor
      break;
    case USB_ProductStringIndex:
      s[2] = 'A';
      s[4] = 'T';
      s[6] = '9';
      s[8] = '0';
      s[10] = 'U';
      s[12] = 'S';
      s[14] = 'B';
      s[0] = 16;
      break;
    case USB_SerialNumberStringIndex:
      s[2] = '0';
      s[4] = '0';
      s[6] = '1';
      s[0] = 8;
      break;
    default:
      s[2] = '?';
      s[0] = 4;
  }
}

static void
UsbDevDisableAndFreeEndpoint(uint8_t i)
{
  UsbDevSelectEndpoint(i);
  UsbDevDisableEndpoint();
  UsbDevClearEndpointAllocBit();
}

static void
UsbDevSetUnconfiguredState(void)
{
  uint8_t i;
  UsbDevConfValue = UsbUnconfiguredState;
  i = UsbNumEndpointsAT90USB;
  while (--i > 0) // free all endpoints but EP0
    UsbDevDisableAndFreeEndpoint(i);
  UsbAllocatedEPs = 1;
}

// A device can have multiple configurations. In the simplest case configurations may differ only in power
    consumption.
```

```
// But configurations can be totally different ( differ in number of interfaces , endpoints , ...
bool
UsbDevSetConfiguration( uint8_t c)
{
  uint8_t i;
  switch (c)
  {
    case 0: // go back to unconfigured (addressed) state
      UsbDevSetUnconfiguredState();
      return true;
      break;
    case 1: // set configuration 1
      if (UsbDevConfValue != c)
      {
        i = USB_MaxInterfaces;
        while (i-->0) AltSettingOfInterface[i] = UsbInterfaceUnconfigured;
      }
      for (i = 0; i < USB_Interfaces[c-1]; i++)
      {
        if (!UsbDevSetInterface(c, i , 0))
        {
          UsbDevSetUnconfiguredState();
          return false;
        }
      }
      UsbDevConfValue = c;
      return true;
      break;
    default:
      Debug("UsbDevSetConfiguration(): configuration does not exist!\r\n");
      return false;
  }
}

// Multiple interfaces can exist at the same time! These interfaces have to use different endpoints.
// For example interface 1 can use ep1 and ep2, and interface 2 can use some of the other endpoints (ep3, ep4, ep5
    , ep6).
// A special problem occurs from FIFO memory management of AT90USB: Memory for FIFO has to be allocated in growing
    order.
// If we use multiple interfaces with multiple alternate settings there may occur memory conflicts concerning
    FIFO memory:
// If we allocate memory for endpoint i, there may be an overlap with FIFO of endpoint i+1 or memory of endpoint i
    +1 may slide.
// This is a result of internal design of AT90USB, see section 21.7 in datasheet.
// But from USB design interfaces should be independent, changes of interface i should not affect interface j#i
// To prevent conflicts , following strategy is useful:
// Use endpoints 1 to n for interface 1, endpoints n+1 to m for interface 2, and endpoints m+1, m+2 ... for
    interface 3.
// Give only more than one alternate setting to the interface with the highest number. So changes of alternate
    setting
// with new FIFO sizes can not disturb other interfaces .
// If you really need more than one interface with different alternate settings (endpoint FIFO sizes) you may try
    to
// insert unused dummy endpoints to prevent memory slides or overlaps. Or use different configurations or force
    reallocation of all endpoints.
// Our application uses only interface 0 with endpoints ep1, ep2, ep3. But the code is designed to support more
    interfaces .
bool
```

```
UsbDevSetInterface( uint8_t conf, uint8_t inf, uint8_t as)
{
  uint8_t i;
  if (conf-- == UsbUnconfiguredState) // each interface should be bound to a configuration
  {
    Debug("UsbDevSetInterface(): called from unconfigured (addressed) state !\r\n");
    return false;
  }
  if ((inf >= USB_Interfaces[conf]) || (as >= USB_AltSettings[conf][inf]))
  {
    Debug("UsbDevSetInterface(): interface not supported!\r\n");
    return false;
  }
  if ((as > 0) && (inf != (USB_Interfaces[conf]-1)))
  {
    Debug("UsbDevSetInterface(): Multiple interfaces with more than one alternate setting => FIFO memory conflicts ↵
        may occur!\r\n");
    return false;
  }
  if ( AltSettingOfInterface [inf] == as) // no changes, reset toggle bits of endpoints of this interface
  {
    if (conf == 0)
    {
      if (inf == 0) // first interface of first configuration
      {
        for (i = 1; i < 4; i++) // reset toggle bit of all endpoints of this interface; an endpoint reset may be ↵
            necessary too
        {
          UsbDevSelectEndpoint(i);
          UsbDevResetEndpoint(i);
          UsbDevResetDataToggleBit();
        }
      }
      else if (inf == 1) // second interface of first configuration
      {
        // reset endpoints of second interface
      }
    }
    else if (conf == 2) // similar operations
    {
    }
    return true;
  }
  if (conf == 0)
  {
    if (inf == 0) // first allocation or reallocation with new alternate setting
    {
      UsbDevDisableAndFreeEndpoint(3);
      UsbDevDisableAndFreeEndpoint(2);
      UsbDevDisableAndFreeEndpoint(1);
      UsbAllocatedEPs = 1;
       AltSettingOfInterface [0] = UsbInterfaceUnconfigured;
      if (as == 0) // use alternate setting 0
      {
        if (UsbDevEP_Setup(1, UsbEP_TypeBulk, EP1_FIFO_Size, 1, UsbEP_DirIn))
        {
          Debug("!!! Successful set up EP1!\r\n");
          //UsbDevSelectEndpoint(1); this ep is already selected by UsbDevEP_Setup()
```

```
        UsbDevEnableNAK_IN_Int(); // trigger interrupt when host got a NAK as a result of a read request
      }
      else
      {
        Debug("!!! Setup of EP1 failed !\r\n"); // should not occur ;-)
        return false;
      }
      if (UsbDevEP_Setup(2, UsbEP_TypeBulk, EP2_FIFO_Size, 2, UsbEP_DirIn))
      {
        Debug("!!! Successful set up EP2!\r\n");
      }
      else
      {
        Debug("!!! Setup of EP2 failed !\r\n");
        return false;
      }
      if (UsbDevEP_Setup(3, UsbEP_TypeBulk, EP3_FIFO_Size, 1, UsbEP_DirOut))
      {
        Debug("!!! Successful set up EP3!\r\n");
        UsbDevEnableReceivedOUT_DATA_Int(); // trigger interrupt when out data is available
      }
      else
      {
        Debug("!!! Setup of EP3 failed !\r\n");
        return false;
      }
    }
    else if (as == 1) // alternate setting 1
    {
      // set up same endpoints with different parameters (FIFO-size)
    }
     AltSettingOfInterface [0] = as;
    UsbStartupFinished = 1;
    return true;
  }
  else if (inf == 1) // setup interface 1
  {
  }
 }
 else if (conf == 1) // similar setup if configuration 2 with other interfaces is selected
 {
 }
 return false; // dummy to suppress compiler warning
}

void
UsbDevProcessVendorRequest(USB_DeviceRequest *req)
{
 ProcessUserCommand(req->bRequest, req->wValue, req->wIndex);
}

// This function is called whenever host tries to read data from ep1
// We send a status byte which indicates success of DAQ operation
void
UsbDevFillEP1FIFO(void)
{
 if UsbDevTransmitterReady()
 {
```

```
      UsbDevClearTransmitterReady();
      UsbDevClearNAK_ResponseInBit();
      UsbDevWriteByte(DAQ_Result);
      UsbDevSendInData();
    }
}

// This function is called whenever OUT FIFO has data for us
void
UsbDevReadEP3FIFO(void)
{
  if (UsbDevHasReceivedOUT_Data())
  {
    UsbDevClearHasReceivedOUT_Data();
    if (UsbDevReadAllowed())
    {
      DDRB = 0xFF;
      PORTB = UsbDevReadByte();
      UsbDevClearFifoControllBit(); // maybe we should use an alias for this macro
    }
  }
}
```

```
// AT90USB/usb_requests.h
// Handling of USB (standard) requests
// S. Salewski 22-MAR-2007

#ifndef _USB_REQUESTS_H_
#define _USB_REQUESTS_H_

#include <stdint.h>
#include <stdbool.h>
#include "usb_spec.h"
#include "usb_api.h" // USB_MaxInterfaces

// USB-Standard-Device-Requests, H.J. Kelm USB 2.0, section 2.9.1, page 108
#define USB_StdDevReqGET_STATUS     0x00
#define USB_StdDevReqCLEAR_FEATURE 0x01
#define USB_StdDevReqSET_FEATURE    0x03
#define USB_StdDevReqSET_ADDRESS    0x05
#define USB_StdDevReqGET_DESCRIPTOR 0x06
#define USB_StdDevReqSET_DESCRIPTOR 0x07
#define USB_StdDevReqGET_CONFIGURATION 0x08
#define USB_StdDevReqSET_CONFIGURATION 0x09
#define USB_StdDevReqGET_INTERFACE 0x0A
#define USB_StdDevReqSET_INTERFACE 0x0B
#define USB_StdDevReqSYNCH_FRAME 0x0C

// Meaning of bmRequestType, H.J. Kelm USB 2.0, section 2.9.1, page 107
#define UsbIsDataHostToDevice(bm)       (!(bm & (1<<7)))
#define UsbIsDataDeviceToHost(bm)       (bm & (1<<7))

#define UsbIsStandardRequest(bm)        (bm & (3<<5)) == 0
#define UsbIsClassRequest(bm)           (bm & (3<<5)) == (1<<5)
#define UsbIsVendorRequest(bm)          (bm & (3<<5)) == (1<<6)

#define UsbIsRequestForDevice(bm)       (bm & (31) == 0
#define UsbIsRequestForInterface (bm)   (bm & (31) == 1
#define UsbIsRequestForEndpoint(bm)     (bm & (31) == 2
#define UsbIsRequestForOther(bm)        (bm & (31) == 3

extern uint8_t UsbDevConfValue; // current configuration of our device; 0 is unconfigured state
extern uint8_t AltSettingOfInterface [USB_MaxInterfaces]; // current alternate setting of active interfaces

void UsbProcessSetupRequest(void);

//void UsbDevWriteDescriptor(void *d, uint8_t *written, uint8_t requested);
//void UsbDevReadBytesN(void *c, uint8_t n);
//void WaitZLP_FromHost(void);
//void UsbProcessSetupRequest(void);
//bool UsbSendDescriptors(uint8_t cdi, uint8_t *written, uint8_t requested);
//bool UsbDevSetInterface(uint8_t inf, uint8_t as);
//bool UsbDevSetConfiguration(uint8_t c);

#endif
```

```
// AT90USB/usb_requests.c
// Handling of USB (standard) requests
// S. Salewski 22-MAR-2007

#include <stdint.h>
#include "macros.h"
#include "usb_spec.h"
#include "usb_drv.h"
#include "usart_debug.h"
#include "usb_api.h" // EP0_FIFO_Size and USB_NumInterfaces
#include "usb_requests.h"

uint8_t UsbDevConfValue = UsbUnconfiguredState;
uint8_t AltSettingOfInterface[4] = {UsbInterfaceUnconfigured, UsbInterfaceUnconfigured, UsbInterfaceUnconfigured,
    UsbInterfaceUnconfigured};
uint8_t RemoteWakeupActive = 0;

static void UsbDevWriteDescriptor(void *d, uint8_t *written, uint8_t requested);
static void UsbDevReadBytesN(void *c, uint8_t n);
static bool UsbSendDescriptors(uint8_t cdi, uint8_t *written, uint8_t requested);

// Send descriptor to host, but don't send more total bytes than requested.
// Stop if there is a ZLP (abort) from host or if all bytes are send, else
// fill FIFO and send FIFO if FIFO is full.
static void
UsbDevWriteDescriptor(void *d, uint8_t *written, uint8_t requested)
{
  uint8_t l;
  l = requested - *written; // we may send l bytes more
  if (l > *(uint8_t*)d) l = *(uint8_t*)d; // clip to size of this descriptor
  while (l--)
  {
    if UsbDevHasReceivedOUT_Data() // ZLP from host -- abort
    {
      *written = requested; // prevent further writes
      return;
    }
    UsbDevWriteByte(*(uint8_t*)d++); // put byte to FIFO
    if (((*written+=1) % EP0_FIFO_Size) == 0) // if FIFO full
    {
      UsbDevSendControlIn(); // send FIFO
      // while (!(UsbDevHasReceivedOUT_Data() || UsbDevTransmitterReady()));
      while (!(UEINTX & ((1<<RXOUTI) | (1<<TXINI)))); // the same -- saves a few bytes
    }
  }
}

// Read n bytes from FIFO; FIFO should contain exactly n bytes
// Limited to n < 256
static void
UsbDevReadBytesN(void *c, uint8_t n)
{
  Assert(UsbDevGetByteCountLow() == n);
  while (n--) *(uint8_t*)c++ = UsbDevReadByte();
}

static void
WaitZLP_FromHost(void)
```

```
{
  while (!UsbDevHasReceivedOUT_Data());
  UsbDevClearHasReceivedOUT_Data();
}

void
UsbProcessSetupRequest(void)
{
  USB_DeviceRequest req;
  union // we can use the same piece of memory for these descriptors
  {
    USB_DeviceDescriptor devDes;
    USB_ConfigurationDescriptor confDes;
    char strDes[USB_MaxStringDescriptorLength];
  } u;
  uint8_t b;
  UsbDevSelectEndpoint(0);
  UsbDevReadBytesN(&req, 8);
  // Caution: We have to delay the AcknowledgeSETUP() if request is a 3 stage-transfer with out data
  // because host may send out data immediately after our acknowledge and may not see our stall request!
  if (!(req.bRequest == USB_StdDevReqSET_DESCRIPTOR)) // ! 3 stage-transfer with out data
    UsbDevAcknowledgeSETUP();
  UsbDumpSetupRequest(&req);
  if (UsbIsVendorRequest(req.bmRequestType))
  {
    ReqDebug("Received UsbVendorRequest");
    UsbDevProcessVendorRequest(&req);
    UsbDevSelectEndpoint(0);
    UsbDevSendControlIn(); // send ZLP
  }
  else if (UsbIsStandardRequest(req.bmRequestType))
  {
    ReqDebug("Received UsbStandardRequest");
    switch (req.bRequest)
    {
      case USB_StdDevReqGET_STATUS: // 3 stages with 2 byte IN-data -- not tested yet
        ReqDebug("USB_StdDevReqGET_STATUS");
        Assert(req.wValue == 0);
        Assert(req.wLength == 2);
        switch (req.bmRequestType)
        {
          case 128: // device: Self-Power-Bit, Remote-Wakeup-Bit set?
            Assert(req.wIndex == 0);
            if (UsbDevConfValue == UsbUnconfiguredState)
            {
              ReqDebug("Error: USB_StdDevReqGET_STATUS device in unconfigured state!");
              UsbDevRequestStallHandshake();
              return;
            }
            UsbGetConfigurationDescriptor(&u.confDes, UsbDevConfValue);
            if (u.confDes.bmAttributes & (1<<6)) b = 1; else b = 0; // self powered?
            if ((u.confDes.bmAttributes & (1<<5)) && (RemoteWakeupActive)) b |= (1<<1);
            UsbDevWriteByte(b);
            break;
          case 129: // interface : Always return 0
            if (UsbDevConfValue == UsbUnconfiguredState)
            {
              ReqDebug("Error: USB_StdDevReqGET_STATUS interface in unconfigured state!");
```

```
            UsbDevRequestStallHandshake();
            return;
          }
          // Assert(req.wIndex < USB_Interfaces[UsbDevConfValue]);
          UsbDevWriteByte(0);
          break;
        case 130: // endpoint: is this endpoint stalled ?
          Assert((MSB(req.wIndex) == 0));
          b = LSB(req.wIndex) & 127; // endpoint number
          if (b >= UsbNumEndpointsAT90USB)
          {
            ReqDebug("Error: USB_StdDevReqGET_STATUS endpoint does not exist!");
            UsbDevRequestStallHandshake();
            return;
          }
          if ((UsbDevConfValue == UsbUnconfiguredState) && (b > 0))
          {
            ReqDebug("Error: USB_StdDevReqGET_STATUS for ep > 0 in unconfigured state!");
            UsbDevRequestStallHandshake();
            return;
          }
          UsbDevSelectEndpoint(b);
          if (UsbDevIsEndpointStalled()) // stalled bit is marked write-only in datasheet -- do we need a ↩
              separate state variable ?
            UsbDevWriteByte(1);
          else
            UsbDevWriteByte(0);
          UsbDevSelectEndpoint(0);
          break;
        default:
          ReqDebug("Error: USB_StdDevReqGET_STATUS!");
          UsbDevRequestStallHandshake();
          return;
      }
      UsbDevWriteByte(0);
      UsbDevSendControlIn();
      WaitZLP_FromHost();
      break;
    case USB_StdDevReqCLEAR_FEATURE: // 2 stages (no data-stage) -- not tested yet
    case USB_StdDevReqSET_FEATURE:
      ReqDebug("USB_StdDevReqCLEAR/SET_FEATURE");
      Assert(req.wLength == 0);
      switch (req.bmRequestType)
      {
        case 0: // device
          Assert(req.wIndex == 0);
          if (req.wValue != 1) // Feature selector != DEVICE_REMOTE_WAKEUP
          {
            ReqDebug("Error: USB_StdDevReqSET/CLEAR_FEATURE wrong feature selector for device!");
            UsbDevRequestStallHandshake();
            return;
          }
          if (req.bRequest == USB_StdDevReqCLEAR_FEATURE)
          {
            RemoteWakeupActive = 0;
          }
          else
          {
```

```
        UsbGetConfigurationDescriptor(&u.confDes, UsbDevConfValue);
        if (!(u.confDes.bmAttributes & (1<<5))) // Remote-Wakeup support set in configuration descriptor ?
        {
          ReqDebug("Error: USB_StdDevReqSET_FEATURE, Remote-Wakeup not supported!");
          UsbDevRequestStallHandshake();
          return;
        }
        RemoteWakeupActive = 1; // here we only set a flag -- of course this is not enough to reactivate the sleeping bus
      }
      break;
    case 1: // interface -- nothing to do. Should we request a stall handshake?
      ReqDebug("Error: USB_StdDevReqSET/CLEAR_FEATURE with receiver == interface!");
      UsbDevRequestStallHandshake();
      return;
      break;
    case 2: // endpoint
      Assert(MSB(req.wIndex) == 0);
      if (req.wValue != 0) // Feature selector != ENDPOINT_STALL
      {
        ReqDebug("Error: USB_StdDevReqSET/CLEAR_FEATURE wrong feature selector for endpoint!");
        UsbDevRequestStallHandshake();
        return;
      }
      b = LSB(req.wIndex) & 127; // endpoint address
      if (b >= UsbNumEndpointsAT90USB)
      {
        ReqDebug("Error: USB_StdDevReqSET/CLEAR_FEATURE endpoint does not exist!");
        UsbDevRequestStallHandshake();
        return;
      }
      if ((UsbDevConfValue == UsbUnconfiguredState) && (b > 0))
      {
        ReqDebug("Error: USB_StdDevReqClearSetFeature for ep > 0 in unconfigured state !");
        UsbDevRequestStallHandshake();
        return;
      }
      // Caution: what shall we do if (b == 0)?
      UsbDevSelectEndpoint(b);
      if (req.bRequest == USB_StdDevReqCLEAR_FEATURE)
      {
        UsbDevClearStallRequest();
        UsbDevResetEndpoint(b); // should we do an endpoint reset ?
        UsbDevResetDataToggleBit();
      }
      else
      {
        UsbDevRequestStallHandshake(); // for endpoint b
      }
      UsbDevSelectEndpoint(0);
      break;
    default:
      ReqDebug("Error: USB_StdDevReqClearSetFeature!");
      UsbDevRequestStallHandshake();
      return;
  }
  UsbDevSendControlIn(); // send ZLP
  break;
```

```
case USB_StdDevReqSET_ADDRESS: // 2 stages (no data−stage)
  ReqDebug("USB_StdDevReqSET_ADDRESS");
  if ((LSB(req.wValue) == 0) && (UsbDevConfValue != UsbUnconfiguredState))
  {
    ReqDebug("Error: USB_StdDevReqSET_ADDRESS address 0 in configured state!");
    UsbDevRequestStallHandshake();
    return;
  }
  Assert(req.bmRequestType == 0);
  Assert(MSB(req.wValue) == 0);
  Assert(req.wIndex == 0);
  Assert(req.wLength == 0);
  UsbDevSetAddress(LSB(req.wValue));
  UsbDevSendControlIn(); // send ZLP
  UsbDevWaitTransmitterReady();
  UsbDevEnableAddress();
  break;
case USB_StdDevReqSET_DESCRIPTOR: // 3 stage−transfer with out data
  ReqDebug("Error: USB_StdDevReqSET_DESCRIPTOR currently not supported!");
  UsbDevRequestStallHandshake(); // RequestStallHandshake() before UsbDevAcknowledgeSETUP()
  UsbDevAcknowledgeSETUP(); // to ensure that host will not send out data!
  // you may implement this request if you have any idea for what you can use it
  break;
case USB_StdDevReqGET_CONFIGURATION: // 3 stages with 1 byte IN−data −− not tested yet
  ReqDebug("USB_StdDevReqGET_CONFIGURATION");
  Assert(req.bmRequestType == 128);
  Assert(req.wValue == 0);
  Assert(req.wIndex == 0);
  Assert(req.wLength == 1);
  UsbDevWriteByte(UsbDevConfValue);
  UsbDevSendControlIn();
  WaitZLP_FromHost();
  break;
case USB_StdDevReqSET_CONFIGURATION: // 2 stages (no data−stage)
  ReqDebug("USB_StdDevReqSET_CONFIGURATION");
  Assert(req.bmRequestType == 0);
  Assert(MSB(req.wValue) == 0);
  Assert(req.wIndex == 0);
  Assert(req.wLength == 0);
  if (UsbDevSetConfiguration(LSB(req.wValue)))
  {
    UsbDevSelectEndpoint(0); // UsbDevSetConfiguration() may select other ep
    UsbDevSendControlIn(); // send ZLP
  }
  else
  {
    ReqDebug("Error: USB_StdDevReqSET_CONFIGURATION unsupported configuration!");
    UsbDevSelectEndpoint(0);
    UsbDevRequestStallHandshake();
  }
  break;
case USB_StdDevReqGET_INTERFACE: // 3 stages with 1 byte IN−data −− not tested yet
  ReqDebug("USB_StdDevReqGET_INTERFACE");
  if (UsbDevConfValue == UsbUnconfiguredState)
  {
    ReqDebug("Error: USB_StdDevReqGET_INTERFACE in unconfigured state!");
    UsbDevRequestStallHandshake();
    return;
```

```
    }
    Assert(req.bmRequestType == 129);
    Assert(req.wValue == 0);
    Assert(MSB(req.wIndex) == 0);
    Assert(req.wLength == 1);
    UsbDevWriteByte(AltSettingOfInterface[(uint8_t) LSB(req.wIndex)]);
    UsbDevSendControlIn();
    WaitZLP_FromHost();
    break;
  case USB_StdDevReqSET_INTERFACE: // 2 stages (no data-stage)
    ReqDebug("USB_StdDevReqSET_INTERFACE");
    if (UsbDevConfValue == UsbUnconfiguredState)
    {
      ReqDebug("Error: USB_StdDevReqSET_INTERFACE in unconfigured state!");
      UsbDevRequestStallHandshake();
      return;
    }
    Assert(req.bmRequestType == 1);
    Assert(MSB(req.wValue) == 0);
    Assert(MSB(req.wIndex) == 0);
    Assert(req.wLength == 0);
    if (UsbDevSetInterface(UsbDevConfValue, LSB(req.wIndex), LSB(req.wValue)))
    {
      UsbDevSelectEndpoint(0); // UsbDevSetInterface() may select other ep
      UsbDevSendControlIn(); // send ZLP
    }
    else
    {
      UsbDevSelectEndpoint(0);
      ReqDebug("Error: USB_StdDevReqSET_INTERFACE alt. setting does not exist!");
      UsbDevRequestStallHandshake();
    }
    break;
  case USB_StdDevReqSYNCH_FRAME: // 3 stages with 2 byte IN-data -- not supported
    ReqDebug("USB_StdDevReqSYNCH_FRAME not supported!");
    UsbDevRequestStallHandshake();
    return;
  case USB_StdDevReqGET_DESCRIPTOR: // 3 stages transfer
    ReqDebug("USB_StdDevReqGET_DESCRIPTOR");
    Assert(req.bmRequestType == 128);
    {
      uint8_t requested;
      uint8_t written = 0;
      if (MSB(req.wLength)) requested = 255; else requested = LSB(req.wLength); // we will never send more ↵
          than 255 bytes
      switch (MSB(req.wValue))
      {
        case 1: // Device-Descriptor
          Assert(req.wIndex == 0);
          Assert(LSB(req.wValue) == 0);
          UsbGetDeviceDescriptor(&u.devDes);
          UsbDumpDeviceDescriptor(&u.devDes);
          UsbDevWriteDescriptor(&u.devDes, &written, requested);
          break;
        case 2: // Configuration-Descriptor
          if (!UsbSendDescriptors(LSB(req.wValue), &written, requested))
          {
            ReqDebug("Error: USB_StdDevReqGET_DESCRIPTOR: can not sent Configuration-Descriptor!");
```

```
                UsbDevRequestStallHandshake();
                return;
              }
              break;
            case 3: // String-Descriptor
              UsbGetStringDescriptor(u.strDes, LSB(req.wValue)); // strDes is an array, so no & is necessary!
              UsbDumpStringDescriptor(u.strDes);
              UsbDevWriteDescriptor(u.strDes, &written, requested);
              break;
            default:
              ReqDebug("USB_StdDevReqGET_DESCRIPTOR: Unknown type!");
              return;
          }
          if UsbDevHasReceivedOUT_Data() // got ZLP from host -- abort!
          {
            UsbDevClearHasReceivedOUT_Data();
            ReqDebug("USB_StdDevReqGET_DESCRIPTOR: Abort from host!");
            return;
          }
          if (((written % EP0_FIFO_Size) != 0) || (written < requested))
            UsbDevSendControlIn();
          WaitZLP_FromHost();
        }
        break;
      default:
        ReqDebug("Received unsupported UsbStandardRequest!");
        UsbDevRequestStallHandshake();
    }
  }
  else
  {
    ReqDebug("Unsupported nonstandard request!");
    UsbDevRequestStallHandshake();
  }
}

static bool
UsbSendDescriptors(uint8_t cdi, uint8_t *written, uint8_t requested)
{
  union // we can use the same piece of memory for these descriptors
  {
    USB_ConfigurationDescriptor confDes;
    USB_InterfaceDescriptor intDes;
    USB_EndpointDescriptor endDes;
  } u;
  // uint8_t cdi; // configuration descriptor index
  uint8_t idi; // interface descriptor index
  uint8_t as; // alternate setting of interface
  uint8_t edi; // endpoint descriptor index
  if (UsbGetConfigurationDescriptor(&u.confDes, cdi))
  {
    UsbDumpConfigurationDescriptor(&u.confDes);
    UsbDevWriteDescriptor(&u.confDes, written, requested);
    idi = 0;
    do
    {
      as = 0;
      while (UsbGetInterfaceDescriptor(&u.intDes, cdi, idi, as))
```

```
      {
        UsbDumpInterfaceDescriptor(&u.intDes);
        UsbDevWriteDescriptor(&u.intDes, written, requested);
        for (edi = 1; edi < UsbNumEndpointsAT90USB; edi++)
        {
          if (UsbGetEndpointDescriptor(&u.endDes, cdi, idi, as, edi))
          {
            UsbDumpEndpointDescriptor(&u.endDes);
            UsbDevWriteDescriptor(&u.endDes, written, requested);
          }
        }
        as++;
      }
      idi++;
    } while (as > 0);
    return true;
  }
  else return false;
}
```

```
// AT90USB/usb_spec.h
// USB datastructures as defined by www.usb.org
// S. Salewski 27-FEB-2007

#ifndef _USB_SPEC_H_
#define _USB_SPEC_H_
#include <stdint.h>

#define USB_Spec1_1     0x0110
#define USB_Spec2_0     0x0200

#define USB_ControlTransfer      0
#define USB_IsochronousTransfer  1
#define USB_BulkTransfer         2
#define USB_InterruptTransfer    3

#define USB_DeviceDescriptorType                0x01
#define USB_ConfigurationDescriptorType         0x02
#define USB_StringDescriptorType                0x03
#define USB_InterfaceDescriptorType             0x04
#define USB_EndpointDescriptorType              0x05
// #define USB_DeviceQualifierDescriptorType 0x06 // only used for high speed devices
// #define USB_OtherSpeedConfigurationDescriptorType 0x07 // only used for high speed devices

#define USB_DeviceDescriptorLength              0x12
#define USB_ConfigurationDescriptorLength       0x09
#define USB_InterfaceDescriptorLength           0x09
#define USB_EndpointDescriptorLength            0x07

#define USB_LanguageDescriptorIndex             0
#define USB_ManufacturerStringIndex             1
#define USB_ProductStringIndex                  2
#define USB_SerialNumberStringIndex             3

// USB uses little endian format, avr-gcc too, so no byte-swap is necessary for 16 bit data

// USB-Device-Request, H.J. Kelm USB 2.0, section 2.9.1, page 107
typedef struct
{
  uint8_t       bmRequestType;
  uint8_t       bRequest;
  uint16_t      wValue;
  uint16_t      wIndex;
  uint16_t      wLength;
} USB_DeviceRequest;

// Device-Descriptor, H.J. Kelm, USB 2.0, section 2.8.2 (page 98) and section 9.4.7 (page 296)
typedef struct
{
  uint8_t       bLength;
  uint8_t       bDescriptorType;
  uint16_t      bcdUSB;
  uint8_t       bDeviceClass;
  uint8_t       bDeviceSubClass;
  uint8_t       bDeviceProtocoll;
  uint8_t       bMaxPacketSize0;
  uint16_t      idVendor;
  uint16_t      idProduct;
```

```
  uint16_t      bcdDevice;
  uint8_t       iManufacturer;
  uint8_t       iProduct;
  uint8_t       iSerialNumber;
  uint8_t       bNumConfigurations;
} USB_DeviceDescriptor;

// Configuration-Descriptor, H.J. Kelm, USB 2.0, section 2.8.3 (page 100) and section 9.4.7 (page 297)
typedef struct
{
  uint8_t       bLength;
  uint8_t       bDescriptorType;
  uint16_t      wTotalLength;
  uint8_t       bNumInterfaces;
  uint8_t       bConfigurationValue;
  uint8_t       iConfiguration;
  uint8_t       bmAttributes;
  uint8_t       MaxPower;
} USB_ConfigurationDescriptor;

// Interface-Descriptor, H.J. Kelm, USB 2.0, section 2.8.4 (page 101) and section 9.4.7 (page 297)
typedef struct
{
  uint8_t       bLength;
  uint8_t       bDescriptorType;
  uint8_t       bInterfaceNumber;
  uint8_t       bAlternateSetting;
  uint8_t       bNumEndpoints;
  uint8_t       bInterfaceClass;
  uint8_t       bInterfaceSubClass;
  uint8_t       bInterfaceProtocol;
  uint8_t       iInterface;
} USB_InterfaceDescriptor;

// Endpoint-Descriptor, H.J. Kelm, USB 2.0, section 2.8.5 (page 102) and section 9.4.7 (page 298)
typedef struct
{
  uint8_t       bLength;
  uint8_t       bDescriptorType;
  uint8_t       bEndpointAddress;
  uint8_t       bmAttributes;
  uint16_t      wMaxPacketSize;
  uint8_t       bInterval;
} USB_EndpointDescriptor;

/* not used
typedef char USB_StringDescriptor[USB_MaxStringDescriptorLength];
void InitDeviceDes(USB_DeviceDescriptor *d);
void InitConfigurationDes(USB_ConfigurationDescriptor *c);
void InitInterfaceDes(USB_InterfaceDescriptor *i);
void InitEndpointDes(USB_EndpointDescriptor *e);
*/

#endif
```

```
// AT90USB/usb_spec.c
//  Initialization  of USB datastructures
// S. Salewski 27-FEB-2007

#include "usb_spec.h"
#include "usb_api.h"

// USB uses little endian format, avr-gcc too, so no byte-swap is necessary for 16 bit data

/* set defaults, we don't really need this

void
InitDeviceDes(USB_DeviceDescriptor *d)
{
  d->bLength = USB_DeviceDescriptorLength;
  d->bDescriptorType = USB_DeviceDescriptorType;
  d->bcdUSB = USB_Spec1_1;
  d->bDeviceClass = 0;
  d->bDeviceSubClass = 0;
  d->bDeviceProtocoll = 0;
  d->bMaxPacketSize0 = EP0_FIFO_Size;
  d->idVendor = MyUSB_VendorID;
  d->idProduct = MyUSB_ProductID;
  d->bcdDevice = MyUSB_DeviceBCD;
  d->iManufacturer = USB_ManufacturerStringIndex;
  d->iProduct = USB_ProductStringIndex;
  d->iSerialNumber = USB_SerialNumberStringIndex;
  d->bNumConfigurations = 1;
}

void
InitConfigurationDes(USB_ConfigurationDescriptor *c)
{
  c->bLength = USB_ConfigurationDescriptorLength;
  c->bDescriptorType = USB_ConfigurationDescriptorType;
  c->wTotalLength = 0; // this and the following fields must be filled with correct values!
  c->bNumInterfaces = 0;
  c->bConfigurationValue = 0;
  c->iConfiguration = 0;
  c->bmAttributes = 0;
  c->MaxPower = 0;
}

void
InitInterfaceDes(USB_InterfaceDescriptor *i)
{
  i->bLength = USB_InterfaceDescriptorLength;
  i->bDescriptorType = USB_InterfaceDescriptorType;
  i->bInterfaceNumber = 0; // this and the following fields must be filled with correct values!
  i->bAlternateSetting = 0;
  i->bNumEndpoints = 0;
  i->bInterfaceClass = 0;
  i->bInterfaceSubClass = 0;
  i->bInterfaceProtocol = 0;
  i->iInterface = 0;
}

void
```

```
InitEndpointDes(USB_EndpointDescriptor *e)
{
  e->bLength = USB_EndpointDescriptorLength;
  e->bDescriptorType = USB_EndpointDescriptorType;
  e->bEndpointAddress = 0; // this and the following fields must be filled with correct values!
  e->bmAttributes = 0;
  e->wMaxPacketSize = 0;
  e->bInterval = 0;
}

*/
```

```
// AT90USB/usb_drv.h
// Macros for access to USB registers of Atmels AT90USB microcontrollers
// This file contains low level register stuff as described in
// Atmels AT90USB datasheet 7593D−AVR−07/06
// S. Salewski 21−MAR−2007

#ifndef _USB_DRV_H_
#define _USB_DRV_H_

#include <avr/io.h>
#include <stdint.h>
#include <stdbool.h>
#include "defines.h"

#define UsbNumEndpointsAT90USB 7

#define UsbEP_TypeControl       0
#define UsbEP_TypeIso           1
#define UsbEP_TypeBulk          2
#define UsbEP_TypeInterrupt     3
#define UsbEP_DirOut            0
#define UsbEP_DirControl        0
#define UsbEP_DirIn             1

#define UsbUnconfiguredState    0
#define UsbInterfaceUnconfigured 0xFF
#define UsbNoInterfaceClass     0xFF
#define UsbNoInterfaceSubClass  0xFF
#define UsbNoInterfaceProtokoll 0xFF
#define UsbNoDeviceClass        0xFF
#define UsbNoDeviceSubClass     0xFF
#define UsbNoDeviceProtokoll    0xFF
#define UsbNoDescriptionString  0

extern uint8_t UsbAllocatedEPs;
extern volatile uint8_t UsbStartupFinished;

void UsbDevLaunchDevice(bool lowspeed);
bool UsbDevEP_Setup(uint8_t num, uint8_t type, uint16_t size, uint8_t banks, uint8_t dir);
void UsbInitialReset (void);
void UsbStartPLL(void);
void UsbDevStartDeviceEP0(void);

// A few macros for bit fiddling
#define SetBit(adr, bit)                (adr |= (1<<bit))
#define ClearBit(adr, bit)              (adr &= ~(1<<bit))
#define BitIsSet(adr, bit)              (adr & (1<<bit))
#define BitIsClear(adr, bit)            (!(adr & (1<<bit)))

// A few simple macros
#define UsbDevWaitStartupFinished()     while (!UsbStartupFinished);
#define UsbOutEndpointAdress(endpointIndex)  (endpointIndex & 15)
#define UsbInEndpointAdress(endpointIndex)   (endpointIndex & 15) | (1<<7)
#define UsbConfigurationValue(confIndex)     (confIndex + 1)          // +1, because 0 indicates
    unconfigured (addressed) state
//#define UsbMaxPower2mA(mA)            (mA/2)
#define UsbConfDesAttrBusPowered        (1<<7)                       // use | to combine these 3
    Attributes
```

```
#define UsbConfDesAttrSelfPowered          (1<<7) | (1<<6)
#define UsbConfDesAttrRemoteWakeup         (1<<7) | (1<<5)

// Section and page references refer to release 7593D−AVR−07/06 of
// Atmels Manual for AT90USB devices

// USB general registers , section 21.12.1, page 263 of datasheet
// UHWCON (UsbHardWareCONfiguration)
#define UsbSetDeviceMode()                 SetBit(UHWCON, UIMOD)      // select host or device mode ↵
    manually
#define UsbSetHostMode()                   ClearBit(UHWCON, UIMOD)
#define UsbEnableUID_ModeSelection()       SetBit(UHWCON, UIDE)       // enable mode selection by UID ↵
    pin
#define UsbDisableUID_ModeSelection()      ClearBit(UHWCON, UIDE)
#define UsbEnableUVCON_PinControl()        SetBit(UHWCON, UVCONE)     // enable UVCON pin control, ↵
    figure 21−7
#define UsbDisableUVCON_PinControl()       ClearBit(UHWCON, UVCON)
#define UsbEnablePadsRegulator()           SetBit(UHWCON, UVREGE)     // USB pads (D+, D−) supply
#define UsbDisablePadsRegulator()          ClearBit(UHWCON, UVREGE)

// USBCON (USB CONfiguration)
#define UsbEnableController()              SetBit(USBCON, USBE)       // USB controller enable
#define UsbDisableController()             ClearBit(USBCON, USBE)     // reset and disable controller
#define UsbIsControllerEnabled()           BitIsSet(USBCON, USBE)
#define UsbSetHostModeReg()                SetBit(USBCON, HOST)       // select multiplexed controller ↵
    registers
#define UsbSetDeviceModeReg()              ClearBit(USBCON, HOST)     //
#define UsbFreezeClock()                   SetBit(USBCON, FRZCLK)     // reduce power consumption
#define UsbEnableClock()                   ClearBit(USBCON, FRZCLK)
#define UsbIsClockFreezed()                BitIsSet(USBCON, FRZCLK)
#define UsbEnableOTG_Pad()                 SetBit(USBCON, OTGPADE)    // ??? is this the UID pad?
#define UsbDisableOTG_Pad()                ClearBit(USBCON, OTGPADE)
#define UsbEnableID_TransitionInt()        SetBit(USBCON, IDTE)       // enable ID transition interrupt ↵
    generation
#define UsbDisableID_TransitionInt()       ClearBit(USBCON, IDTE)
#define UsbEnableVBUS_TransitionInt()      SetBit(USBCON, VBUSTE)     // enable VBUS transition ↵
    interrupt
#define UsbDisableVBUS_TransitionInt()     ClearBit(USBCON, VBUSTE)

// USBSTA (USBSTAtus, read only)
#define UsbIsFullSpeedMode()               BitIsSet(USBSTA, SPEED)    // set by hardware if controller ↵
    is in fullspeed mode,
#define UsbIsLowSpeedMode()                BitIsClear(USBSTA, SPEED)  // use in host mode only, ↵
    indeterminate in device mode
#define UsbIsUID_PinHigh()                 BitIsSet(USBSTA, ID)       // query UID pad/pin
#define UsbIsVBUS_PinHigh()                BitIsSet(USBSTA, VBUS)     // query VBUS pad/pin

// USBINT (USBINTerrupt)
#define UsbIsIDTI_FlagSet()                BitIsSet(USBINT, IDTI)     // set by hardware if ID pin ↵
    transition detected
#define UsbClearIDTI_Flag()                ClearBit(USBINT, IDTI)     // shall be cleared by software
#define UsbIsVBUSTI_FlagSet()              BitIsSet(USBINT, VBUSTI)   // set by hardware if transition ↵
    on VBUS pad is detected
#define UsbClearVBUSTI_Flag()              ClearBit(USBINT, VBUSTI)   // shall be cleared by software

// OTGCON (OnTheGoCONfiguration)
// to do ...
```

```
// OTGTCON (OnTheGoTimerCONfiguration)
// to do ...

// OTGIEN (OnTheGoInterruptEnable)
// to do ...

// OTGINT (OnTheGoINTerrupt)
// to do ...

// USB device general registers , section 22.19.1, page 281
// UDCON (Usb Device CONfiguration)
#define UsbDevSelectLowSpeed()                SetBit(UDCON, LSM)          // Page 258, figure 21-14
#define UsbDevSelectFullSpeed()               ClearBit(UDCON, LSM)
#define UsbDevIsLowSpeedSelected()            BitIsSet(UDCON, LSM)
#define UsbDevInitiateRemoteWakeup()          SetBit(UDCON, RMWKUP)       // cleared by hardware, see ↵
    section 22.11, page 273
#define UsbDevIsRemoteWakeupPending()         BitIsSet(UDCON, RMWKUP)
#define UsbDevDetach()                        SetBit(UDCON, DETACH)       // disconnect internal pull-up on ↵
    D+ or D-
#define UsbDevAttach()                        ClearBit(UDCON, DETACH)     // connect internal pull-up on D+ ↵
    or D-
#define UsbDevIsDetached()                    BitIsSet(UDCON, DETACH)
#define UsbDevIsAttached()                    BitIsClear(UDCON, DETACH)

// UDINT (Usb Device INTerrupt)
#define UsbDevIsUpstreamResumeFlagSet() BitIsSet(UDINT, UPRSMI)     // set by hardware
#define UsbDevClearUpstreamResumeFlag() ClearBit(UDINT, UPRSMI)     // shall be cleared by software
#define UsbDevIsEndOfResumeFlagSet()          BitIsSet(UDINT, EORSMI)     // set by hardware if host sends " ↵
    end of resume"
#define UsbDevClearEndOfResumeFlag()          ClearBit(UDINT, EORSMI)     // shall be cleared by software
#define UsbDevIsWakeupCPU_FlagSet()           BitIsSet(UDINT, WAKEUPI)    // set by hardware when USB ↵
    controller is reactivated
#define UsbDevClearWakeupCPU_Flag()           ClearBit(UDINT, WAKEUPI)    // shall be cleared by software ↵
    after enabling USB clock inputs
#define UsbDevIsEndOfResetFlagSet()           BitIsSet(UDINT, EORSTI)     // set by hardware when USB ↵
    controller has detected "End Of Reset"
#define UsbDevClearEndOfResetFlag()           ClearBit(UDINT, EORSTI)     // shall be cleared by software
#define UsbDevIsStartOfFrameFlagSet()         BitIsSet(UDINT, SOFI)       // set by hardware when an USB " ↵
    Start Of Frame" has been detected
#define UsbDevClearStartOfFrameFlag()         ClearBit(UDINT, SOFI)
#define UsbDevIsSuspendFlagSet()              BitIsSet(UDINT, SUSPI)      // set by hardware when USB bus is ↵
     idle
#define UsbDevClearSuspendFlag()              ClearBit(UDINT, SUSPI)      // shall be cleared by software

// UDIEN (Usb Device Interrupt ENable)
#define UsbDevEnableUpstreamResumeInt() SetBit(UDIEN, UPRSME)
#define UsbDevDisableUpstreamResumeInt()      ClearBit(UDIEN, UPRSME)
#define UsbDevEnableEndOfResumeInt()          SetBit(UDIEN, EORSME)
#define UsbDevDisableEndOfResumeInt()         Clear(UDIEN, EORSME)
#define UsbDevEnableWakeupCPU_Int()           SetBit(UDIEN, WAKEUPE)
#define UsbDevDisableWakeupCPU_Int()          ClearBit(UDIEN, WAKEUPE)
#define UsbDevEnableEndOfResetInt()           SetBit(UDIEN, EORSTE)
#define UsbDevDisableEndOfResetInt()          ClearBit(UDIEN, EORSTE)
#define UsbDevEnableStartOfFrameInt()         SetBit(UDIEN, SOFE)
#define UsbDevDisableStartOfFrameInt()        ClearBit(UDIEN, SOFE)
#define UsbDevEnableSuspendInt()              SetBit(UDIEN, SUSPE)
#define UsbDevDisableSuspendInt()             ClearBit(UDIEN, SUSPE)
```

```
// UDADDR (Usb Device ADDRess)
#define UsbDevEnableAddress()                SetBit(UDADDR, ADDEN)      // cleared by hardware
#define UsbDevSetAddress(adr)                UDADDR = (adr & 127)       // set by software

// UDFNUMH (Usb Device Frame NUMber High)
#define UsbDevGetFrameNumberHigh(num)        num = UDFNUMH              // 3 MSB of 11-bits frame number; set by hardware

// UDFNUML (Usb Device Frame NUMber Low)
#define UsbDevGetFrameNumberLow(num)         num = UDFNUML              // 8 LSB of 11-bits frame number; set by hardware

// UDMFN
#define UsbDevIsFrameNumberCRC_Error()       BitIsSet(UDMFN, FNCERR)    // set by hardware if CRC error of framenumber

// USB device endpoint registers, section 22.19.2 page 284
// UENUM (Usb Endpoint NUMber)
#define UsbDevSelectEndpoint(num)            UENUM = (num & 7)          // num == 0, 1, ..., 6

// UERST (Usb Endpoint ReSeT)
#define UsbDevResetEndpoints(mask)           UERST = mask & 127; UERST = 0 // reset selected endpoints; is UERST = 0 necessary?
#define UsbDevResetEndpoint(num)             UERST |= (1<<num); UERST = 0 // reset this endpoint

// UECONX (Usb Endpoint CONfiguration X)
#define UsbDevRequestStallHandshake()        SetBit(UECONX, STALLRQ)    // request a STALL answer to the host for next handshake
#define UsbDevClearStallRequest()            SetBit(UECONX,STALLRQC)    // disable STALL handshake mechanism
#define UsbDevIsEndpointStalled()            BitIsSet(UECONX, STALLRQ)  // Caution: This bit is marked write-only in datasheet
#define UsbDevResetDataToggleBit()           SetBit(UECONX, RSTDT)      // next packet is data0
#define UsbDevEnableEndpoint()               SetBit(UECONX, EPEN)       // enable endpoint according to device configuration
#define UsbDevDisableEndpoint()              ClearBit(UECONX, EPEN)
#define UsbDevIsEndpointEnabled()            BitIsSet(UECONX, EPEN)

// UECFG0X (Usb Endpoint ConFiGuration 0X)
#define UsbDevSetEndpointTypeControl()       UECFG0X &= ~((1<<EPTYPE1)|(1<<EPTYPE0))
#define UsbDevSetEndpointTypeIso()           UECFG0X = (UECFG0X & ~(1<<EPTYPE1)) | (1<<EPTYPE0)
#define UsbDevSetEndpointTypeBulk()          UECFG0X = (UECFG0X & ~(1<<EPTYPE0)) | (1<<EPTYPE1)
#define UsbDevSetEndpointTypeInt()           UECFG0X |= ((1<<EPTYPE1)|(1<<EPTYPE0))
#define UsbDevSetEnpointDirectionIn()        SetBit(UECFG0X, EPDIR)
#define UsbDevSetEnpointDirectionOut()       ClearBit(UECFG0X, EPDIR)
#define UsbDevSetEnpointDirectionControl()   ClearBit(UECFG0X, EPDIR)

// UECFG1X (Usb Endpoint ConFiGuration 1X)
// here we access bits direct, without using bitnames
#define UsbDevSetEndpointSize8()             UECFG1X = (UECFG1X & (4+8)) // preserve bank, clear ALLOC, set size
#define UsbDevSetEndpointSize16()            UECFG1X = (UECFG1X & (4+8)) | (1<<4)
#define UsbDevSetEndpointSize32()            UECFG1X = (UECFG1X & (4+8)) | (2<<4)
#define UsbDevSetEndpointSize64()            UECFG1X = (UECFG1X & (4+8)) | (3<<4)
#define UsbDevSetEndpointSize128()           UECFG1X = (UECFG1X & (4+8)) | (4<<4)
#define UsbDevSetEndpointSize256()           UECFG1X = (UECFG1X & (4+8)) | (5<<4)
#define UsbDevSetEndpointSize512()           UECFG1X = (UECFG1X & (4+8)) | (6<<4)
#define UsbDevSetEndpointOneBank()           UECFG1X = (UECFG1X & (7<<4)) // preserve size, clear ALLOC, set
```

```
    bank
#define UsbDevSetEndpointDoubleBank()          UECFG1X = (UECFG1X & (7<<4)) | (1<<EPBK0)
#define UsbDevSetEndpointAllocBit()            SetBit(UECFG1X, ALLOC)      // allocate endpoint memory
#define UsbDevClearEndpointAllocBit()          ClearBit(UECFG1X, ALLOC)    // free endpoint memory

// UESTA0X (Usb Endpoint STAtus 0X)
#define UsbDevIsConfigurationOk()              BitIsSet(UESTA0X, CFGOK)    // updated when ALLOC bit is set
#define UsbDevIsOverflowErrorFlagSet()         BitIsSet(UESTA0X, OVERFL)   // indicates overflow in ↵
    isochronous endpoint
#define UsbDevClearOverflowErrorFlag()         ClearBit(UESTA0X, OVERFL)   // shall be cleared by software
#define UsbDevIsUnderflowErrorFlagSet()        BitIsSet(UESTA0X, UNDERFL)  // indicates underflow in ↵
    isochronous endpoint
#define UsbDevClearUnderflowErrorFlag()        ClearBit(UESTA0X, UNDERFL)  // shall be cleared by software
#define UsbDevZeroLengthPackedSeen()           BitIsSet(UESTA0X, ZLPSEEN)
#define UsbDevClearZeroLengthPackedFlag()      ClearBit(UESTA0X, ZLPSEEN)  // shall be cleared by software
#define UsbDevIsData0()                        UESTA0X & ((1<<DTSEQ0) | (1<<DTSEQ1)) == 0
#define UsbDevIsData1()                        UESTA0X & ((1<<DTSEQ0) | (1<<DTSEQ1)) == (1<<DTSEQ0)
#define UsbDevGetNumberOfBusyBanks()           (UESTA0X & 3)               // if 0 then all banks are free, ↵
    if 1 then 1 bank is busy ...

// UESTA1X (Usb Endpoint STAtus 1X)
#define UsbDevIsControlDirectionIn()           BitIsSet(UESTA1X, CTRLDIR)  // direction of next packed after ↵
    SETUP packed
#define UsbDevIsControlDirectionOut()          BitIsClear(UESTA1X, CTRLDIR) // set and cleared by hardware
#define UsbDevGetCurrentBank()                 (UESTA1X & 3)               // set by hardware, number of ↵
    current bank, 0 or 1

// UEINTX (Usb Endpoint INTerrupt X)
#define UsbDevIsFifoControllBitSet()           BitIsSet(UEINTX, FIFOCON)   // see page 289
#define UsbDevClearFifoControllBit()           ClearBit(UEINTX, FIFOCON)
#define UsbDevSendInData()                     ClearBit(UEINTX, FIFOCON)   // see section 22.15, page 277
#define UsbDevNAK_ResponseSendToInRequest()    BitIsSet(UEINTX, NAKINI)
#define UsbDevClearNAK_ResponseInBit()         ClearBit(UEINTX, NAKINI)    // shall be cleared by software
//#define UsbDevReadWriteAllowed()             BitIsSet(UEINTX, RWAL)      // set by hardware, don't use for ↵
    control endpoint
#define UsbDevReadAllowed()                    BitIsSet(UEINTX, RWAL)
#define UsbDevWriteAllowed()                   BitIsSet(UEINTX, RWAL)
#define UsbDevNAK_ResponseSendToOutRequest()   BitIsSet(UEINTX, NAKOUTI)
#define UsbDevClearNAK_ResponseOutBit()        ClearBit(UEINTX, NAKOUTI)   // shall be cleared by software
#define UsbDevHasReceivedSETUP()               BitIsSet(UEINTX, RXSTPI)    // set by hardware if current bank ↵
    contains a valid SETUP packet
//#define UsbDevClearHasReceivedSETUP()        ClearBit(UEINTX, RXSTPI)    // shall be cleared by software
#define UsbDevAcknowledgeSETUP()               ClearBit(UEINTX, RXSTPI)    // acknowledge request and clear ↵
    fifo, see section 22.13
#define UsbDevHasReceivedOUT_Data()            BitIsSet(UEINTX, RXOUTI)    // set by hardware if current bank ↵
    contains a new packed
#define UsbDevClearHasReceivedOUT_Data()       ClearBit(UEINTX, RXOUTI)    // shall be cleared by software
#define UsbDevKillLastWrittenBank()            SetBit(UEINTX, RXOUTI)      // see page 278 for abort ↵
    operation
#define UsbDevSTALLHandshakeSend()             BitIsSet(UEINTX, STALLEDI)  // STALL send or CRC error in OUT ↵
    isochronous endpoint
#define UsbDevTransmitterReady()               BitIsSet(UEINTX, TXINI)     // current bank is free and can be ↵
    filled
#define UsbDevWaitTransmitterReady()           while (!(UEINTX & (1<<TXINI)));
#define UsbDevClearTransmitterReady()          ClearBit(UEINTX, TXINI)     // shall be cleared by software
#define UsbDevSendControlIn()                  ClearBit(UEINTX, TXINI)     // see section 22.13, page 274 (↵
    control endpoint management)
#define UsbDevAcknowledgeInBankFreeInt()       ClearBit(UEINTX, TXINI)     // TXINI is set by hardware if in ↵
```

```
    bank becomes free, see section 22.15

// UEIENX (Usb Endpoint Interupt ENable X)
#define UsbDevEnableFlowErrorInt()              SetBit(UEIENX, FLERRE)
#define UsbDevDisableFlowErrorInt()             ClearBit(UEIENX, FLERRE)
#define UsbDevEnableNAK_IN_Int()                SetBit(UEIENX, NAKINE)
#define UsbDevDisableNAK_IN_Int()               ClearBit(UEIENX, NAKINE)
#define UsbDevEnableNAK_OUT_Int()               SetBit(UEIENX, NAKOUTE)
#define UsbDevDisableNAK_OUT_Int()              ClearBit(UEIENX, NAKOUTE)
#define UsbDevEnableReceivedSETUP_Int()         SetBit(UEIENX, RXSTPE)
#define UsbDevDisableReceivedSETUP_Int()        ClearBit(UEIENX, RXSTPE)
#define UsbDevEnableReceivedOUT_DATA_Int()      SetBit(UEIENX, RXOUTE)
#define UsbDevDisableReceivedOUT_DATA_Int()     ClearBit(UEIENX, RXOUTE)
#define UsbDevEnableSTALLED_Int()               SetBit(UEIENX, STALLEDE)
#define UsbDevDisableSTALLED_Int()              ClearBit(UEIENX, STALLEDE)
#define UsbDevEnableTransmitterReadyInt()       SetBit(UEIENX, TXINE)
#define UsbDevDisableTransmitterReadyInt()      ClearBit(UEIENX, TXINE)

// UEDATX (Usb Endpoint DATa X)
#define UsbDevReadByte()                        UEDATX                  // read byte from endpoint FIFO ↵
    selected by EPNUM
#define UsbDevWriteByte(byte)                   UEDATX = byte           // write byte to endpoint FIFO ↵
    selected by EPNUM

// UEBCHX (Usb Endpoint Byte Count High X)
#define UsbDevGetByteCountHigh()                UEBCHX                  // 3 MSB of the byte count of the ↵
    FIFO endpoint

// UEBCLX (Usb Endpoint Byte Count Low X)
#define UsbDevGetByteCountLow()                 UEBCLX                  // 8 LSB of the byte count of the ↵
    FIFO endpoint

// UEINT (Usb Endpoint INTerrupt)
#define UsbDevGetEndpointIntBits()              UEINT                   // set by hardware when interrupt ↵
    is triggered, cleared if int source is served

// PLL clock for USB interface, section 6.11, page 50
// PLLCSR (PLL Control and Status Register)
// set PLL prescaler according to XTAL crystal frequency
//#define UsbXTALFrequencyIs2MHz()              PLLCSR = (PLLCSR & ~(15<<1))
//#define UsbXTALFrequencyIs4MHz()              PLLCSR = (PLLCSR & ~(15<<1)) | (1<<2)
//#define UsbXTALFrequencyIs6MHz()              PLLCSR = (PLLCSR & ~(15<<1)) | (2<<2)
//#define UsbXTALFrequencyIs8MHz()              PLLCSR = (PLLCSR & ~(15<<1)) | (3<<2)
//#define UsbXTALFrequencyIs12MHz()             PLLCSR = (PLLCSR & ~(15<<1)) | (4<<2)
//#define UsbXTALFrequencyIs16MHz()             PLLCSR = (PLLCSR & ~(15<<1)) | (5<<2)
#if (F_XTAL == 2000000)
#define _pre_ 0
#elif (F_XTAL == 4000000)
#define _pre_ 1
#elif (F_XTAL == 6000000)
#define _pre_ 2
#elif (F_XTAL == 8000000)
#define _pre_ 3
#elif (F_XTAL == 12000000)
#define _pre_ 4
#elif (F_XTAL == 16000000)
#define _pre_ 5
#else
```

```
  #error "XTAL-Frequency has to be 2, 4, 6, 8, 12 or 16 MHz for USB devices!"
#endif
#define UsbSetPLL_XTAL_Frequency()          PLLCSR = (_pre_<<2)
#define UsbEnablePLL()                      SetBit(PLLCSR, PLLE)
#define UsbDisablePLL()                     ClearBit(PLLCSR, PLLE)
#define UsbIsPLL_Locked()                   BitIsSet(PLLCSR, PLOCK)
#define UsbWaitPLL_Locked()                 while (!(PLLCSR & (1<<PLOCK)));

#endif
```

```
// AT90USB/usb_drv.c
// Basic functions for Atmels AT90USB microcontrollers
// Based on Atmels AT90USB datasheet 7593D-AVR-07/06
// S. Salewski 21-MAR-2007

#include <stdbool.h>
#include <stdint.h>
#include <avr/interrupt.h> // sei()
#include "usb_drv.h"
#include "usart_debug.h"
#include "usb_api.h" // EP0_FIFO_Size

uint8_t UsbAllocatedEPs = 0;
volatile uint8_t UsbStartupFinished = 0;

// Set some registers to their initial (reset) value.
// Reason: Atmels bootloader activates some interrupts.
// This function deactivates it, so we have clean start
// conditions if our program is started from bootloader.
void
UsbInitialReset (void)
{
  USBCON      = (1<<FRZCLK);
  OTGIEN      = 0;
  UDIEN       = 0;
  UHIEN       = 0;
  UEIENX      = 0;
  UPIENX      = 0;
  UHWCON      = (1<<UIMOD);
}

// Start PLL and enable clock
void
UsbStartPLL(void)
{
  UsbSetPLL_XTAL_Frequency();
  UsbEnablePLL();
  UsbWaitPLL_Locked();
  UsbEnableClock();
}

// Basic USB activation necessary to trigger a wakeup interrupt
// Wakeup ISR will start PLL clock, then USB-END-OF-RESET interrupts are recognized
void
UsbDevLaunchDevice(bool lowspeed)
{
  UsbInitialReset();
  if (1) // set it to (1) if you need very small code size (i.e. bootloader, saves 78 bytes)
  {
    UHWCON = ((1<<UIMOD) | (1<<UVREGE));
    USBCON = ((1<<USBE) | (1<<OTGPADE) | (1<<FRZCLK));
    asm volatile ("nop"); // nop may be are necessary if mpu is clocked with 16 MHz (deactivated prescaler)
    // asm volatile ("nop");
    if (lowspeed) UDCON = (1<<LSM);
    USBCON = ((1<<USBE) | (1<<OTGPADE));
    USBCON = ((1<<USBE) | (1<<OTGPADE) | (1<<FRZCLK)); // setting FRZCLK again in not necessary (see below)
    UDIEN = ((1<<WAKEUPE) | (1<<EORSTE));
  }
```

```
  else // do the same thing, bit for bit, with named macros
  {
    UsbEnablePadsRegulator();
    UsbEnableController();
    UsbEnableOTG_Pad(); // necessary to power on the device (undocumented feature)
    UsbDisableUID_ModeSelection();
    UsbSetDeviceMode();
    UsbSetDeviceModeReg();
    if (lowspeed) UsbDevSelectLowSpeed(); // fullspeed is default
    UsbEnableClock(); // without this no (wakeup) interrupt is triggered !!!
    UsbFreezeClock(); // FreezeClock() is not really necessary, but we may feel better because PLL in not running ↵
        yet
    UsbDevEnableWakeupCPU_Int();
    UsbDevEnableEndOfResetInt(); // call this AFTER (UsbEnableController(); UsbEnableOTG_Pad();)
  }
  UsbDevAttach();
  sei();
}

// Called by USB-END-OF-RESET interrupt
// USB reset resets EP0! (see section 22.5)
void
UsbDevStartDeviceEP0(void)
{
  UsbDevSelectEndpoint(0);
  if UsbDevIsEndpointEnabled() // can this be true?
    Debug("~~~ EP0 already enabled!\r\n");
  else
  {
    Debug("~~~ Setting up EP0...\r\n");
    UsbAllocatedEPs = 0;
    if (UsbDevEP_Setup(0, UsbEP_TypeControl, EP0_FIFO_Size, 1, UsbEP_DirControl))
      Debug("~~~ Successful set up EP0!\r\n");
    else
      Debug("~~~ Setup of EP0 failed!\r\n");
  }
  UsbDevEnableReceivedSETUP_Int();
}

// To reduce codesize, you may comment this function out and allocate your ep with low level macros
// num: 0...6
// type: UsbEP_TypeControl, UsbEP_TypeIso, UsbEP_TypeBulk, UsbEP_TypeInterrupt
// size: 8, 16, 32, 64. Only for isochronous ep 128, 256 or 512.
// banks: 1 or 2
// dir: UsbEP_DirOut, UsbEP_DirControl, UsbEP_DirIn
// more than one control ep or more than one bank for control ep0 may work, but is not recommended
bool
UsbDevEP_Setup(uint8_t num, uint8_t type, uint16_t size, uint8_t banks, uint8_t dir)
{
  uint8_t i, j;
  banks--;
  if ((num > 6) || (dir > 1) || (banks > 1) || (type > 3) || (size & 0xFC07)) return false;
  i = (uint8_t)(size / 8);
  if ((UsbDevIsLowSpeedSelected()) && ((type == UsbEP_TypeBulk) || (type == UsbEP_TypeIso) || (i > 1))) return ↵
      false;
  if (!((i == 1) || (i == 2) || (i == 4) || (i == 8)))
    if (!((i == 16) || (i == 32) || (i == 64)) && (type == UsbEP_TypeIso))) return false;
  if ((type == UsbEP_TypeControl) && ((dir != UsbEP_DirControl) || (banks > 0))) return false;
```

```
  if (num == 0)
  {
    if ((dir != UsbEP_DirControl) || (type != UsbEP_TypeControl)) return false; // EP0 is always control ep
  }
  else
  {
    if ((num != UsbAllocatedEPs) || // allocate eps in growing order, see section 21.7
        (type == UsbEP_TypeControl)) return false; // more than one control ep may be ok?
  }
  UsbDevSelectEndpoint(num);
  UsbDevEnableEndpoint(); // enable ep before memory is allocated? Yes, see figure 22-2
  j = 0;
  while ((i = (i >> 1))) j++;
  UECFG0X = ((type << 6) | (dir));
  UECFG1X = ((j << 4) | (banks << 2));
  UECFG1X |= (1 << ALLOC);
  if (UESTA0X & (1 << CFGOK))
  {
    UsbAllocatedEPs++;
    return true;
  }
  else
  {
    UsbDevDisableEndpoint();
    return false;
  }
}
```

```
// AT90USB/usart_drv.h
// Basic output routines for USART, adapted to Atmels AT90USB microcontrollers
// Based on AT90USB datasheet (7593D−AVR−07/06), chapter 18 and examples from
// http ://www.mikrocontroller.net/ articles /AVR−GCC−Tutorial and
// http ://www.roboternetz.de/wissen/index.php/UART_mit_avr−gcc
// S. Salewski 29−JAN−2007

#ifndef _USART_DRV_H_
#define _USART_DRV_H_

#include <stdint.h>
#include "defines.h"

#ifndef NOUART
void USART_Init(void); // initialize USART, 8N1 mode
void USART_WriteChar(char c);
void USART_WriteHex(unsigned char c);
void USART_WriteHexW(uint16_t w);
void USART_WriteString(char *s);
void USART_NewLine(void);
#else
#define USART_Init()
#define USART_WriteChar(c)
#define USART_WriteHex(c)
#define USART_WriteHexW(w)
#define USART_WriteString(s)
#define USART_NewLine()
#endif

#endif
```

```
// AT90USB/usart_drv.c
// Basic output routines for USART, adapted to Atmels AT90USB microcontrollers
// Based on AT90USB Datasheet (7593D−AVR−07/06), chapter 18 and examples from
// http ://www.mikrocontroller.net/ articles /AVR−GCC−Tutorial and
// http ://www.roboternetz.de/wissen/index.php/UART_mit_avr−gcc
// S. Salewski 27−FEB−2007

#include <avr/io.h>
#include "defines.h"
#include "usart_drv.h"
#include "macros.h"

#ifndef NOUART

#define Wait_USART_Ready() while (!(UCSR1A & (1<<UDRE1)))
#define UART_UBRR (F_CPU/(16L*USART_BAUD)−1)

// initialize USART, 8N1 mode
void
USART_Init(void)
{
  UBRR1 = UART_UBRR;
  UCSR1C = (1<<UCSZ10) | (1<<UCSZ11);
  UCSR1B = (1<<TXEN1);
}

void
USART_WriteChar(char c)
{
  Wait_USART_Ready();
  UDR1 = c;
}

void
USART_WriteHex(unsigned char c)
{
  unsigned char nibble;
  nibble = (c >> 4);
  if (nibble < 10) nibble += '0'; else nibble += ('A'−10);
  Wait_USART_Ready();
  UDR1 = nibble;
  nibble = (c & 0x0F);
  if (nibble < 10) nibble += '0'; else nibble += ('A'−10);
  Wait_USART_Ready();
  UDR1 = nibble;
}

void
USART_WriteHexW(uint16_t w)
{
  USART_WriteHex(MSB(w));
  USART_WriteHex(LSB(w));
}

void
USART_WriteString(char *s)
{
  while (*s) USART_WriteChar(*s++);
```

```
}

void
USART_NewLine(void)
{
  Wait_USART_Ready();
  UDR1 = '\r';
  Wait_USART_Ready();
  UDR1 = '\n';
}

#endif
```

```
// AT90USB/usart_debug.h
// A few macros for debugging by sending messages over the serial port
// S. Salewski 22-MAR-2007

#ifndef _USART_DEBUG_H_
#define _USART_DEBUG_H_

#include "defines.h"
#include "usart_drv.h"
#include "usb_spec.h"

#ifdef DEBUG
#define Debug(msg) USART_WriteString(msg)
//#define ReqDebug(msg) USART_WriteString(msg " (" __FILE__ ")\r\n")
#define ReqDebug(msg) USART_WriteString("|" msg "\r\n")
#else
#define Debug(msg)
#define ReqDebug(msg)
#endif

#ifdef DEBUG
#define Assert(exp) if (exp == 0) USART_WriteString("Error: " #exp " == 0 (" __FILE__ ")\r\n");
#else
#define Assert(exp)
#endif

#ifdef DEBUG
void UsbDumpDeviceDescriptor(USB_DeviceDescriptor *d);
void UsbDumpConfigurationDescriptor(USB_ConfigurationDescriptor *c);
void UsbDumpInterfaceDescriptor(USB_InterfaceDescriptor *i);
void UsbDumpEndpointDescriptor(USB_EndpointDescriptor *e);
void UsbDumpStringDescriptor(char *s);
void UsbDumpSetupRequest(USB_DeviceRequest *req);
#else
#define UsbDumpDeviceDescriptor(d)
#define UsbDumpConfigurationDescriptor(c)
#define UsbDumpInterfaceDescriptor(i)
#define UsbDumpEndpointDescriptor(e)
#define UsbDumpStringDescriptor(s)
#define UsbDumpSetupRequest(req)
#endif

#endif
```

```
// AT90USB/usart_debug.c
// Debugging by sending messages over the serial port
// S. Salewski 22-MAR-2007

#include "defines.h"

#ifdef DEBUG

#include "usart_debug.h"
#include "usb_spec.h"
#include "usart_drv.h"

void
UsbDumpDeviceDescriptor(USB_DeviceDescriptor *d)
{
  USART_WriteString("USB Device-Descriptor:");
  USART_WriteString("\r\n d->bLength: "); USART_WriteHex(d->bLength);
  USART_WriteString("\r\n d->bDescriptorType: "); USART_WriteHex(d->bDescriptorType);
  USART_WriteString("\r\n d->bcdUSB: "); USART_WriteHexW(d->bcdUSB);
  USART_WriteString("\r\n d->bDeviceClass: "); USART_WriteHex(d->bDeviceClass);
  USART_WriteString("\r\n d->bDeviceSubClass: "); USART_WriteHex(d->bDeviceSubClass);
  USART_WriteString("\r\n d->bDeviceProtocoll: "); USART_WriteHex(d->bDeviceProtocoll);
  USART_WriteString("\r\n d->bMaxPacketSize0: "); USART_WriteHex(d->bMaxPacketSize0);
  USART_WriteString("\r\n d->idVendor: "); USART_WriteHexW(d->idVendor);
  USART_WriteString("\r\n d->idProduct: "); USART_WriteHexW(d->idProduct);
  USART_WriteString("\r\n d->bcdDevice: "); USART_WriteHexW(d->bcdDevice);
  USART_WriteString("\r\n d->iManufacturer: "); USART_WriteHex(d->iManufacturer);
  USART_WriteString("\r\n d->iProduct: "); USART_WriteHex(d->iProduct);
  USART_WriteString("\r\n d->iSerialNumber: "); USART_WriteHex(d->iSerialNumber);
  USART_WriteString("\r\n d->bNumConfigurations: "); USART_WriteHex(d->bNumConfigurations);
  USART_NewLine();
}

void
UsbDumpConfigurationDescriptor(USB_ConfigurationDescriptor *c)
{
  USART_WriteString(" -USB Configuration-Descriptor:");
  USART_WriteString("\r\n   c->bLength: "); USART_WriteHex(c->bLength);
  USART_WriteString("\r\n   c->bDescriptorType: "); USART_WriteHex(c->bDescriptorType);
  USART_WriteString("\r\n   c->wTotalLength: "); USART_WriteHexW(c->wTotalLength);
  USART_WriteString("\r\n   c->bNumInterfaces: "); USART_WriteHex(c->bNumInterfaces);
  USART_WriteString("\r\n   c->bConfigurationValue: "); USART_WriteHex(c->bConfigurationValue);
  USART_WriteString("\r\n   c->iConfiguration: "); USART_WriteHex(c->iConfiguration);
  USART_WriteString("\r\n   c->bmAttributes: "); USART_WriteHex(c->bmAttributes);
  USART_WriteString("\r\n   c->MaxPower: "); USART_WriteHex(c->MaxPower);
  USART_NewLine();
}

void
UsbDumpInterfaceDescriptor(USB_InterfaceDescriptor *i)
{
  USART_WriteString("  -USB Interface-Descriptor:");
  USART_WriteString("\r\n     i->bLength: "); USART_WriteHex(i->bLength);
  USART_WriteString("\r\n     i->bDescriptorType: "); USART_WriteHex(i->bDescriptorType);
  USART_WriteString("\r\n     i->bInterfaceNumber: "); USART_WriteHex(i->bInterfaceNumber);
  USART_WriteString("\r\n     i->bAlternateSetting: "); USART_WriteHex(i->bAlternateSetting);
  USART_WriteString("\r\n     i->bNumEndpoints: "); USART_WriteHex(i->bNumEndpoints);
  USART_WriteString("\r\n     i->bInterfaceClass: "); USART_WriteHex(i->bInterfaceClass);
```

```
  USART_WriteString("\r\n      i->bInterfaceSubClass: "); USART_WriteHex(i->bInterfaceSubClass);
  USART_WriteString("\r\n      i->bInterfaceProtocol: "); USART_WriteHex(i->bInterfaceProtocol);
  USART_WriteString("\r\n      i->iInterface: "); USART_WriteHex(i->iInterface);
  USART_NewLine();
}

void
UsbDumpEndpointDescriptor(USB_EndpointDescriptor *e)
{
  USART_WriteString("    -USB Endpoint-Descriptor:");
  USART_WriteString("\r\n        e->bLength: "); USART_WriteHex(e->bLength);
  USART_WriteString("\r\n        e->bDescriptorType: "); USART_WriteHex(e->bDescriptorType);
  USART_WriteString("\r\n        e->bEndpointAddress: "); USART_WriteHex(e->bEndpointAddress);
  USART_WriteString("\r\n        e->bmAttributes: "); USART_WriteHex(e->bmAttributes);
  USART_WriteString("\r\n        e->wMaxPacketSize: "); USART_WriteHexW(e->wMaxPacketSize);
  USART_WriteString("\r\n        e->bInterval: "); USART_WriteHex(e->bInterval);
  USART_NewLine();
}

void
UsbDumpStringDescriptor(char *s)
{
   uint8_t  i;
  i = *s;
  USART_WriteString("USB String-Descriptor:\r\n");
  while (i--) USART_WriteHex(*s++);
  USART_NewLine();
}

void
UsbDumpSetupRequest(USB_DeviceRequest *req)
{
  USART_WriteString("Setup Request:");
  USART_WriteString(" \r\nbmRequestType: "); USART_WriteHex(req->bmRequestType);
  USART_WriteString(" \r\nbRequest:      "); USART_WriteHex(req->bRequest);
  USART_WriteString(" \r\nwValue:       "); USART_WriteHexW(req->wValue);
  USART_WriteString(" \r\nwIndex:       "); USART_WriteHexW(req->wIndex);
  USART_WriteString(" \r\nwLength:      "); USART_WriteHexW(req->wLength);
  USART_NewLine();
}

#endif
```

```
// AT90USB/ringbuffer.h
// Simple Ring-Buffer (FIFO) for Elements of type Q
// S. Salewski, 20-MAR-2007

#ifndef _RING_BUFFER_H_
#define _RING_BUFFER_H_

#include <stdint.h>

#define BufElements 1024
#define Q uint16_t

extern uint16_t RB_Entries;

#define RB_FreeSpace() (BufElements - RB_Entries)
#define RB_IsFull() (RB_Entries == BufElements)
#define RB_IsEmpty() (RB_Entries == 0)

void RB_Init(void);
void RB_Write(Q el);
Q RB_Read(void);

#endif
```

```
// AT90USB/ringbuffer.c
// Simple Ring-Buffer (FIFO) for Elements of type Q
// S. Salewski, 19-MAR-2007

/*
t-> o
    o <-w
    x
    x <-r
b-> x
*/

#include <stdint.h>
#include "ringbuffer.h"

static Q buf[BufElements];
uint16_t RB_Entries;

#define t &buf[BufElements - 1]
#define b &buf[0]

//Q *t = &buf[BufElements - 1];
//Q *b = &buf[0];
Q *r; // position from where we can read (if RB_Entries > 0)
Q *w; // next free position (if RB_Entries < BufElements))

void
RB_Init(void)
{
  r = b;
  w = b;
  RB_Entries = 0;
}

Q
RB_Read(void)
{
//Assert(RB_Entries > 0);
  RB_Entries--;
  if (r > t) r = b;
  return *r++;
}

void
RB_Write(Q el)
{
//Assert(RB_Entries < BufElements);
  RB_Entries++;
  if (w > t) w = b;
  *w++ = el;
}
```

```
// AT90USB/usb_isr.c
// USB Interrupt Service Routines
// S. Salewski 22-MAR-2007

#include <avr/interrupt.h>
#include <stdint.h>
#include "usb_drv.h"
#include "usart_debug.h"
#include "usb_api.h"
#include "usb_requests.h"

// USB General Interrupt Handler (Figure 21.11)
// USB Registers: USBINT.0, USBINT.1, UDINT
ISR(USB_GEN_vect)
{
  Debug("ISR(USB_GEN_vect)\r\n");
  if UsbIsIDTI_FlagSet()
  {
    UsbClearIDTI_Flag();
    Debug("=== IDTI_FlagSet\r\n");
  }
  if UsbIsVBUSTI_FlagSet()
  {
    UsbClearVBUSTI_Flag();
    Debug("=== VBUSTI_FlagSet\r\n");
  }
  if UsbDevIsEndOfResetFlagSet()
  {
    UsbDevClearEndOfResetFlag();
    UsbDevStartDeviceEP0();
    Debug("=== EndOfResetFlagSet\r\n");
  }
  if UsbDevIsWakeupCPU_FlagSet()
  {
    UsbDevDisableWakeupCPU_Int();
    UsbDevClearWakeupCPU_Flag();
    UsbStartPLL();
    Debug("=== WakeupCPU_FlagSet\r\n");
  }
}

// USB Endpoint/Pipe Interrupt Handler (Figure 21.12)
// Endpoint Registers: UEINTX, UESTAX.6 and UESTAX.5
// Setup-Request may reset endpoints, so we process data endpoints first!
// User defined actions have to acknowledge the interrupt!
ISR(USB_COM_vect)
{
  uint8_t mask;
  uint8_t ep;
  Debug("ISR(USB_COM_vect)\r\n");
  mask = UsbDevGetEndpointIntBits();
  ep = UsbNumEndpointsAT90USB;
  while (ep-- > 0)
  {
    if (mask & (1<<ep))
    {
      UsbDevSelectEndpoint(ep);
      switch (ep)
```

```
    {
      case 0:
        if UsbDevHasReceivedSETUP()
          UsbProcessSetupRequest();
        break;
      case 1:
        UsbDevEP1IntAction();
        break;
      case 2:
        UsbDevEP2IntAction();
        break;
      case 3:
        UsbDevEP3IntAction();
        break;
      case 4:
        UsbDevEP4IntAction();
        break;
      case 5:
        UsbDevEP5IntAction();
        break;
      case 6:
        UsbDevEP6IntAction();
        break;
      default:
        Debug("Error in ISR(USB_COM_vect)\r\n");
    }
  }
 }
}
```

```
// AT90USB/com_def.h
// Common defines for firmware and PC program
// S. Salewski, 23-MAR-2007

#ifndef _COM_DEF_H_
#define _COM_DEF_H_

#define MyUSB_VendorID 0x03eb // Atmel code
#define MyUSB_ProductID 0x0001 // arbitrary value
#define USB_VendorRequestCode (1<<6)

// Endpoint 1, used for transferring status information:
// Bulk IN, 8 byte FIFO
// Filled by "NAK-IN-WAS-SEND" ISR
#define EP1_FIFO_Size 8

// Endpoint 2, used for transferring DAQ data:
// Bulk IN, 64 byte dual bank FIFO
// Filled from within timer ISR
#define EP2_FIFO_Size 64

// Endpoint 3, used to set digital port B
// Bulk OUT, 8 byte FIFO
// Read by "OUT-FIFO-IS-FILLED" ISR
#define EP3_FIFO_Size 8

#define MaxSamples 65535

// Status codes
#define UsbDevStatusOK 0
#define UsbDevStatusDAQ_BufferOverflow 1
#define UsbDevStatusDAQ_TimerOverflow 2
#define UsbDevStatusDAQ_ConversionNotFinished 3

#endif
```

```
// AT90USB/defines.h
// Adapt the constants in this file to your project
// S. Salewski 29-JAN-2007

#ifndef _DEFINES_H_
#define _DEFINES_H_

#define DEBUG // send debug messages over serial port
//#undef DEBUG

//#define NOUART // do not use usart_drv.c at all
#undef NOUART

#define F_XTAL          16000000        // XTAL frequency (Hz). Use a crystal with 2, 4, 6, 8, 12 or 16 MHz if ↵
    USB is used.
#define F_CPU           16000000UL      // mpu frequency (may be divided by prescaler, default prescaler is 8)
#define USART_BAUD      9600UL          // baud(rate) for serial port
#define UART_BAUD       USART_BAUD      // alias

#endif
```

```
// AT90USB/macros.h
// S. Salewski 29-JAN-2007

#ifndef _MACROS_H_
#define _MACROS_H_

#define MSB(w) ((char*) &w)[1]
#define LSB(w) ((char*) &w)[0]

#endif
```

ATMEL AT90USB Datasheet (7593D−AVR−07/06)

There exists some errors or at least unclear statements as listed below:

21.4 General Operating Modes
21.4.2 Power−on and reset (page 253)
The SPDCONF bits can be set by software.
!!! QUESTION: What is SPDCONF

Figure 21−12. USB Endpoint/Pipe Interrupt vector sources (page 256)
Figure 22−5. USB Device Controller Endpoint Interrupt System (page 280)
TXOUTE UEIENX.3
!!! Name TXOUTE should be RXSTPE for Device−Mode

Figure 21−12. USB Endpoint/Pipe Interrupt vector sources (page 256)
Figure 23−5. USB Device Controller Pipe Interrupt System (page 300)
UPIEN
!!! Correct Name of register is UPIENX

21.6.1 Device mode (page 258)
!!! UDSS register mentioned in text does not exists.

21.6.2 Host mode (page 259)
When the USB interface is configured in device mode, internal Pull Down resistors are activated on both UDP UDM lines and the interface detects the type of device connected.
!!! It should be "... configured in host mode ,..."

21.12.1 USB general registers (page 264)
4 − OTGPADE: OTG Pad Enable
Set to enable the OTG pad. Clear to disable the OTG pad.
!!! This bit seems to be important to activate the USB device, but its not clear
!!! why and what OTG Pad is.

21.13 USB Software Operating modes (page 267)
Wait USB pads regulator ready state
!!! How to detect ready state ?

21.13 USB Software Operating modes (page 268)
Set USB suspend clock
Clear USB suspend clock
!!! What is suspend clock

22.6 Endpoint selection (page 270)
Clearing EPNUMS
!!! Register EPNUMS does not exists!

Figure 22−2. Endpoint activation flow: (page 271)
the Not Yet Disable feature
!!! What is the Not Yet Disable feature

22.13 CONTROL endpoint management (page 274)
CONTROL endpoints should not be managed by interrupts, but only by polling the status bits.
!!! What does this mean? Using interrupts seems to work fine ?

22.12.2 STALL handshake and Retry mechanism (page 274)
The Retry mechanism has priority over the STALL handshake. A STALL handshake is sent if the STALLRQ request bit is set and if there is no retry required.
!!! What is the "Retry mechanism"?

22.19.1 USB device general registers (page 281)
2−0 − FNUM10:8 − Frame Number Upper Flag
FNUM is updated if a corrupted SOF is received.
!!! Strange statement!

22.19.2 USB device endpoint register
6−0 − EPRST6:0 − Endpoint FIFO Reset Bits
Then, cleared by software to complete the reset operation and start using the endpoint.
!!! Is clearing by software really necessary? In most similar cases this is done by hardware!

22.19.2 USB device endpoint registers (page 285) ***NEW
Register UECONX: Bit 5 − STALLRQ − STALL Request Handshake Bit is marked WRITE−ONLY
!!! This seems to be wrong, we can read this bit. Reading is necessary to query stalled state!

22.16 Isochronous mode (page 278)
For Isochronous IN endpoints, it is possible to automatically switch the banks on each start of frame (SOF). This is done by setting ISOSW. The CPU has to fill the bank of the endpoint; the bank switching will be automatic as soon as a SOF is seen by the hardware.
!!! ISOSW is mentioned here and in the 31. Register Summary (page 414), but no where else.
!!! In Register Summary (page 414) Flags SOSW, AUTOSW, NYETSDIS are mentioned, but not explained!

25.4 Prescaling and Conversion Timing (page 319) ***NEW
setting the ADHSM bit in ADCSRB allows an increased ADC clock frequency
!!! ADHSM bit is mentioned multiple times in datasheet, but it does not exist.

25.8.2 ADC Control and Status Register A − ADCSRA (332) ***NEW
Bits 2:0 − ADPS2:0: ADC Prescaler Select Bits
These bits determine the division factor between the XTAL frequency and the input clock to the ADC.
!!! Input clock of ADC prescaler seems to be not the XTAL frequency, but the (already prescaled)
!!! mpu frequency.

Stefan Salewski, 20−MAR−2007