

PDIUSBD12 MIDI Adapter

Manuel Odendahl

June 8, 2007

Contents

1 Main application code for PDIUSBD12 MIDI Adapter

```
#include "app.h"

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

#include "common.h"
#include "d12.h"
#include "usb.h"
#include "midi_link_desc.h"
#include "uart.h"
#include "usb_midi.h"
```

Callback for outgoing MIDI messages. Send midi buffer if not empty.

```
void midi_ep_in_callback(uint8_t ep) {
    uint8_t buf[1];
    d12_read_cmd(D12_CMD_READ_LAST_TX_STATUS + ep, buf, 1);

    if (!d12_device_is_configured())
        return;

    usb_midi_send_buf();
}
```

Buffer for receiving midi messages.

```
static volatile uint8_t recv_bytes = 0;
static volatile uint8_t recv_buf[64];
```

Process MIDI data from the host, called when data is received on the OUT endpoint.

```

void midi_ep_out_callback(uint8_t ep) {
    uint8_t buf[1];
    d12_read_cmd(D12_CMD_READ_LAST_TX_STATUS + ep, buf, 1);
    if (!d12_device_is_configured())
        return;
}

```

Read data from D12.

```

recv_bytes = d12_read_ep_pkt(ep, recv_buf, sizeof(recv_buf));
if (!(recv_bytes % 4)) {
    uint8_t count = 0;
    uint8_t i;
}

```

Dispatch to MIDI stack (same as on AT90USB1286

```

    for (i = 0; i < recv_bytes; i += 4, count++) {
        if (!usb_midi_handle_out_msg(0, (usb_midi_message_t *) (recv_buf + i))) {
            break;
        }
    }
}
}
}

```

Main loop for MIDI.

```

void midi_link_main(void) {
    if (d12_device_is_configured()) {
}

```

Receive data on UART if usb is configured.

```

    while (uart_avail()) {
        uint8_t c = uart_getc();
        usb_midi_handle_rx_byte(c);
    }
}
}

```

Main routine for midi-link.

```

int main(void) {

```

Disable watchdog.

```

    wdt_disable();

```

setbits for the LEDs.

```

    DDRB = _BV(1) | _BV(2);

```

Initialize UART.

```

    uart_init();
    usb_midi_init();

```

Initialize D12 pins and stack.

```

    d12_pins_init();
    d12_init();

```

Enable interrupts.

```

    sei();

    for (;;) {

```

Handle usb status.

```

        d12_main();

```

Handle midi from UART and from USB.

```

        midi_link_main();
    }
}

```

2 Communication with PDIUSB12

```

#include "app.h"

#include <avr/io.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <avr/sleep.h>

#include "d12.h"
#include "usb.h"
#include "midi_link_desc.h"

```

Configured USB device address, set during enumeration

```

static uint8_t d12_dev_address = 0;

```

D12 chip configuration flags. These flags are set mostly during enumeration according to messages received from the server. Flags are also used to signal a suspend state.

```

static uint8_t d12_flags = 0;

```

Device has been configured during enumeration.

```

#define D12_FLAG_DEV_CONFIGURED 0x01

```

Device address received, need to set correct address.

```

#define D12_FLAG_TX_ADDRESS      0x02

```

Read data to be sent on endpoint 0 from flash memory and not from working memory

```

#define D12_FLAG_FLASH           0x04

```

Need to enable endpoints (configuration successful).

```
    #define D12_FLAG_ENABLE_EP      0x08
```

USB bus has gone into suspend, suspend flash. XXX need to set INT0 interrupt to wake up.

```
    #define D12_FLAG_SUSPEND        0x10
```

Send buffer for ep0, because messages can be split between different interrupt states.

```
    static uint8_t *tx_ptr, tx_bytes;
```

Check the suspend line from D12.

```
    static inline uint8_t d12_is_suspend(void) {  
        return IS_BIT_SET(PINB, SUSPEND_PIN);  
    }
```

Check the interruptline from d12 (low active).

```
    static inline uint8_t d12_is_irq(void) {  
        return !IS_BIT_SET(PIND, 2);  
    }
```

External access to configured flag.

```
    uint8_t d12_device_is_configured(void) {  
        return (d12_flags & D12_FLAG_DEV_CONFIGURED) && !(d12_flags & D12_FLAG_ENABLE_EP);  
    }
```

2.1 Communication with D12

Write and read the data lines to d12, and bitbang RD, WR and A0 accordingly.
Set the IO pins to output and write a byte to the data lines

```
    static inline void d12_write_to_output(uint8_t data) {  
        DDRC = 0x3F;  
        DDRD |= _BV(6) | _BV(7);  
        PORTC = data & 0x3F;  
        PORTD = (data & 0xC0) | (PORTD & 0x3F);  
    }
```

Switch the IO pins to input.

```
    static inline void d12_set_to_input(void) {  
        DDRC = 0x00;  
        DDRD &= ~_BV(6);  
        DDRD &= ~_BV(7);  
        PORTC = 0x3F;  
        PORTD |= 0xC0;  
    }
```

Initialize the pins needed for the communication with D12.

```

void d12_pins_init(void) {
    DDRD |= _BV(A0_PIN) | _BV(RD_PIN) | _BV(WR_PIN);
    DDRB |= _BV(0); /* suspend pin */
    DDRD &= ~_BV(2); /* interrupt pin */
    PORTD &= ~_BV(2);
}

static void d12_write_data(uint8_t data);

```

Write a command byte to d12.

```

static void d12_write_cmd_byte(uint8_t cmd) {
    SET_A0();
    CLEAR_WR();
    d12_write_to_output(cmd);
    asm("nop");
    SET_WR();
}

```

Write a command and its data to D12.

```

void d12_write_cmd(uint8_t cmd, uint8_t *data, uint8_t cnt) {
    d12_write_cmd_byte(cmd);
    uint8_t i;
    for (i = 0; i < cnt; i++) {
        d12_write_data(data[i]);
    }
}

```

Write a command and its data (from flash) to D12.

```

void d12_write_cmd_flash(uint8_t cmd, uint8_t *data, uint8_t cnt) {
    d12_write_cmd_byte(cmd);
    uint8_t i;
    for (i = 0; i < cnt; i++) {
        d12_write_data(pgm_read_byte(data + i));
    }
}

```

Write a data byte to D12.

```

static void d12_write_data(uint8_t data) {
    CLEAR_A0();
    CLEAR_WR();
    d12_write_to_output(data);
    asm("nop");
    SET_WR();
}

```

Read a data byte from D12.

```

static uint8_t d12_read_data(void) {
    d12_set_to_input();
    CLEAR_A0();
}

```

```

    CLEAR_RD();
    asm("nop");
    uint8_t data = (PINC & 0x3F) | (PIND & 0xC0);
    SET_RD();
    return data;
}

```

Write a command byte to D12 and read answer data.

```

void d12_read_cmd(uint8_t cmd, uint8_t *buf, uint8_t cnt) {
    d12_write_cmd_byte(cmd);

    uint8_t i;
    for (i = 0; i < cnt; i++) {
        buf[i] = d12_read_data();
    }
}

```

Write a data packet to an endpoint of D12, validate the buffer afterwards.

```

void d12_write_ep_pkt(uint8_t ep, uint8_t *buf, uint8_t cnt) {
    /* select endpoint */
    uint8_t status;
    d12_read_cmd(D12_CMD_SELECT_EP + ep, &status, 1);
    d12_write_cmd_byte(D12_CMD_WRITE_BUFFER);
    d12_write_data(0x00);
    d12_write_data(cnt);
    uint8_t i;
    for (i = 0; i < cnt; i++) {
        d12_write_data(buf[i]);
    }
    d12_write_cmd(D12_CMD_VALIDATE_BUFFER, NULL, 0);
}

```

Write a data packet form flash to D12 endpoint.

```

void d12_write_ep_pkt_flash(uint8_t ep, uint8_t *buf, uint8_t cnt) {
    /* select endpoint */
    uint8_t status;
    d12_read_cmd(D12_CMD_SELECT_EP + ep, &status, 1);
    d12_write_cmd_byte(D12_CMD_WRITE_BUFFER);
    d12_write_data(0x00);
    d12_write_data(cnt);
    uint8_t i;
    for (i = 0; i < cnt; i++) {
        d12_write_data(pgm_read_byte(buf + i));
    }
    d12_write_cmd(D12_CMD_VALIDATE_BUFFER, NULL, 0);
}

```

Read an endpoint of D12 and clear the buffer afterwards. Returns the received length.

```

uint8_t d12_read_ep_pkt(uint8_t ep, uint8_t *buf, uint8_t cnt) {
    /* select endpoint */
    uint8_t status;
    d12_read_cmd(D12_CMD_SELECT_EP + ep, &status, 1);
    if (status & 0x01) {
        /* data available */
        uint8_t len[2];
        d12_read_cmd(D12_CMD_READ_BUFFER, len, 2);
        if (len[1]) {
            if (len[1] > cnt)
                len[1] = cnt;
            uint8_t i;
            for (i = 0; i < len[1]; i++) {
                buf[i] = d12_read_data();
            }
            d12_write_cmd(D12_CMD_CLEAR_BUFFER, buf, 0);
        }
        return len[1];
    } else {
        return 0;
    }
}

```

Write the ep0 buffer to endpoint 0.

```

void d12_write_ep0_buf(void) {
    uint8_t to_send = 0;

    if (tx_bytes == 0)
        return;

    to_send = tx_bytes;
    if (to_send > D12_EPO_SIZE)
        to_send = D12_EPO_SIZE;

    if (d12_flags & D12_FLAG_FLASH)
        d12_write_ep_pkt_flash(D12_EPO_IN, tx_ptr, to_send);
    else
        d12_write_ep_pkt(D12_EPO_IN, tx_ptr, to_send);

    tx_ptr += to_send;
    tx_bytes -= to_send;
}

```

Stall the control endpoint (in case a command can't be answered).

```

void d12_stall_ep0(void) {
    uint8_t cmd = 1;
    /* stall pid in response to next DATA stage TX */
    d12_write_cmd(D12_CMD_SET_EP_STATUS + D12_EPO_IN, &cmd, 1);
    /* stall pid in response to the next status stage */
    d12_write_cmd(D12_CMD_SET_EP_STATUS + D12_EPO_OUT, &cmd, 1);
}

```

Stall an endpoint.

```

void d12_stall_ep(uint8_t ep) {
    uint8_t cmd ;
    d12_read_cmd(D12_CMD_READ_LAST_TX_STATUS + ep, &cmd, 1);
    cmd = 1; /* stall ep */
    d12_write_cmd(D12_CMD_SET_EP_STATUS + ep, &cmd, 1);
}

```

Initialize the D12 stack.

```

void d12_init(void) {
    d12_flags = 0;
    tx_bytes = 0;
    tx_ptr = NULL;
    uint8_t buf[2];

    /* set address and enable */
    buf[0] = 0x80;
    d12_write_cmd(D12_CMD_SET_ADDRESS, buf, 1);

    /* enable generic endpoints */
    buf[0] = 0x01;
    d12_write_cmd(D12_CMD_SET_EP_ENABLE, buf, 1);

    buf[0] = _BV(6) // ep index 4 interrupt enable
        | _BV(7) // ep index 5 interrupt enable
        ;
    // d12_write_cmd(D12_CMD_SET_DMA, buf, 1);

    buf[0] =
        _BV(1) // no lazyclock
        | _BV(2) // clock running, even during suspend
        | _BV(3) // interrupt mode
        | _BV(4) // softconnect
        // mode 0
        ;
    buf[1] = 0x5; // clock division 48 / (5 + 1) = 8 Mhz
    d12_write_cmd(D12_CMD_SET_MODE, buf, 2);
}

```

Structure to hold the callbacks for different D12 interrupt status.

```

typedef struct {
    uint8_t mask;
    void (*function)(uint8_t);
    uint8_t param;
} usb_event_list_t;

```

Switch to atmel power saving mode. XXX need to enable interrupt on INT0 here!

```

void set_power_save(void) {
    MCUCR = _BV(SE) | _BV(SM0) | _BV(SM1) | (MCUCR & 0x0F);
}

```

Handler for USB suspend state. Switch to power save mode. XXX interrupt handler for INT0


```

void d12_suspend_handler(void) {
    cli();
    if (!(d12_flags & D12_FLAG_SUSPEND))
        goto end;

    if (!d12_is_suspend())
        goto end;
    sei();
    _delay_ms(1);
    if (!d12_is_suspend())
        goto end;

    set_power_save();
    sleep_cpu();

    cli();

end:
    sei();
    return;
}

```

Main routine for d12, poll interrupts and check suspend state.

```

void d12_main(void) {
    d12_interrupt_handler();
    d12_suspend_handler();
}

void d12_suspend_callback(uint8_t bla) {
    if (d12_is_suspend()) {
        d12_flags |= D12_FLAG_SUSPEND;
    } else {
        d12_flags &= ~D12_FLAG_SUSPEND;
    }
}

```

Reset the D12 stack after a bus reset.

```

void d12_bus_reset(uint8_t i) {
    d12_flags = 0;
    tx_bytes = 0;
    tx_ptr = NULL;

    uint8_t buf[1];

    /* set address and enable */
    buf[0] = 0x80;
    d12_write_cmd(D12_CMD_SET_ADDRESS, buf, 1);

    /* enable generic endpoints */
    buf[0] = 0x01;
    d12_write_cmd(D12_CMD_SET_EP_ENABLE, buf, 1);
}

```

2.2 USB enumeration handling.

The USB enumeration process is mostly the same as on the AT90USB1286, except that interrupts from the USB controller have to be fetched explicitly and acted upon. This dispatching is done through an event list with callback function pointers. Generic endpoint callback, just clear interrupt status, don't answer.

```
void d12_generic_ep_callback(uint8_t ep) {
    uint8_t buf[1];
    d12_read_cmd(D12_CMD_READ_LAST_TX_STATUS + ep, buf, 1);
}

#ifndef D12_EP2_IN_CALLBACK
#define D12_EP2_IN_CALLBACK d12_generic_ep_callback
#endif /* D12_EP2_IN_CALLBACK */
#ifndef D12_EP2_OUT_CALLBACK
#define D12_EP2_OUT_CALLBACK d12_generic_ep_callback
#endif /* D12_EP2_OUT_CALLBACK */

#ifndef D12_EP1_IN_CALLBACK
#define D12_EP1_IN_CALLBACK d12_generic_ep_callback
#endif /* D12_EP1_IN_CALLBACK */
#ifndef D12_EP1_OUT_CALLBACK
#define D12_EP1_OUT_CALLBACK d12_generic_ep_callback
#endif /* D12_EP1_OUT_CALLBACK */
```

Standard string descriptor.

```
static const PROGMEM usb_language_id_descriptor_t usb_langid_string = {
    sizeof(usb_language_id_descriptor_t), /* bLength */
    USB_DESCRIPTOR_STRING,               /* bDescriptorType */
    USB_LANGID_US_EN                     /* wLANGID */
};

#include "midi_link_str.h"
```

USB String list to answer GET_STRING_DESCRIPTOR requests. The strings themselves are in midi_link_str.h

```
static const PROGMEM usb_string_list_t usb_string_list[] = {
    { 0, sizeof(usb_langid_string), (uint8_t PROGMEM *)&usb_langid_string },
#ifdef USB_STR_MANUFACTURER
    { 1, sizeof(usb_manufacturer_string), usb_manufacturer_string },
#endif
#ifdef USB_STR_PRODUCT
    { 2, sizeof(usb_product_string), usb_product_string },
#endif
#ifdef USB_STR_SERIAL
    { 3, sizeof(usb_serial_string), usb_serial_string },
#endif
#ifdef USB_STR_CONFIGURATION
    { 4, sizeof(usb_configuration_string), usb_configuration_string },
#endif
#ifdef USB_STR_INTERFACE
```

```

        { 5, sizeof(usb_interface_string), usb_interface_string },
    #endif
};

```

Answer a GET_DESCRIPTOR request on ep0.

```

void d12_get_descriptor(usb_setup_request_t *setup_pkt) {
    tx_ptr = NULL;
    tx_bytes = 0;
    d12_flags &= ~D12_FLAG_FLASH;

    switch (setup_pkt->wValue >> 8) {
    case USB_DESCRIPTOR_DEVICE:
        tx_ptr = (uint8_t *)&usb_device_descriptor;
        tx_bytes = sizeof(usb_device_descriptor);
        d12_flags |= D12_FLAG_FLASH;
        break;

    case USB_DESCRIPTOR_CONFIGURATION:
        tx_ptr = (uint8_t *)&usb_configuration_descriptor;
        tx_bytes = sizeof(usb_configuration_descriptor);
        d12_flags |= D12_FLAG_FLASH;
        break;

    case USB_DESCRIPTOR_STRING:
        {
            uint8_t i;
            for (i = 0; i < sizeof(usb_string_list) / sizeof(usb_string_list_t); i++) {
                if (pgm_read_byte(&usb_string_list[i].index) == (setup_pkt->wValue & 0xFF)) {
                    tx_ptr = GET_FLASH_PTR(&usb_string_list[i].str);
                    tx_bytes = pgm_read_byte(&usb_string_list[i].size);
                    d12_flags |= D12_FLAG_FLASH;
                }
            }
            break;
        }
    }

    if (tx_bytes) {
        if (tx_bytes > setup_pkt->wLength)
            tx_bytes = setup_pkt->wLength;
        d12_write_ep0_buf();
    } else {
        d12_stall_ep0();
    }
}

```

Callback for Endpoint 0 IN, write any data available if necessary, else act according to D12 flags. This interrupt is called periodically apparently, but maybe we should move the code to themain loop.

```

void d12_ep0_in_callback(uint8_t ep) {
    uint8_t status;
    d12_read_cmd(D12_CMD_READ_LAST_TX_STATUS + D12_EP0_IN, &status, 1);
    /* discard status */
}

```

```

if (d12_flags & D12_FLAG_TX_ADDRESS) {
    d12_write_ep_pkt(D12_EP0_IN, NULL, 0);
    d12_write_cmd(D12_CMD_SET_ADDRESS, &d12_dev_address, 1);
    d12_flags &= ~D12_FLAG_TX_ADDRESS;
} else if (d12_flags & D12_FLAG_ENABLE_EP) {
    uint8_t buf[2];
    if (d12_flags & D12_FLAG_DEV_CONFIGURED) {
        buf[0] = 1;
    } else {
        buf[0] = 0;
    }
    /* enable generic endpoints */
    d12_write_cmd(D12_CMD_SET_EP_ENABLE, buf, 1);
    d12_flags &= ~D12_FLAG_ENABLE_EP;
} else {
    /* data tx may be in progress, send more */
    d12_write_ep0_buf();
}
}
}

```

Handle a DEVICE setup packet.

```

void d12_handle_dev_setup_pkt(usb_setup_request_t *setup_pkt) {
    uint8_t buf[2];

    switch (setup_pkt->bRequest) {
    case USB_REQ_GET_STATUS:
        buf[0] = USB_STATUS_SELF_POWERED;
        buf[1] = 0x00;
        d12_write_ep_pkt(D12_EP0_IN, buf, 2);
        break;

    case USB_REQ_CLEAR_FEATURE:
    case USB_REQ_SET_FEATURE:
        /* unsupported */
        d12_stall_ep0();
        break;

    case USB_REQ_SET_ADDRESS:
        d12_dev_address = setup_pkt->wValue | 0x80;
        d12_flags |= D12_FLAG_TX_ADDRESS;
        break;

    case USB_REQ_GET_DESCRIPTOR:
        d12_get_descriptor(setup_pkt);
        break;

    case USB_REQ_GET_CONFIGURATION:
        if (d12_flags & D12_FLAG_DEV_CONFIGURED)
            buf[0] = 1;
        else
            buf[0] = 0;
        d12_write_ep_pkt(D12_EP0_IN, buf, 1);
        break;
    }
}

```

```

case USB_REQ_SET_CONFIGURATION:
    if (setup_pkt->wValue & 0xFF) {
        buf[0] = 1;
        d12_flags |= D12_FLAG_DEV_CONFIGURED | D12_FLAG_ENABLE_EP;
    }
    else {
        buf[0] = 0;
        d12_flags &= ~D12_FLAG_DEV_CONFIGURED | D12_FLAG_ENABLE_EP;
    }
    d12_write_ep_pkt(D12_EP0_IN, NULL, 0);
    break;

case USB_REQ_SET_DESCRIPTOR:
    d12_stall_ep0();
    break;

default:
    d12_stall_ep0();
}
}

```

Handle an interface setup packet.

```

void d12_handle_interface_setup_pkt(usb_setup_request_t *setup_pkt) {
    uint8_t buf[2];
    switch (setup_pkt->bRequest) {
case USB_REQ_GET_STATUS:
        buf[0] = 0x00;
        buf[1] = 0x00;
        d12_write_ep_pkt(D12_EP0_IN, buf, 2);
        break;

case USB_REQ_SET_INTERFACE:
        /* support only first interface */
        if ((setup_pkt->wIndex == 0) && (setup_pkt->wValue == 0))
            d12_write_ep_pkt(D12_EP0_IN, NULL, 0);
        else
            d12_stall_ep0();
        break;

case USB_REQ_GET_DESCRIPTOR:
        d12_get_descriptor(setup_pkt);
        break;

case USB_REQ_GET_INTERFACE:
        if (setup_pkt->wIndex == 0) {
            buf[0] = 0;
            d12_write_ep_pkt(D12_EP0_IN, buf, 1);
        } else {
            d12_stall_ep0();
        }
        break;

case USB_REQ_CLEAR_FEATURE:

```

```

        case USB_REQ_SET_FEATURE:
            d12_stall_ep0();
            break;

        default:
            d12_stall_ep0();
            break;
    }
}

```

Handle an endpoint setup packet.

```

void d12_handle_ep_setup_pkt(usb_setup_request_t *setup_pkt) {
    uint8_t buf[2];

    switch (setup_pkt->bRequest) {
    case USB_REQ_CLEAR_FEATURE:
    case USB_REQ_SET_FEATURE:
        /* supports only STALL feature */
        if (setup_pkt->wValue == USB_FEATURE_EP_HALT) {
            if (setup_pkt->bRequest == USB_REQ_CLEAR_FEATURE)
                buf[0] = 0;
            else
                buf[0] = 1;

            uint8_t ep = setup_pkt->wIndex & 0x7F;
            if (ep > 2) {
                d12_stall_ep0();
            } else {
                if (setup_pkt->wIndex & USB_DIR_IN) {
                    d12_write_cmd(D12_CMD_SET_EP_STATUS + ep * 2 + 1, buf, 1);
                } else {
                    d12_write_cmd(D12_CMD_SET_EP_STATUS + ep * 2, buf, 1);
                }
                d12_write_ep_pkt(D12_EP0_IN, NULL, 0);
            }
        } else {
            d12_stall_ep0();
        }
        break;

    case USB_REQ_GET_STATUS:
        {
            /* return 1 if endpoint is stalled */
            uint8_t ep = setup_pkt->wIndex & 0x7F;
            if (ep > 2) {
                d12_stall_ep0();
            } else {
                if (setup_pkt->wIndex & USB_DIR_IN) {
                    d12_read_cmd(D12_CMD_READ_EP_STATUS + ep * 2 + 1, buf, 1);
                } else {
                    d12_read_cmd(D12_CMD_READ_EP_STATUS + ep * 2, buf, 1);
                }
            }
            if (buf[0] & 0x80) {
                buf[0] = 1; /* endpoint is stalled */
            }
        }
    }
}

```

```

    } else {
        buf[0] = 0;
    }
    d12_write_ep_pkt(D12_EP0_IN, buf, 2);
    }
    }
    break;

default:
    d12_stall_ep0();
    break;
}
}

```

Dispatch an incoming setup packet, ignore vendor and class requests (not needed for MIDI).

```

void d12_handle_setup_pkt(usb_setup_request_t *setup_pkt) {
    switch (setup_pkt->bmRequestType & USB_REQ_TYPE_MASK) {
    case USB_REQ_STANDARD:
        switch (setup_pkt->bmRequestType & USB_REQ_TARGET_MASK) {
        case USB_REQ_DEV:
            d12_handle_dev_setup_pkt(setup_pkt);
            break;

        case USB_REQ_IF:
            d12_handle_interface_setup_pkt(setup_pkt);
            break;

        case USB_REQ_EP:
            d12_handle_ep_setup_pkt(setup_pkt);
            break;

        default:
            d12_stall_ep0();
            break;
        }
        break;

    case USB_REQ_VENDOR:
        /* ignore */
        d12_write_ep_pkt(D12_EP0_IN, NULL, 0);
        break;

    case USB_REQ_CLASS:
        d12_write_ep_pkt(D12_EP0_IN, NULL, 0);
        break;

    default:
        d12_stall_ep0();
        break;
    }
}

```

Receive a packet on endpoint 0 (only act on setup packets).

```

void d12_ep0_out_callback(uint8_t ep) {
    uint8_t buf[2];
    d12_read_cmd(D12_CMD_READ_LAST_TX_STATUS + D12_EP0_OUT, buf, 1);

    if (buf[0] & 0x20) {
        usb_setup_request_t setup_pkt;
        /* setup packet */
        uint8_t ret = d12_read_ep_pkt(D12_EP0_OUT, (uint8_t *)&setup_pkt, sizeof(setup_pkt));
        ret = 0;
        /* ack setup pkt */
        d12_write_cmd(D12_CMD_ACK_SETUP, NULL, 0);
        /* flush buffer after ack */
        d12_write_cmd(D12_CMD_CLEAR_BUFFER, NULL, 0);
        d12_write_cmd(D12_CMD_SELECT_EP + D12_EP0_IN, NULL, 0);
        d12_write_cmd(D12_CMD_ACK_SETUP, NULL, 0);
        d12_write_cmd(D12_CMD_CLEAR_BUFFER, NULL, 0);

        d12_handle_setup_pkt(&setup_pkt);
    }
}

```

Interrupt state callback list.

```

static const PROGMEM usb_event_list_t event_list[] = {
    { 0x80, d12_suspend_callback, 0 },
    { 0x40, d12_bus_reset, 0 },
    { 0x20, D12_EP2_IN_CALLBACK, D12_EP2_IN },
    { 0x10, D12_EP2_OUT_CALLBACK, D12_EP2_OUT },
    { 0x08, D12_EP1_IN_CALLBACK, D12_EP1_IN },
    { 0x04, D12_EP1_OUT_CALLBACK, D12_EP1_OUT },
    { 0x02, d12_ep0_in_callback, D12_EP0_IN },
    { 0x01, d12_ep0_out_callback, D12_EP0_OUT },
    // { 0x02, d12_generic_ep_callback, D12_EP0_IN },
    // { 0x01, d12_generic_ep_callback, D12_EP0_OUT },
};

//static uint8_t int_cnt = 0;

```

Read the interrupt status if D12 is in interrupt state (INT0 pulled down).

```

void d12_interrupt_handler(void) {
    if (!d12_is_irq())
        return;
    uint8_t irq[2];
    d12_read_cmd(D12_CMD_READ_INT_REG, irq, 2);

    uint8_t i;
    for (i = 0; i < sizeof(event_list) / sizeof(usb_event_list_t); i++) {
        if (irq[0] & GET_FLASH(&(event_list[i].mask))) {
            void (*function)(uint8_t);
            function = GET_FLASH_PTR(&(event_list[i].function));
            function(GET_FLASH(&(event_list[i].param)));
        }
    }
}

```


3 USB MIDI implementation

```
#include "app.h"

#include <avr/interrupt.h>
#include <util/delay.h>
#include "common.h"
#include "usb.h"
#include "usb_midi.h"
#include "uart.h"
#include "d12.h"
#include "midi_link_desc.h"
```

Conversion table from “real” midi to USB MIDI.

First parameter is the “real” midi message type, second is the state-machine state used to parse the incoming “real” midi message.

```
const midi_to_usb_t midi_to_usb[] = {
    { MIDI_NOTE_OFF,          midi_wait_byte_2,    CIN_NOTEOFF },
    { MIDI_NOTE_ON,           midi_wait_byte_2,    CIN_NOTEON },
    { MIDI_AFTER_TOUCH,       midi_wait_byte_2,    CIN_POLYKEYPRESS }, /* ??? */
    { MIDI_CONTROL_CHANGE,    midi_wait_byte_2,    CIN_CONTROL_CHANGE },
    { MIDI_PROGRAM_CHANGE,    midi_wait_byte_1,    CIN_PROGRAM_CHANGE },
    { MIDI_CHANNEL_PRESSURE,  midi_wait_byte_1,    CIN_CHANNEL_PRESSURE },
    { MIDI_PITCH_WHEEL,       midi_wait_byte_2,    CIN_PITCHBEND_CHANGE },
    /* special handling for SYSEX */
    { MIDI_MTC_QUARTER_FRAME, midi_wait_byte_1,    CIN_SYSCOMMON_2BYTES },
    { MIDI_SONG_POSITION_PTR, midi_wait_byte_2,    CIN_SYSCOMMON_3BYTES },
    { MIDI_SONG_SELECT,       midi_wait_byte_1,    CIN_SYSCOMMON_2BYTES },
    { MIDI_TUNE_REQUEST,      midi_wait_status,    CIN_SYSCOMMON_1BYTE },
    { 0, 0, 0 }
};
```

volatile is important - gcc bug

```
static volatile midi_state_t    in_state;
static volatile uint8_t         last_status = 0;
static volatile uint8_t         running_status = 0;
static volatile uint8_t         in_msg_len;
static volatile usb_midi_message_t in_msg;

static volatile uint8_t midi_buf[MIDI_BUF_SIZE];
static volatile uint8_t midi_buf_len;

#define PRINTF_APP(args, ...)
```

Initialise the usb midi implementation (the receiving state machine).

```
void usb_midi_init(void) {
    last_status = 0;
    in_state = midi_ignore_message;
    in_msg.cable = 0x00;
    midi_buf_len = 0;
}
```

```

// XXX optimize sending without buffers
void usb_midi_send_buf(void) {
    if (midi_buf_len > 0) {
        d12_write_ep_pkt(D12_MIDI_EP_IN, midi_buf, midi_buf_len);
        midi_buf_len = 0;
    }
}

```

handle a byte from the MIDI IN port. It parses midi messages and generates USB messages appropriately.

```

void usb_midi_handle_rx_byte(uint8_t byte) {
    again:
    /* realtime messages have to be sent directly */
    if (MIDI_IS_REALTIME_STATUS_BYTE(byte)) {
        usb_midi_message_t msg;
        msg.cable = 0x00;
        msg.cin = CIN_SINGLE_BYTE;
        msg.msg[0] = byte;
        msg.msg[1] = msg.msg[2] = 0x00;
        usb_midi_send_message(&msg);
        return;
    }

    switch (in_state) {
    case midi_ignore_message:
        if (MIDI_IS_STATUS_BYTE(byte)) {
            in_state = midi_wait_status;
            goto again;
        } else {
            /* ignore */
        }
        break;

        /* handle sysex messages */
    case midi_wait_sysex:
        if (MIDI_IS_STATUS_BYTE(byte) && (byte != MIDI_SYSEX_END)) {
            /* a status byte interrupts a sysex request */
            in_state = midi_wait_status;
            goto again;
        }

        /* copy byte into message */
        in_msg.msg[in_msg_len] = byte;

        if (byte == MIDI_SYSEX_END) {
            /* end a sysex request */
            if (in_msg_len == 0) {
                in_msg.cin = CIN_SYSCOMMON_1BYTE;
                in_msg.msg[1] = in_msg.msg[2] = 0;
            } else if (in_msg_len == 1) {
                in_msg.cin = CIN_SYSEX_END_2BYTES;
                in_msg.msg[2] = 0;
            } else if (in_msg_len == 2) {

```

```

in_msg.cin = CIN_SYSEX_END_3BYTES;
}

usb_midi_send_message(&in_msg);
in_state = midi_wait_status;
return;
}

in_msg_len++;
if (in_msg_len >= 3) {
    in_msg.cin = CIN_SYSEX_START;
    usb_midi_send_message(&in_msg);
    in_msg_len = 0;
}
break;

case midi_wait_status:
{
    /* special handling for sysex */
    if (byte == MIDI_SYSEX_START) {
        in_state = midi_wait_sysex;
        in_msg_len = 1;
        in_msg.cin = CIN_SYSEX_START;
        in_msg.msg[0] = byte;
        running_status = 0;
        last_status = 0;
        return;
    }

    if (MIDI_IS_STATUS_BYTE(byte)) {
        last_status = byte;
        running_status = 0;
    } else {
        if (last_status == 0)
            break;
        running_status = 1;
    }

    uint8_t status = last_status;
    if (MIDI_IS_VOICE_STATUS_BYTE(status)) {
        status = MIDI_VOICE_TYPE_NIBBLE(status);
    }

    int i;
    for (i = 0; midi_to_usb[i].midi_status != 0; i++) {
        if (midi_to_usb[i].midi_status == status) {
            in_msg.cin = midi_to_usb[i].cin;
            in_state = midi_to_usb[i].next_state;
            in_msg.msg[0] = last_status;
            in_msg_len = 1;

            /* if we don't have to wait for data send message */
            if (in_state == midi_wait_status)
                usb_midi_send_message(&in_msg);

```

```

        break;
    }
    }
    /* status byte not found */
    if (midi_to_usb[i].midi_status == 0) {
PRINTF_APP("Ignore message\n");
in_state = midi_ignore_message;
return;
    }
    if (running_status)
goto again;
    }
    break;

case midi_wait_byte_1:
    in_msg.msg[in_msg_len++] = byte;
    usb_midi_send_message(&in_msg);
    in_state = midi_wait_status;
    break;

case midi_wait_byte_2:
    in_msg.msg[in_msg_len++] = byte;
    in_state = midi_wait_byte_1;
    break;
}
}

```

helper to send 16 bit value over sysex

```

void usb_midi_send_time(uint16_t timer) {
    uint8_t buf[16];
    uint8_t i = 0;
    buf[i++] = 0xf0;

    if (timer == 0) {
        buf[i++] = '0';
    } else {
        for (; (i < 14) && (timer != 0) ; i++) {
            uint8_t ch = timer & 0xf;
            if (ch < 10)
                buf[i] = '0' + ch;
            else
                buf[i] = 'a' + (ch - 10);
            timer >>= 4;
        }
    }
    uint8_t len = i;
    buf[len] = 0xf7;
    if (i > 1) {
        for (i--; i > (len / 2); i--) {
            uint8_t tmp = buf[len - i];
            buf[len - i] = buf[i];
            buf[i] = tmp;
        }
    }
}

```

```

    usb_midi_send_sysex(buf, len + 1);
}

```

helper function to send a description of an array over MIDI to debug on the receiving side.

```

void usb_midi_send_key_state(uint8_t *arr, uint8_t len) {
    uint8_t buf[32];
    uint8_t i = 0;
    uint8_t j = 0;
    buf[i++] = 0xf0;

    for (j = 0; j < len; j++) {
        uint8_t ch = arr[j] & 0xf;
        if (ch < 10)
            buf[i++] = '0' + ch;
        else
            buf[i++] = 'a' + (ch - 10);
        ch = (arr[j] >> 4) & 0xf;
        if (ch < 10)
            buf[i++] = '0' + ch;
        else
            buf[i++] = 'a' + (ch - 10);
    }
    buf[i++] = 0xf7;
    usb_midi_send_sysex(buf, i);
}

```

Send a MIDI message on the usb interface by writing a usb package to the endpoint.

```

void usb_midi_send_message(usb_midi_message_t *msg) {
    if (msg->msg[0] != 0xFE)
        PRINTF_APP("in_msg: cin: %x, b0: %x, b1: %x, b2: %x\n",
            msg->cin, msg->msg[0], msg->msg[1], msg->msg[2]);

    again:
    if (midi_buf_len + 4 <= MIDI_BUF_SIZE) {
        midi_buf[midi_buf_len++] = (msg->cin) | (msg->cable << 4);
        midi_buf[midi_buf_len++] = msg->msg[0];
        midi_buf[midi_buf_len++] = msg->msg[1];
        midi_buf[midi_buf_len++] = msg->msg[2];
        return;
    } else {

```

If midi send buffer is full, loop a bit through USB stack to get rid of the buffer (through midi EP in callback.

```

        d12_main();

        goto again;
    }
}

```

Send a note message on the given channel, with given code, key and velocity.

```

void usb_midi_send_note_message(uint8_t channel, uint8_t code, uint8_t key, uint8_t velocity)
{
    usb_midi_message_t msg;
    if (key > 0xF7)
        return;
    msg.cable = 0x00;
    if (code == MIDI_NOTE_ON) {
        msg.cin = CIN_NOTEON;
    } else {
        msg.cin = CIN_NOTEOFF;
    }
    msg.msg[0] = code | channel;
    msg.msg[1] = key;
    msg.msg[2] = velocity;

    usb_midi_send_message(&msg);
}

```

Send a program change on the given channel.

```

void usb_midi_send_program_change(uint8_t channel, uint8_t program) {
    usb_midi_message_t msg;
    if (program > 0x7F)
        return;
    msg.cable = 0x00;
    msg.cin = CIN_PROGRAM_CHANGE;
    msg.msg[0] = MIDI_PROGRAM_CHANGE | channel;
    msg.msg[1] = program;
    msg.msg[2] = 0;

    usb_midi_send_message(&msg);
}

```

Send a channel pressure message on the given channel.

```

void usb_midi_send_channel_pressure(uint8_t channel, uint8_t pressure) {
    usb_midi_message_t msg;
    if (pressure > 0x7F)
        return;
    msg.cable = 0x00;
    msg.cin = CIN_CHANNEL_PRESSURE;
    msg.msg[0] = MIDI_CHANNEL_PRESSURE | channel;
    msg.msg[1] = pressure;
    msg.msg[2] = 0;

    usb_midi_send_message(&msg);
}

```

Send a sysex message.

```

void usb_midi_send_sysex(uint8_t *buf, uint16_t len) {
    usb_midi_message_t msg;
    int i;

    msg.cable = 0x00;

```

```

    for (i = 0; i < len - 3; i+=3) {
        msg.cin = CIN_SYSEX_START;
        msg.msg[0] = buf[i];
        msg.msg[1] = buf[i+1];
        msg.msg[2] = buf[i+2];
        usb_midi_send_message(&msg);
    }
    uint8_t togo = len - i;
    if (togo == 1) {
        msg.cin = CIN_SYSCOMMON_1BYTE;
        msg.msg[0] = buf[i];
        msg.msg[1] = 0;
        msg.msg[2] = 0;
        usb_midi_send_message(&msg);
    } else if (togo == 2) {
        msg.cin = CIN_SYSEX_END_2BYTES;
        msg.msg[0] = buf[i];
        msg.msg[1] = buf[i+1];
        msg.msg[2] = 0;
        usb_midi_send_message(&msg);
    } else if (togo == 3) {
        msg.cin = CIN_SYSEX_END_3BYTES;
        msg.msg[0] = buf[i];
        msg.msg[1] = buf[i+1];
        msg.msg[2] = buf[i+2];
        usb_midi_send_message(&msg);
    }
}

```

Send a debug string stored in program memory as sysex.

```

void usb_midi_send_debug(PGM_VOID_P buf, uint16_t len) {
    uint8_t buf2[len + 2];
    buf2[0] = 0xF0;
    uint8_t i;
    for (i = 0; i < len; i++) {
        buf2[i+1] = pgm_read_byte_near(buf + i);
    }
    buf2[i+1] = 0xF7;
    usb_midi_send_sysex(buf2, len + 2);
}

```

Write a midi byte to the serial interface (MIDI OUT).

```

void midi_uart_putc(uint8_t byte) {
    PRINTF_APP("out: %x\n", byte);
    uart_putc(byte);
}

```

Handle an incoming USB MIDI message. Running status is commented out.

```

int usb_midi_handle_out_msg(uint8_t ep, usb_midi_message_t *out_msg) {
    static uint8_t out_last_status = 0;

```

```

PRINTF_APP("out_msg: cin: %x, b0: %x, b1: %x, b2: %x\n",
           out_msg->cin, out_msg->msg[0], out_msg->msg[1], out_msg->msg[2]);

if (out_msg->cable != 0) {
    PRINTF_APP("No such cable: %d\n", out_msg->cable);
    return 0;
}

switch (out_msg->cin) {
case CIN_SINGLE_BYTE:
case CIN_SYSCOMMON_1BYTE:
    if (!MIDI_IS_REALTIME_STATUS_BYTE(out_msg->msg[0]))
        out_last_status = 0;

    midi_uart_putc(out_msg->msg[0]);
    break;

case CIN_SYSCOMMON_2BYTES:
case CIN_SYSEX_END_2BYTES:
case CIN_PROGRAM_CHANGE:
case CIN_CHANNEL_PRESSURE:
    #if 0

```

This would be running status.

```

        if (MIDI_IS_STATUS_BYTE(out_msg->msg[0]) &&
            (out_last_status != out_msg->msg[0])) {
            midi_uart_putc(out_msg->msg[0]);
            out_last_status = out_msg->msg[0];
        }
    #endif
    midi_uart_putc(out_msg->msg[0]);
    midi_uart_putc(out_msg->msg[1]);
    break;

case CIN_SYSCOMMON_3BYTES:
case CIN_SYSEX_START:
case CIN_SYSEX_END_3BYTES:
case CIN_NOTEON:
case CIN_NOTEOFF:
case CIN_POLYKEYPRESS:
case CIN_CONTROL_CHANGE:
case CIN_PITCHBEND_CHANGE:
    #if 0

```

This would be running status.

```

        if (MIDI_IS_STATUS_BYTE(out_msg->msg[0]) &&
            (out_last_status != out_msg->msg[0])) {
            midi_uart_putc(out_msg->msg[0]);
            out_last_status = out_msg->msg[0];
        }
    #endif
    midi_uart_putc(out_msg->msg[0]);

```



```
        midi_uart_putc(out_msg->msg[1]);
        midi_uart_putc(out_msg->msg[2]);
        break;

    case CIN_MISC:
    case CIN_CABLE:
    default:
        PRINTF_APP("Ignoring undefined midi request\n");
        break;
}

return 1;
}
```