kbdwtchdg Documentation

Release 0.9

Philipp Rathmanner

Aug 31, 2017

Contents

CHAPTER 1

Overview

A watchdog running V-USB on an Attiny85 that identifies itself as a USB keyboard and sends keyboard strokes.

CHAPTER 2

What Can kbdwtchdg Do?

WTCHDG Mode

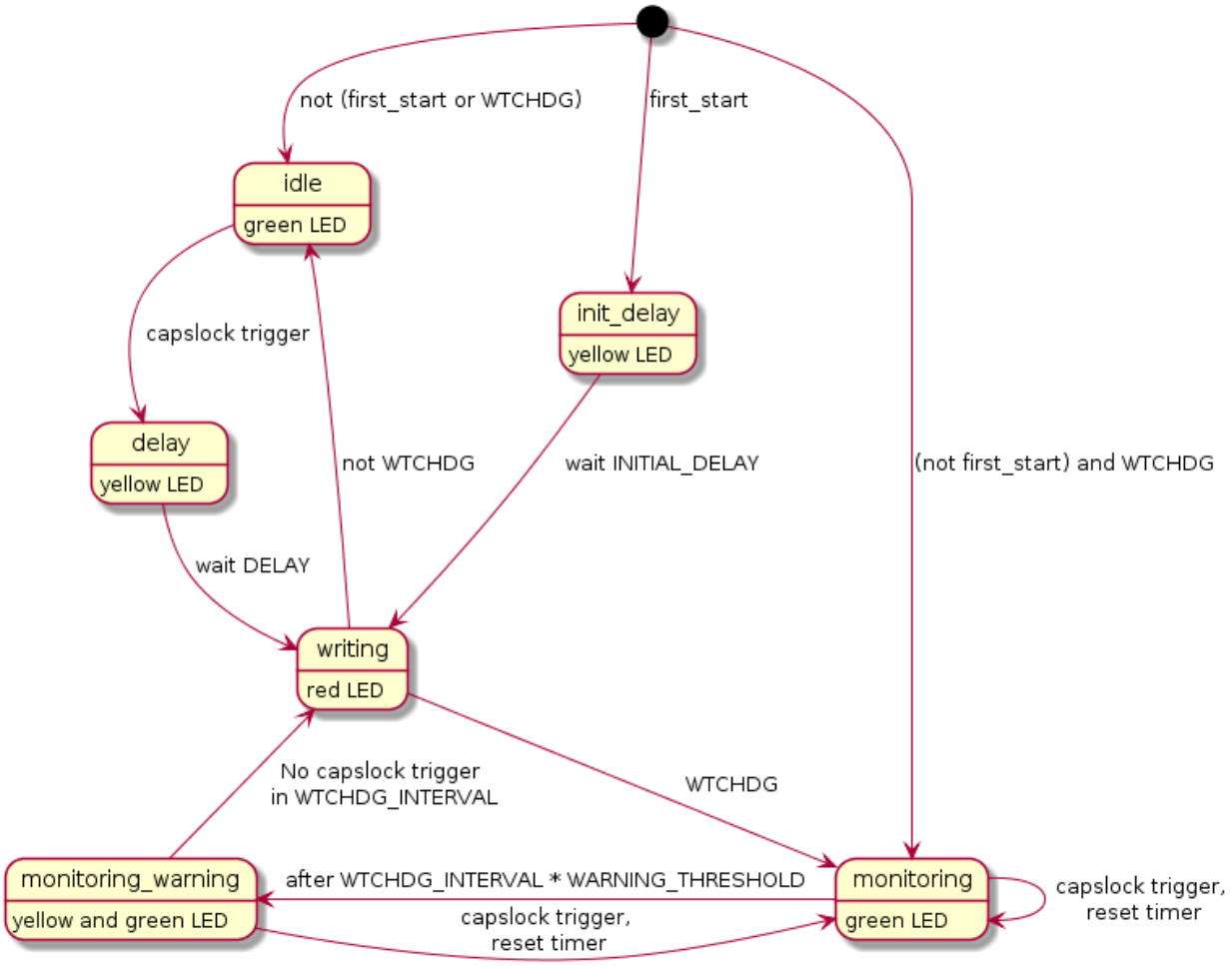
In WTCHDG mode, kbdwtchdg listens for the capslock trigger to occur during WTCHDG_INTERVAL. If no trigger occurs, the text is written. If a trigger occurs, the timer is reset.

Non-WTCHDG Mode

After receiving the capslock trigger and waiting for DELAY time the text is written.

Initial Writing

If first_start is set the text is written after INITIAL_DELAY after power up.


not (first_start or WTCHDG)
first_start
idle
green LED
capslock trigger
init_delay
yellow LED
delay
yellow LED
not WTCHDG
wait INITIAL_DELAY
(not first_start) and WTCHDG
wait DELAY
writing
red LED
No capslock trigger
in WTCHDG_INTERVAL
WTCHDG
monitoring_warning
yellow and green LED
after WTCHDG_INTERVAL * WARNING_THRESHOLD
monitoring
green LED
capslock trigger,
reset timer
capslock trigger,
reset timer

CHAPTER 3

How to Use

The repo contains an AtmelStudio 7 project file that is preconfigured for kbdwtchdg.

CHAPTER 4

Acknowledgements

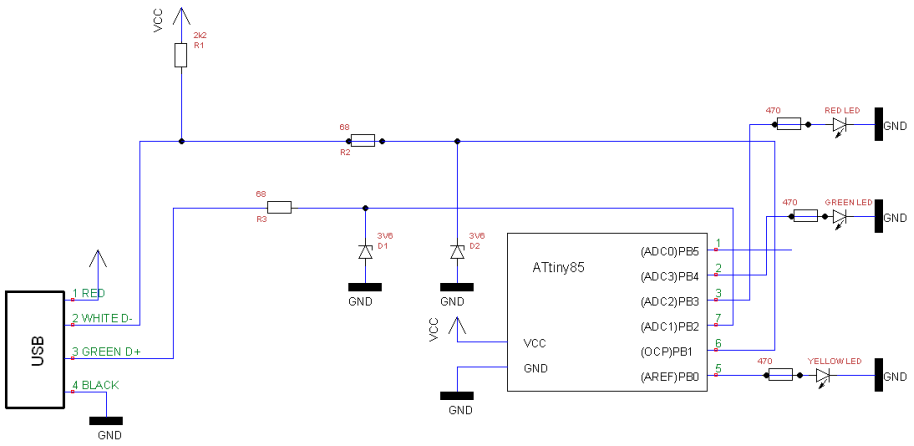
The code of this project is based on Frank Zhao's USB business card and built based on Dovydas R.'s circuit diagram for usb_pass_input_with_buttons.

This documentation was built using antiweb

Project setup

Hardware preparation

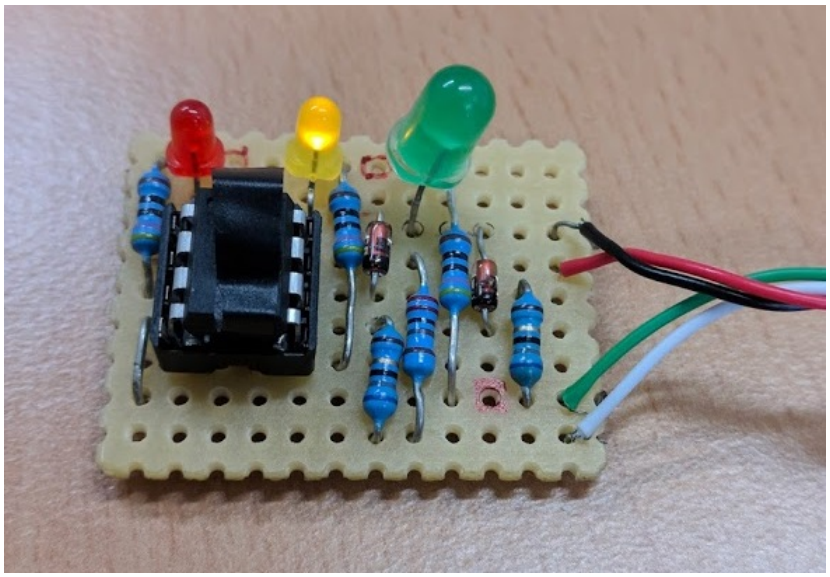
Before programming your Attiny85 to run kbdwtchdg you need to build your circuit. We built our project according to the following diagram:



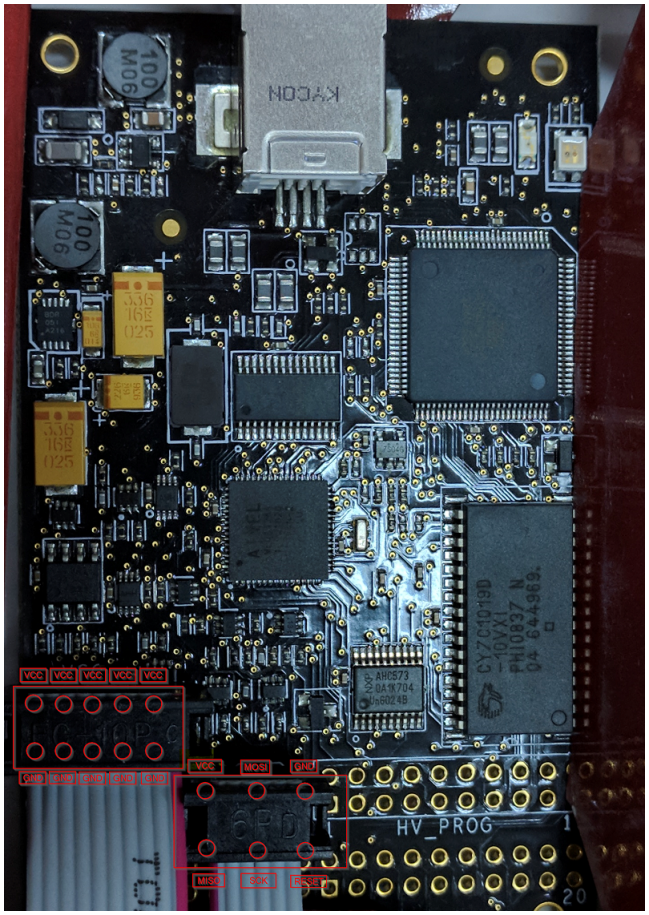
Title	kbdwtchdg	
Author	Dovydas R. & Philipp R.	
[redacted]		Document
Revision 1.0	Date 14.08.2017	Sheets 1 of 1

Below you can find our suggested layout for the soldering:

This is the finished ATtiny85 board:

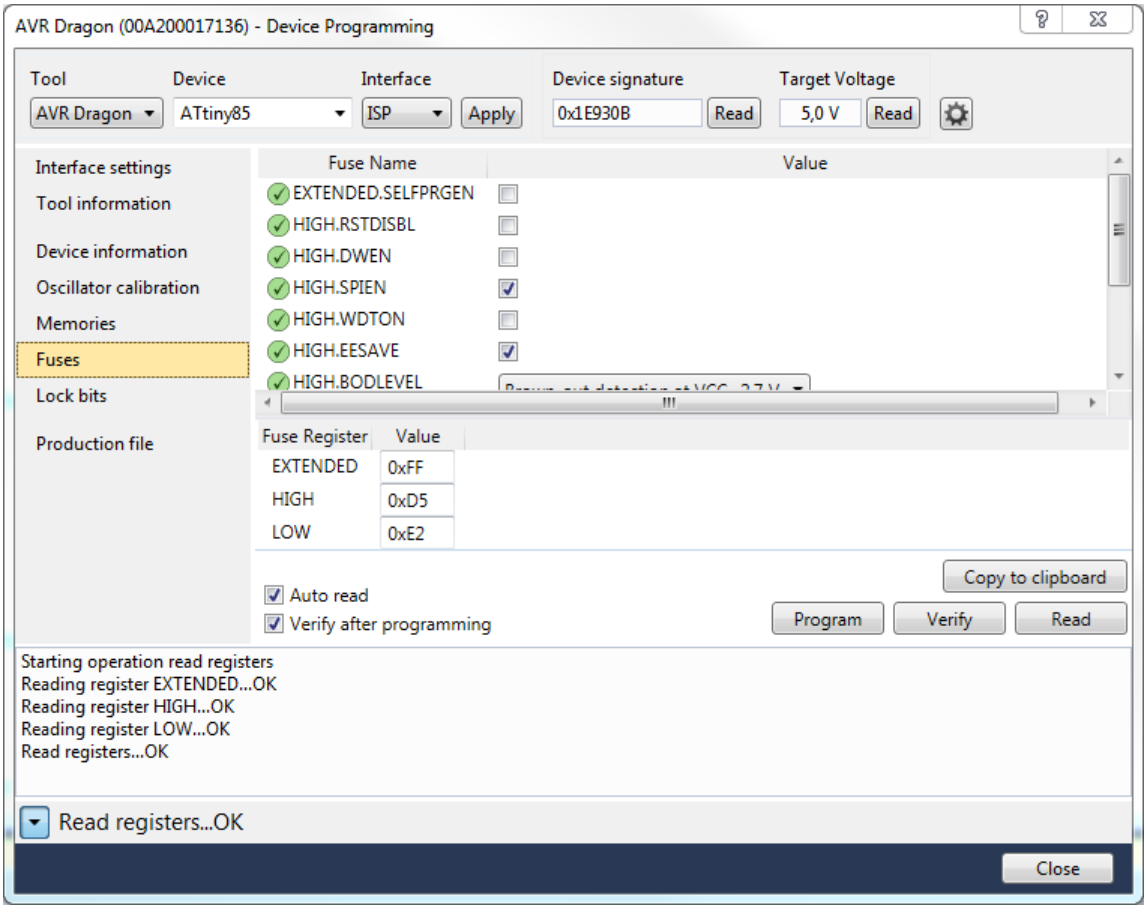


Below is an example photo of the connections on the AVR Dragon programmer:



AtmelStudio 7

There is an AtmelStudio 7 project file inside the repository (kbdwtchdg.atsln). It is preconfigured to use the kbdwtchdg folder as its project folder. After you made sure all your wires are connected and you selected your programmer (you can check that by reading the voltage and device signature in Tools -> Device Programming) you can build the project using Build/Build kbdwtchdg. After that, you need to set the correct fuses. Go to Device Programming -> Fuses and set `EXTENDED`, `HIGH` and `LOW` to the following values. All the fuses will be set correctly after clicking program:



If there are no errors you can proceed to load the project onto your Microcontroller using Debug -> Start without Debugging (Ctrl+Alt+F5).

For problems concerning V-USB, please take a look at their Troubleshooting site.

Configuring the Project

See Variables.

Source code

main() function

The `main()` function consists of four important parts:

- The setup calls to initiate a connection,
- The WTCHDG mode, which will write text in a specific interval if not reset by capslock.
- The waiting mode, which will write text on start up and after a capslock trigger.
- a function called `usbPoll();` which will keep the connection alive

```
int main()
{
   uint8_t calibrationValue = eeprom_read_byte(0); /* calibration value from last␣
↪time */

   if (calibrationValue != 0xFF)
   {
      OSCCAL = calibrationValue;
   }

   setup_timer();

   sei(); // enable global interrupt

   stdout = &mystdout; // set default stream

   // initialize report (I never assume it's initialized to 0 automatically)
   keyboard_report_reset();

   wdt_disable(); // disable ATtiny85's internal watchdog (no connection to our␣
↪wtchdg)
               // good habit if you don't use it

   // enforce USB re-enumeration by pretending to disconnect and reconnect
   usbDeviceDisconnect();
   _delay_ms(250);
   usbDeviceConnect();

   // initialize various modules
   usbInit();

   State state;
   if (first_start)
   {
      state = init_delay; // do a first start
      activate_led(LED_YELLOW); // activate yellow led on startup
   }
   else // skip the first_start
   {
      if (WTCHDG) // we are in WTCHDG mode
      {
         state = monitoring; // skip initial state and writing state, go to monitoring
      }
      else
      {
         state = idle; // skip initial and writing state, go to idle
      }
   }

  while (1) // main loop, do forever
  {
      // check the current state to choose the next appropriate state in the chain
      switch (state)
      {
         case init_delay: // perform a delay using INITIAL_DELAY

            if (!delay_started)
            {
```

```
            start_delay();
         }

         if (timer_count >= (begin_delay + INITIAL_DELAY)) // initial delay at␣
↪first start
         {
            state = writing;
         }
         break;

      case delay: // capsloock has been triggered, perform a delay

        // starting the delay before writing
         if (!delay_started) // dont't enter if the delay interval already started
         {
            start_delay();
         }

         if (timer_count >= (begin_delay + DELAY)) // delay after capslock trigger
         {
               state = writing;
         }
         break;

      case monitoring: // while in monitoring state, check for capslock triggers

         activate_led(LED_GREEN);

         state = check_trigger(state);

         if(timer_count > WARNING_INTERVAL)
         {
            state = monitoring_warning; // go to monitoring_warning
         }
         break;

      case monitoring_warning:

         activate_2_leds(LED_GREEN, LED_YELLOW); // 2 LEDs, indicate warning

         state = check_trigger(state); // check if user has sent trigger to reset␣
↪timer

         if (timer_count > WTCHDG_INTERVAL) // no trigger in interval
         {
            state = writing; // write after interval has passed in WTCHDG mode
         }
         break;

      case writing: // print out our text, proceed to next state

         write();

         if (WTCHDG) // we are in WTCHDG mode
         {
            state = monitoring;
         }
         else
```

```
            {
               state = idle;
            }
            break;

         case idle: // wait for capslock trigger

            activate_led(LED_GREEN); // Turn on Green LED to indicate idle state
            state = check_trigger(state);
            break;
    } // switch

      // perform usb related background tasks
      usbPoll(); // this needs to be called at least once every 10 ms
      // this is also called in send_report_once

  } // while
   return 0;
}
```

Variables

The user can edit the following variables to adjust kbdwtchdg:

```
//USER VARIABLES

#define WTCHDG 1 // Change between two modes. If 1, WTCHDG mode is active
                 //(press capslock at least "THRESHOLD" times in the defined interval,
                 //otherwise write TEXT).
                 //If 0, waiting mode is active (press capslock > THRESHOLD to write␣
↪TEXT).

#define WTCHDG_INTERVAL 3000 // Set interval for WTCHDG mode (in 1/100 seconds)

#define WARNING_THRESHOLD 0.8 // Percentage (given between 0 and 1) of WTCHDG_INTERVAL
                              // after which monitoring_warning state is entered

#define BLINK_INTERVAL 25 //set interval for blinking LED

#define DELAY 600 // delay (in 1/100th of seconds) to wait after pressing capslock
                  // before writing string; max: ~ 5.8*10^9 years
                  // has no effect in WTCHDG mode

#define INITIAL_DELAY 300  //Delay (in 1/100th of seconds) after power
                           // before writing string; max: ~ 5.8*10^9 years

uint8_t first_start = 1; //set to 1 if you want kbdwtchdg to write
                         //on power up. Otherwise set to 0

#define THRESHOLD 3 //pressing capslock more than 3 times triggers the counter

#define TEXT PSTR("Hello World! This is kbdwatchdog!\n") //Text to be written

#define INTER_KEY_DELAY 100 // delay between key presses in milliseconds
                            //comment out whole definition if no delay is desired
```

```
// Defining the bits to set LED outputs:

#define LED_RED (1 << PB3) //Turn on red led on PB3
#define LED_GREEN (1 << PB4) //Turn on green led on PB4
#define LED_YELLOW (1 << PB0) //Turn on yellow led on PB0

// End of USER VARIABLES
```

Timer setup

To perform our delays without using `_delay_ms` (which would prevent our ATtiny85 from talking to the computer). We use interrupts which are caused by `timer0` in CTC mode:

```
volatile uint64_t timer_count = 0;
volatile uint64_t wtchdg_blink;
volatile uint64_t begin_delay;
volatile uint8_t delay_started = 0;

typedef enum state { init_delay, writing, idle, monitoring, monitoring_warning, delay
↪} State;

void setup_timer()
{
   DDRB = OUTPUT_BITS; // Setting the output bits

   TCCR0A |= (1 << WGM01); // Configure timer0 to CTC mode

   TIMSK |= (1 << OCIE0A); // Enable CTC interrupt

   OCR0A = F_CPU/1024 * 0.01 - 1; // Get the value to compare our timer with

   TCCR0B |= (1 << CS02)|(1 << CS00); // 1024 Prescaler
}
```

For more information on which bits need to be set, consider looking at the Datasheet

The following function called `start_delay` initiates the delay after which text is being written.

```
void start_delay()
{
   activate_led(LED_YELLOW); // Turn on Yellow LED to indicate waiting state

   begin_delay = timer_count; // remember beginning of delay interval
   delay_started = 1; // delay interval has started
}
```

Interrupt

The following function is called every **1/100 seconds** by `timer0`, it will continue counting to its maximum if not reset.

```
ISR(TIM0_COMPA_vect)
{
  timer_count++; // counting up until reset
```

```
  wtchdg_blink++; // counting up until reset
}
```

Capslock counter

When an output report is received (in our case the LED status of capslock is the only possible output report) the `blink_count` of capslock is being raised.

```
usbMsgLen_t usbFunctionWrite(uint8_t * data, uchar len)
{
   if (data[0] != LED_state)
   {
      // increment count when LED has toggled
      blink_count = blink_count < 10 ? blink_count + 1 : blink_count;
   }

   LED_state = data[0];

   return 1; // 1 byte read
}
```

Activating/toggling an LED

We are turning off all LEDs by doing a bitwise `&` between the current `PORTB` register and the negation of turning on the three LEDs. Afterwards one or two specific LEDs are turned on by a bitwise `|`:

```
void activate_led(uint8_t led)
{
   // turn all LEDs off
   PORTB &= ~(LED_YELLOW | LED_RED | LED_GREEN);

   // turn on specific LED
   PORTB |= (led);

}

void activate_2_leds(uint8_t led1, uint8_t led2)
{
  // turn all LEDs off
  PORTB &= ~(LED_YELLOW | LED_RED | LED_GREEN);

  // turn on 2 LEDs
  PORTB |= ((led1) | (led2));
}
```

Writing Procedure

The writing prodecure consists of turning the RED LED on (to indicate writing) and writing the defined text.

Afterwards `timer_count``and ``blink_count` are reset, `delay startet` and `first_start` are set to false (0).

- `timer_count` is set to 0 so the timer restarts

- `blink_count` needs to be reset to 0 so we can start counting again
- `delay_started` is set to false (0) because the delay already finished
- `first_start` needs to be set to false (0), as the initial delay/first start has already finished

```
void write()
{

  activate_led(LED_RED); // Turn red LED on to represent writing state

  printf_P(TEXT); // Printing our TEXT

  reset_timer();
  first_start = 0; // no first start anymore
  delay_started = 0; // reset delay interval
}
```

Delay Keystrokes

To set a delay between the key presses, the following function will call a delay of 5ms and then `usbPoll();`. This sequence is being repeated until the defined delay is reached.

```
void delay_keystrokes(uint64_t ms)
{
   const uint8_t milliseconds = 5;
   uint64_t loop_count = ms/milliseconds; // get the amount of loops necessary
   uint64_t i;

   // a delay bigger than 10ms would kill the connection, so we split
   // the delay up into little delays that do not harm our connection
   for (i = 0; i <= loop_count; i++)
   {
      _delay_ms(milliseconds);
      usbPoll();
   }
}
```

ASCII to Keycode

To get appropriate keycodes we can send to the computer, each ASCII character needs to be converted to its corresponding keycode:

```
// translates ASCII to appropriate keyboard report, taking into consideration the␣
↪status of caps lock
void ASCII_to_keycode(uint8_t ascii)
{
   keyboard_report.keycode[0] = 0x00;
   keyboard_report.modifier = 0x00;

   // see scancode.doc appendix C

   // delay between the keystrokes
   #ifdef INTER_KEY_DELAY
      delay_keystrokes(INTER_KEY_DELAY);
   #endif
```

```
if (ascii >= 'A' && ascii <= 'Z')
{
   keyboard_report.keycode[0] = 4 + ascii - 'A'; // set letter
   if (bit_is_set(LED_state, 1)) // if caps is on
   {
      keyboard_report.modifier = 0x00; // no shift
   }
   else
   {
      keyboard_report.modifier = _BV(1); // hold shift // hold shift
   }
}
else if (ascii >= 'a' && ascii <= 'z')
{
   keyboard_report.keycode[0] = 4 + ascii - 'a'; // set letter
   if (bit_is_set(LED_state, 1)) // if caps is on
   {
      keyboard_report.modifier = _BV(1); // hold shift // hold shift
   }
   else
   {
      keyboard_report.modifier = 0x00; // no shift
   }
}
else if (ascii >= '0' && ascii <= '9')
{
   keyboard_report.modifier = 0x00;
   if (ascii == '0')
   {
      keyboard_report.keycode[0] = 0x27;
   }
   else
   {
      keyboard_report.keycode[0] = 30 + ascii - '1';
   }
}
else
{
   switch (ascii) // convert ascii to keycode according to documentation
   {
      case '!':
      keyboard_report.modifier = _BV(1); // hold shift
      keyboard_report.keycode[0] = 29 + 1;
      break;
      case '@':
      keyboard_report.modifier = _BV(1); // hold shift
      keyboard_report.keycode[0] = 29 + 2;
      break;
      case '#':
      keyboard_report.modifier = _BV(1); // hold shift
      keyboard_report.keycode[0] = 29 + 3;
      break;
      case '$':
      keyboard_report.modifier = _BV(1); // hold shift
      keyboard_report.keycode[0] = 29 + 4;
      break;
      case '%':
```

```
        keyboard_report.modifier = _BV(1); // hold shift
        keyboard_report.keycode[0] = 29 + 5;
        break;
        case '^':
        keyboard_report.modifier = _BV(1); // hold shift
        keyboard_report.keycode[0] = 29 + 6;
        break;
        case '&':
        keyboard_report.modifier = _BV(1); // hold shift
        keyboard_report.keycode[0] = 29 + 7;
        break;
        case '*':
        keyboard_report.modifier = _BV(1); // hold shift
        keyboard_report.keycode[0] = 29 + 8;
        break;
        case '(':
        keyboard_report.modifier = _BV(1); // hold shift
        keyboard_report.keycode[0] = 29 + 9;
        break;
        case ')':
        keyboard_report.modifier = _BV(1); // hold shift
        keyboard_report.keycode[0] = 0x27;
        break;
        case '~':
        keyboard_report.modifier = _BV(1); // hold shift
        // fall through
        case '`':
        keyboard_report.keycode[0] = 0x35;
        break;
        case '_':
        keyboard_report.modifier = _BV(1); // hold shift
        // fall through
        case '-':
        keyboard_report.keycode[0] = 0x2D;
        break;
        case '+':
        keyboard_report.modifier = _BV(1); // hold shift
        // fall through
        case '=':
        keyboard_report.keycode[0] = 0x2E;
        break;
        case '{':
        keyboard_report.modifier = _BV(1); // hold shift
        // fall through
        case '[':
        keyboard_report.keycode[0] = 0x2F;
        break;
        case '}':
        keyboard_report.modifier = _BV(1); // hold shift
        // fall through
        case ']':
        keyboard_report.keycode[0] = 0x30;
        break;
        case '|':
        keyboard_report.modifier = _BV(1); // hold shift
        // fall through
        case '\\':
        keyboard_report.keycode[0] = 0x31;
```

```
            break;
            case ':':
            keyboard_report.modifier = _BV(1); // hold shift
            // fall through
            case ';':
            keyboard_report.keycode[0] = 0x33;
            break;
            case '"':
            keyboard_report.modifier = _BV(1); // hold shift
            // fall through
            case '\'':
            keyboard_report.keycode[0] = 0x34;
            break;
            case '<':
            keyboard_report.modifier = _BV(1); // hold shift
            // fall through
            case ',':
            keyboard_report.keycode[0] = 0x36;
            break;
            case '>':
            keyboard_report.modifier = _BV(1); // hold shift
            // fall through
            case '.':
            keyboard_report.keycode[0] = 0x37;
            break;
            case '?':
            keyboard_report.modifier = _BV(1); // hold shift
            // fall through
            case '/':
            keyboard_report.keycode[0] = 0x38;
            break;
            case ' ':
            keyboard_report.keycode[0] = 0x2C;
            break;
            case '\t':
            keyboard_report.keycode[0] = 0x2B; // tab
            break;
            case '\n':
            keyboard_report.keycode[0] = 0x28; // enter
            break;
            case '\b':
            keyboard_report.keycode[0] = 0x2A; // backspace
        }
    }
}
```

HID Report Descriptor

The ATtiny85 Microcontroller needs some definitions to be recognized as a HID (Human Interface Device), or keyboard. Those definitions are stored inside the `usbHidReportDescriptor`. The descriptor defines which kind of device your ATtiny85 pretends to be and which keys are available. It gives the user the ability to define many different aspects of a HID. More information on HIDs: USB.org

```
// USB HID report descriptor for boot protocol keyboard
// see HID1_11.pdf appendix B section 1
// USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH is defined in usbconfig
```

```
PROGMEM char usbHidReportDescriptor[USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH] = {
   0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)
   0x09, 0x06,                    // USAGE (Keyboard)
   0xa1, 0x01,                    // COLLECTION (Application)
   0x75, 0x01,                    //   REPORT_SIZE (1)
   0x95, 0x08,                    //   REPORT_COUNT (8)
   0x05, 0x07,                    //   USAGE_PAGE (Keyboard)(Key Codes)
   0x19, 0xe0,                    //   USAGE_MINIMUM (Keyboard LeftControl)(224)
   0x29, 0xe7,                    //   USAGE_MAXIMUM (Keyboard Right GUI)(231)
   0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
   0x25, 0x01,                    //   LOGICAL_MAXIMUM (1)
   0x81, 0x02,                    //   INPUT (Data,Var,Abs) ; Modifier byte
   0x95, 0x01,                    //   REPORT_COUNT (1)
   0x75, 0x08,                    //   REPORT_SIZE (8)
   0x81, 0x03,                    //   INPUT (Cnst,Var,Abs) ; Reserved byte
   0x95, 0x05,                    //   REPORT_COUNT (5)
   0x75, 0x01,                    //   REPORT_SIZE (1)
   0x05, 0x08,                    //   USAGE_PAGE (LEDs)
   0x19, 0x01,                    //   USAGE_MINIMUM (Num Lock)
   0x29, 0x05,                    //   USAGE_MAXIMUM (Kana)
   0x91, 0x02,                    //   OUTPUT (Data,Var,Abs) ; LED report
   0x95, 0x01,                    //   REPORT_COUNT (1)
   0x75, 0x03,                    //   REPORT_SIZE (3)
   0x91, 0x03,                    //   OUTPUT (Cnst,Var,Abs) ; LED report padding
   0x95, 0x06,                    //   REPORT_COUNT (6)
   0x75, 0x08,                    //   REPORT_SIZE (8)
   0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
   0x25, 0x65,                    //   LOGICAL_MAXIMUM (101)
   0x05, 0x07,                    //   USAGE_PAGE (Keyboard)(Key Codes)
   0x19, 0x00,                    //   USAGE_MINIMUM (Reserved (no event␣
↪indicated))(0)
   0x29, 0x65,                    //   USAGE_MAXIMUM (Keyboard Application)(101)
   0x81, 0x00,                    //   INPUT (Data,Ary,Abs)
   0xc0                           // END_COLLECTION
};

// data structure for boot protocol keyboard report
// see HID1_11.pdf appendix B section 1
typedef struct {
   uint8_t modifier;
   uint8_t reserved;
   uint8_t keycode[6];
} keyboard_report_t;

// global variables

static keyboard_report_t keyboard_report;
#define keyboard_report_reset() keyboard_report.modifier=0;keyboard_report.reserved=0;
↪keyboard_report.keycode[0]=0;keyboard_report.keycode[1]=0;keyboard_report.
↪keycode[2]=0;keyboard_report.keycode[3]=0;keyboard_report.keycode[4]=0;keyboard_
↪report.keycode[5]=0;
static uint8_t idle_rate = 500 / 4; // see HID1_11.pdf sect 7.2.4
static uint8_t protocol_version = 0; // see HID1_11.pdf sect 7.2.6
static uint8_t LED_state = 0; // see HID1_11.pdf appendix B section 1
static uint8_t blink_count = 0; // keep track of how many times caps lock have toggled
```

USB Setup Function

The following function is called to receive reports and process them.

```
// see http://vusb.wikidot.com/driver-api
// constants are found in usbdrv.h
usbMsgLen_t usbFunctionSetup(uint8_t data[8])
{
   // see HID1_11.pdf sect 7.2 and http://vusb.wikidot.com/driver-api
   usbRequest_t *rq = (void *)data;

   if ((rq->bmRequestType & USBRQ_TYPE_MASK) != USBRQ_TYPE_CLASS)
   return 0; // ignore request if it's not a class specific request

   // see HID1_11.pdf sect 7.2
   switch (rq->bRequest)
   {
      case USBRQ_HID_GET_IDLE:
      usbMsgPtr = &idle_rate; // send data starting from this byte
      return 1; // send 1 byte
      case USBRQ_HID_SET_IDLE:
      idle_rate = rq->wValue.bytes[1]; // read in idle rate
      return 0; // send nothing
      case USBRQ_HID_GET_PROTOCOL:
      usbMsgPtr = &protocol_version; // send data starting from this byte
      return 1; // send 1 byte
      case USBRQ_HID_SET_PROTOCOL:
      protocol_version = rq->wValue.bytes[1];
      return 0; // send nothing
      case USBRQ_HID_GET_REPORT:
      usbMsgPtr = &keyboard_report; // send the report data
      return sizeof(keyboard_report);
      case USBRQ_HID_SET_REPORT:
      if (rq->wLength.word == 1) // check data is available
      {
         // 1 byte, we don't check report type (it can only be output or feature)
         // we never implemented "feature" reports so it can't be feature
         // so assume "output" reports
         // this means set LED status
         // since it's the only one in the descriptor
         return USB_NO_MSG; // send nothing but call usbFunctionWrite
      }
      else // no data or do not understand data, ignore
      {
         return 0; // send nothing
      }
      default: // do not understand data, ignore
      return 0; // send nothing
   }
}
```

Oscillator Calibration

Calibrating Attiny85's integrated Oscillator to 8.25 MHz:

```
// section copied from EasyLogger
/* Calibrate the RC oscillator to 8.25 MHz. The core clock of 16.5 MHz is
```

```
 * derived from the 66 MHz peripheral clock by dividing. Our timing reference
 * is the Start Of Frame signal (a single SE0 bit) available immediately after
 * a USB RESET. We first do a binary search for the OSCCAL value and then
 * optimize this value with a neighboorhod search.
 * This algorithm may also be used to calibrate the RC oscillator directly to
 * 12 MHz (no PLL involved, can therefore be used on almost ALL AVRs), but this
 * is wide outside the spec for the OSCCAL value and the required precision for
 * the 12 MHz clock! Use the RC oscillator calibrated to 12 MHz for
 * experimental purposes only!
 */
static void calibrateOscillator(void)
{
   uchar       step = 128;
   uchar       trialValue = 0, optimumValue;
   int         x, optimumDev, targetValue = (unsigned)(1499 * (double)F_CPU / 10.5e6␣
↪+ 0.5);

    /* do a binary search: */
    do{
        OSCCAL = trialValue + step;
        x = usbMeasureFrameLength();    /* proportional to current real frequency */
        if(x < targetValue)             /* frequency still too low */
            trialValue += step;
        step >>= 1;
    }while(step > 0);
    /* We have a precision of +/- 1 for optimum OSCCAL here */
    /* now do a neighborhood search for optimum value */
    optimumValue = trialValue;
    optimumDev = x; /* this is certainly far away from optimum */
    for(OSCCAL = trialValue - 1; OSCCAL <= trialValue + 1; OSCCAL++){
        x = usbMeasureFrameLength() - targetValue;
        if(x < 0)
            x = -x;
        if(x < optimumDev){
            optimumDev = x;
            optimumValue = OSCCAL;
        }
    }
    OSCCAL = optimumValue;
}
/*
Note: This calibration algorithm may try OSCCAL values of up to 192 even if
the optimum value is far below 192. It may therefore exceed the allowed clock
frequency of the CPU in low voltage designs!
You may replace this search algorithm with any other algorithm you like if
you have additional constraints such as a maximum CPU clock.
For version 5.x RC oscillators (those with a split range of 2x128 steps, e.g.
ATTiny25, ATTiny45, ATTiny85), it may be useful to search for the optimum in
both regions.
*/

void usbEventResetReady(void)
{
   calibrateOscillator();
   eeprom_update_byte(0, OSCCAL);   /* store the calibrated value in EEPROM */
}
```

Background tasks

Performing obligatory background tasks:

```
void send_report_once()
{
   // perform usb background tasks until the report can be sent, then send it
   while (1)
   {
      usbPoll(); // this needs to be called at least once every 10 ms


      if (usbInterruptIsReady())
      {
         usbSetInterrupt(&keyboard_report, sizeof(keyboard_report)); // send

         break;

         // see http://vusb.wikidot.com/driver-api
      }
   }
}

// stdio's stream will use this funct to type out characters in a string
void type_out_char(uint8_t ascii, FILE *stream)
{
   ASCII_to_keycode(ascii);
   send_report_once();
   keyboard_report_reset(); // release keys
   send_report_once();
}

static FILE mystdout = FDEV_SETUP_STREAM(type_out_char, NULL, _FDEV_SETUP_WRITE); //␣
↪setup writing stream
```

Definitions

The following libraries need to be included:

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include <avr/pgmspace.h>
#include <avr/eeprom.h>
#include <stdio.h>

#include "usbdrv/usbdrv.h"
#include "usbdrv/usbconfig.h"

#define F_CPU 16500000L //Defining a CPU Frequency of 16.5 MHz
#include <util/delay.h>
```

Copyright

```
/*
 This program is free software: you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation, either version 3 of the License, or
 (at your option) any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program.  If not, see <http://www.gnu.org/licenses/>.

Copyright by Frank Zhao (http://www.frank-zhao.com), Philipp Rathmanner (https://
↪github.com/Yarmek) and Christian Eitner (https://github.com/7enderhead)
 */

//The code of this project is based on Frank Zhao's USB business card(http://www.
↪instructables.com/id/USB-PCB-Business-Card/)
//and built based on Dovydas R.'s circuit diagram for "usb_pass_input_with_buttons
↪"(https://github.com/Dovydas-R/usb_pass_input_with_buttons).
```

Extract documentation

A tool called antiweb can extract `.rst` files for use with Sphinx from comments in your code.

Installation

You can use `pip install antiweb` to get the latest version of antiweb.

Usage

You need to set special directives inside your comments. Those are being interpreted by antiweb.

To generate the *.rst* file, navigate to the folder where `main.c` (or your source file) is located and type:

```
antiweb <source file> <options>
```

In this case `antiweb main.c`. That will generate a file called main.rst which can then be used by Sphinx for documentation.

More information

If you want to learn more about antiweb and its usage, click here.

Changelog

- **11.08.2017:** Initial commit

- **16.08.2017:** re-organizing folder structure

  adapting GPL
- **17.08.2017:** added pictures and diagrams

  added documentation
- **21.08.2017:** Added antiweb how-to
- **23.08.2017:** Added WTCHDG mode
- **28.08.2017:** Added state indicator

  new state diagrams

  reworked main() with switch statement
- **30.08.2017:** Added inter-key-delay

  Added documentation for settig the fuses