

Gecko library for Atmels AT90USB and ATMEGA_USB microcontroller

Firmware is compatible with avr-gcc version 4.3.3

D. Commons May-2010

Files :

README.txt	This text page.
Gecko.pdf	This entire pdf document including the listings .
Errata.tex	Errata for Chameleon hardware and Gecko library api.
stdio_driver.c	This file contains uart functions that support the microcontrollers hardware serial port. The basic functions are putchar(), getchar() and rxready() to prevent receiver blocking. In addition support; a stdio stream is provided for use with avrlibc printf functions. This stream handler includes a buffered getchar function which includes editing.
usb_api.c	The api is the main entry point to the Gecko library. The main functions allow the USB port to be initialized and provides callbacks for all IN/OUT transactions.
usb_standard_request.c	The basic USB enumeration code
usb_task.c	Low level USB hardware management This is a low level device driver for managing the hardware. This file contains the interrupt handlers for the general USB interrupt. Additional functions included to handle 'standard_requests' via 'usb_task()'. To support device initialization 'usb_task_init()' enables the USB device to connect and attach to the bus.
usb_debug.c	Debug utilities that can be optionally compiled into the library.
usb_CDC_descriptors.c	Descriptor file for CDC operation
usb_HID_descriptors.c	Descriptor file for HID operation
usb_IAD_descriptors.c	Descriptor file for dual CDC operation using interface associated descriptor
Makefile:	type "make" to generate all binary files, or "make debug" for serial debug.
Gecko.tex	Script for generating pdf printout of source code using "pdflatex Gecko.tex"
Include files :	
usb_debug.h	Debug print options
usb_CDC_descriptors.h	Descriptor for CDC operation
usb_HID_descriptors.h	Descriptor for HID operation
usb_IAD_descriptors.h	Descriptor for dual CDC operation using interface associated descriptor
usb_standard_request.h	The basic USB enumeration code
usb_task.h	Low level USB hardware management

Gecko Files usb_api.c**//Public functions**

```
void usb_init(u08 xtal, u16 Vid, u16 Pid);
void usb_user_endpoint_init(u08 conf_nb);
void usb_OUT_Callback(void *fct);
void usb_IN_Callback(void *fct);
void Vendor_Callback(void *fct);
void *Usb_Vendor_Callback;
void Trace_Interrupt(void);
```

//Private functions

```
void OneMsTimer(void);
void usb_init_ep(u08 num, u08 type, u08 dir, u08 nyet, u08 size, u08 bank, u08 interrupt);
void usb_user_endpoint_init(u08 conf_nb);
void usb_config_ep(u08 config0, u08 config1);
void usb_init_device (void);
void SetupEndpoints(void);
u08 usb_configured(void);
void bootloader_start(void);

u08 Endpoint_WaitUntilReady(void);
u08 Endpoint_RW_Stream(const void* Buffer, u16 Length);
u08 Endpoint_Write_Control_Stream(const void* Buffer, u16 Length);
u08 Endpoint_Read_Control_Stream(void* Buffer, u16 Length);
```

usb_debug.c**//Public functions**

```
void debug_CDC_line_code(void);
void debug_to_uart(void);
void debug_HexDump(char* buf, int size);
void debug_ep_init(unsigned char cfg0, unsigned char cfg1);
void debug_dev_interrupt(void);
void debug_ep_interrupt(void);
```

usb_standard_request.c**Public functions**

```
void handle_get_request(void);
void usb_process_request( void);
```

Gecko Errata

- simpleport reap error every second run. I presume this is due to some error in the way standard requests are handled. This is because of the Toggle bit. For some reason I don't see the select interface setup request that would normally reset the Toggle bit.
- HID operation fails to enumerate for AVROpendous modifiable keyboard.
- Standard request doesn't handle short control endpoints. The code to limit the packet size of sent packets is missing or in error.
- Why don't I see a setup request for selecting the interface? I think that the request must come during the current packet processing so that it gets lost. Perhaps I can clear the setup so that I start processing the setup. I should add a print before finishing the current setup request. This would tell me the interrupt state.
- Is the issue that I clear the fifo rather than the interrupt. As a result I loose the data?
- Why do I endup stuck waiting for an OUT request.

Gecko Strategy

- Consider clearing the interrupt bits after displaying the Interrupt trace
- Migrate to the new usb_api calls
 - start with BULK IN apps
 - test on CDC, UrJtag and Simpleport
 - migrate to BULK OUT apps
 - finally use the control streams within the library itself
- Tidy Gecko standard request to use native calls
 - compare to Lufa and Atmel with native calls
 - update source code appropriately

AVR Chameleon V1.0 Errata Mar 18 2009

- The BAT54C pad and pin numbers are out of order. The Eagle symbol needs to be re-worked.
- The connection of the CPLD to the HWB line prevents the HWB line from ever going low. The solution is to isolate this signal from the CPLD.
- The Rx and Tx data lines are connected to TMS and TDI which prevents reliable JTAG operations. Isolate TMS and TDI.
- Resistor R6 is 330ohms in the schematic and it should be 220ohms.
- The HWB time constant seems a little to slow. Try lowering the time by 50% (8Mohm to 4Mohm)
- A 10K pullup resistor is required on both sides of the CPLDs RST- line to the AVR.
- Note that debugWire is an open-collector bi-directional signal. Currently I haven't figured out how to configure the CPLD to handle this so debugWire doesn't work.
- When I try to directly connect the Chameleon to the ICE the 10pin and 14 pin headers won't allow direct insertion. So then I use a 1:1 ribbon cable execept that this flips the A and B row pins which requires a special CPLD load. Alternatively I can use the Atmel squid cable.

Further Testing Required

- I am uncertain about the power demands of the CPLD during use and programming. This needs to be tested.
- I also need to verify the de-coupling works for the CPLD
- Make the EP configuration include the control EP so that only the descriptor.c file defines the EP sizes.
- The AVR Chameleon can be programmed using the ICE however certain precautions are required. First it is important to use slew rate limited IO especially for the SCK signal.

```

#ifndef _USB_API_H_
#define _USB_API_H_
// ----- I N C L U D E S -----
#include "typedefs.h"           // Type definition
#include <stdio.h>
#include <avr/pgmspace.h>

// ----- Usb FIFO init -----
typedef struct{
    u08 valid ;
    u08 ConfigOK;
    u08 cfg0;
    u08 cfg1;
    u08 interrupt ;
} S_usb_endpoint;
#define RX_INT 0
#define TX_INT 1
#define NO_INT 2

//I need to define the EP number based on the processor 7 or 5
#define possible_eps 5
S_usb_endpoint ep_init [ possible_eps ];

//Note that because interrupts occur at any time, an ISR might change the
//value of the endpoint after a "Select_Endpoint" operation. To prevent this
//all ISRs should save and restore the EP number with these defines .
#define SaveEP() EP_temp = UENUM
#define RestoreEP() UENUM = EP_temp

//For stream handlers
//
#define USB_STREAM_TIMEOUT_MS 5
#define Stream_OK 0 //Command completed successfully, no error.
#define Stream_Stalled 1 //The endpoint was stalled during the
                        //stream transfer by the host or device.
#define Stream_Disconnected 1 //Device was disconnected from the host
                        //during the transfer .
#define Stream_Timeout 2 //The host failed to accept or send the
                        //next packet within the software timeout
                        //period set by the USB_STREAM_TIMEOUT_MS macro.

// Define the action to associate to each USB event
// The usb state machine is time dependant so it is important that you don't
// block or delay them with your user functions

#define Usb_sof_action () OneMsTimer();
#define Usb_wake_up_action()
#define Usb_resume_action()
#define Usb_suspend_action ();
#define Usb_reset_action ()
#define Usb_vbus_on_action ()
#define Usb_vbus_off_action ()
#define Usb_set_configuration_action ()
#define Usb_id_action ();

// ----- D E F I N I T I O N S -----
#define usb_task_time 5; //ms

```

```

#define usb_flush_time          10;      //ms

volatile u08 CPU, EP_temp;
volatile u16 Vid, Pid, timer1 ,timer2;
u08 rx_counter , RX_EP, TX_EP;

//Public functions
void usb_init (u08 xtal , u16 Vid, u16 Pid);
void usb_user_endpoint_init (u08 conf_nb);
void usb_OUT_Callback(void *fct);
void usb_IN_Callback(void *fct );
void Vendor_Callback(void *fct );
void *Usb_Vendor_Callback;
void Trace_Interrupt (void);

//Private functions
void OneMsTimer(void);
void usb_init_ep (u08 num, u08 type, u08 dir , u08 nyet, u08 size , u08 bank, u08 interrupt );
void usb_user_endpoint_init (u08 conf_nb);
void usb_config_ep (u08 config0, u08 config1 );
void usb_init_device (void);
void SetupEndpoints(void);
u08 usb_configured (void);
void bootloader_start (void);

u08 Endpoint_WaitUntilReady(void);
u08 Endpoint_RW_Stream(const void* Buffer, u16 Length);
u08 Endpoint_Write_Control_Stream(const void* Buffer, u16 Length);
u08 Endpoint_Read_Control_Stream(void* Buffer, u16 Length);

#endif // _USB_API_H_

```

```

//To Do
//1) Get rid of option to enable interrupts . Always enable with null handler
//2) Shorten device structures if possible
//3) Make all of the library DEBUG code work without stdio. Smaller library
//4) Migrate to a better general I/O structure for both uart and usb.
//
// ----- I N C L U D E S -----
#include <avr/io.h>
#include <avr/interrupt.h>
#include "include/CDC.api.h"
#include "include/typedefs.h"
#include "include/usb.api.h"
#include "include/usb.task.h"
#include "include/usb.driv.h"
#include "include/usb_descriptors.h"
#include "include/usb_standard_request.h"
#include <avr/eeprom.h>
#include <avr/power.h>           // clock prescaler
#include <avr/pgmspace.h>
#include <stdio.h>
#include "include/stdio_driver.h"
#include "include/usb_debug.h"
#define USB_STDIO

#if defined __AVR_AT90USB1287__ | __AVR_ATmega32U4__
void (*avrupdate_jump_to_boot)( void ) = (void *) 0x1e000; //0x0E000
#elif defined __AVR_AT90USB162__
void (*avrupdate_jump_to_boot)( void ) = (void *) 0x3000; //0x03000
#elif defined __AVR_ATmega32U4__
void (*avrupdate_jump_to_boot)( void ) = (void *) 0x6000; //0x06000
#endif

u08 ee_bootloader EEMEM = 1;
u16 debug_timer=0;

//
//Define this function in your main application code if you want the
//usb_OUT_Callback = Rx fifo data to be processed.
//usb_IN_Callback = Tx fifo data to be processed.
//Vendor_Callback = to process vendor specific setup requests .
//
void *OUT_Callback;
void *IN_Callback;
void *Usb_Vendor_Callback;

void usb_OUT_Callback(void *fct)
{
    OUT_Callback = fct;
}
void usb_IN_Callback(void *fct)
{
    IN_Callback = fct;
}
void Vendor_Callback(void *fct)
{
    Usb_Vendor_Callback = fct;
}
u08 No_user_handler(void)

```

```

{
    return FALSE;
}

//This is driven by the 1ms start of frame requests from the host
void OneMsTimer(void)
{
    u08 EP_temp;
    EP_temp = UENUM;                                //SaveEP();
    if (timer1 > 0) {--timer1;}                       //usb_task
    if (timer1 == 0) {usb_task(); timer1 = usb_task_time;}
#ifdef CDC
    if (timer2 > 0) {--timer2;}                       //usb_task
    if (timer2 == 0) {usb_flush(); timer2 = usb_flush_time;}
#endif
    debug_timer++;
    UENUM = EP_temp;                                //RestoreEP();
}

void usb_init (u08 xtal, u16 Vendorid, u16 Productid)
{
    //Manage watchdog timer after the bootloader finishes
    asm ("WDR");                                     //stop watchdog timer
    MCUSR= !(1<<WDRF);                               //Wdt_clear_flag
    WDTCSR |= (1<<WDCE) | (1<<WDE);                  //Wdt_change_enable
    WDTCSR = 0x00;                                   //Wdt_stop

    //Setup master clock
    CPU = xtal;
    if (CPU == 16){
        clock_prescale_set (0);                      //prescaler divides by 1
    } else if (CPU == 8){
        clock_prescale_set (1);                      //prescaler divides by 2
    }
    GTCCR = 0;                                       //enable prescaler

    usb_user_request_callback (No_user_handler);
    // usb_OUT_Callback(No_user_handler);
    // usb_IN_Callback(No_user_handler);
    // Vendor_Callback(No_user_handler);

    debug_to_uart ();                               //Setup debug if requested
    SetupEndpoints();                               //get the ep's from descriptor
    Vid = Vendorid;                                 //Setup USB VID and PID
    Pid = Productid;
    timer1 = usb_task_time;                         //Setup usb task timers
    timer2 = usb_flush_time;

    // Initialize usb software
    usb_task_init ();                               //Note: this must come 1st.
                                                //since it enables interrupts
    UDIEN |= (1<<SOFE);                             //Enable 1ms sof interrupt from host
}

u08 usb_configured (void)
{
    if (usb_configuration_nb != 0){
        return TRUE;
    } else {

```

```

        return FALSE;
    }
}

void usb_init_ep (u08 num, u08 type, u08 dir, u08 nyet, u08 size, u08 bank, u08 interrupt )
{
    if (num < possible_eps){
        ep_init [num].valid = TRUE;
        ep_init [num].cfg0 = ((type) | (nyet) | (dir));
        ep_init [num].cfg1 = ((size) | (bank<<2));
        ep_init [num].interrupt = interrupt ;
        debug_init ("\\ nusb_init_ep [%d]", num);
        debug_ep_init ( ep_init [num].cfg0, ep_init [num].cfg1);
        debug_init ("\\ ninterrupt = %d", interrupt );
        debug_init (" Config OK = %d\\n", ConfigOK());
    }
}

u08 Desc_size_toEP_size (u08 size)
{
    u16 volatile root, i;
    u08 power;
    root = size /8;
    power = 0;
    i=1;
    while(i != root){
        power = power+1;
        i = i*2;
    }
    size = power<<4;
    return size ;
}

```

//This routine reads the descriptor table and determines how to configure the physical hardware. This is done through three function calls as illustrated .

//1) SetupEndpoints is called to read the descriptor tables and extract the ep information

//2) usb_init_ep is called by 1 above to fill in a data structure that holds the actual data to be written to the hardware. This is done so that the hardware can be written from the lowest to highest ep number in sequence as required by the hardware.

//3) usb_user_endpoint_init actually writes the endpoint data to the hardware. It also saves the result status in the configuration table .

```

void SetupEndpoints(void)
{
    u08 ep, type, dir, size, bank, intr ;
    u08 volatile length ;
    u08 volatile * structure_ptr ;
    u08 volatile * end_of_structure ;
    length = (int)(conf_desc [0]. size);
    structure_ptr = (u08*)conf_desc [0]. conf_desc_ptr ;
    end_of_structure = structure_ptr +length;
    while( structure_ptr < end_of_structure ){
        structure_ptr += pgm_read_byte( structure_ptr );
        if (pgm_read_byte( structure_ptr +1) == ENDPOINT_DESCRIPTOR){
            ep = pgm_read_byte( structure_ptr +2)&0x0f;
            dir = pgm_read_byte( structure_ptr +2)>>7 ;

```



```

    type = pgm_read_byte( structure_ptr +3)<<6;
    size = pgm_read_byte( structure_ptr +4);

    size = Desc_size.toEP_size ( size );

    bank = (int)( EP_definition [ep].BitBank);
    intr = (int)( EP_definition [ep]. BitInterrupt );
    usb_init_ep (ep, type, dir, NYET_ENABLED, size, bank, intr);

    if ((type == BULK<<6) && (dir == OUT )){
        RX_EP = (int)( EP_definition [ep]. BitEndpoint );}; //CDC
    if ((type == INTR<<6) && (dir == IN )){
        TX_EP = (int)( EP_definition [ep]. BitEndpoint );}; //HID
    if ((type == BULK<<6) && (dir == IN )){
        TX_EP = (int)( EP_definition [ep]. BitEndpoint );}; //CDC
    };
};
}

// This function configures an endpoint with the selected type.
// Note that NYET should be enabled for USB16 to work

void usb_config_ep (u08 config0, u08 config1 )
{
    Endpoint_enable_endpoint ();
    UECFG0X = config0;
    UECFG1X = config1;
    Endpoint_allocate_memory ();
}

// This function initializes the USB device controller and
// configures the Control Endpoint 0.
//
void usb_init_device (void)
{
    u08 size;
    Endpoint_select_endpoint (EP_CONTROL);
    if (! Is_usb_endpoint_enabled ()) { //only set if unallocated
        size = pgm_read_byte(&usb_dev_desc.bMaxPacketSize0);
        size = Desc_size.toEP_size ( size );
        usb_config_ep (( Control|OUT|NYET_ENABLED),(size|(BANK_1<<2)));
    }
}

// This function is called when the host requests that the device is
// configured. The data to write to the hardware is stored in the
// ep_init structure which has been filled in by the SetupEndpoints
// call. Note that the hardware ep's are written from lowest to
// highest as required by the AT90USB devices.
//
// To do: Note that I need a common endpoint handler for each interrupt .
//
void usb_user_endpoint_init (u08 conf_nb)
{
    int ep;
    for(ep = 1; ep < possible_eps ; ep++){
        if ( ep_init [ep]. valid == TRUE){ //only fill actual endpoints
            Endpoint_select_endpoint (ep);

```

```

usb_config_ep ( ep_init [ep].cfg0, ep_init [ep].cfg1);

debug_init ("\\ nusb_user_endpoint_init [%d]",ep);
debug_ep_init ( ep_init [ep].cfg0, ep_init [ep].cfg1);
debug_init (" Config OK = %d\\n",ConfigOK());

ep_init [ep].ConfigOK = ConfigOK();           // flag for config status
Endpoint_reset_endpoint (ep);
if ( ep_init [ep]. interrupt == RX_INT){
    UEIENX |= 1<<RXOUTE;                     // enable RXOUTI for RX_EP
}
if ( ep_init [ep]. interrupt == TX_INT){
    UEIENX |= 1<<TXINE;                       // enable TXINI for TX_EP
}
}
}

//
//Note that this is not as fast as it could be. Because we don't free the
//Rx fifo until after the packet is processed we effectively run the USB
//transfer in series with the application rather than in parallel . We could
//use two buffers to fix this or re-work the design. For now its good enough.
//
SIGNAL(USB_COM_vect)
{
    u08 EP_temp;
    EP_temp = UENUM;                          //SaveEP();
    void (*ptr)();                             //
    Endpoint_select_endpoint (RX_EP);          // Select RX FIFO endpoint
    if (Endpoint_OUT_received_int() & (UEIENX & (1<<RXOUTE))) { //wait for host data
        UEINTX &= ~(1<<RXOUTI);
        ptr = OUT_Callback;                   // call application with buffer
        (*ptr)();                             //data to be processed
    }                                          //or (*ptr)(&buf);

#ifdef CDC
    Endpoint_select_endpoint (TX_EP);
    if ( Is_usb_in_ready () & (UEIENX & (1<<TXINE))) { //wait for host IN request
        ptr = IN_Callback;                   // call application with buffer
        (*ptr)();                             //data to be processed
    };
#endif
    UENUM = EP_temp;                          //RestoreEP();
}

void bootloader_start (void)
{
    eeprom_write_byte(&ee.bootloader,0xFF); //force bootloader to start
    eeprom_busy_wait ();                     //
    Usb_disable ();
    Usb_detach();
    avrupdate_jump_to_boot ();               //Jump to bootloader sector
}

```

```

//
// usb_debug.h
//

#ifndef _USB_DEBUG_H_
#define _USB_DEBUG_H_
#include "typedefs.h"          //debug definition

//Use the following options to enable the debug level required. The
//DEBUG flag must always be set to enable debugging; after that any
//combination of flags can be set by un-commenting the following defines.
//Note that because these options generate a lot of text messages the
//code size of your application will be increased substantially.
//Also note that the DEBUG_EP flag when selected will generate messages
//every 1ms of 1000/sec. Consider yourself warned!
#ifdef DEBUG
#define DEBUG_INIT           //Trace the device hardware init.
// #define DEBUG_ENUM       //Trace the enumeration process
// #define DEBUG_EP         //Trace the EP IN responses
// #define DEBUG_HID        //Trace the HID enumeration and report
// #define DEBUG_CDC        //Trace the CDC and line settings
#endif

// Function declarations
void debug_CDC_line_code(void);
void debug_to_uart(void);
void debug_HexDump(char* buf,int size);
void debug_ep_init(unsigned char cfg0,unsigned char cfg1);
void debug_dev_interrupt(void);
void debug_ep_interrupt(void);

#ifdef DEBUG
#define debug1(string,...) ; fprintf_P(stdout,PSTR(string),__VA_ARGS__)
#else
#define debug1(string,...) ; // this compiles to no code for all optimizations
#endif

#ifdef DEBUG_INIT
#define debug_init(string,...) ; fprintf_P(stdout,PSTR(string),__VA_ARGS__)
#else
#define debug_init(string,...) ; // this compiles to no code for all optimizations
#endif

#ifdef DEBUG_ENUM
#define debug_enum(string,...) ; fprintf_P(stdout,PSTR(string),__VA_ARGS__)
#else
#define debug_enum(string,...) ; // this compiles to no code for all optimizations
#endif

#ifdef DEBUG_EP
#define debug_ep_int(x)      debug_ep_int(x)
#define debug_ep(string,a)  printf_P(PSTR("\nEP%d "),UENUM);\
                           printf_P(PSTR("T=%d "),Usb_data_toggle());\
                           printf_P(PSTR(string),a);
#else
#define debug_ep_int(x)
#define debug_ep(string,a) // this compiles to no code for all optimizations
#endif

```

```
#ifdef DEBUG_CDC
#define debug_line( string ,... ) ; fprintf_P ( stdout ,PSTR(string),__VA_ARGS__)
#else
#define debug_line( string ,... ) ; // this compiles to no code for all optimizations
#endif

#ifdef DEBUG_HID
#define debug_hid( string ,... ) ; fprintf_P ( stdout ,PSTR(string),__VA_ARGS__)
#else
#define debug_hid( string ,... ) ; // this compiles to no code for all optimizations
#endif

#endif /* _USB_DEBUG_H_ */
```

```

// usb_debug.c
//
// The following debug routines can be used to help isolate USB problems
//when developing code. Each routine is conditionally included to limit
//code size . The DEBUG flag must be set for any of these routines to be
//available . Additionally the INIT,EP,CDC,HID flags can also be set to
//include specific routines as required . The hierarchy is as follows:
// DEBUG
//      debug_to_uart ()
//      debug_HexDump()
//      DEBUG_INIT
//      debug_ep_init ()
//      DEBUG_EP
//      debug_dev_interrupt ()
//      debug_ep_interrupt ()
//      DEBUG_CDC
//      debug_CDC_line_code()
//
//
// ----- I N C L U D E S -----
#include <avr/io.h>
#include "include/typedefs.h"
#include "include/usb_drv.h"
#include "include/usb_descriptors.h"
#include <stdio.h>
#include "include/stdio_driver.h"
#include "include/usb_debug.h"

#ifdef DEBUG
//connect USB driver to stdio
FILE uart_str = FDEV_SETUP_STREAM(uart_std_putchar, uart_std_getchar, _FDEV_SETUP_RW);

//This routine connects the uart to the STDIO print stream and sets the baud
//rate to 115200bps
void debug_to_uart (void)
{
    uart_init (115200);
    stdout = stdin = stderr = &uart_str;          //connect serial I/O stream
    printf_P (PSTR("\nUart debug started ... \n"));
}

//Print the content of size bytes of buf. in hex format
//The results are displayed as lines of 16 bytes.
void debug_HexDump(char* buf,int size)
{
    int i,j;
    for(i = 0; i<=(size/16); i++){
        printf_P (PSTR("\n"));
        if (size-(i*16) >16){
            // print full 16 byte lines
            for(j=0; j<16; j++){
                printf_P (PSTR(" 0x%2.2x"),buf[i*16+j]);
            }
        }
        else{
            for(j=0; j<(size-(i*16)); j++){
                // print shorter remainder line
                printf_P (PSTR(" 0x%2.2x"),buf[i*16+j]);
            }
        }
    }
}

```

```

    printf_P (PSTR("\n"));
}
#else
void debug_to_uart (void){}           //In case the option is disabled
void debug_HexDump(char* buf,int size){} //compile to nothing
#endif

#ifdef DEBUG_INIT
// It is often useful to decode the configuration of the AVR's DPRAM fifo. This
// routine prints each data field in the config0,1 registers as english text .
// Note that the configuration values are only valid after the USB device has
// been configured, before that they are undefined. That is to say if you call
// this routine from your code make sure that the device is configured first .
void debug_ep_init (unsigned char cfg0,unsigned char cfg1)
{
//Decode cfg0
    printf_P (PSTR("\nCFG0 Type = "));
    switch(((cfg0>>6)&0x03)){           //decode transfer type
        case 0x00:
            printf_P (PSTR("CONTROL "));
            break;
        case 0x01:
            printf_P (PSTR("ISOCHRONOUS "));
            break;
        case 0x02:
            printf_P (PSTR("BULK      "));
            break;
        case 0x03:
            printf_P (PSTR("INTERRUPT "));
            break;
        default :
            printf_P (PSTR("UNKNOWN "));
            break;
    }
    printf_P (PSTR("Direction = "));           //decode transfer direction
    if ((cfg0 & 0x01) == 0){
        printf_P (PSTR("OUT "));
    } else {
        printf_P (PSTR("IN"));
    }
}
//Decode cfg1
    printf_P (PSTR("\nCFG1 Size = "));
    switch(((cfg1>>4)&0x07)){           //decode ep size in bytes
        case 0x00:
            printf_P (PSTR("8"));
            break;
        case 0x01:
            printf_P (PSTR("16"));
            break;
        case 0x02:
            printf_P (PSTR("32"));
            break;
        case 0x03:
            printf_P (PSTR("64"));
            break;
        case 0x04:
            printf_P (PSTR("128"));
            break;
    }
}

```

```

    case 0x05:
        printf_P (PSTR("256"));
        break;
    case 0x06:
        printf_P (PSTR("512"));
        break;
    case 0x07:
        printf_P (PSTR("Reserved"));
        break;
}
printf_P (PSTR(" Banks = "));           //decode number of banks per ep
if ((cfg1 >> 2 & 0x01) == 0){
    printf_P (PSTR("ONE"));
} else {
    printf_P (PSTR("TWO"));
}
printf_P (PSTR(" Allocated = "));       //decode if the ep is allocated
if ((cfg1 >> 1 & 0x01) == 0){
    printf_P (PSTR("NO"));
} else {
    printf_P (PSTR("YES"));
}
}
#else
void debug_ep_init (unsigned char cfg0, unsigned char cfg1){}
//In case the option is disabled
//compile to nothing

#endif

#ifdef DEBUG_EP
//Decode device interrupts and display as english text
void debug_dev_interrupt (void)
{
    printf_P (PSTR("UERST 0x%x\n"), UERST);
    printf_P (PSTR("UDINT ="));
    if (UDINT & 1 << UPRSMI) {printf_P(PSTR(" Upstream resume "));};
    if (UDINT & 1 << EORSMI) {printf_P(PSTR(" End of resume "));};
    if (UDINT & 1 << WAKEUPI) {printf_P(PSTR(" Wakeup "));};
    if (UDINT & 1 << EORSTI) {printf_P(PSTR(" End of reset "));};
    if (UDINT & 1 << SOFI) {printf_P(PSTR(" Start of frame "));};
    if (UDINT & 1 << SUSPI) {printf_P(PSTR(" Suspend "));};
    printf_P (PSTR("\n"));
}

//Decode endpoint interrupts and display as english text
void debug_ep_interrupt (void)
{
    u08 i;
    u08 EP_temp;
    EP_temp = UENUM;                       //SaveEP();
    for (i=0; i<5; i++){
        Endpoint_select_endpoint (i);
        if ((UEINTX & (1 << RXOUTI)) | (UEINTX & (1 << TXINI)) | (UEINTX & (1 << RXSTPI))){
            if (UEINTX){
                printf_P (PSTR("EP%d enabled "), UENUM);
                printf_P (PSTR("time = %d "), debug_timer);
                if (UEINTX & (1 << FIFOCON)) {printf_P(PSTR(" FIFOCON "));};
                if (UEINTX & (1 << NAKINI)) {printf_P(PSTR(" Nak IN "));};
            }
        }
    }
}

```

```

        if (UEINTX & (1 << RWAL)) {printf_P(PSTR(" R/W allowed"))};
        if (UEINTX & (1 << NAKOUTI)) {printf_P(PSTR(" Rx Nak out "))};
        if (UEINTX & (1 << RXSTPI)) {printf_P(PSTR(" Rx stop "))};
        if (UEINTX & (1 << RXOUTI)) {printf_P(PSTR(" Rx OUT "))};
        if (UEINTX & (1 << STALLEDI)) {printf_P(PSTR(" Stalled "))};
        if (UEINTX & (1 << TXINI)) {printf_P(PSTR(" Tx IN      "))};
        printf_P (PSTR("\n"));
    }
}
}
UENUM = EP_temp;                //RestoreEP();
}
#else
void debug_dev_interrupt (void){}    //In case the option is disabled
void debug_ep_interrupt (void){}    //compile to nothing
#endif

#ifdef DEBUG_CDC
//Decode the USB CDC line coding and display the stop bits , party and line rate
void debug_CDC_line_code(void)
{
    printf_P (PSTR("      Baud: %lu bps "), (u32) line_coding .dwDTERate);
    switch( line_coding .bCharFormat){
    case 0:
        printf_P (PSTR(" 0"));
        break;
    case 1:
        printf_P (PSTR(" 1.5"));
        break;
    case 2:
        printf_P (PSTR(" 2"));
        break;
    };
    printf_P (PSTR(" stop bits Parity : "));
    switch( line_coding .bParityType){
    case 0:
        printf_P (PSTR("none"));
        break;
    case 1:
        printf_P (PSTR("odd"));
        break;
    case 2:
        printf_P (PSTR("even"));
        break;
    case 3:
        printf_P (PSTR("mark"));
        break;
    case 4:
        printf_P (PSTR("space"));
        break;
    };
    printf (" data bits : %d\n", line_coding .bDataBits);
}
#else
void debug_CDC_line_code(void){}    //In case the option is disabled
//compile to nothing
#endif

```



```

// usb_standard_request.h
//
// This file contains the USB endpoint 0 management routines corresponding to
// the standard enumeration process (refer to chapter 9 of the USB
// specification). The enumeration parameters (descriptor tables) are
// contained in the usb_descriptors.c file.

#ifndef _USB_STANDARD_REQUEST_H_
#define _USB_STANDARD_REQUEST_H_

// ----- I N C L U D E S -----
#include "usb_descriptors.h"

// ----- S T A N D A R D   D E F I N I T I O N S -----
// Device descriptor types for bmRequestType
// format of bmRequestType byte
#define bmRT_DIR_MASK          (0x1<<7)
#define bmRT_DIR_IN            (1<<7)
#define bmRT_DIR_OUT           (0<<7)
#define StdRequest             0x00
#define ClassRequest           0x20
#define VendorRequest          0x40

#define CTRL                   0x00
#define ISOC                   0x01
#define BULK                    0x02
#define INTR                   0x03

// Standard Requests
#define GET_STATUS              0x00
#define CLEAR_FEATURE           0x01           // see FEATURES below
#define SET_FEATURE             0x03           // see FEATURES below
#define SET_ADDRESS             0x05
#define GET_DESCRIPTOR          0x06
#define SET_DESCRIPTOR          0x07
#define GET_CONFIGURATION       0x08
#define SET_CONFIGURATION       0x09
#define GET_INTERFACE           0x0A
#define SET_INTERFACE           0x0B
#define SYNCH_FRAME             0x0C

#define REQUEST_DEVICE_STATUS   0x80
#define REQUEST_INTERFACE_STATUS 0x81
#define REQUEST_ENDPOINT_STATUS 0x82
#define ZERO_TYPE               0x00
#define INTERFACE_TYPE          0x01
#define ENDPOINT_TYPE           0x02

// Descriptor Types
#define DEVICE_DESCRIPTOR        0x01
#define CONFIGURATION_DESCRIPTOR 0x02
#define STRING_DESCRIPTOR        0x03
#define INTERFACE_DESCRIPTOR     0x04
#define ENDPOINT_DESCRIPTOR      0x05
#define DEVICE_QUALIFIER         0x06
#define IAD_DESCRIPTOR           0x0B

// Standard Features

```

```

// #define FEATURE_DEVICE_REMOTE_WAKEUP 0x01
#define FEATURE_ENDPOINT_HALT          0x00

// Device Status
#define USB_BUSPOWERED                  0x80
// #define USB_SELFPOWERED              0xC0
// #define USB_REMOTE_WAKEUP           0xA0

// ----- D E C L A R A T I O N -----
Bool    usb_user_read_request (u08 type, u08 request);
u08      bmRequestType, data_to_transfer, bmRequest;
u16      wValue, wIndex, wLength;
volatile u08    *pbuffer;
u08 stateep1, stateep2;

// ----- Public functions -----
void    handle_get_request (void);
void    usb_process_request ( void);
#endif  // _USB_STANDARD_REQUEST_H_

```

```

// usb_standard_request.c
//
// This file contains the USB control (EP0) management routines corresponding
// to the standard enumeration process (refer to chapter 9 of the USB
// specification). Non-standard requests are managed with a call to
// usb) user_read_request(). This provides a user call back for CDC, HID etc.
// The descriptor tables are contained in the usb_descriptors.c file.
// ----- I N C L U D E S -----
#include <avr/io.h>
#include <stdio.h>
#include <avr/pgmspace.h>
#include "include/usb_api.h"
#include "include/usb_drv.h"
#include "include/usb_standard_request.h"
#include "include/CDC_api.h"
#include "include/stdio_driver.h"
#include "include/usb_debug.h"

// ----- P R I V A T E   D E C L A R A T I O N -----
static void usb_get_descriptor (void);
static void usb_set_address (void);
static void usb_set_configuration (void);
static void usb_get_configuration (void);
static void usb_vendor_request (void);

// ----- D E C L A R A T I O N -----
static u08 zlp;
volatile u08 Unicode = FALSE;
volatile u08 AddHeader;
u16 wValue, wIndex, wLength;
u08 bmRequestType, bmRequest;
u08 usb_configuration_nb;
extern u08 usb_connected;
extern S_usb_device_descriptor usb_user_device_descriptor;
extern S_line_coding line_coding;

void * user_request_callback;
u08 (*ptr)();
u08 FIFO_depth = 0;

void usb_user_request_callback (void * fct)
{
    user_request_callback = fct;
}

//
// This function reads the SETUP request sent to the default control
// endpoint 0 and calls the appropriate function. On exiting the
// device is ready to manage the next request.
//
void usb_process_request (void)
{
    u08 bmRequest;

    bmRequestType = Usb_read_u08();
    bmRequest = Usb_read_u08();
    LSB(wValue) = Usb_read_u08();
    MSB(wValue) = Usb_read_u08();
    //read setup parameters

```

```

LSB(wIndex)    = Usb_read_u08();
MSB(wIndex)    = Usb_read_u08();
LSB(wLength)   = Usb_read_u08();
MSB(wLength)   = Usb_read_u08();
debug_enum("\n-----", ' ');
debug_enum("\nSetup request: bmRequestType=0x%x ",bmRequestType);
debug_enum("bmRequest=0x%x ",bmRequest);
debug_enum("wValue=0x%x ",wValue);
debug_enum("wIndex=0x%x ",wIndex);
debug_enum("wLength=0x%x\n",wLength);
switch(bmRequestType & 0x60){                                //mask request type
case StdRequest:
    switch(bmRequest){
        case GET_DESCRIPTOR:
            if (0x80 == bmRequestType) {
                debug_enum("Get descriptor %c",'\n');
                usb_get_descriptor ();
                return;};

        case GET_CONFIGURATION:
            if (0x80 == bmRequestType) {
                debug_enum("Get configuration %c",'\n');
                usb_get_configuration ();
                return;};

        case SET_ADDRESS:
            if (0x00 == bmRequestType) {
                debug_enum("Set address %c",'\n');
                usb_set_address ();
                return;};

        case SET_CONFIGURATION:
            if (0x00 == bmRequestType) {
                debug_enum("Set configuration %c",'\n');
                usb_set_configuration ();
                return;};

        default :
            debug_enum("    Unknown std_request%c", ' ');
            break;
    }; //end of bmRequest case "End of standard requests"
// case ClassRequest:
//     debug_enum("Class request%c",'\n');
//     break;
case VendorRequest:
    debug_enum("Vendor request%c",'\n');
    usb_vendor_request ();
    break;
default :                                //un-supported request
    if ( usb_user_read_request (bmRequestType, bmRequest) == FALSE){
        debug_enum("Un-supported request%c\n", ' ');
        Usb_stall_and_ack_setup ();
    };
}; //end of bmRequestType case
}

// This function manages the custom vendor requests . If a vendor_callback
// function is provided control is transfered otherwise the vendor request

```

```

// triggers a reboot of the bootlader.
//
void usb_vendor_request (void)
{
    u08 (*ptr)(); // define vendor callback
    Endpoint_ack_setup_receive (); //Occurs before writing reply
    if (Usb_Vendor_Callback){ // call vendor setup handler
        debug_ep("Usb_Vendor_Callback: 0x%x\n",Usb_Vendor_Callback);
        ptr = Usb_Vendor_Callback; // if it is defined
        (*ptr)(); // actual call
    } else {
        debug_ep(" bootloader_start !%c",'\n');
//Comment out this line so that the Ubuntu Linux doesn't confuse the device ito re-booting
// bootloader_start ();
    }
    Usb_send_control_in (); //Send packet
    debug_ep(" usb_process_request : VendorRequest ep#%d\n",UENUM);
    while(! Is_usb_in_ready ()); //waits for status phase done
}

// This function manages the SET ADDRESS request. When complete, the device
// will filter requests using the new address.
//
void usb_set_address (void)
{
    Endpoint_ack_setup_receive ();
    while(! Is_usb_in_ready ()); //Wait for the control fifo available
    Usb_send_control_in (); //send a ZLP for STATUS phase
    debug_ep(" usb_set_address : ep#%d\n",UENUM);
    Usb_configure_address (LSB(wValue)); // set address
    while(! Is_usb_in_ready ()); //waits for status phase done(fifo empty)
    Usb_enable_address (); //before using the new address
    debug_enum(" usb_set_address : address=0x%x\n",LSB(wValue));
}

// This function manages the SET CONFIGURATION request. If the selected
// configuration is valid, this function call the usb_user_endpoint_init ()
// function that will configure the endpoints following the configuration
// number.
//
void usb_set_configuration ( void )
{
    u08 i,EP_temp;
    u08 configuration_number;
    Endpoint_ack_setup_receive ();
    configuration_number = LSB(wValue);
    if ( configuration_number <= NB_CONFIGURATION){
        usb_configuration_nb = configuration_number;
    } else {
        Usb_stall_and_ack_setup ();
    }
    Usb_send_control_in (); //< send a ZLP for STATUS phase
    debug_ep(" usb_set_configuration : ep#%d\n",UENUM);
    usb_user_endpoint_init ( usb_configuration_nb ); //< endpoint configuration
    Usb_set_configuration_action ();
    debug_enum(" usb_set_configuration : config=0x%x\n",configuration_number);

    EP_temp = UENUM; //SaveEP();

```

```
//This function returns the user string descriptors
//The data for the strings is stored in Flash memory as a null
//terminated string . The USB protocol requires a 16 bit non-zero
//terminated UNICODE string; so we we set a UNICODE flag and
// correct the output as it is played out in the handle_get_descriptor ()
// routine .
```

```
// This function manages the GET_DESCRIPTOR request. The device descriptor,
// and the configuration descriptor are supported. All
// other descriptors must be supported by the usb_user_get_descriptor
// function. Only 1 configuration is supported. All data is stored in Flash.
//
```

```
void usb_get_descriptor (void)  
{  
    u08 descriptor_type , string_type ;  
    //>>>>>>>>>>Is the ZLP definition required It is handled later in this code
```

```

zlp          = FALSE;                //no zero length packet
string_type  = LSB(wValue);
descriptor_type = MSB(wValue);
debug_enum("    usb_get_descriptor : string_type=0x%x ", string_type );
debug_enum(" descriptor_type=0x%x\n", descriptor_type );
switch ( descriptor_type ){
    case DEVICE_DESCRIPTOR:
        data_to_transfer = Usb_get_dev_desc_length ; // size
        pBuffer          = Usb_get_dev_desc_pointer ; //data
        debug_enum("    DEVICE_DESCRIPTOR%c",' ');
        break;
    case CONFIGURATION_DESCRIPTOR:
        data_to_transfer = (int)(conf_desc [0]. size ); // size
        pBuffer          = (u08*)conf_desc [0]. conf_desc_ptr ; //data
        debug_enum("    CONFIGURATION_DESCRIPTOR%c",' ');
        break;
    case STRING_DESCRIPTOR:
        debug_enum("    STRING_DESCRIPTOR%c",' ');
        usb_string_descriptor ( string_type );
        break;
    case DEVICE_QUALIFIER:
        debug_enum("    DEVICE_QUALIFIER%c",' \n');
    default :
        debug_enum("    Unknown descriptor%c",' ');
        Usb_stall_and_ack_setup () ;
        return;
}
handle_get_request () ;
//This code looks to see if we have received a second setup request while
//processing the current request. If so we need to stall the host so that
//it resends the request after we return from this interrupt ; otherwise we
//miss the setup request.
/* if( Endpoint_setup_received_int () ){
    Endpoint_enable_stall () ;
    Endpoint_ack_setup_receive () ;
}*/
}

// This code creates a Unicode packet in the USB fifo. The first time this
// is called the AddHeader flag should be set to append the 2 byte Unicode
// header to the packet. That is the length and string descriptor .
// Subsequent data consists of 16 bit Unicode in the form of the byte
// followed by a 0x00 byte.
//
void Send_UNICODE(void)
{
    FIFO_depth +=2;
    if (AddHeader){
        Usb_write_byte (wLength*2+2);
        Usb_write_byte (STRING_DESCRIPTOR);
        AddHeader = FALSE;
        debug_enum(" 0x%x ",wLength*2+2);
        // length=2xbytes+hdr
        // only add hdr once
    } else {
        debug_enum("%c",pgm_read_byte(pBuffer));
        Usb_write_byte (pgm_read_byte(pBuffer++)); // write low byte
        Usb_write_byte (0x00); // write high byte
        wLength --;
    }
};

```

```

}
// This code handles the ram based VID and PID as setup by the usb_init ()
// function . The Flash based descriptors stored in descriptor .c are
// dynamically over-written with the SRAM values.
void Dynamic_VID_PID(void)
{
//Handle ram based VID and PID
    if (pbuffer == (u08 *)&usb_dev_desc.idVendor){
        Usb_write_byte ((u08)(Vid&0xFF));
        Usb_write_byte ((u08)(Vid>>8 &0xFF));
        Usb_write_byte ((u08)(Pid&0xFF));
        Usb_write_byte ((u08)(Pid>>8 &0xFF));
        debug_enum("0x%x ",(u08)(Vid&0xFF));
        debug_enum("0x%x ",(u08)(Vid>>8&0xFF));
        debug_enum("0x%x ",(u08)(Pid&0xFF));
        debug_enum("0x%x ",(u08)(Pid>>8&0xFF));
        pbuffer +=4;           //send VID 2 bytes plus
        wLength -=4;          //send PID 2 byte = 4
        FIFO_depth +=4;
    }
}

void Format_hex_print (void)
{
// Format the debug string to show 8 hex characters per line
    debug_enum("0x%x ",pgm_read_byte(pbuffer));
// debug_enum("--[0x%x] ",FIFO_depth);
    if ((FIFO_depth&0x07) == 7){
        if (FIFO_depth != EP_CONTROL_LENGTH){
            if ((wLength!=1)){
                debug_enum("\n    Data:%c",' ');
            };
        };
    };
}

void handle_get_request (void)
{
    Endpoint_ack_setup_receive ();           //acknowledge the setup request
    debug_enum("\nCSetupRx--UEINTX &= ~(1<<RXSTPI);",' ');
    if (wLength > data_to_transfer ){        // limit the packet size to the lesser
        wLength = data_to_transfer ;        // of datato transfer or wLength
    }
    AddHeader = TRUE;                       //Add a header of
                                           //( length ,STRING) to
                                           // the UNICODE playback
                                           // process the entire transfer (s)

    while(wLength){
        FIFO_depth = 0;
        while(! Is_usb_read_control_enabled ()){ //wait for the control fifo available
            if (Endpoint_OUT_received_int ()){
                debug_enum("\nCSetupRx--UEINTX &= ~(1<<RXSTPI);",' ');
                UEINTX &= ~(1<<RXSTPI);
                return;
            }
        }
        debug_ep(" UEINTX & (1 << TXINI)=%d ",(UEINTX & (1 << TXINI)));
        debug_enum("\n    byte_counter ()=0x%x ",Usb_byte_counter());
        debug_enum("wLength=0x%x ",wLength);
    }
}

```



```

    debug_enum("pbuffer=0x%x\n    Data: ", pbuffer);
    while(wLength && (FIFO_depth<EP_CONTROL_LENGTH)){//no bigger than EP0
        if (Unicode){//Convert to UNICODE?
            Send.UNICODE();
        } else {
            Dynamic.VID_PID();//replace Flash values with sram VID/PID
            Usb_write_byte(pgm_read_byte(pbuffer));//write byte
            Format_hex_print();
            pbuffer++;
            wLength--;
            FIFO_depth++;
        }
    }
    zlp = (FIFO_depth == EP_CONTROL_LENGTH);
    debug_enum("\nCSI--UEINTX &= ~(1<<TXINI);", ' ');
    debug_ep(" handle_get_request :SEND", ' ');
    Usb_send_control_in();//Send packet
}

    Unicode = FALSE;//Only set for string descriptors

if(zlp) {
    debug_enum("handle_get_request :ZLP %c", '\n');
    while(! Is_usb_read_control_enabled());//wait for the control fifo available
    Usb_send_control_in();
    debug_enum("\nZLPCSI--UEINTX &= ~(1<<TXINI);", ' ');
}
while(! Endpoint_OUT_received_int());
debug_enum("\nCSO--UEINTX &= ~(1<<RXOUTI);", ' ');
UEINTX &= ~(1<<RXOUTI);
debug_ep(" handle_get_request :ACK\n", ' ');//ep send
}

// This function manages the GET CONFIGURATION request. The current
// configuration number is returned.
//
void usb_get_configuration (void)
{
    Endpoint_ack_setup_receive();
    Usb_write_byte(usb_configuration_nb);
    Usb_ack_in_ready();
    debug_ep("usb_get_configuration1 \n", ' ');//ep send
    while(!Endpoint_OUT_received_int());
    Usb_ack_receive_out();
    debug_ep("usb_get_configuration2 \n", ' ');//ep send
    debug_enum("usb_get_configuration : nb=0x%x\nData: ",usb_configuration_nb);
}

// -----
// This function is called by the standard usb read request function when
// the Usb request is not supported. This function returns TRUE when the
// request is processed. This function returns FALSE if the request is not
// supported. In this case, a STALL handshake will be automatically
// sent by the standard usb read request function.
//
Bool usb_user_read_request (u08 type, u08 request)
{
    ptr = user_request_callback;
    return((* ptr)(&type,&request));
}

```

```
}
```

```
//  
// usb_task.h  
//  
// This file contains the function declarations  
  
#ifndef _USB_TASK_H_  
#define _USB_TASK_H_  
// ----- D E C L A R A T I O N S -----  
void usb_task_init (void);  
void usb_start_device (void);  
void usb_task (void);  
u08 usb_configuration_nb ;  
  
#endif /* _USB_TASK_H_ */
```

```

// The USB task checks the arrival of new requests from the USB Host.
// Other class specific requests are also processed in this file .
// This file manages all the USB events:
// Suspend / Resume / Reset / Start Of Frame / Wake Up / Vbus events / Id transition
//
// ----- I N C L U D E S -----
#include <avr/io.h>
#include <avr/interrupt.h>
#include "include/usb_api.h"
#include "include/usb_task.h"
#include "include/usb_drv.h"
#include "include/usb_standard_request.h"

//
// This function enables the USB controller and init the USB interrupts .
// The aim is to allow the USB connection detection in order to send
// the appropriate USB event to the operating mode manager.
//
void usb_task_init (void)
{
    sei (); //enable interrupts
    Usb_force_device_mode();
    Usb_select_device ();
    Endpoint_select_endpoint (EP_CONTROL);
    Usb_enable();
    Usb_enable_vbus_pad();
    Usb_enable_vbus_interrupt ();
    UDIEN |= (1<<EORSTE); //Enable reset interrupt
    usb_start_device ();
}

//
// This function enables the USB controller and init the USB interrupts .
// The aim is to allow the USB connection detection in order to send
// the appropriate USB event to the operating mode manager.
//
void usb_start_device (void)
{
    #if defined __AVR_AT90USB1287__
        Usb_enable_regulator ();
        if (CPU == 8){
            PLLCSR = ((0<<PLLP2)|(1<<PLLP1)|(1<<PLLP0)|(1<<PLLE)); //8MHz
        } else if (CPU == 16){
            PLLCSR = ((1<<PLLP2)|(0<<PLLP1)|(1<<PLLP0)|(1<<PLLE)); //16MHz
        }
    #elif defined __AVR_ATmega32U4__
        Usb_enable_regulator ();
        PLLFRQ = (1<<PDIV1 | 1<<PDIV3); //96MHz PLL
        PLLFRQ |= 1<<PLLUSB; //48MHz to USB
        if (CPU == 8){
            PLLCSR &= ~(1<<PINDIV);
        } else if (CPU == 16){
            PLLCSR |= 1<<PINDIV;
        }
        PLLCSR |= 1<<PLLE;
    #elif defined __AVR_AT90USB162__

```

```

    if (CPU == 8){
        PLLCSR = ((0<<PLL2)|(0<<PLL1)|(0<<PLL0)|(1<<PLLE)); //8MHz
    } else if (CPU == 16){
        PLLCSR = ((0<<PLL2)|(0<<PLL1)|(1<<PLL0)|(1<<PLLE)); //16MHz
    }
}

#endif

while (!(PLLCSR & (1<<PLOCK)));           //wait for PLL to lock
Usb_unfreeze_clock ();
UDIEN |= (1<<SUSPE);                       //Enable suspend interrupt
usb_init_device ();                         //configure EP0
// usb_user_endpoint_init (0);             // this is a hack ... to force
                                           // the device to configuration 0
                                           // untill a set config is requested
UDIEN = 1<<EORSME | 1<<EORSTE | 1<<WAKEUPE; //resume, reset & wakeup int
Usb_attach ();
}

//
// This function is the entry point of the USB management.
// If a Setup request occurs on the Default Control Endpoint,
// the usb_process_request () function is call
void usb_task (void)
{
    Endpoint_select_endpoint (EP_CONTROL);
    if ( Endpoint_setup_received_int () ){
        usb_process_request ();
    }
}

//
// This function is called each time a USB interrupt occurs.
// The following USB events are processed:
// - VBus On / Off           -only on USB128
// - Id transition          -only on USB128
// - Start Of Frame
// - Suspend
// - Wake-Up
// - Resume
// - Reset
// The file usb_api.h defines what user action takes place for each
// interrupt .
//
SIGNAL(USB_GEN_vect)
{
    u08 EP_temp;
    EP_temp = UENUM;                       //SaveEP();
    #if defined __AVR_AT90USB1287__ | defined __AVR_ATmega32U4__
        if ( Is_usb_vbus_transition () ){   //Check for VBUS transitions
            Usb_ack_vbus_transition ();
            if ( Is_usb_vbus_high () ){      //usb connected
                Usb_vbus_on_action ();
                Usb_enable_reset_interrupt ();
                usb_start_device ();
            } else {
                Usb_vbus_off_action ();      //usb detached
            }
        }
    }
}

```



```

// ../usb/library/include/usb_descriptors.h
//
// This file contains the usb descriptor structures
//

#ifndef _USB_DESCRIPTOR_H_
#define _USB_DESCRIPTOR_H_

// ----- I N C L U D E S -----
#include "usb_api.h"

// ----- M A C R O S -----
#define Usb_get_dev_desc_pointer    &(usb_dev_desc.bLength)
#define Usb_get_dev_desc_length    sizeof (usb_dev_desc)

// ----- U S B   D E F I N E -----
#define CONF_ATTRIBUTES            USB_BUSPOWERED
//USB Device Descriptor
#define NB_CONFIGURATION            1
#ifndef CDC
#define EP_CONTROL_LENGTH            32                //64 for full speed USB2.0
#else
#define EP_CONTROL_LENGTH            64                //64 for full speed USB2.0
//#define EP_CONTROL_LENGTH            8                //64 for full speed USB2.0
#endif

#define LANG_ID                    0
#define MAN_INDEX                    1
#define PROD_INDEX                    2
#define SN_INDEX                    3
#define Language_ID                0x0409            //English

#define DEVICE_STATUS                0x00            // TBD
#define INTERFACE_STATUS            0x00            // TBD

#define EP0 0                        // Control
#define EP1 1                        // CDC Tx
#define EP2 2                        // CDC Rx
#define EP3 3                        // CDC Int
#define EP4 4
#define EP5 5
#define EP6 6
#define EP7 7

// -----Usb      Request-----
typedef struct {
    u08 bmRequestType;    u08 bRequest;                u16 wValue;
    u16 wIndex;            u16 wLength;
} S_UsbRequest;

// -----Usb      Device      Descriptor-----
typedef struct {
    u08 bLength;            u08 bDescriptorType;        u16 bscUSB;
    u08 bDeviceClass;        u08 bDeviceSubClass;        u08 bDeviceProtocol;
    u08 bMaxPacketSize0;    u16 idVendor;            u16 idProduct;
    u16 bcdDevice;            u08 iManufacturer;        u08 iProduct;
    u08 iSerialNumber;        u08 bNumConfigurations;
} S_usb_device_descriptor ;

```

```

// -----Usb Configuration Descriptor-----
typedef struct {
    u08 bLength;          u08 bDescriptorType;          u16 wTotalLength;
    u08 bNumInterfaces;    u08 bConfigurationValue;    u08 iConfiguration ;
    u08 bmAttributes;      u08 MaxPower;
} S_usb_configuration_descriptor ;

// -----Usb Interface Associated Descriptor-----
typedef struct {
    u08 bLength;          u08 bDescriptorType;          u08 bFirstInterface ;
    u08 bInterfaceCount;  u08 bFunctionClass;          u08 FunctionSubClass;
    u08 bFunctionProtocol; u08 iFunction ;
} S_usb_Interface_Association_descriptor ;

// -----Usb Interface Descriptor-----
typedef struct {
    u08 bLength;          u08 bDescriptorType;          u08 bInterfaceNumber;
    u08 bAlternateSetting; u08 bNumEndpoints;          u08 bInterfaceClass ;
    u08 bInterfaceSubClass; u08 bInterfaceProtocol ;    u08 iInterface ;
} S_usb_interface_descriptor ;

// -----Usb Endpoint Descriptor-----
typedef struct {
    u08 bLength;          u08 bDescriptorType;          u08 bEndpointAddress;
    u08 bmAttributes;      u16 wMaxPacketSize;          u08 bInterval ;
} S_usb_endpoint_descriptor ;

// -----Usb Device Qualifier Descriptor-----
typedef struct {
    u08 bLength;          u08 bDescriptorType;          u16 bscUSB;
    u08 bDeviceClass;      u08 bDeviceSubClass;          u08 bDeviceProtocol;
    u08 bMaxPacketSize0;   u08 bNumConfigurations;        u08 bReserved;
} S_usb_device_qualifier_descriptor ;

// -----Usb Language Descriptor-----
typedef struct {
    u08 bLength;          u08 bDescriptorType;          u16 wlangid;
} S_usb_language_descriptor ;

// -----Usb User Descriptor-----
typedef struct {
    S_usb_configuration_descriptor    cfg;
    S_usb_interface_descriptor        ifc0;
    S_usb_endpoint_descriptor         ep1;
    S_usb_endpoint_descriptor         ep2;
} S_usb_user_configuration_descriptor ;

S_usb_device_descriptor    usb_dev_desc ;
S_usb_device_qualifier_descriptor    usb_qual_desc ;
S_usb_language_descriptor    usb_user_language_id ;

// -----CDC Functional Descriptors-----
//See CDC Specification 1.1 for details

// -----Header Functional Descriptor-----
typedef struct {
    u08 bLength;          u08 bDescriptorType;          u08 bDscSubType;
    u16 bcdCDC;

```



```

} S_usb__CDC_header_descriptor;

// ----- Abstract Control Management Descriptor -----
typedef struct {
    u08 bLength;          u08 bDescriptorType;          u08 bDscSubType;
    u08 bmCapabilities;
} S_usb_CDC_ACM_descriptor;

// ----- Union Functional Descriptor -----
typedef struct {
    u08 bLength;          u08 bDescriptorType;          u08 bDscSubType;
    u08 bMasterIntf;      u08 bSlaveIntf0;
} S_usb_CDC_union_descriptor;

// ----- Call Management Functional Descriptor -----
typedef struct {
    u08 bLength;          u08 bDescriptorType;          u08 bDscSubType;
    u08 bmCapabilities;   u08 bDataInterface;
} S_usb_CDC_call_management_descriptor;

// ----- Usb CDC Descriptor -----
typedef struct {
    S_usb_configuration_descriptor    cfg;
    S_usb_interface_descriptor        ifc0;
    S_usb__CDC_header_descriptor      cdca;
    S_usb_CDC_call_management_descriptor cdcba;
    S_usb_CDC_ACM_descriptor          cdcc;
    S_usb_CDC_union_descriptor        cdcd;
    S_usb_endpoint_descriptor         ep3;
    S_usb_interface_descriptor        ifc1;
    S_usb_endpoint_descriptor         ep1;
    S_usb_endpoint_descriptor         ep2;
} S_usb_CDC_descriptor;

// ----- Usb CDC IA Descriptor -----
typedef struct {
    S_usb_configuration_descriptor    cfg;
    S_usb_Interface_Association_descriptor iad1;
    S_usb_interface_descriptor        ifc0;
    S_usb__CDC_header_descriptor      cdca;
    S_usb_CDC_call_management_descriptor cdcba;
    S_usb_CDC_ACM_descriptor          cdcc;
    S_usb_CDC_union_descriptor        cdcd;
    S_usb_endpoint_descriptor         ep3;
    S_usb_interface_descriptor        ifc1;
    S_usb_endpoint_descriptor         ep1;
    S_usb_endpoint_descriptor         ep2;
} S_usb_CDCIA_descriptor;

// ----- Usb CDC with UrJTAG Descriptor -----
typedef struct {
    S_usb_configuration_descriptor    cfg;
    S_usb_interface_descriptor        ifc0;
    S_usb_endpoint_descriptor         ep1;
    S_usb_endpoint_descriptor         ep2;
    S_usb_Interface_Association_descriptor iad1;
    S_usb_interface_descriptor        ifc1;
    S_usb__CDC_header_descriptor      cdca;

```

```

    S_usb_CDC_call_management_descriptor    cdc_b;
    S_usb_CDC_ACM_descriptor               cdcc;
    S_usb_CDC_union_descriptor             cdcd;
    S_usb_endpoint_descriptor              ep3;
    S_usb_interface_descriptor             ifc2 ;
    S_usb_endpoint_descriptor              ep4;
    S_usb_endpoint_descriptor              ep5;
} S_usb_UrJTAG_descriptor;

// ----- U S B   H I D   D E S C R I P T O R -----
typedef struct {
    u08 bLength;                          // Size of this descriptor in bytes
    u08 bDescriptorType;                  // HID descriptor type
    u16 bscHID;                           // Binay Coded Decimal Spec. release
    u08 bCountryCode;                     // Hardware target country
    u08 bNumDescriptors;                  // Number of HID class descriptors to follow
    u08 bRDescriptorType;                 // Report descriptor type
    u16 wDescriptorLength;                // Total length of Report descriptor
} S_usb_hid_descriptor ;

typedef struct {
    int      size ;                       // Size of report in bytes
    u08*     report_ptr ;                 // pointer to start of HID_report
} S_hid_report ;

// extern S_hid_report HID_report[2];
extern S_hid_report HID_report [3];

typedef struct
{
    S_usb_configuration_descriptor        cfg_kbd;
    S_usb_interface_descriptor            ifc_kbd ;
    S_usb_hid_descriptor                  hid_kbd;
    S_usb_endpoint_descriptor             ep1_kbd;
    S_usb_interface_descriptor            ifc1 ;
    S_usb__CDC_header_descriptor           cdca;
    S_usb_CDC_call_management_descriptor   cdc_b;
    S_usb_CDC_ACM_descriptor              cdcc;
    S_usb_CDC_union_descriptor            cdcd;
    S_usb_endpoint_descriptor             ep2;
    S_usb_interface_descriptor            ifc2 ;
    S_usb_endpoint_descriptor             ep3;
    S_usb_endpoint_descriptor             ep4;
} S_usb_HID_KBD_CDC_descriptor;

typedef struct
{
    S_usb_configuration_descriptor        cfg_kbd;
    S_usb_interface_descriptor            ifc_kbd ;
    S_usb_hid_descriptor                  hid_kbd;
    S_usb_endpoint_descriptor             ep1_kbd;
} S_usb_HID_keyboard_descriptor ;

typedef struct
{
    S_usb_configuration_descriptor        cfg_mouse;
    S_usb_interface_descriptor            ifc_mouse;

```

```

        S_usb_hid_descriptor          hid_mouse;
        S_usb_endpoint_descriptor     ep1_mouse;
    } S_usb_HID_mouse_descriptor;

```

typedef struct

```

{
    S_usb_configuration_descriptor    cfg_mouse;
    S_usb_interface_descriptor        ifc_mouse;
    S_usb_hid_descriptor              hid_mouse;
    S_usb_endpoint_descriptor         ep1_mouse;
    S_usb_interface_descriptor        ifc_kbd ;
    S_usb_hid_descriptor              hid_kbd;
    S_usb_endpoint_descriptor         ep2_kbd;
} S_usb_HID_composite_descriptor ;

```

typedef struct

```

{
    S_usb_configuration_descriptor    cfg_mouse;
    S_usb_interface_descriptor        ifc_mouse;
    S_usb_hid_descriptor              hid_mouse;
    S_usb_endpoint_descriptor         ep1_mouse;
    S_usb_interface_descriptor        ifc_kbd ;
    S_usb_hid_descriptor              hid_kbd;
    S_usb_endpoint_descriptor         ep2_kbd;
    S_usb_interface_descriptor        ifc_listen ;
    S_usb_hid_descriptor              hid_listen ;
    S_usb_endpoint_descriptor         ep3_listen ;
    S_usb_endpoint_descriptor         ep4_listen ;
} S_usb_HID_composite_listen_descriptor ;

```

```

/*{
    S_usb_configuration_descriptor    cfg_mouse;
    S_usb_interface_descriptor        ifc_mouse;
    S_usb_hid_descriptor              hid_mouse;
    S_usb_endpoint_descriptor         ep1_mouse;
    S_usb_interface_descriptor        ifc_kbd ;
    S_usb_hid_descriptor              hid_kbd;
    S_usb_endpoint_descriptor         ep2_kbd;
    S_usb_configuration_descriptor    cfg_listen ;
    S_usb_interface_descriptor        ifc_listen ;
    S_usb_endpoint_descriptor         ep3_listen ;
    S_usb_endpoint_descriptor         ep4_listen ;
} S_usb_HID_composite_listen_descriptor ;*/

```

typedef struct

```

{
    S_usb_configuration_descriptor    cfg_mouse;
    S_usb_interface_descriptor        ifc_mouse;
    S_usb_hid_descriptor              hid_mouse;
    S_usb_endpoint_descriptor         ep1_mouse;
    S_usb_interface_descriptor        ifc_kbd ;
    S_usb_hid_descriptor              hid_kbd;
    S_usb_endpoint_descriptor         ep2_kbd;
    S_usb_interface_descriptor        ifc_control ;
    S_usb_endpoint_descriptor         ep3_Tx;
    S_usb_endpoint_descriptor         ep4_Rx;
} S_super_composite_descriptor ;

```

*//This structure allows us to define which conf_desc to use while
 //maintaining a constant size structure as required by the library use.*

```
typedef struct
{
    int      size;           // size of configuration descriptor in bytes
    u08*     conf_desc_ptr;  // pointer to configuration descriptor
} S_conf_desc;
```

```
extern S_conf_desc conf_desc [1];
```

```
typedef struct
{
    u08*     string;
    u08      size;
} S_string_desc;
```

```
extern S_string_desc string_desc [3];
```

```
typedef struct
{
    u08      BitInterrupt :2,
            BitBank:2,
            BitEndpoint:4;
} S_EP_definition;
```

```
extern S_EP_definition EP_definition [6];
```

```
#endif  // _USB_DESCRIPTOR_H_
```

```

//
// ../ usb_library /usb_CDC_descriptors.c
//This file contains the usb parameters that uniquely identify the
//usbCDC16 application through descriptor tables .
// ----- I N C L U D E S -----

#include " ../ Gecko_library /include /usb_api .h"
#include " ../ Gecko_library /include /usb_drv .h"
#include " ../ Gecko_library /include /usb_descriptors .h"
#include " ../ Gecko_library /include /usb_standard_request .h"
#include <avr/pgmspace.h>

#define Manufacturer "USB16"
#define Product "Serial CDC demo"
#define Serial "GPL/GPL2"

// usb device descriptor
S_usb_device_descriptor PROGMEM usb_dev_desc =
{
    sizeof(usb_dev_desc),           // Size of descriptor in bytes
    DEVICE_DESCRIPTOR,              // DEVICE descriptor type
    0x0200,                         // BCD – USB Spec Release Number
    0x02,                           // Class Code
    0,                              // Subclass code
    0,                              // Protocol code
    EP_CONTROL_LENGTH,              // Max size EP0, usb_descriptors .h
    0x03eb,                         // Vendor ID
    0x2020,                         // Product ID
    0x1000,                         // BCD – Device release number
    MAN_INDEX,                     // Manufacturer string index
    PROD_INDEX,                   // Product string index
    SN_INDEX,                      // Serial number string index
    1                              // Number of configurations
};

// usb configuration descriptor
S_usb_CDC_descriptor PROGMEM usb_conf_desc = {
    { sizeof( S_usb_configuration_descriptor ), // Size of descriptor in bytes
      CONFIGURATION_DESCRIPTOR,                // CONFIGURATION descriptor type
      sizeof(usb_conf_desc),                   // Total length of data for cfg
      0x02,                                    // Number of interfaces in cfg
      0x01,                                    // Index value of configuration
      0,                                       // Configuration string index
      CONF_ATTRIBUTES,                        // Attributes , usb_descriptors .h
      100,                                    // Max power consumption (2X mA)
    },
    // Interface 0 Descriptor
    { sizeof( S_usb_interface_descriptor ), // Size of descriptor in bytes
      INTERFACE_DESCRIPTOR,                // INTERFACE descriptor type
      0,                                  // Interface Number
      0,                                  // Alternate Setting Number
      1,                                  // Number of endpoints in intf
      0x02,                               // Class code where CDC ACM=0x02
      0x02,                               // Subclass: 0x02=communication
      0x01,                               // Protocol: 0x01=AT commands
      0,
    },
};
// CDC Class-Specific Descriptors

```

```

// see Table 25 in USB CDC Specification 1.1
{
    sizeof(S_usb__CDC_header_descriptor), //
    0x24, //CS_INTERFACE
    0x00, //DSC_FN_HEADER
    0x0110 //
},
{
    sizeof(S_usb_CDC_call_management_descriptor), //
    0x24, //CS_INTERFACE
    0x01, //DSC_FN_CALL_MGT
    0x03, //
    0x01 //CDC_DATA_INTF_ID
},
{
    sizeof(S_usb_CDC_ACM_descriptor), //
    0x24, //CS_INTERFACE
    0x02, //DSC_FN_ACM— Abstract Control Management
    0x06 //
},
{
    sizeof(S_usb_CDC_union_descriptor), //
    0x24, //CS_INTERFACE
    0x06, //DSC_FN_UNION
    0x00, //CDC_COMM_INTF_ID
    0x01 //CDC_DATA_INTF_ID
},
//Usb Endpoint 3 Descriptor
{
    sizeof( S_usb_endpoint_descriptor ), //
    ENDPOINT_DESCRIPTOR, //
    (0x80|EP3), //Endpoint number
    INTR, //Transfer type BULK,INTR,ISOC
    32, //EP fifo size in bytes
    0xFF, //Interval ms
},
// Interface 1 Descriptor
{
    sizeof( S_usb_interface_descriptor ), // Size of descriptor in bytes
    INTERFACE_DESCRIPTOR, // INTERFACE descriptor type
    1, // Interface Number
    0, // Alternate Setting Number
    2, // Number of endpoints in intf
    0x0A, // Class code:0x0A=CDC ACM i/f
    0, // Subclass code
    0, // Protocol code
    0, // Interface string index
},
//Usb Endpoint 1 Descriptor
{
    sizeof( S_usb_endpoint_descriptor ), // Size of endpoint in bytes
    ENDPOINT_DESCRIPTOR, //
    (0x80|EP1), //Endpoint number
    BULK, //EP transfer type attribute
    32, //EP fifo size in bytes
    0x00, //Endpoint interval
},
//Usb Endpoint 2 Descriptor
{
    sizeof( S_usb_endpoint_descriptor ), // Size of endpoint in bytes
    ENDPOINT_DESCRIPTOR, //
    EP2, //Endpoint number
    BULK, //EP transfer type attribute
    32, //EP fifo size in bytes
    0x00 //Endpoint interval
}

```

```

};

S_conf_desc conf_desc[]={
    { sizeof (usb_conf_desc),
      (u08*)&usb_conf_desc}
};

// usb_user_language_id
S_usb_language_descriptor PROGMEM usb_user_language_id = {
    sizeof (usb_user_language_id)
,   STRING_DESCRIPTOR
,   Language_ID                      //LANGUAGE_ID = English
};

u08 Manufacturer_string [] PROGMEM = {Manufacturer};
u08 Product_string [] PROGMEM = {Product};
u08 Serial_string [] PROGMEM = {Serial};

S_string_desc string_desc [] = {
    { Manufacturer_string , sizeof (Manufacturer)-1},
    { Product_string , sizeof (Product)-1},
    { Serial_string , sizeof ( Serial )-1}
};

S_EP_definition EP_definition [] = {
    {NO_INT,BANK_1,EP0},
    {NO_INT,BANK_1,EP1},
    {NO_INT,BANK_1,EP2},
    {NO_INT,BANK_1,EP3}
};

```

```
// stdio_driver .h
#ifndef _stdio_driver_
#define _stdio_driver_
#include <stdio.h>
#include "typedefs.h"           //my favourite typedefs

void uart_init (u32 baud);
u08 uart_rx_ready (void);
u08 uart_getchar (void);
void uart_putchar (u08 data);
int  uart_std_putchar (char c, FILE *stream);
int  uart_std_getchar (FILE *stream);

//connect USB driver to stdio
FILE uart_str ;

#endif
```



```

// stdio_driver .c
// ----- I N C L U D E S -----
#include "include/stdio_driver.h"
#include "include/usb16.h"

void uart_init (u32 baud)
{
    SerialDDR |= 1<<TXD;           //allow txd to be an output
    UBRR1H = (u08)((u32)F_CPU/((u32)baud*8)-1)>>8);
    UBRR1L = (u08)((u32)F_CPU/((u32)baud*8)-1);
    UCSRA = 1<<U2X1;              //double rate allows a lower
    UCSRB = ((1<<RXEN) | (1<<TXEN)); //bit error at 115200
}

u08 uart_rx_ready (void)
{
    return(UCSRA & (1<<RXC));
}

u08 uart_getchar (void)
{
    while ( !(UCSRA & (1<<RXC)) ){}
    return(UDR);
}

void uart_putchar (u08 data)
{
    while ( !(UCSRA & (1<<UDRE)) ){}
    UDR1 = data;           // Start transmission
}

// -----The following two functions provide stdio-----
//
// Send usb character to host. Note that the data is only queued to be sent.
// The actual data transfer occurs when either the tx buffer is full or when
// the flush timer times out.
//
int uart_std_putchar (char c, FILE *stream)
{
    uart_putchar (c);
    if (c == '\n'){
        uart_putchar ('\r');
    }
    return 0;
}

/*
 * Receive a character from the uart port.
 *
 * This features a simple line-editor that allows for delete to
 * edit the characters entered, until either CR or NL is entered.
 * Printable characters entered will be echoed using uart_std_putchar ().
 *
 * Editing characters :
 *
 * \b (BS) or \177 (DEL) delete the previous character
 * \t (tab) will be replaced by a single space
 *
 * All other control characters will be ignored.

```

```

*
* The internal line buffer is RX_BUFSIZE characters long, which
* includes the terminating \n (but no terminating \0). If the buffer
* is full (i. e., at RX_BUFSIZE-1 characters in order to keep space for
* the trailing \n), any further input attempts will send a \a (BEL character)
* to uart_putchar() , although line editing is still allowed.
*
* Successive calls to uart_getchar() will be satisfied from the
* internal buffer until that buffer is emptied again.
*/
#define RX_BUFSIZE 20

u08 *uart_rxp = 0;
int uart_std_getchar (FILE *stream)
{
    u08 RxChar, *CharPtr, TextBuffer[RX_BUFSIZE];

    if (uart_rxp == 0)
        for (CharPtr = TextBuffer ;;) {
            RxChar = uart_getchar ();
            if (RxChar == CR){                                     // behaviour similar to Unix stty ICRNL
                RxChar = new_line;
            }
            if (RxChar == new_line){                               // CR to newline
                *CharPtr = RxChar;
                uart_std_putchar (RxChar, stream);
                uart_rxp = TextBuffer;
                break;
            } else if (RxChar == tab){                             // tab to space
                RxChar = ' ';
            };

            if ((RxChar >= (u08)' ' && RxChar <= (u08)'\x7e') || RxChar >= (u08)'\xa0'){
                if (CharPtr == TextBuffer + RX_BUFSIZE - 1){
                    uart_std_putchar (bell, stream); //no room in buffer
                } else {
                    *CharPtr++ = RxChar;             //add ch to buffer
                    uart_std_putchar (RxChar, stream); //echo character
                }
                continue;
            }
        }
        if ((RxChar == '\b') | (RxChar == '\x7f')){ //backspace or del
            if (CharPtr > TextBuffer){
                uart_std_putchar (backspace, stream);
                uart_std_putchar (space, stream);
                uart_std_putchar (backspace, stream);
                CharPtr--;
            }
        }
        RxChar = *uart_rxp++;                                // last char in buffer
        if (RxChar == new_line){
            uart_rxp = 0;
        }
    return RxChar;
}

```

```

//usb_drv.h
#ifndef _USB_DRV_H_
#define _USB_DRV_H_

#ifdef __AVR_AT90USB1287__
    #define boot_vector 0x1e000;    //4K boot sector at address 0xf000*2=0x1e000
#endif
#ifdef __AVR_AT90USB162__
    #define boot_vector 0x3800;    //2K boot sector at address 0x1C00*2=0x03800
#endif
#ifdef __AVR_ATmega32U4__
    #define boot_vector 0x7000;    //2K boot sector at address 0x3800*2=0x07000
#endif

#define Usb_data_toggle ()        ((UESTA0X&MSK_DTSEQ) >> 2)
#define MSK_DTSEQ                  0x0C

// ----- MACROS -----
//keep this order (set StallRq/clear RxSetup) or an
//OUT request following the SETUP may be acknowledged
#define Usb_stall_and_ack_setup () \
    Endpoint_enable_stall (); \
    Endpoint_ack_setup_receive (); \
    return;

#define clock_prescale_set (x) \
{ \
    uint8_t tmp = _BV(CLKPCE); \
    __asm__ __volatile__ ( \
        "in __tmp_reg__, __SREG__" "\n\t" \
        "cli" "\n\t" \
        "sts %1, %0" "\n\t" \
        "sts %1, %2" "\n\t" \
        "out __SREG__, __tmp_reg__" \
        : /* no outputs */ \
        : "d" (tmp), \
        "M" (_SFR_MEM_ADDR(CLKPR)), \
        "d" (x) \
        : "r0"); \
}

//USB EndPoint
#define EP_CONTROL                  0
#define MSK_EP_DIR                  0x7F
#define MSK_UADD                    0x7F
//Parameters for endpoint configuration
#define Control                     0<<6
#define Isochronous                 1<<6
#define Bulk                        2<<6
#define Interrupt                   3<<6

#define OUT                         0
#define IN                          1

#define SIZE_8                      0<<4
#define SIZE_16                    1<<4
#define SIZE_32                    2<<4
#define SIZE_64                    3<<4

```

```

#define SIZE_128          4<<4
#define SIZE_256          5<<4
#define SIZE_512          6<<4
#define SIZE_1024         7<<4

#define BANK_1             0      //0<<2
#define BANK_2             1      //1<<2

#define NYET_ENABLED       0<<1
#define NYET_DISABLED     1<<1

#define usb_configure_endpoint (num, type, dir, size, bank, nyet)\
    ( Usb_select_endpoint (num),\
      usb_config_ep (((type) |(nyet) |(dir)),((size) |(bank))))

//General USB management
#define Usb_enable_regulator ()      (UHWCON |= (1<<UVREGE))
#define Usb_disable_uid_pin ()      (UHWCON &= ~(1<<UIDE))

#if defined __AVR_AT90USB1287__
#define Usb_force_device_mode()      ( Usb_disable_uid_pin (), UHWCON |= (1<<UIMOD))
#define Usb_select_device ()          (USBCON &= ~(1<<HOST))
#else
#define Usb_force_device_mode()
#define Usb_select_device ()
#endif

#if defined __AVR_AT90USB1287__ | __AVR_ATmega32U4__
#define Usb_enable_vbus_pad()          (USBCON |= (1<<OTGPADE))
#define Usb_disable_vbus_pad ()        (USBCON &= ~(1<<OTGPADE))
#define Usb_enable_vbus_interrupt ()    (USBCON |= (1<<VBUSTE))
#else
#define Usb_enable_vbus_pad()
#define Usb_disable_vbus_pad ()
#define Usb_enable_vbus_interrupt ()
#endif

#define Usb_enable()                (USBCON |= (1<<USBE))
#define Usb_disable ()               (USBCON &= ~(1<<USBE))
#define Usb_freeze_clock ()          (USBCON |= (1<<FRZCLK))
#define Usb_unfreeze_clock ()        (USBCON &= ~(1<<FRZCLK))

#define Usb_enable_reset_interrupt () (UDIEN |= (1<<EORSTE))

#define Is_usb_vbus_high ()           (USBSTA & (1<<VBUS))
#define Is_usb_id_device ()           (USBSTA & (1<<ID))

#define Usb_ack_id_transition ()      (USBINT = ~(1<<IDTI))
#define Usb_ack_vbus_transition ()    (USBINT = ~(1<<VBUSTI))
#define Is_usb_id_transition ()       (USBINT & (1<<IDTI))
#define Is_usb_vbus_transition ()     (USBINT & (1<<VBUSTI))

// USB Device management
#define Usb_attach ()                 (UDCON &= ~(1<<DETACH))
#define Usb_detach()                 (UDCON |= 1<<DETACH)
#define Usb_ack_resume()              (UDINT = ~(1<<EORSMI))
#define Usb_ack_wake_up()             (UDINT = ~(1<<WAKEUPI))

```

```

#define Usb_ack_reset () (UDINT = ~(1<<EORSTI))
#define Usb_ack_sof () (UDINT = ~(1<<SOFI))
#define Usb_ack_suspend () (UDINT = ~(1<<SUSPI))
#define Is_usb_wake_up () (UDINT & (1<<WAKEUPI))
#define Usb_enable_address () (UDADDR |= (1<<ADDEN))
#define Usb_configure_address (addr) (UDADDR = (UDADDR & (1<<ADDEN)) | ((u08)addr & MSK_UADD))

// General endpoint management
#define Endpoint_select_endpoint (ep) (UENUM = (u08)ep )
#define Endpoint_reset_endpoint (ep) (UERST = 1 << (u08)ep, UERST = 0)
#define ENDPOINT_DIR_OUT (0 << EPDIR)
#define ENDPOINT_DIR_IN (1 << EPDIR)
#define Endpoint_SetEndpointDirection ( dir) UECFG0X = ((UECFG0X & ~ENDPOINT_DIR_IN) | (dir));

#define Endpoint_enable_endpoint () (UECONX |= (1<<EPEN))
#define Endpoint_disable_endpoint () (UECONX &= ~(1<<EPEN))
#define Endpoint_enable_stall () (UECONX |= (1<<STALLRQ))
#define Endpoint_reset_data_toggle () (UECONX |= (1<<RSTDT))
#define Endpoint_clear_stall () (UECONX |= (1<<STALLRQC))
#define Is_usb_endpoint_enabled () (UECONX & (1<<EPEN))

#define Endpoint_allocate_memory () (UECFG1X |= (1<<ALLOC))
#define Endpoint_deallocate_memory () (UECFG1X &= ~(1<<ALLOC))
#define ConfigOK () (UESTA0X & (1<<CFGOK))
#define IN_Busy () (UESTA0X & (1<<NBUSYBK1 | 1<<NBUSYBK0))

#define Usb_FIFO_available () (UEINTX&(1<<RWAL))
#define Usb_FIFO_full () !(UEINTX&(1<<RWAL))
#define Is_usb_read_control_enabled () (UEINTX&(1<<TXINI))
#define Endpoint_setup_received_int () (UEINTX&(1<<RXSTPI))
#define Endpoint_OUT_received_int () (UEINTX&(1<<RXOUTI))
#define Is_usb_in_ready () (UEINTX&(1<<TXINI))

#define Endpoint_ack_setup_receive () (UEINTX &= ~(1<<RXSTPI));
#define Usb_ack_receive_out () (UEINTX &= ~(1<<RXOUTI); \
UEINTX &= ~(1<<FIFOCON);

#define Endpoint_clear_out () (UEINTX &= ~(1<<RXOUTI);
#define Usb_ack_in_ready () (UEINTX &= ~(1<<TXINI); \
UEINTX &= ~(1<<FIFOCON);

#define Usb_send_in () (UEINTX &= ~(1<<FIFOCON));
#define Usb_send_control_in () (UEINTX &= ~(1<<TXINI));

#define Usb_read_u08 () (UEDATX)
#define Usb_write_byte (byte) (UEDATX = (u08)byte)
#define Usb_write_u16(u16) (UEDATX = (u08)&u16[0]), \
(UEDATX = (u08)&u16[1])
#define Usb_write_u32(u32) (UEDATX = (u08)&u32[0]), \
(UEDATX = (u08)&u32[1]), \
(UEDATX = (u08)&u32[2]), \
(UEDATX = (u08)&u32[3])

//#define Usb128_byte_counter() (((u16)UEBCHX) << 8) | (UEBCLX)
//#define Usb16_byte_counter() ((u08)UEBCLX)

#if defined __AVR_AT90USB1287__ | __AVR_ATmega32U4__
#define Usb_byte_counter () (((u16)UEBCHX) << 8) | (UEBCLX))
#elif defined atmega32u4

```

```

#define      Usb_byte_counter ()          (((u16)UEBCHX) << 8) | (UEBCLX))
#elif defined __AVR_AT90USB162__
#define      Usb_byte_counter ()          ((u08)UEBCLX)
#endif

#define Usb_write_line_coding1 \
    Usb_write_byte(LSB0(line_coding [1]. dwDTERate));\
    Usb_write_byte(LSB1(line_coding [1]. dwDTERate));\
    Usb_write_byte(LSB2(line_coding [1]. dwDTERate));\
    Usb_write_byte(LSB3(line_coding [1]. dwDTERate));\
    Usb_write_byte( line_coding [1]. bCharFormat);\
    Usb_write_byte( line_coding [1]. bParityType);\
    Usb_write_byte( line_coding [1]. bDataBits);\
    print_line_code ();

#define Usb_read_line_coding1 \
    LSB0(line_coding [1]. dwDTERate) = Usb_read_u08();\
    LSB1(line_coding [1]. dwDTERate) = Usb_read_u08();\
    LSB2(line_coding [1]. dwDTERate) = Usb_read_u08();\
    LSB3(line_coding [1]. dwDTERate) = Usb_read_u08();\
    line_coding [1]. bCharFormat = Usb_read_u08();\
    line_coding [1]. bParityType = Usb_read_u08();\
    line_coding [1]. bDataBits = Usb_read_u08();\
    print_line_code ();

#define Usb_write_line_coding2 \
    Usb_write_byte(LSB0(line_coding [2]. dwDTERate));\
    Usb_write_byte(LSB1(line_coding [2]. dwDTERate));\
    Usb_write_byte(LSB2(line_coding [2]. dwDTERate));\
    Usb_write_byte(LSB3(line_coding [2]. dwDTERate));\
    Usb_write_byte( line_coding [2]. bCharFormat);\
    Usb_write_byte( line_coding [2]. bParityType);\
    Usb_write_byte( line_coding [2]. bDataBits);\
    print_line_code ();

#define Usb_read_line_coding2 \
    LSB0(line_coding [2]. dwDTERate) = Usb_read_u08();\
    LSB1(line_coding [2]. dwDTERate) = Usb_read_u08();\
    LSB2(line_coding [2]. dwDTERate) = Usb_read_u08();\
    LSB3(line_coding [2]. dwDTERate) = Usb_read_u08();\
    line_coding [2]. bCharFormat = Usb_read_u08();\
    line_coding [2]. bParityType = Usb_read_u08();\
    line_coding [2]. bDataBits = Usb_read_u08();\
    print_line_code ();

#define Usb_interrupt_flags ()          (UEINT != 0x00)

#endif // _USB_DRV_H_

```